Oligonucleotide Design and Codon Optimization for PCR-based Gene Synthesis

by

Paul Jamesen Steiner

S.B. Computer Science and Engineering
MIT, 2007

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

September 2009

Signature of Author: _____
Department of Electrical Engineering and Computer Science
August 3, 2009

Certified by: _____
Thomas F. Knight Jr.
Senior Research Scientist
Thesis Supervisor

Certified by: _____
Brian C. Williams
Professor
Thesis Co-Supervisor

Accepted by: _____
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Oligonucleotide Design and Codon Optimization for PCR-based Gene Synthesis

by
Paul Jamesen Steiner

Submitted to the Department of Electrical Engineering and Computer Science on August 3, 2009 in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science.

Abstract

If synthetic biologists are to engineer novel biological functionality, they must be able to fabricate the DNA encoding it. A number of companies synthesize DNA for a fee, but their service is opaque. Researchers can alternatively perform their own syntheses, but the process is time-consuming and error-prone. This thesis introduces a software tool designed to make it simpler and more reliable.

DNA is synthesized from overlapping oligonucleotides by ligation or PCR; this thesis focuses on PCR-based methods. Many sets of oligonucleotides can be used to synthesize a given sequence; choosing the optimal set is a computational problem. A number of software tools for oligonucleotide design exist, but none are adequate. Some employ poorly-designed algorithms, while others place unnecessary restrictions on oligonucleotide length or overlap size.

An optimal set of oligonucleotides for PCR-based synthesis has no potential for mispriming and has maximally uniform overlap melting temperatures. We present an algorithm that finds such a set. Unlike similar algorithms, it places no restrictions on oligo length or overlap size except those given by the user. Mason, a tool employing this algorithm, has been implemented in Common Lisp.

The space of potential sets of oligos is much larger when the DNA to be synthesized contains protein-coding regions; because the genetic code is degenerate, a combinatorial number of different sequences can encode the same protein. If the primary concern is a protein sequence, codons can be changed to synonymous codons with little consequence, making it possible to remove problematic repetitive elements. We show that our algorithm can theoretically be extended and used with constraint optimization algorithms to solve the more difficult problem of simultaneously optimizing codon usage and designing oligonucleotides for synthesis.

Thesis Supervisor: Thomas F. Knight Jr.
Title: Senior Research Scientist

Thesis Co-Supervisor: Brian C. Williams
Title: Professor

Acknowledgments

Thanks to Tom Knight for making it obvious how exciting biology is and for being always available with insight, wisdom, and a joke at the expense of the tasteless.

Thanks to Brian Williams for introducing me to the world of constraints and for reminding me to enjoy the opportunity to spend all my time thinking about interesting problems.

Thanks to Pete Carr and Dave Kong for advice when I started, and thanks to Dave for pointing me to Tom.

Finally, thanks to my mother, my father, and my sisters: Margaret, Ramsey, and Madeline.

Contents

1 Introduction 11

2 Background: Molecular Biology 13

3 Gene Synthesis 21

4 Previous Work & Available Software 27

5 The Oligo Design Problem 33

6 An Algorithm for the Oligo Design Problem 41

7 Implementation: Mason 53

8 The Codon Optimization Problem 55

9 Background: Constraint Satisfaction 59

10 An Algorithm for the Codon Optimization Problem 67

11 Conclusion & Future Work 77

References 79

List of Figures

2.1 DNA. 14
2.2 A DNA hairpin. 15
2.3 Primer extension. 18
2.4 Polymerase Chain Reaction. 19
3.1 Ligation assembly. 22
3.2 Polymerase Cycling Assembly. 23
3.3 Types of misannealing events. 25
5.1 A solution to the oligo design problem. 34
5.2 Overlapping oligos. 35
5.3 Mispriming. 36
6.1 Arrangement of nodes in the graph. 43
6.2 Edges in the graph. 44
6.3 A path through the graph. 45
6.4 Recursively finding the shortest path. 46
9.1 Graph to be 3-colored. 60
9.2 Branch and bound search tree. 63
10.1 Input sequence with amino acids and variables. 67
10.2 Partial assignment to codons. 70
10.3 Replacing undefined regions with unique characters. 71

List of Tables

2.1 The standard genetic code. 16

List of Algorithms

6.1 Shortest path algorithm. 47
6.2 Find all maximal repeats. 50
6.3 Finding mispriming oligos. 51
6.4 Finding self-priming hairpins. 52
9.1 General branch and bound algorithm. 62
9.2 Conflict-directed A*. 65
10.1 Finding all maximal repeats in a partially undefined sequence. 71
10.2 Extracting conflicts for sequence constraints. 74
10.3 Extracting a minimal conflict from a complete assignment. 75
10.4 Extracting minimal conflicts. 75

1 Introduction

The ability to synthesize novel DNA is the most fundamental prerequisite for synthetic biology. Engineering novel biological behavior is impossible if, at the end of the day, it is too difficult to synthesize the DNA encoding that behavior and observe its performance in a living system. Despite its importance, the problem of gene synthesis has not yet been solved, and synthetic biologists are finding that waiting for DNA is the rate-limiting step in their research.

There are two options for synthesizing DNA: paying a professional, or doing it oneself. Commercial synthesis has the benefits of convenience and, increasingly, affordability; however, the process is opaque and introduces an often unwanted dependence on external companies. The DIY approach has its own drawbacks: existing synthesis methods are unreliable and can become major time sinks.

DNA is normally synthesized from sets of overlapping oligonucleotides using ligation or PCR. The selection of the oligonucleotides used is a major computational problem: an enormous number of different sets can be used to synthesize a given sequence, but many sets — maybe most — will not work at all. Many oligonucleotide design tools exist, but those available are inflexible and employ poorly-designed algorithms.

This thesis presents and justifies a well-designed algorithm for the design of oligonucleotides for PCR-based gene synthesis. It introduces a software tool, Mason, that implements this algorithm. Finally, it discusses how the algorithm could be extended to simultaneously perform codon optimization and oligonucleotide design using constraint optimization.

The remainder of this document is organized as follows:

- Chapter 2 introduces molecular biology, DNA thermodynamics, and PCR for those unfamiliar with them.

- Chapter 3 discusses ligation-based and PCR-based DNA synthesis and the difficulties associated with each method.
- Chapter 4 reviews some currently available software for gene synthesis and explains where each is lacking.
- Chapter 5 formally states the problem of oligonucleotide design for PCR-based gene synthesis.
- Chapter 6 develops an algorithm that efficiently solves that problem.
- Chapter 7 introduces Mason, a tool employing this algorithm.
- Chapter 8 formally states the problem of oligo design with codon optimization.
- Chapter 9 introduces constraint satisfaction and constraint optimization for those unfamiliar with those fields of computer science.
- Chapter 10 discusses how to apply constraint optimization to the problem of oligo design with codon optimization.
- Chapter 11 discusses future work and concludes.

Though it takes a different approach, this work is indebted to Wozniak [2005], previous work on the development of a well-designed algorithm for oligonucleotide design.

2 Background: Molecular Biology

This chapter is a brief introduction to molecular biology, and should provide the non-biologist with the background needed to understand the problem this thesis solves. Three topics are addressed: DNA, RNA, protein, and the central dogma of molecular biology; DNA thermodynamics; and polymerase chain reaction (PCR). For a more in depth introduction, see Watson et al. [1987].

Those with a background in biology may want to skip this chapter.

2.1 Molecular Biology

2.1.1 DNA

DNA — deoxyribonucleic acid — is the molecule that stores the genetic information in a cell. It can exist as single-stranded DNA (ssDNA) or as double-stranded DNA (dsDNA or a *duplex*). A strand of DNA is a string of deoxyribonucleotides[1] joined by phosphodiester links. Each deoxyribonucleotide is composed of two units: a sugar (deoxyribose) with a phosphate group attached, and one of four bases — adenine, thymine, guanine, or cytosine (`A`, `T`, `G`, or `C`). Strands of DNA are directional: one end is called the 5' end; the other is called the 3' end.[2]

The two strands of a DNA duplex run in opposite directions: one from 5' → 3', one from 3' → 5'. They are twisted in a double helix; and held together by loose hydrogen bonds between the bases of paired nucleotides on opposite strands. This *base pairing* is only possible for two of the eight possible base permutations:

[1] 'Deoxyribonucleotide' is often shortened to 'nucleotide', which is abbreviated 'nt'.

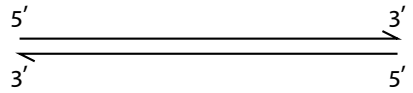
[2] 5' and 3' refer to carbon atoms in the deoxyribose molecule. The phosphodiester linkage between nucleotides connects to the 5' carbon of one deoxyribose molecule and to the 3' carbon of the next. The nucleotide at the 5' end of the strand has a phosphate group attached to its 5' carbon, but this phosphate is not attached to any other nucleotide. The nucleotide at the 3' end of the strand has no phosphate attached to its 3' carbon.

adenine–thymine, and guanine–cytosine. Therefore, if an `A` (`G`) appears in one strand, a `T` (`C`) must appear in the corresponding position on the other strand.

Figure 2.1 shows two common representations of DNA on paper. That shown in figure 2.1b will be used throughout this thesis.

```
5'-AGTGGACCGAT-3'
3'-TCACCTGGCTA-5'
```

(a) Double-stranded DNA shown textually.



(b) A schematic representation of DNA. Arrows appear at the 3′ ends of each strand.

Figure 2.1 – Common paper representations of DNA.

Because each base can only pair with one other, the sequence of one strand exactly specifies the sequence of the other. If we read both from 5' → 3', the two sequences are *reverse complements*: reversing one sequence and replacing each base with its complement[3] yields the other sequence. For this reason, normally just one sequence is given for double-stranded DNA. This sequence is always understood to be written from 5' → 3', and is called the *top strand*, *plus strand*, or *coding strand*.[4]

DNA can form more complicated structures than a double-helix. Single-stranded DNA is flexible enough to form double-stranded DNA by looping and base-pairing with itself. Such structures are called *hairpins* or *stem-loops*. They consist of a double-stranded stem with a single-stranded loop at one end (see figure 2.2). More complicated structures formed from multiple strands of DNA with many stems and loops are also possible.

2.1.2 RNA

RNA — ribonucleic acid — is a molecule very similar to DNA, but with a few major differences:

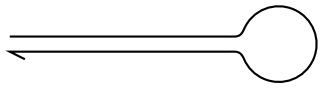
1. Each nucleotide in RNA contains a ribose molecule instead of a deoxyribose molecule.

[3] `A` with `T`, `T` with `A`, `G` with `C`, `C` with `G`.

[4] The other strand is, predictably, the *bottom strand*, *minus strand*, or *non-coding strand*.

```
                A G
              T     A
5'-GAACCAGTGA        G
3'-CTTGGTCACT        G
              C     T
                A T
```

(a) Shown textually.



(b) Shown schematically.

Figure 2.2 – A DNA hairpin.

2. Naturally occurring RNA never contains thymine. Instead, it contains the similar base uracil (U), which can also base pair with adenine.
3. Naturally occurring RNA is mostly single-stranded. However, RNA can be double-stranded, and a single RNA strand can form a duplex with a single DNA strand.

In cells, RNA acts as short-term storage: when the information stored in a cell's DNA is needed, a single-stranded RNA 'working copy' is made.

2.1.3 Protein

Proteins are chains of amino acids linked by peptide bonds. These chains fold into complex, often compact, three-dimensional shapes in the cell. Twenty different amino acids appear in naturally occurring proteins.

Functionally, proteins are incredibly important: among other things, they act as enzymes that catalyze the chemical reactions critical to life.

2.1.4 The Genetic Code

The major role of DNA is to store the sequences of the proteins cells must continually synthesize to grow and divide. Twenty different amino acids appear in naturally occurring proteins, but only four nucleotides appear in DNA. Therefore, a minimum of three nucleotides is needed to uniquely represent each amino acid.[5] The genetic code, the scheme by which DNA encodes protein sequences,

[5] One nucleotide can encode four amino acids ($4^1 = 4$); two nucleotides can encode sixteen amino acids ($4^2 = 16$); three nucleotides can encode sixty-four amino acids ($4^3 = 64$).

works in exactly this way.

Each three-base unit in a sequence of DNA that encodes a protein is called a *codon*. Codons are read in sequence, without gaps, from 5' → to 3'. This gives six possible *reading frames* for a given sequence of DNA: a reading frame can be on one of two strands, and at one of three offsets within a strand. The start of a protein sequence is marked by the presence of the *start codon* (normally `ATG`); the end of a protein sequence is marked by one of a few *stop codons* (normally `TAA`, `TAG`, and `TGA`). A start codon, followed by any number of codons, and finally followed by a stop codon in the same frame is termed an *open reading frame* (ORF).

Table 2.1 shows the standard[6] genetic code. Note that it is degenerate: because there are sixty-four codons, but only twenty amino acids, most amino acids are encoded by multiple codons.

	T	C	A	G	
T	TTT Phe	TCT Ser	TAT Tyr	TGT Cys	T
	TTC Phe	TCC Ser	TAC Tyr	TGC Cys	C
	TTA Leu	TCA Ser	TAA*	TGA*	A
	TTG Leu	TCG Ser	TAG*	TGG Trp	G
C	CTT Leu	CCT Pro	CAT His	CGT Arg	T
	CTC Leu	CCC Pro	CAC His	CGC Arg	C
	CTA Leu	CCA Pro	CAA Gln	CGA Arg	A
	CTG Leu	CCG Pro	CAG Gln	CGG Arg	G
A	ATT Ile	ACT Thr	AAT Asn	AGT Ser	T
	ATC Ile	ACC Thr	AAC Asn	AGC Ser	C
	ATA Ile	ACA Thr	AAA Lys	AGA Arg	A
	ATG† Met	ACG Thr	AAG Lys	AGG Arg	G
G	GTT Val	GCT Ala	GAT Asp	GGT Gly	T
	GTC Val	GCC Ala	GAC Asp	GGC Gly	C
	GTA Val	GCA Ala	GAA Gln	GGA Gly	A
	GTG Val	GCG Ala	GAG Gln	GGG Gly	G

Table 2.1 – The standard genetic code. This table shows the mapping from codons to amino acids. Standard three-letter abbreviations are used for amino acids.

† The start codon.

* The stop codons.

[6] Organisms using non-standard codes exist.

2.1.5 The Central Dogma; Transcription and Translation

The central dogma of molecular biology states that information in biological systems is transferred from DNA to RNA to protein. DNA acts as long-term information storage. To synthesize a protein, the cell makes a single-stranded RNA copy of the relevant DNA. This process is called *transcription*; the single-stranded RNA that results is called *messenger RNA* (mRNA). The protein is then synthesized from the mRNA by a piece of cellular machinery called a *ribosome*; this process is called *translation*.

2.2 DNA Melting Temperature

Complementary strands of DNA will automatically form double-stranded DNA. However, when heated, double-stranded DNA can melt: the hydrogen bonds holding the two strands together can break. The *melting temperature* (or T_m) of a DNA duplex is defined as the temperature at which half the strands that form a duplex are bound and half are free (single-stranded) [SantaLucia, 1998].

We will find it important to be able to estimate DNA melting temperature, which can be done using the enthalpy and entropy of duplex formation:

$$T_m = \frac{\Delta H}{\Delta S + R \log \frac{C_T}{4}}$$

C_T is the total concentration of the two strands that form the duplex (which are assumed to be different and to be present in equal concentrations — each at $\frac{C_T}{2}$) and R is the ideal gas constant [SantaLucia, 1998].[7]

The nearest-neighbors method can be used to calculate ΔH and ΔS for DNA duplex formation [SantaLucia and Hicks, 2004]. This method assumes the stacking energies in the duplex — the energies between neighboring base pairs — make additive contributions to both quantities. For example, ΔH of the duplex with top

[7] Suppose A and B are the two species forming the duplex, each present at $[\frac{C_T}{2}]$. Then $K = \frac{[AB]}{[A][B]}$. The melting temperature is defined as the temperature at which half the strands are in the duplex, i.e. where $[A] = [B] = [AB] = \frac{C_T}{4}$, so $K = \frac{4}{C_T}$. In general, $T = \frac{\Delta H}{\Delta S - R \log K}$, plugging in $K = \frac{4}{C_T}$ gives the above equation for T_m.

strand `5'-ATGGCAT-3'` would be calculated as:

$$\Delta H_{tot} = \Delta H_{\substack{\mathtt{AT}\\\mathtt{TA}}} + \Delta H_{\substack{\mathtt{TG}\\\mathtt{AC}}} + \Delta H_{\substack{\mathtt{GG}\\\mathtt{CC}}} + \Delta H_{\substack{\mathtt{GC}\\\mathtt{CG}}} + \Delta H_{\substack{\mathtt{CA}\\\mathtt{GT}}} + \Delta H_{\substack{\mathtt{AT}\\\mathtt{TA}}}$$

…plus a term for initiation and a few other constant terms added only for some sequences. ΔS is calculated the same way.

2.3 PCR

Polymerase chain reaction (PCR) is a technique used in molecular biology to amplify (increase the concentration of) DNA already present in solution. It makes use of DNA polymerase, a naturally occurring enzyme that is able to synthesize a complementary strand onto single-stranded DNA in place. Polymerase cannot start with single-stranded DNA alone: it requires a small part of the complementary strand, a primer, to already be bound to the longer single-stranded DNA, the template. Given a primer, polymerase can add complementary nucleotides to its 3' end one at a time until reaching the end of the template strand (see figure 2.3).

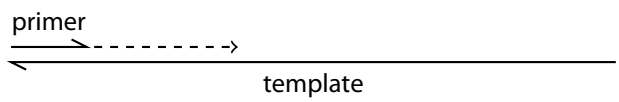


Figure 2.3 – Extension of a primer by DNA polymerase

PCR is performed on double-stranded DNA. Short primers that match the 5' ends of both strands in the duplex (the top strand and the bottom strand) are added to a solution containing the double-stranded DNA to be amplified. Then, the following steps are repeated many times:

1. The solution is heated, causing the double-stranded DNA to melt into two complementary pieces of single-stranded DNA.
2. The solution is cooled, allowing duplexes to form again. Primers will anneal to some full-length single strands, but some full-length duplexes will simply reform.
3. The solution is warmed slightly to a temperature at which the polymerase is active. Each primer bound to a full-length strand is extended so that it too becomes a full-length strand.

Each time a primer bound to the top strand is extended, a strand of DNA complementary to it is produced, effectively duplicating the bottom strand. Likewise, each time a primer bound to the bottom strand is extended, the top strand is duplicated. During each cycle, some fraction of the top and bottom template strands are extended, so the amount of full double-stranded DNA is increased by some factor.[8] There is therefore exponential growth in the amount of full double-stranded DNA over time. Figure 2.4 graphically illustrates a single cycle of PCR.

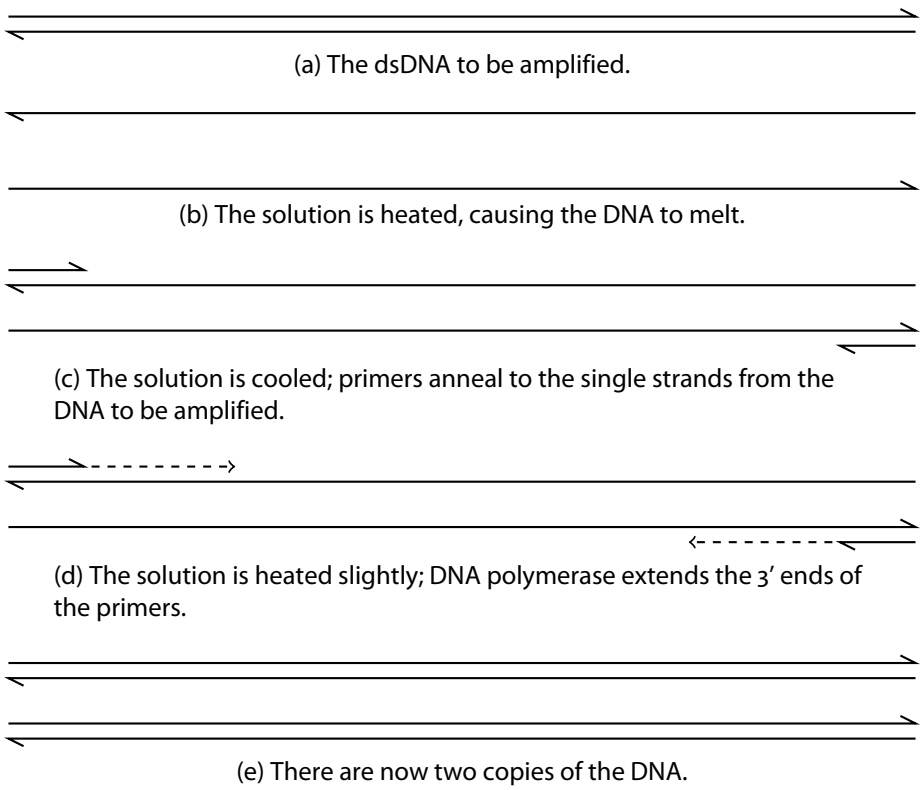


Figure 2.4 – The steps of a single PCR cycle.

There is one important limitation to PCR: because primers complementary to the 5' ends of the top and bottom strands must be added, the sequence at both ends of the full-length duplex must be known. This does not prevent amplification of sequences for which the interior region is unknown.

[8] Theoretically, the amount of full-length DNA could be doubled each cycle.

3 Gene Synthesis

Gene synthesis[1], the *de novo* synthesis of large[2] molecules of double-stranded DNA, is critical for synthetic biology. Synthetic biologists need to:

- mix and match regulatory elements and protein coding regions,
- combine unrelated protein domains,
- alter codon usage, and
- recode genes for organisms with unusual genetic codes.

Sequence engineering has traditionally been performed using recombinant DNA technology. This technology has served biologists (and synthetic biologists [Knight et al., 2003]) well, but does not provide the power needed to accomplish the tasks just listed.

The dominant approach to gene synthesis has been to assemble long double-stranded DNA from *oligonucleotides* (oligos) — short pieces of single-stranded DNA. Oligos are synthesized chemically (no biological machinery is used) one nucleotide at a time [Caruthers, 1985]. Unfortunately, this places limits on their length. With a constant efficiency of nucleotide addition, the probability of producing a correct product drops geometrically: even assuming 99.9% efficiency, less than 82% of two hundred-nucleotide oligos will be correct. Therefore, oligos are generally somewhere between thirty and sixty nucleotides in length.

There are two main strategies for assembly of DNA from oligos: ligation-based methods and PCR-based methods. Both methods allow many different sets of oligos to be used to assemble a given sequence. However, not all such sets are created equal: many will not work well, and many will not work at all.

[1] The term 'gene synthesis' is something of a misnomer: it is perfectly common to synthesize DNA that contains no genes.

[2] Large, in this case, means on the order of kilobases.

This chapter introduces both synthesis techniques and discusses the considerations that make oligo design challenging for each technique. It then explains why this thesis will focus on PCR-based synthesis.

3.1 Ligation

The ligation method of gene synthesis was one of the first methods used [Itakura et al., 1977]. Using this technique, full-length double-stranded DNA is assembled from overlapping oligos that together form the entire sequence. They are allowed to assemble as in figure 3.1 and are then joined by DNA ligase, an enzyme that can repair single-stranded nicks in DNA (see figure 3.1a).[3] Ligase requires the 5' ends to be phosphorylated, so the entire pool of oligos used in the gene synthesis must be phosphorylated before being ligated[4].

Synthesis by ligation produces only a small amount of product, so PCR is typically performed after assembly.

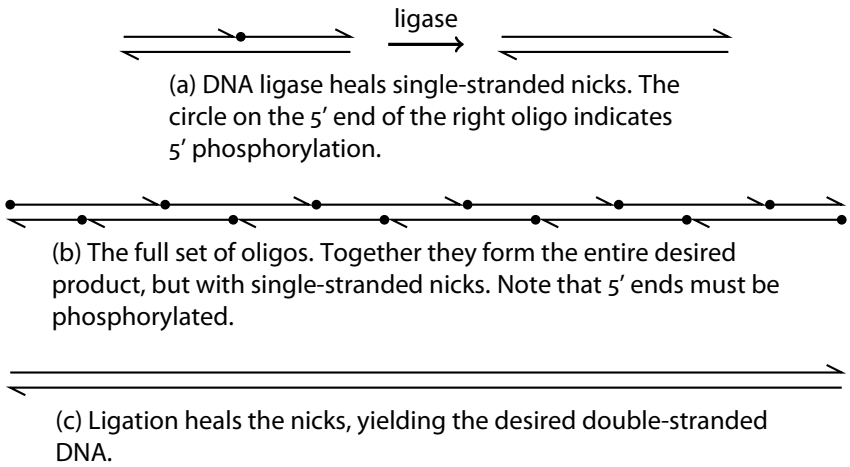


Figure 3.1 – Ligation assembly.

[3] A single-stranded nick is a break in one strand of double-stranded DNA. DNA ligase can also heal double-stranded breaks, but that ability isn't important here.

[4] It is also possible to purchase oligos with phosphorylated 5' ends, but this can be expensive.

Challenges

This method relies on the oligos self-assembling; if any unintended duplexes form, an incorrect product could result. To avoid such *misannealing* events, every oligo overlap must be sufficiently unique. Furthermore, oligos should not form hairpins or other structures that might compete with the formation of the desired duplexes. Finally, the melting temperatures of the overlaps between oligos should be as uniform as possible; this allows assembly to take place in stringent thermal conditions, reducing the probability of unwanted duplexes forming [Stewart and Burgin, 2005].

3.2 Polymerase Cycling Assembly

Polymerase cycling assembly (PCA), a second method of gene synthesis, is based on PCR [Stemmer et al., 1995]. As in ligation assembly, the full DNA duplex is assembled from overlapping oligos; however, the oligos used need not form the entire double-stranded product — there can be gaps between oligos on the same strand (see figure 3.2a).

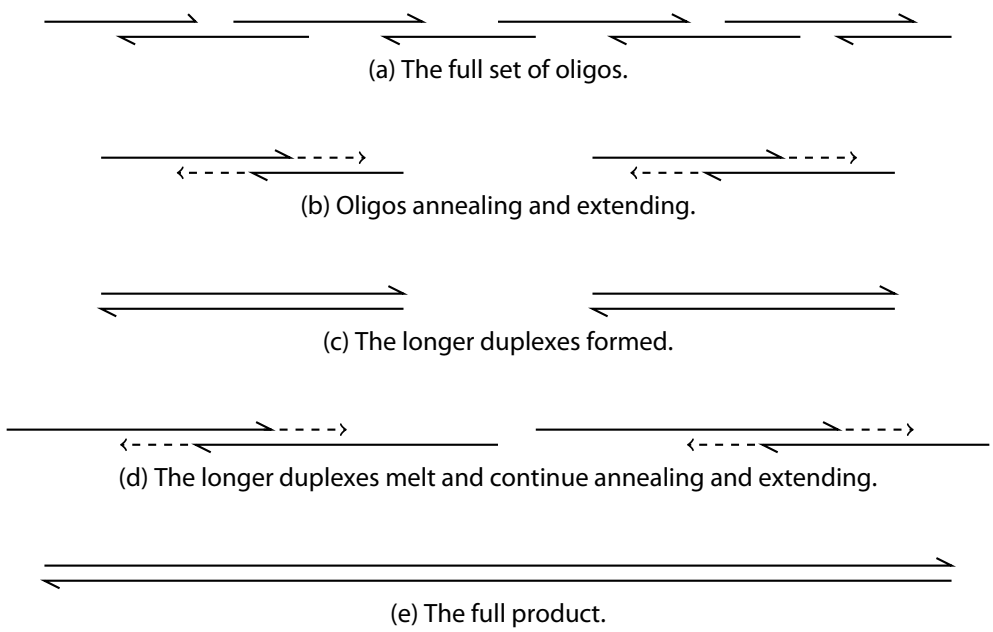


Figure 3.2 – The progression of PCA.

A solution containing all of the oligos (and DNA polymerase) is subjected to the same thermal cycling as in PCR. During each cycle, pairs of oligos anneal and the 3' ends of each are extended, forming a duplex spanning the sequence between the extreme ends of the two overlapping oligos. During subsequent cycles, these longer duplexes will melt and continue to anneal and extend. Larger and larger duplexes will be formed; after many cycles, full-length duplexes will exist. Figure 3.2 graphically illustrates this process.

Because DNA polymerase can only extend 3' ends, the first[5] oligo must be on the top strand and the last[6] oligo must be on the bottom strand. If the first were on the bottom strand, the very beginning of the sequence would never be synthesized; if the last were on the top strand, the very end of the sequence would never be synthesized.

PCA as described does not result in exponential growth of the product. PCR can be performed after the PCA reaction — a two-step synthesis [Stemmer et al., 1995] — or PCR primers can be included in the PCA reaction itself — a one-step synthesis [Wu et al., 2006].

Challenges

The challenges of PCA oligo design are quite different from the challenges of ligation assembly oligo design. In ligation assembly, the concern is that the oligos fit together correctly and that no thermodynamically stable competing structures can form. In PCA, the concern is that only intentionally overlapping primers anneal and extend — if two primers misanneal and then are extended, incorrect DNA will be produced. If such *mispriming* happens frequently, incorrect products of different lengths will be formed in addition to the correct product, decreasing yield and purity. Such events are not limited to those between two different oligos — hairpin-forming oligos that can self-prime are also problematic.

Mispriming events occur only when the 3' end of an oligo can actually be extended; therefore, many possible duplexes or hairpins are perfectly acceptable. Figure 3.3 illustrates the two situations: figures 3.3a and 3.3c show mispriming

[5] Leftmost in figure 3.2a.
[6] Rightmost in figure 3.2a.

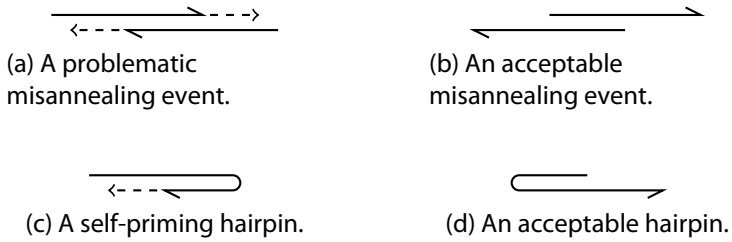


Figure 3.3 – Types of misannealing events.

events; figures 3.3b and 3.3d show misannealing events that can be tolerated because no extension is possible.

Like ligation assembly, PCA is most effective when the oligo overlaps have a uniform melting temperature.

3.3 Important Differences

Ligation assembly and PCA are very different techniques. The set of oligos used in ligation assembly must cover the entire sequence to be synthesized; the set of oligos used in PCA can have gaps. Therefore, the number of sets that could be used to synthesize a given sequence with PCA is orders of magnitude larger than the number of sets that could be used with ligation.

The challenges the two methods present are fundamentally different. With ligation, repeats anywhere in the sequence are a problem, but only if they are thermodynamically stable at the chosen assembly temperature. With PCA, only mispriming repeats are problematic, but those repeats do not need to be long. For ligation, the trick is to avoid stable, undesired duplexes and secondary structures. For PCA, the trick is to avoid any possible mispriming.

The remainder of this thesis will be focused on the oligo design problem for PCR-based gene synthesis. Because of the much larger number of possible sets, design for PCR-based gene synthesis is, in one sense, a more difficult problem. However, the increased flexibility resulting from this larger number of possible solutions also makes it more likely that some very good solution exists.

4 Previous Work & Available Software

A number of software packages that can design oligos for gene synthesis are available; however, no available software is sufficiently flexible and rigorous. This section provides a brief review of some currently available tools and notes the shortcomings of each.

Many of the tools here are intended to be complete gene design solutions. Some can perform codon optimization and restriction site insertion; others offer the ability to split large sequences into smaller 'synthons' for synthesis. However, every available tool lacks a carefully designed algorithm for oligo design. This chapter will focus on implementations of that functionality.

4.1 DNAWorks

DNAWorks is a web-based oligo design tool implemented in Fortran 90. The original version of the software is described in Hoover and Lubkowski [2002]. The currently available version is described in Hel [2009].

DNAWorks allows the user to provide either:

- an amino acid sequence and flanking nucleotide sequences, in which case the amino acid sequence can be codon optimized, or
- a nucleotide sequence, which will not be codon optimized.

The programs' oligo design algorithm begins by dividing the sequence into regions of relatively uniform T_m corresponding to oligo overlaps. In the original version of the program, these regions were contiguous; in the currently available version of the program, gaps between adjacent oligos on the same strand are permitted.

The algorithm then optimizes the section locations (and, for amino acid input, codon usage) using a variant of simulated annealing.[1] Individual sections are scored using an objective function that incorporates:

- T_m,
- codons used,
- presence of repeats,
- potential for mispriming,
- `GC` content,
- `AT` content,
- length, and…
- presence of forbidden subsequences.

The score of the full set of oligos and codon choices is the sum of the scores of all regions.

The program offers two modes. In one, all oligos designed (except the first and last) are of the same length. In the other, oligo length is allowed to vary.

DNAWorks has two major problems: its objective function and its use of simulated annealing. The form of the objective function is arbitrary, and no consideration is given to the weights used for each factor it incorporates: by default, each is given the weight 1. The use of simulated annealing, a stochastic algorithm, means that DNAWorks only explores a small subset of possible solutions; using the option that allows oligo lengths to vary, three runs on the same input give three different results.

4.2 Gene2Oligo

Gene2Oligo is a web-based tool written in Java [Rouillard et al., 2004]. Unlike DNAWorks, Gene2Oligo does not provide codon optimization functionality, nor does it allow for gaps between oligos.

Gene2Oligo offers two major modes of operation:[2] the first mode prioritizes uniformity of oligo length, the second mode prioritizes uniformity of overlap

[1] Simulated annealing is a stochastic optimization algorithm that has nothing to do with the annealing of DNA.

[2] A third mode simply cuts the sequence into oligos of equal size with no thought.

melting temperature. The program adds flanking sequences to the input sequence in order to increase the number of solutions; these sequences must be removed by the user via PCR.

Gene2Oligo begins by computing the T_m of all possible overlaps in the sequence. It then uses BLAST to find candidate oligos that partially match other places in the sequence and checks the T_m of the undesired duplexes that could be formed; oligos that form stable undesired duplex are flagged.

The program then selects the two best oligos that begin at each index in the sequence. This forms a binary tree: each oligo points to the index after its ending index; this index points to two new oligos. This tree is searched using depth-first search with backup for a string of oligos covering the sequence. This appears to only design oligos for one strand; it is not clear how oligos for the other strand are designed.

There are a few problems with the approach of Gene2Oligo. First, it is inflexible: oligo length cannot vary significantly, and the program does not allow gaps in the oligo set for PCR-based synthesis. Second, when it selects two oligos for each index in the sequence, it removes many potentially good candidates. It is therefore only exploring a small subset of the possible solution space, even given its restrictions on oligo length.

4.3 GeMS

GeMS is a stand-alone complete gene design tool developed in Python [Jayaraj et al., 2005]. It offers a number of features, including codon optimization and restriction site insertion. Unfortunately, its oligo design functionality is simplistic.

Given a sequence produced by the codon optimization, etc. modules of GeMS, the program designs a set of 40 nt oligos with 20 bp overlaps. Neither of these numbers can be changed. The lengths of the oligos at the extreme ends of the sequence are variable.

GeMS designs such a set of nucleotides (there are only a few possible solutions with such restrictive parameters) and then looks for mispriming oligos. If an oligo could misprime, two random base pairs are added to the end of the sequence and a new oligo set is designed and checked for mispriming. If no solution is found,

codon optimization (a stochastic process in GeMS) is performed again.

With no variability in oligo length or overlap size, the algorithm used by GeMS is clearly inadequate. It does not check for the presence of hairpins and, though it checks for mispriming, its inflexibility prevents it from effectively avoiding mispriming.

4.4 GeneDesign

GeneDesign is a web-based complete gene design tool developed in Perl and C [Richardson et al., 2006]. The oligo design module focuses on $\geq$ 60 nt oligos with $\geq$ 20 bp overlaps. GeneDesign automatically breaks large sequences into chunks of $\approx$ 500 bp. The chunk of DNA to be synthesized is broken up into an even number of oligos of the given length, each with overlaps of the given size. The oligos are then adjusted in length to fit the size of the given sequence.

Next, the program determines a target T_m for overlaps in the chunk of DNA to be synthesized. Then lengths of oligos are adjusted to make the overlap melting temperatures approach this value.

The most obvious problem with GeneDesign's oligo design algorithm is that it does not check for mispriming or hairpins. Without this feature, it is unlikely the algorithm could be used to design oligo sets that assemble reliably.

4.5 Gene Composer

Gene Composer is a stand-alone complete gene design tool implemented in C++ [Lorimer et al., 2009]. Its oligo design module allows oligo length to range between two user-provided values and aims for overlaps of a user-provided size, but does not allow for gaps.

The algorithm randomly cuts the top strand into adjacent oligos of allowed sizes. When it reaches the end of the sequence, the cuts are moved backwards until the final oligo is of an allowed length. Bottom strand oligos are designed by placing cuts approximately halfway between the cuts on the top strand.

This oligo design step is repeated thousands of times. Candidate sets that include an oligo with a sequence similar to that of another oligo or include an oligo

that forms a stable hairpin are discarded[3]. The ideal set is the set remaining with the highest average overlap T_m and the lowest overlap T_m variance.

The algorithm used by Gene Composer has many of the same problems as the other algorithms discussed here. Its strategy of randomly assembling thousands of solutions and returning the best found is poor, because there is an enormous number of solutions ($\gg 1000$); this algorithm will never explore more than a tiny portion of the search space.

4.6 TmPrime

TmPrime provides oligo design functionality for ligation-based synthesis or PCR-based synthesis [Bode et al., 2009]. As it addresses both methods, it does not allow gaps between adjacent oligos on the same strand.

The program first divides the sequence into regions of relatively uniform melting temperature; these regions correspond to overlaps between nucleotides. An oligo set is then constructed by concatenating adjacent overlap regions.

TmPrime also has the ability to look for misannealing oligos and hairpin forming oligos. This functionality appears to be geared toward ligation-based assemblies.

Because it seems focused on ligation-based assembly and does not allow for gaps between same-strand oligos, TmPrime is not an ideal tool for PCR-based gene synthesis.

4.7 Summary

This chapter has given just a brief overview of some available oligo design software. However, it should be clear that synthetic biology is missing an oligo design tool that employs a carefully designed, robust algorithm. Such a tool should:

- be focused on oligo design for either ligation-based synthesis or PCR-based synthesis, as the two problems are very different;
- be flexible, with no artificial restrictions on oligo length or overlap size; and

[3]The algorithm also incorporates the ΔG of intramolecular folding. It is not clear how this value is calculated nor how it is used.

- employ a well-designed algorithm that returns the same optimal answer every time it is run.

The next three chapters will formalize the oligo-design problem for PCR-based synthesis, present a carefully designed algorithm that solves that problem, and introduce an software tool employing that algorithm.

5 The Oligo Design Problem

We now turn to the problem of designing oligo sets for gene synthesis. To begin, we need to define our task. This chapter presents and justifies a precise formulation of the oligo design problem for PCR-based gene synthesis.

Throughout this section and the rest of this thesis, we will use the notation $\overline{S}$ to represent the reverse complement of a DNA sequence S.

5.1 Basic Parameters

Our goal is to find a set of overlapping oligos that can be used to synthesize a DNA sequence S of length L. Both the length of the oligos and the size of the overlaps between oligos can vary. However, to make the problem tractable, we need limits on oligo length and overlap size. At a minimum, we need four parameters:

P1. l_{min} The minimum allowed oligo length.

P2. l_{max} The maximum allowed oligo length.

P3. o_{min} The minimum allowed overlap between two oligos.

P4. o_{max} The maximum allowed overlap between two oligos.

Call our set of oligos $S_1, S_2, S_3, \ldots S_N$.[1] The numbering is from left to right — the ith oligo overlaps the $i+1^{\text{th}}$ oligo.

Any oligo used to synthesize S must be a subsequence of S or $\overline{S}$. Recall from chapter 3 that the first oligo in the set must be on the top strand (a subsequence of S), the last oligo in the set must be on the bottom strand (a subsequence of $\overline{S}$),

[1] N is not a fixed parameter, but is simply used as shorthand for the number of oligos in a given set.

and that oligos must alternate between the top and bottom strands. This makes two things clear:

1. N, the number of oligos in our set $\{S_i\}$, must be even, and
2. S_i is on the top strand if i is odd, or on the bottom strand if i is even.

Knowing this, we can identify the position of each oligo S_i with the starting and ending indices of a subsequence of S — a_i and b_i. If i is even, S_i is $S_{a_i \ldots b_i}$; if i is odd, S_i is the reverse complement of $S_{a_i \ldots b_i}$.

We now write down a few constraints on $\{(a_i, b_i)\}$:

C1. $a_1 = 1$
The first oligo must begin at the beginning of the sequence.

C2. $b_N = L$
The last oligo must end at the end of the sequence.

C3. $l_{min} \leq b_i - a_i + 1 \leq l_{max}, i = 1, 2, 3, \ldots N$
The length of each oligo is between the minimum and maximum allowed lengths.

C4. $o_{min} \leq b_i - a_{i+1} + 1 \leq o_{max}, i = 1, 2, 3, \ldots N - 1$
The size of each overlap is between the minimum and maximum allowed sizes.

Figure 5.1 shows a solution that satisfies these constraints.

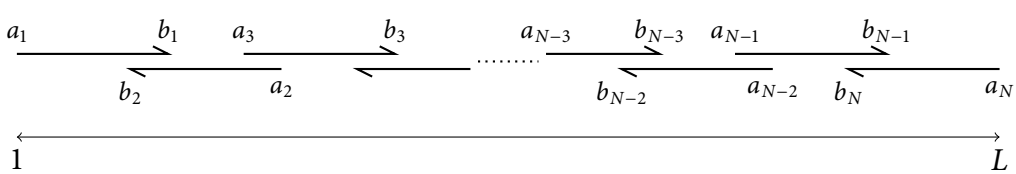


Figure 5.1 – A solution to the oligo design problem.

Note that the constraints do not actually prevent two adjacent primers on the same strand (i.e. some S_i and S_{i+2}) from overlapping as in figure 5.2 — such overlapping is not a problem for PCA.[2]

[2] An oligo will only anneal to one other oligo during a given cycle. Because S_i and S_{i+2} will never anneal to S_{i+1} simultaneously, their overlapping does not matter.

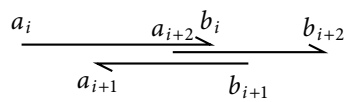


Figure 5.2 – Two adjacent oligos on the same strand overlapping.

A solution that satisfies the above constraints 'makes sense' — the oligos fit together as they should — but will not necessarily assemble well. The rest of this chapter introduces parameters and constraints meant to ensure successful assembly.

5.2 Overlap Melting Temperature

Recall from chapter 3 that PCA is most reliable when the melting temperatures of all oligo overlaps are uniform. To account for this, we'll add two additional parameters to our formulation:

P5. T_0 The target overlap melting temperature.

P6. ΔT The maximum allowed absolute deviation from T_0.

…and we'll add another constraint:

C5. $|T_m(S_i, S_{i+1}) - T_0| \leq \Delta T, i = 1, 2, 3, \ldots N - 1$
The melting temperature of each overlap falls within ΔT of the target melting temperature.
($T_m(S_i, S_{i+1})$ is the melting temperature of the overlap between the ith and $i + 1$th oligo.)

The constraint ensures that no solution will have an oligo overlap with a melting temperature outside the target range.

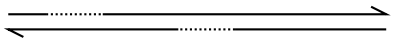
5.3 Avoiding Mispriming

For PCA to work correctly, it is critical that the chance of mispriming is minimized (see chapter 3). Consider the conditions necessary for mispriming: an

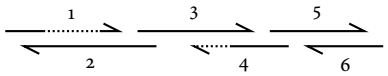
oligo must misanneal in such a way that its 3' end can be extended by DNA polymerase. Only the 3' end needs to be part of a duplex for polymerase to extend it.

This can happen whenever the sequence at the 3' end of an oligo occurs elsewhere in S or in the reverse complement of S. If this is the case, the 3' end of the oligo will be able to anneal to the strand complementary to the other occurrence and can then be extended. The repeat must be long enough that the duplex it forms is stable at the annealing phase of the PCR cycle — very short repeats are not problematic.[3] Figure 5.3 shows this process graphically.

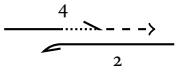
Mispriming can also occur when the 5' end of an oligo occurs elsewhere in S or in its reverse complement. Suppose the 5' end of S_i is a repeated subsequence. Some other oligo (either S_{i-1} or S_{i+1}) will anneal to S_i and be extended so that its 3' end is complementary to the 5' end of S_i. Because that 5' end was a repeat, its complement is also a repeat[4], so the 3' end of the extended neighbor of S_i will be a repeated sequence that could misprime.



(a) A full sequence containing a repeated subsequence, indicated by the dashed region.



(b) A set of oligos with a repeat at the 3′ end of one oligo.



(c) Oligo 4 anneals to oligo 2 and is extended: a mispriming event. Note that oligo 2 is not extended, because its extreme 3′ end is not a repeat.

Figure 5.3 – Mispriming illustrated. The incorrectly extended oligo 4 will now be able to anneal and extend where only oligo 1 should, causing further errors.

We conclude that neither the 3' end nor the the 5' end of any oligo can be a repeated subsequence. In other words, no 'long' prefix or suffix of any oligo in

[3] In fact, because there are only four possible bases, short repeats are unavoidable.

[4] We're dealing with double-stranded DNA, so if a subsequence occurs twice, its complement will also occur twice.

the set can occur anywhere else in S or $\overline{S}$. To quantify 'long', we need another parameter:

P7. R_{min} The shortest repeated subsequence that could cause mispriming.

…and we'll add one new constraint:

C6. $\nexists\ (x, y) \in R$ s.t. $(b_i \leq y$ and $b_i - x + 1 \geq R_{min})$ or $(a_i \geq x$ and $y - a_i + 1 \geq R_{min})$
No oligo has a suffix or prefix that is a repeat of length $\geq R_{min}$.

In the above constraint, R is the set of all regions corresponding to maximal repeats longer than R_{min} in S and $\overline{S}$. A maximal repeat is one which could not be extended to either the right or left and remain a repeat [Gusfield, 1997]. 'In S and $\overline{S}$' means that, of the two instances of a repeat:

1. both can be found in S,
2. both can be found in $\overline{S}$, or…
3. one instance can be found in S and one can be found in $\overline{S}$.

5.4 Avoiding Hairpins

Self-priming hairpins are just as problematic as mispriming oligos. A hairpin can occur any time an inverted repeat — a subsequence followed soon after by its reverse complement — occurs in the sequence. A oligo can form a self-priming hairpin if:

1. the oligo contains an inverted repeat,
2. the oligo's 3' end is in one end of the repeat, and
3. the hairpin the oligo forms leaves room for 3' extension.

Because the two repeat regions forming a hairpin stem are on the same oligo, it is much more likely that they will anneal. It is therefore sensible to put a length bound on repeats leading to self-priming hairpins that is even shorter than R_{min}:

P8. H_{min} The minimum stem length of a self-priming hairpin we consider a problem.

We also add a corresponding constraint:

C7. $S_i \notin H, i = 1, 2, 3, \ldots N$
No oligos can form self-priming hairpins with stem length $\geq H_{min}$, where H is the set of all such oligos.[5]

5.5 The Optimal Solution

We now have a full set of parameters and a set constraints that must be satisfied by any acceptable set of oligos. Now, we have to evaluate each: which of the oligo sets that satisfy the constraints given is best?

The constraints listed in this chapter will prevent the majority of mispriming events. Therefore, our biggest concern is ensuring that the melting temperatures of all overlaps are uniform. Of the many ways to quantify uniformity, the most appropriate is maximum absolute deviation from the target melting temperature: $\max_{i=1}^{N-1} |T_m(S_i, S_{i+1}) - T_0|$. Other quantities — like mean deviation from T_0 or variance around T_0 — are problematic because they will be small in the undesirable case where most overlaps have melting temperatures close to T_0 but a few have temperatures very far from T_0.

5.6 Summary

The oligo design problem for PCR-based synthesis is formally summarized below.

Given a sequence S and the following parameters:

P1. l_{min}

P2. l_{max}

P3. o_{min}

P4. o_{max}

P5. T_0

P6. ΔT

P7. R_{min}

[5] Writing a closed form expression to describe oligos that form self-priming hairpins is difficult, but finding those oligos is not difficult: see algorithm 6.4.

P8. H_{min}

...find a set of oligos $\{(a_i, b_i)\} = \{S_i\}$ that minimizes $\max_i |T_m(S_i, S_{i+1}) - T_0|$, $i = 1, 2, 3, \ldots N-1$ (where N is the number of oligos in the set) subject to the following constraints:

C1. $a_1 = 1$

C2. $b_N = L$

C3. $l_{min} \leq b_i - a_i + 1 \leq l_{max}$, $i = 1, 2, 3, \ldots N$

C4. $o_{min} \leq b_i - a_{i+1} + 1 \leq o_{max}$, $i = 1, 2, 3, \ldots N-1$

C5. $|T_m(S_i, S_{i+1}) - T_0| \leq \Delta T$, $i = 1, 2, 3, \ldots N-1$

C6. $\nexists\ (x, y) \in R$ s.t. $(b_i \leq y$ and $b_i - x + 1 \geq R_{min})$ or $(a_i \geq x$ and $y - a_i + 1 \geq R_{min})$

C7. $S_i \notin H$, $i = 1, 2, 3, \ldots N$

6 An Algorithm for the Oligo Design Problem

This chapter presents an algorithm for finding the optimal set of oligos for PCR-based synthesis of a DNA sequence S. The algorithm casts the oligo design problem as a graph problem: it constructs a graph with a node for every possible oligo and with an edge between any two nodes corresponding to oligos that could overlap. Acceptable sets of oligos correspond to paths through the graph.

Section 6.1 describes how to construct the graph given some basic inputs and shows that paths through the graph correspond to sets of oligos. Section 6.2 gives an algorithm for finding optimal paths. Section 6.3 describes how the constraints given in chapter 5 affect the graph and gives algorithms for the application of each constraint.

6.1 Constructing The Graph

We begin by describing the construction of a directed acyclic graph (DAG) given an instance of the oligo design problem. Throughout this section, we'll work with the following example input:

$$
\begin{aligned}
S &= \texttt{GACATGACCA} \\
l_{min} &= 2 \\
l_{max} &= 5 \\
o_{min} &= 2 \\
o_{min} &= 3
\end{aligned}
$$

These values are unrealistic, but will allow us to draw a graph on one page.[1]

[1] In a real problem, S might a kilobase in length, and we might have $l_{min} = 40$, $l_{max} = 60$, $o_{min} = 10$, and $o_{max} = 20$.

6.1.1 Nodes

Recall that every possible oligo used to synthesize the sequence S is either a subsequence of S or a subsequence of $\overline{S}$. We can identify any such oligo as a triplet (i, j, s), where i is the starting index of the oligo, j is the end of the oligo in S, and s is $+$ if the oligo is a subsequence of S or $-$ if the oligo is a subsequence of $\overline{S}$. In our example, $S/\overline{S}$ is:

```
5'-GACATGACCA-3'
3'-CTGTACTGGT-5'
```

...so $(3, 7, +)$ would be the oligo `5'-CATGA-3'` and $(5, 10, -)$ would be the oligo `5'-TGGTCA-3'`.

We now construct a graph with a node for every possible oligo $(i, j, \pm)$ arranged as in figure 6.1. The nodes are split into two main groups: one for top strand oligos $(i, j, +)$ and one for bottom strand oligos $(i, j, -)$. Within each group, nodes are arranged in columns by the starting index of the oligos they correspond to: every node in the xth column corresponds to an oligo $(x, j, \pm)$. Within each column, nodes are arranged by oligo length: the node corresponding to the shortest oligo is at the top of the column; the node corresponding to the longest oligo is at the bottom. Not all columns will be of the same length — at the right end of the graph longer oligo lengths aren't possible.

6.1.2 Edges

We now add an edge between every pair of oligos which could overlap given the values of o_{min} and o_{max}. The edge is directed from the left to right (from the oligo with the lower starting index to the oligo with the higher starting index). Because overlapping oligos must be on opposite strands, all edges in the graph cross from the top group of nodes to the bottom group, or vice versa. Figure 6.2, which continues the example begun in figure 6.1, shows all outgoing edges from one node.

6.1.3 From A to Ω

At this point, paths through the graph correspond to sets of oligos that overlap each other. However, recall constraints 1 and 2 from chapter 5: in acceptable sets of oligos, the first oligo must be of the form $(1, j, +)$ and the last oligo must be of

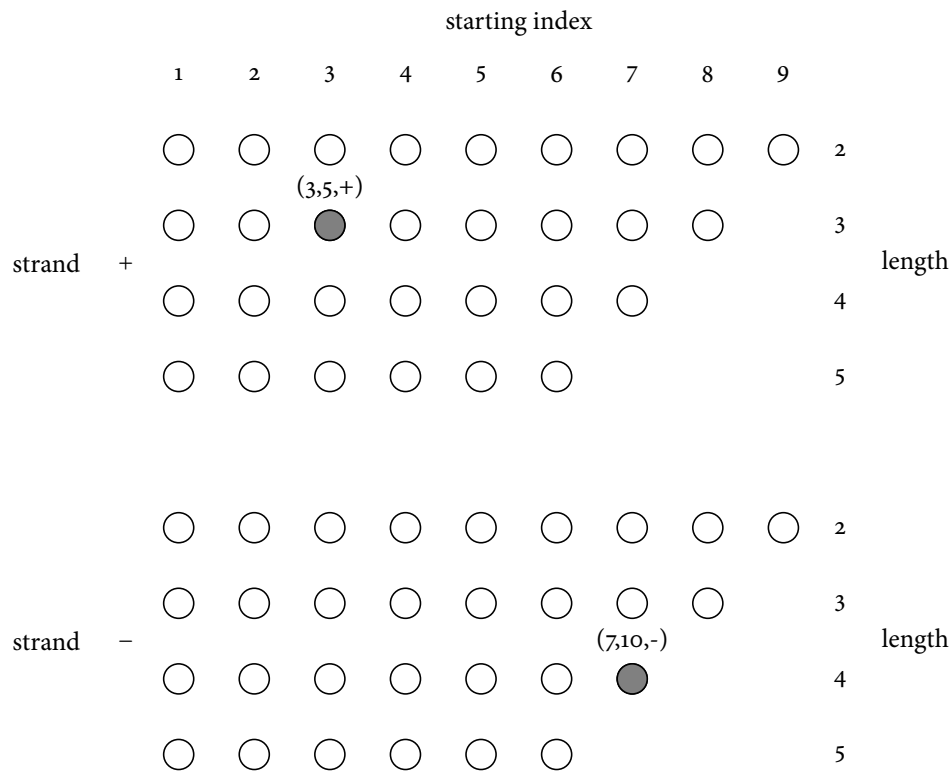


Figure 6.1 – Example arrangement of nodes in a graph for a sequence of length 10 with $l_{min} = 2$ and $l_{max} = 5$. A few nodes are labeled to illustrate naming.

the form $(i, L, -)$. To make it easy to find all such sets, we add two special nodes to the graph: A and Ω: A has an edge going to every node of the form $(1, j, +)$; Ω has an edge from every node of the form $(j, L, -)$.

With this done, every path from A to Ω corresponds to a set of oligos for the synthesis of the sequence S. The graph contains a node for every possible oligo and an edge for every possible overlap. Therefore, paths through the graph correspond to coherent sets of overlapping oligos. Every valid set begins with an oligo $(1, j, +)$, so there is an edge from A to the first node in the corresponding path. Every valid set ends with an oligo $(j, L, -)$, so there must be an edge from the last node in the corresponding path to Ω. Thus, the set of all paths from A to Ω is the set of all paths corresponding to valid sets of oligos. One path is shown

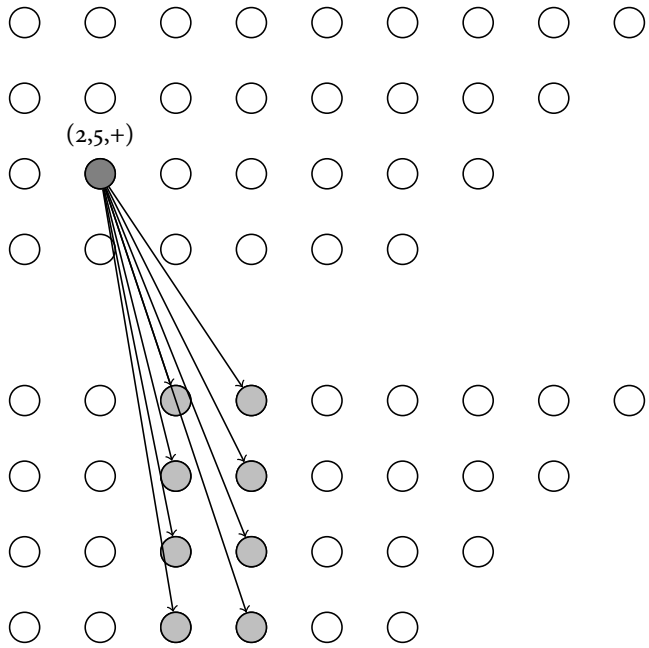


Figure 6.2 – Example edges out of one node in a graph for a sequence of length 10 with $l_{min} = 2$, $l_{max} = 5$, $o_{min} = 2$, and $o_{max} = 3$. (These are again unrealistic, but convenient, parameter values.)

in figure 6.3.

6.1.4 Edge Weights and Optimal Solutions

It remains to add weights to each edge. Recall from chapter 5 that we are concerned with the absolute deviation of the melting temperature of each overlap from the target, T_0. This is the weight we'll use: the edge between nodes α and β has the weight $|T_m(\alpha, \beta) - T_0|$, where $T_m(\alpha, \beta)$ is the melting temperature of the overlap between the oligos corresponding to α and β.

Recall that we are trying to find a set of oligos that minimizes:

$$F = \max_{i=1}^{N-1} |T_m(S_i, S_{i+1}) - T_0|$$

Each path from A to Ω in our graph corresponds to a set of oligos; the value of F for that set is the weight of the heaviest edge in the path. Therefore, an optimal set of oligos is one that corresponds to a path from A to Ω that minimizes F — we've reduced the oligo design problem to a path-finding problem.

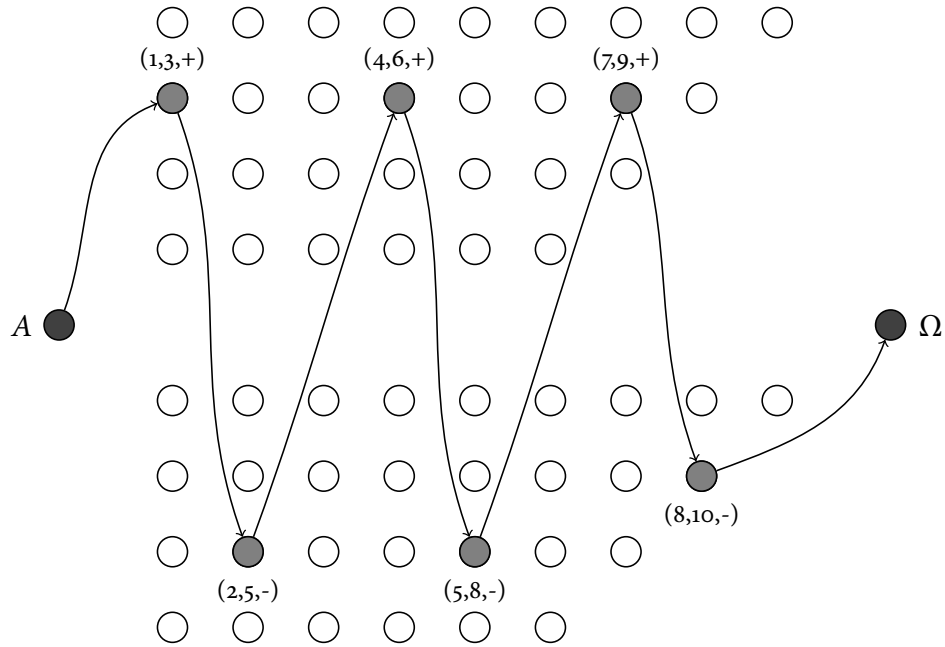


Figure 6.3 – One path from A to Ω through the graph.

6.2 Finding the Optimal Solution

We now turn to the question of finding such a path from A to Ω. Using the technique of dynamic programming, we can find this path in $\Theta(V+E)$ time (V is the number of vertices in the graph, and E is the number of edges). The algorithm we'll use is a variant of the shortest path algorithm for directed acyclic graphs [Cormen et al., 2001].

For the remainder of this chapter, we'll define the 'length' of a path as the maximum weight of all edges in that path. Using this definition of length, the path we are looking for is the shortest path from A to Ω.

Recall that the graph we have constructed is a directed acyclic graph (DAG) — all edges are directed, and since edges always point from left to right, no cycles exist. Note the following property of shortest paths in DAGs: suppose $\{u_i\}$ is the set of all vertices in a graph G with edges directed to some vertex v, and that the weight of the edge between u_i and v is w_i. Assume that $d[u_i]$ is the length of the shortest path from some vertex a to u_i and that $\pi[u_i]$ is the predecessor of u_i in this path. Then the length of the shortest path from a to v that goes

through u_i is $\max(d[u_i], w_i)$. Since all paths to v must go through some u_i (no other nodes have edges to v), the length of the shortest path from a to v — $d[v]$ — is $\min_i [\max(d[u_i], w_i)]$, and $\pi[v]$ is equal to u_i for the i chosen. Figure 6.4 graphically illustrates this property.

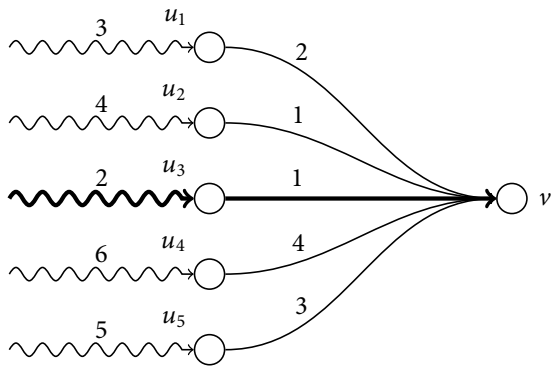


Figure 6.4 – Recursively finding the shortest path from a to v. Wavy lines are labeled with the length of the shortest path from a to each u_i. The path selected, with length 2, is drawn in bold.

There is a base case that will allow us to gain a foothold: the length of the shortest path from a to a itself. Define that length, $d[a]$, as 0, and define $\pi[a]$ as NIL. With $d[a]$ and $\pi[a]$ defined, we can begin calculating $d[u]$ and $\pi[u]$ for all other nodes u. There is one remaining difficulty: we must look at nodes in the correct order. By the time we start calculating the shortest path to v, we must already know the shortest path to every node with an edge to v.

Fortunately, a topological sort of the DAG orders the vertices in just the way we need: it orders the vertices such that if there is an edge from u to v, then u appears before v in the ordering [Cormen et al., 2001]. Even better, the arrangement of nodes first shown in figure 6.1 easily yields a topological sort — we need only read off every node in a column from top to bottom for every column from left to right.

The complete algorithm for finding the shortest path between a and b in a DAG G is shown in algorithm 6.1. We visit each node u in topologically sorted order and lower $d[v]$ for each node v to which u is adjacent if the path from a to v through u is shorter than any path from a to v seen so far. Because the nodes are visited in topologically sorted order, we will have performed this lowering step for

every node leading to a node u before visiting u, meaning we will have computed $\min_i [\max(d[u_i], w_i)]$.

Algorithm 6.1 Shortest path algorithm. Finds the shortest path between a and b in the DAG G.

```
OPTIMAL-PATH(G, a, b)
1  for all u in G
2      do d[u] ← ∞
3         π[u] ← NIL
4  d[a] ← 0
5  for u from a to b in topologically sorted order
6      do for all v adjacent to u with edge weight w
7             do if max(d[u], w) < d[v]
8                   then d[v] ← max(d[u], w)
9                        π[v] ← u
```

Because we are only interested in the shortest path from a to b, we only examine the nodes between a and b in the topologically sorted order (line 5). There can be no edges from a to any nodes before it in this order, so no shortest path from a to b can contain any such nodes. There can be no edges from any node after b in this order to b, so no shortest path from a to b can contain any such nodes.

Consider the running time of algorithm 6.1. The loop starting on line 1 takes $\Theta(V)$ time. Line 5 requires us to topologically sort the nodes of G, which takes no time at all given the structure of our graph. The loop starting on line 5 runs once for each vertex, and the loop starting on line 6 runs once for each edge outgoing from it. Together, that is one iteration of lines 7–9 (which run in constant time) for every edge in G. If there are more edges than nodes in the graph — which is true in our case — the whole algorithm runs in $\Theta(E)$ time.

Suppose we say $\Delta l = l_{max} - l_{min} + 1$ and $\Delta o = o_{max} - omin + 1$. Then there are $\leq 2L\Delta l$ nodes in the graph and each has $\leq \Delta L \Delta o$ outgoing edges. Therefore, there are $\leq 2L\Delta L^2 \Delta o$ edges in the graph, so the algorithm is $O(L)$.

6.3 Applying Constraints

We now turn to the application of the constraints given in chapter 5. First, consider how the application of each will affect the graph. We have a constraint on overlaps:

C5. $|T_m(S_i, S_{i+1}) - T_0| \leq \Delta T, i = 1, 2, 3, \ldots N-1$

Because overlaps correspond to edges in the graph, the application of this constraint corresponds to deletion of edges.

We also have constraints on oligos:

C6. $\nexists\ (x, y) \in R$ s.t. $(b_i \leq y$ and $b_i - x + 1 \geq R_{min})$ or $(a_i \geq x$ and $y - a_i + 1 \geq R_{min})$
C7. $S_i \notin H, i = 1, 2, 3, \ldots N$

Because oligos correspond to nodes in the graph, the application of these constraints corresponds to deletion of nodes.

With that in mind, we now examine the application of each constraint in detail.

6.3.1 Finding Repeats

Constraint 6 refers to R, the set of all maximal repeat regions in S and $\overline{S}$ with length $\geq R_{min}$. Given a DNA sequence S, we must be able to compute this set efficiently; we can do so using a suffix tree [Gusfield, 1997].

Recall that we are concerned not only with repeats that occur in the sequence of the top strand, but with repeats that occur once in the top strand and once in the bottom strand. For example, the sequence `5'-ATGGGACTTACCCAT-3'` might not appear to contain any repeats at first glance, but the complementary strand (its bottom strand) is `5'-ATGGGTAAGTCCCAT-3'`. The two strands, taken together, contain two repeats of length five.

In order to find such repeats, we construct a sequence S' by concatenating S and $\overline{S}$, separated by a unique character which does not appear in either. We then use the method described in Gusfield [1997] to find all maximal pairs[2] of

[2] A maximal pair is a pair of identical subsequences at different locations in a sequence which can not be extended to the left or right and remain identical. For example, in `ACATGCATT`, `CAT` (starting at positions 1 and 5) is a maximal pair, but `CA` (again starting at positions 1 and 5) is not.

length $\geq R_{min}$ in S'. The unique character ensures that no maximal pairs consist of subsequences that cross the boundary between S and its reverse complement in S', which would be nonsensical. (Since the character only appears once in S', no repeat can contain it.)

Each maximal pair is computed as (i, j, l), where i and j are the starting indices of the two instances of the repeat and l is the length of the repeat. We must convert i and j, which are indices of S', to indices of S — repeats found in the second half of S' are repeats in the bottom strand of S and must be mapped to their location in S. This is straightforward: any $i \leq L$ in S' maps to i in S; $i = L+1$ cannot appear in any maximal pair, because it is a unique character; and any $i > L+1$ is a part of the reverse complement of S and maps to $2L + 3 - i - l$.

We note one complexity: many repeats will appear multiple times in the list of maximal pairs in S'. For any pair in which both instances occur in the top strand ($i, j \leq L$), there will be a second pair where both instances occur in the bottom strand ($i, j \geq 2L+2$) — any repeat in the top strand will necessarily have a corresponding repeat in the bottom strand. Inverted repeats, where one instance occurs in the top strand and on in the bottom strand, will also appear twice for the same reason. To avoid duplicates, we adopt the convention of ignoring (i, j, l) if both instances are in the bottom strand or if they are in different strands and (after conversion to S indices) $j > i$.

Algorithm 6.2 describes the function Find-Repeats, which finds all maximal repeats of length $\geq R_{min}$ in S. It returns a set $R = (i, j, l, d)$ of repeat regions in S, where i and j are the starts of a repeated substring, l is that substring's length, and d is + if the repeats are in the same strand or − if the are from opposite strands (an inverted repeat). The algorithm ensures that $i \leq j$. It assumes the existence of a function Maximal-Pairs which returns a set of maximal pairs (i, j, l) in a string, where $i < j$.[3]

Now that we can find all repeats in our DNA duplex, we can find all oligos which could misprime or cause mispriming. Recall that an oligo is problematic if either its 5' end or its 3' end includes a repeat of length $\geq R_{min}$. Therefore, we must look for all oligos which include at least R_{min} bases from any of the repeats found by Find-Repeats.

[3]See Gusfield [1997] for the details of implementing such a function.

Algorithm 6.2 Find all maximal repeats of length $\geq R_{min}$ in both strands of S.

```
FIND-REPEATS(S, R_min)
S' ← S + SEPARATOR + S̄
P ← MAXIMAL-PAIRS(S', R_min)
R ← ∅
for all (i, j, l) ∈ P
      do if i ≤ L and j ≤ L
            then R ← R ∪ {(i, j, l, +)}
         if i ≤ L and j ≥ L + 2
            then a ← i
                 b ← 2L + 3 − j − l
                 if a ≤ b
                    then R ← R ∪ {(a, b, l, −)}
return R
```

Algorithm 6.3 illustrates this procedure in detail. Lines 2 through 4 flatten the list of pairs returned by FIND-REPEATS into a list of individual repeat regions. The loop beginning on line 7 adds all oligos with rightmost ends inside a repeat; the loop beginning on line 11 adds all oligos with leftmost ends inside a repeat.

6.3.2 Finding Hairpins

We can also find all oligos that form self-priming hairpins using FIND-REPEATS. We need only the inverted repeats — the repeats of the form $(i, j, l, -)$. Oligos containing both regions comprising an inverted repeat placed in such a way as to allow self-priming are then tagged as unusable (see algorithm 6.4). The loop beginning on line 9 finds all oligos on the bottom strand that could self-prime due to a given inverted repeat. The loop beginning on line 15 finds all oligos on the top strand that could self-prime due to a given inverted repeat.

6.4 Generality and Extensibility

Chapter 5 presented a handful of constraints; each was addressed in this chapter. However, it should be noted that the algorithm given here is completely general: any constraint on individual oligos or on oligo overlaps can be enforced by delet-

Algorithm 6.3 Finding mispriming oligos. Returns the set of oligos $\{(i, j, s)\}$ that could misprime.

```
FIND-MISPRIMING-OLIGOS(S, R_min, l_min, l_max)
R ← FIND-REPEATS(S, R_min)
R′ ← ∅
for (i, j, l, d) ∈ R
    do R′ ← R′ ∪ {(i, i + l − 1), (j, j + l − 1)}
O ← ∅
for (i, j, l, d) ∈ R′
    do for y from min (i + R_min, L) to j
            do for x from y − l_max to y − l_min
                    do if x > 0
                            then O ← O ∪ {(x, y, +), (x, y, −)}
        for x from max (b − R_min, 0) down to i
            do for y from x + l_min to x + l_max
                    do if y ≤ L
                            then O ← O ∪ {(x, y, +), (x, y, −)}
return O
```

ing nodes or edges from the graph. Furthermore, the objective function optimized can be changed: any function that results in a problem with the property of optimal substructure can be used.[4] This algorithm is therefore an excellent framework for solving the oligo design problem with arbitrary constraints.

[4] See Cormen et al. [2001] for a detailed discussion of optimal substructure. In short, a problem with optimal substructure has the property that its optimal solution contains optimal solutions to its subproblems.

Algorithm 6.4 Finding self-priming hairpins. Returns the set of oligos $\{i, j, s\}$ that could self-prime.

```
FIND-SELF-PRIMING-HAIRPINS(S, H_min, l_min, l_max)
R ← FIND-REPEATS(S, H_min)
H ← ∅
for (i, j, l, d) ∈ R
    do if d = −
          then a ← i
               b ← i + l − 1
               c ← j
               d ← j + l − 1
               for x from a to b − H_min + 1
                   do for y from x + l_min − 1 to x + l_max − 1
                          do l_hang ← y − (c + b − x)
                             l_loop ← b + c − 2x − 2H_min + 1
                             if l_hang > 0 and l_loop ≥ 3
                                then H ← H ∪ {(x, y, −)}
               for y from c − H_min + 1 to d
                   do for x from y − l_max + 1 to y − l_min + 1
                          do l_hang ← (c + b − y) − x
                             l_loop ← −b − c + 2y − 2H_min + 1
                             if l_hang > 0 and l_loop ≥ 3
                                then H ← H ∪ {(x, y, +)}
return H
```

7 Implementation: Mason

The algorithm described in chapter 6 has been implemented in the software tool Mason[1], written in the programming language Common Lisp.

The current version of Mason consists of a number of modules:

- `graph`
  An implementation of a DAG and the DAG shortest path algorithm.
- `util`
  Utility functions relating to DNA.
- `thermo`
  DNA thermodynamics functions (T_m estimation).
- `suffix-tree`
  A naive implementation of a suffix tree.
- `repeats`
  Functions for finding repeats, mispriming oligos, and self-priming hairpins.
- `bl`
  The top level program logic.

In addition, a suite of unit tests for each module is partially completed.

Future versions of Mason will be more modular, allowing easy integration of custom constraints and easy use of alternative objective functions.

[1] **M**ason **A**ssembles **S**ynthetic **O**ligo**n**ucleotides

8 The Codon Optimization Problem

So far, we have addressed the problem of synthesizing an exact DNA sequence. However, the DNA usually encodes a protein sequence and related features. Because the genetic code is degenerate, many DNA sequences exist that code for a given protein sequence. If the protein is the primary concern, we are free to change codons to different but synonymous ones with little consequence.[1] This might allow us to:

- remove problematic repeats,
- optimize melting temperatures,
- improve gene expression, or…
- avoid specific sequences (e.g. restriction sites).

The next three chapters address a new a more difficult problem: how can we simultaneously change codons in a sequence while finding a good set of oligos with which to synthesize it? This work is detailed, but theoretical: the algorithm described has not yet been implemented in Mason.

8.1 Specifying Coding Regions

If we are to change codons in protein-coding regions, we must first know where those regions are. We need another parameter:

P9. C The set of codons in the input sequence that may be changed.

The exact form of C depends on how the input sequence is provided. The sequence could be provided as a nucleotide sequence that is backtranslated:

```
TTATGAGCAGTGA
```

[1] But not without consequence — see section 8.2.1

…or as a hybrid of nucleotides and amino acids:

```
TTMetSerSerGA
```

8.2 The Optimal Solution

We will need a new definition of an optimal solution for the codon optimization problem. First, consider the form of a solution. It includes a set of oligos $\{S_i\} = \{(a_i, b_i)\}$. In addition, it must include an assignment of three nucleotides to every codon in the input sequence: $c_1 = v_1, c_2 = v_2, c_3 = v_3, \ldots c_{|C|} = v_{|C|}$, where $|C|$ is the number of codons in the sequence.

It would be difficult to create a meaningful objective function of both $\{S_i\}$ and $\{c_i = v_i\}$, so we must choose between two options:

1. Optimize a function of $\{S_i\}$. We continue looking for the set of oligos with a minimal maximum deviation from T_0. (With the freedom to change codons, the search space becomes much bigger; but, it also becomes possible to find better solutions.)
2. Optimize a function of $\{c_i = v_i\}$. We optimize some function of the codons chosen. (We must also ensure we are able to find a good set of oligos for synthesis.)

We will choose the second option and optimize a function of codon choice.

To guarantee any solution we choose has an acceptable set of oligos for synthesis, we will use the parameter ΔT from the algorithm in chapter 6. ΔT was included there to prune clearly undesirable oligo overlaps from the search space; here we will use it to set a threshold for acceptable solutions. We add the constraint that a solution exists:

C8. Existence of a solution to the oligo design problem.
There is a solution to the oligo design problem for the sequence S that results from all codon choices $\{c_i = v_i\}$.

Using ΔT in this way, we are guaranteed that *any* solution to the oligo design problem is a good solution. We are free to optimize for codon choice.

Now we define our objective function, $f(\{c_i\})$. For reasons explained in chapter 9, we will want f to be in one of two forms:

$$f(\{c_i\}) = \sum_{i=1}^{|C|} h(c_i) \quad \text{or} \quad f(\{c_i\}) = \prod_{i=1}^{|C|} h(c_i)$$

The value of our objective function for a set of codon choices $\{c_i\}$ is the sum or product over all choices of some function h of a single choice. Our f will be of the second form (a product). To understand why such an objective function is a good choice, we need to discuss the usage of synonymous codons in genes.

8.2.1 Codon Preference

Though amino acids can be encoded by as many as six codons (see table 2.1), not all codons are created equal. Organisms show a marked preference for certain codons over other synonymous ones: the use of rare codons is correlated with lower gene expression [Gouy and Gautier, 1982].

Our objective function will assign a score to each of the sixty-four possible codons; since genes are synthesized for expression in specific organisms, the score for each codon will be a function of that codon's prevalence in the target organism. This requires two more parameters:

P10. M The genetic code used by the target organism. Formally, an injection from amino acids to codons.

P11. U A codon usage table for the target organism. This table gives, for each codon, the fraction of times that codon is used to encode its amino acid.

Our function of individual codons will be $h(c) = U[c]$, and our objective function will be:

$$f(\{c_i\}) = \prod_{i=1}^{|C|} U[c_i]$$

Intuitively, this objective function avoids very rare codons at all costs: a solution that uses mostly very common codons but a few very rare codons will have a very poor score, while a solution that uses all moderately common codons will have a decent score.

8.3 Avoiding Specific Subsequences

Because we can now change the input sequence, we have the ability to avoid specific subsequences in the modified sequence we eventually choose. This could allow us to remove (or avoid introducing) restriction sites, or to avoid false ribosomal binding sites.

We therefore introduce a new parameter:

P12. X A list of sequences that may not appear in the final sequence.

…and a new constraint:

C9. $\forall x \in X$, x is not a subsequence of S.
No sequence in X appears in S, where S is the sequence produced by the set of codon assignments $\{c_i = v_i\}$.

8.4 Summary

We've now completed our formulation of the codon optimization problem, which is formally summarized below.

Given a sequence S and the following parameters:

P9. C
P10. M
P11. U
P12. X

…find a set of codons $\{c_i\}$ that maximizes:

$$f(\{c_i\}) = \prod_{i=1}^{|C|} U[c_i]$$

…while satisfying the following constraints:

C8. Existence of a solution to the oligo design problem.
C9. $\forall x \in X$, x is not a subsequence of S.

9 Background: Constraint Satisfaction

In order to solve the oligo design problem with codon optimization, we'll need to take advantage of some algorithms commonly used in the field of constraint optimization. This chapter briefly introduces constraint satisfaction problems and constraint optimization problems. It then describes two algorithms, branch and bound and conflict-directed A*, which could be used to solve the codon optimization problem.

9.1 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is a tuple $\langle X, D, C \rangle$, where:

- X is a set of variables $x_1, x_2, x_3, \ldots x_n$,
- D is an associated set of domains $d_1, d_2, d_3, \ldots d_n$, and…
- C is a set of constraints.

Each variable x_i can be assigned a single value from its domain d_i. An *assignment* is a mapping from variables to values. An assignment that maps every variable to an element of its domain is a *complete assignment*; an assignment that maps only some variables is a *partial assignment*. An assignment A is a subset of an assignment B ($A \subseteq B$), if every assignment $x_i = v \in d_i$ in A is also in B.[1]

Each constraint $c \in C$ consists of two parts:

- a subset of the CSP's variables, and…
- a list of assignments to that subset.

An assignment A satisfies a constraint c if there exists an assignment B in c such that $B \subseteq A$. A *solution* to a CSP is a complete assignment which satisfies all constraints. See Russell and Norvig [2002] for a more detailed introduction to CSPs.

[1] For example, if $A = \{x_1 = 1, x_2 = 2\}$ and $B = \{x_1 = 1, x_2 = 2, x_3 = 3\}$, then $A \subset B$.

As an example, consider the problem of 3-coloring the graph shown in figure 9.1. Our goal is to color each node red, green, or blue in such a way that no adjacent nodes are the same color.

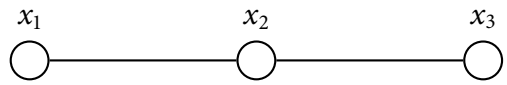


Figure 9.1 – A graph to be 3-colored.

Formulating this problem as a CSP is straightforward. We have three variables: x_1, x_2, and x_3. Each variable has the same domain: $d_1 = d_2 = d_3 = \{R, G, B\}$. Each edge introduces the constraint that the nodes it connects cannot have the same color. We have two constraints:

$$c_1 = \left\langle \{x_1, x_2\}, \left\{ \begin{array}{lll} \{x_1 = R, x_2 = G\}, & \{x_1 = R, x_2 = B\}, & \{x_1 = G, x_2 = R\}, \\ \{x_1 = G, x_2 = B\}, & \{x_1 = B, x_2 = R\}, & \{x_1 = B, x_2 = G\} \end{array} \right\} \right\rangle$$

…and:

$$c_2 = \left\langle \{x_2, x_3\}, \left\{ \begin{array}{lll} \{x_2 = R, x_3 = G\}, & \{x_2 = R, x_3 = B\}, & \{x_2 = G, x_3 = R\}, \\ \{x_2 = G, x_3 = B\}, & \{x_2 = B, x_3 = R\}, & \{x_2 = B, x_3 = G\} \end{array} \right\} \right\rangle$$

There are many solutions to this CSP. One solution is $\{x_1 = R, x_2 = G, x_3 = B\}$.

Though this method of specifying constraints is simple and general, it is unwieldy. Normally, some mathematical language is defined that allows constraints to be expressed more concisely. Using such a language, we might write the constraints above as $x_1 \neq x_2$ and $x_2 \neq x_3$.

9.2 Constraint Optimization Problems

A constraint optimization problem (COP) is a tuple $\langle X, D, C, f \rangle$. X, D, and C comprise a CSP; f is a function that maps solutions of that CSP to real values. The goal is to find a solution which maximizes f, i.e. to find a solution Z such that $Z = \arg\max_{\sigma \in \Sigma} f(\sigma)$, where Σ is the set of all solutions to the CSP [Tsang, 1993].

Suppose we make the CSP above into a COP by adding the following objective function:

$$f(\{x_i\}) = \sum_{i=1}^{3} \left\{ \begin{array}{ll} 1 & \text{if } x_i = R \\ 2 & \text{if } x_i = G \\ 3 & \text{if } x_i = B \end{array} \right.$$

Each of the many solutions to the earlier CSP is now valued by this objective function. It is obvious that there is a single solution that maximizes f: $\{x_1 = B, x_2 = G, x_3 = B\}$, for which $f = 8$.

9.3 Solving Constraint Optimization Problems

Though the example COP above was quite easy to solve by inspection, the general problem of solving COPs is difficult. This section presents two algorithms that solve that problem: branch and bound and conflict-directed A*.

9.3.1 Branch and Bound

Branch and bound is a search technique used to solve many types of optimization problems. It treats optimization as a tree search and accelerates the search by using heuristics to prune entire subtrees all at once [Tsang, 1993].

Each node of the branch and bound search tree corresponds to an assignment: the root node is the empty assignment, internal nodes are partial assignments, and leaves are complete assignments. Generally, a level of the tree corresponds to a single variable — an new assignment is constructed at each node in that level by adding an assignment to that variable.

Branch and bound requires a heuristic function h that, for any partial assignment A, returns an upper bound on the scores of all assignments $B \supseteq A$. In the language of search trees, h estimates the best solution we could find in the subtree rooted at a given node. For correct behavior, h must be admissible — it must always return an accurate upper bound.

Branch and bound is shown in algorithm 9.1. It maintains a lower bound on the score of an optimal solution while performing a depth-first search on the tree. This bound is initialized to $-\infty$ (line 3); it is updated whenever a solution better than the best seen so far is found. Because of the check in line 14, any leaf visited by the search corresponds to a solution. Whenever the algorithm visits a leaf, it evaluates the corresponding solution using the objective function; if that solution has a higher score than any other seen so far, the algorithm records it and raises the lower bound (lines 6 – 9). Whenever the algorithm visits an internal node,

Algorithm 9.1 General branch and bound algorithm.

```
BRANCH-AND-BOUND(⟨X, D, C, f⟩, h)
1  A ← ∅
2  Z ← NIL
3  x ← −∞
4  VISIT(⟨X, D, C, f⟩, h, A, x)
5  return ⟨A, x⟩
VISIT(⟨X, D, C, F⟩, h, A, x)
6  if A is a solution
7     then if f(A) > x
8             then Z ← A
9                  x ← f(A)
10 elseif h(⟨X, D, C, f⟩, A) > x
11    then y ← the first variable in X not in A
12          for d ∈ the domain of y
13              do B ← A ∪ {y = d}
14                 if B violates no constraint c ∈ C
15                    then VISIT(⟨X, D, C, f⟩, h, B, x)
16 else return
```

it evaluates the corresponding partial assignment using the heuristic function h. If result is greater than the value of the best solution seen so far, the algorithm continues exploring the subtree rooted at that node (lines 10–15). Otherwise, the algorithm prunes the subtree rooted at that node: no leaf in that subtree could possibly correspond to a solution better than the best already found (line 15).

As an example, we'll solve the constraint optimization problem given in section 9.2. Recall that we are trying to 3-color the graph given in figure 9.1 while maximizing the function:

$$f(\{x_i\}) = \sum_{i=1}^{3} \begin{cases} 1 & \text{if } x_i = R \\ 2 & \text{if } x_i = G \\ 3 & \text{if } x_i = B \end{cases}$$

…where x_i is the color of each node.

For branch and bound, we will need an admissible heuristic function h. The

following simple function will work:

$$h(\{x_i\}) = \sum_{i=1}^{3} \begin{cases} 1 & \text{if } x_i = R \\ 2 & \text{if } x_i = G \\ 3 & \text{if } x_i = B \\ 3 & \text{if } x_i \text{ is unassigned} \end{cases}$$

This h scores each assigned variable correctly and assumes every unassigned variable will make the largest possible contribution to the score; it is clearly an upper bound for the scores of all supersets of a given assignment.

Figure 9.2 shows the search tree generated by branch and bound using the heuristic h. Note that an optimal solution is found quickly, and two thirds of the tree is pruned rather than searched.

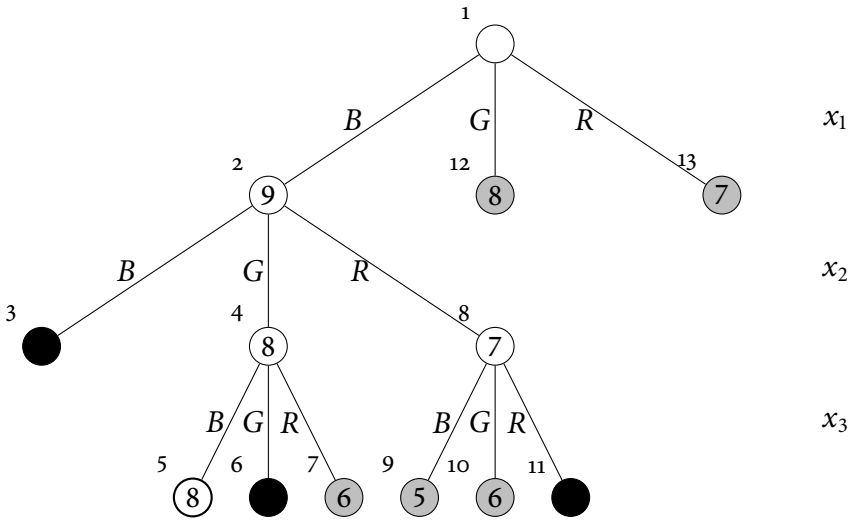


Figure 9.2 – Branch and bound search tree. Nodes are numbered in the order they were visited and labeled with their score or score estimate. Black nodes contain assignments that violate the constraints. Gray nodes were pruned or ignored. Nodes with bold borders replaced the best known solution when visited.

9.3.2 Conflict-directed A*

Conflict-directed A* (CDA*) is a second algorithm for solving COPs [Williams and Ragno, 2007]. The algorithm takes an intuitive approach: it generates full assignments in best-first order. If an assignment is a solution, CDA* returns it; if

it is not a solution, CDA* generalizes whatever causes it to violate the constraints and does not generate solutions with the same problem in the future.

It accomplishes this by extracting *conflicts* from complete assignments that violate constraints. A conflict is an assignment A with the property that no assignment $B \subseteq A$ could satisfy the constraints. A *minimal conflict* is a conflict that is not a conflict if any one of its individual assignments is removed.

In our 3-coloring example, the assignment $A = \{x_1 = B, x_2 = B\}$ is a conflict: no assignment $B \subseteq A$ could satisfy the constraint $x_1 \neq x_2$. Furthermore, A is a minimal conflict: neither $\{x_1 = B\}$ nor $\{x_2 = B\}$ — the two assignments we could obtain by removing a single variable assignment from A — is a conflict.

Because smaller conflicts allow us to skip over larger numbers of assignments, it is in our interest to find the smallest conflicts possible: we want to find minimal conflicts.

In order to efficiently generate assignments that contain no conflicts, CDA* imposes an additional restriction on the problem: it must possess the property of mutual, preferential independence (MPI). This property means that we maximize the objective function by choosing some best value for each variable, i.e. we maximize the objective function by maximizing some function of single variables for each variable individually. Any COP with an objective function which is the sum or product of a function evaluated on each variable individually exhibits MPI. This explains the choice of objective function in chapter 8.

A high-level picture of conflict-directed A* is given in algorithm 9.2. Its behavior is simple: it generates the best complete assignment that contains none of the conflicts discovered so far and checks if that assignment is a solution. If it is, the algorithm is done; if it is not, the algorithm extracts minimal conflicts and generates the next assignment.

Despite this apparent simplicity, the details of CDA* are complex; see Williams and Ragno [2007] for the details of implementing Next-Best-Assignment and Merge-Conflicts. The form of Extract-Conflicts depends on the type of CSP being optimized. (It will be addressed in chapter 10.)

We'll now illustrate the use of conflict-directed A* on our 3-coloring example. We won't need to delve into the implementation details, because it will be easy to list assignments in best-first order and to extract conflicts by inspection.

Algorithm 9.2 Conflict-directed A*.

```
CONFLICT-DIRECTED-A*(⟨X, D, C, f⟩)
K ← ∅
while true
    do A ← NEXT-BEST-ASSIGNMENT(⟨X, D, C, f⟩, K)
       if A satisfies all constraints in C
          then return A
       elseif A does not satisfy all constraints in C
          then κ ← EXTRACT-CONFLICTS(A)
               K ← MERGE-CONFLICTS(K, κ)
       else return
```

The best-scoring assignment is $\{x_1 = x_2 = x_3 = B\}$, for which $f = 9$. This assignment violates both constraints: $x_1 \neq x_2$ and $x_2 \neq x_3$. Both are minimal conflicts; we'll arbitrarily choose to extract $x_1 = x_2 = B$.

There is a three-way tie for the next best scoring assignment. The three assignments are:

$$\begin{array}{lll} \{x_1 = G, & x_2 = B, & x_3 = B\}, \\ \{x_1 = B, & x_2 = G, & x_3 = B\}, \text{ and...} \\ \{x_1 = B, & x_2 = B, & x_3 = G\} \end{array}$$

The third assignment does not resolve our conflict $x_1 = x_2 = B$, so we skip it. We try $\{x_1 = G, x_2 = B, x_3 = B\}$, but find that it violates the constraint $x_2 \neq x_3$. We extract the conflict $\{x_2 = B, x_3 = B\}$ and move to the next assignment, $\{x_1 = B, x_2 = G, x_3 = B\}$. This assignment satisfies all constraints, so it is the optimal solution.

10 An Algorithm for the Codon Optimization Problem

We now to turn to the question of finding the optimal solution to the oligo design problem with codon optimization. We will frame the problem as a constraint optimization problem and then see how we could use branch and bound or conflict-directed A* to solve it (see chapter 9).

10.1 Formulating the Problem

To begin, we'll formulate the codon optimization problem as a COP.

10.1.1 Variables and Domains

The variables will be $\{c_i\}$, one variable for each codon in S named by C. Each c_i encodes a specified amino acid; its domain d_i is the set of three-nucleotide sequences that code for that amino acid according to the supplied genetic code M.

```
GTGA??? ??? ??? ??? ??? ??? ???GTAACAG??? ??? ??? ??? ??? ???AAAAG
     Met Gly Leu Trp Ile Asp □          Met Lys Val Asp Gly □
      c1  c2  c3  c4  c5  c6  c7         c8  c9 c10 c11 c12 c13
```

Figure 10.1 – An input sequence with two hypothetical protein-coding regions and the corresponding amino acids and COP variables.

Figure 10.1 shows an example of the variables for a short input sequence with two protein-coding regions. If M were the standard genetic code shown in table 2.1, the variable c_2 would have the domain $\{\texttt{GGT}, \texttt{GGC}, \texttt{GGA}, \texttt{GGG}\}$ — the set of three-nucleotide sequences that code for Gly.

For the rest of this chapter, the sequence of an assignment or solution will refer to the sequence obtained by placing each codon's assigned three-nucleotide sequence in its position in S.

10.1.2 Constraints

Given a complete assignment to the variables $\{c_i\}$, we must be able to check that all the constraints listed in chapter 8 are satisfied. There are two constraints: the constraint on the presence of subsequences in the final sequence, and the constraint on the existence of a set of acceptable oligos for assembly of that sequence.

Checking the first constraint is simple. We construct S by filling in each codon c_i with its assigned value. We then check for the presence of each x in S. Every such check is an instance of the string matching problem, which is easily solved [Cormen et al., 2001].

To check for the existence of an acceptable set of oligos for assembly, we execute the algorithm from chapter 6 on the sequence. If that algorithm returns an answer, an acceptable set of oligos exists.

10.1.3 The Objective Function

Recall from chapter 8 that our objective function is:

$$f(\{c_i\}) = \prod_{i=1}^{|C|} U[c_i]$$

...where C is a list of codon locations and U is a table of codon usage for the target organism. Given assignments to each c_i, evaluating this function is straightforward.

10.2 Using Branch and Bound

At this point, we can check that complete assignments satisfy our constraints, but we cannot yet use either of the COP algorithms described in chapter 9. We'll start by extending our methods so that we can use branch and bound.

Branch and bound (algorithm 9.1) requires two things we cannot yet do:

1. line 10: evaluate the heuristic function for a partial assignment.
2. line 14: check a partial assignment for consistency with the constraints.

10.2.1 The Heuristic

For the first, we need an admissible heuristic function. One optimistic estimate assumes that each unassigned variable will have the codon in U with the largest usage. We therefore define our heuristic as:

$$h(\{c_i\}) = \begin{cases} U[c_i] & \text{if } c_i \text{ is assigned} \\ \max\limits_{v \in d_i} U[v] & \text{if } c_i \text{ is unassigned} \end{cases}$$

It is clear that no assignment B that is a superset of an assignment A can have a score higher than $h(A)$ — the additional assignments in B will either lower h or leave it unchanged — so h is an admissible heuristic.

10.2.2 Checking Partial Assignments

The second extension, the ability to check partial assignments for consistency, is more involved. We'll address the constraint on the sequence and the constraint on the set of oligos for synthesis separately.

10.2.3 The Sequence Constraint

Our constraint is that no $x \in X$ is a subsequence of the full sequence. We can check a partial assignment against this constraint by constructing a sequence S in which codons that are assigned are written out as nucleotides and codons that are unassigned are written out as the three special non-nucleotide characters — `???` — that do not appear in any $x \in X$ (figure 10.2). We then search this sequence for forbidden subsequences as though it was a full assignment. If an instance of a forbidden subsequence is found, no assignment that is a superset of the assignment we are checking could possibly satisfy the constraints — the forbidden subsequence would appear in all such assignments. Otherwise, we can say nothing definitive about assignments that are supersets of the assignment we're checking: it's possible that some such assignments could satisfy the forbidden sequence constraint.

```
GTGAATGGGT???????ATT???TGAGTAACAGAGT???????????GGG???AAAAG
```

Met	Gly	Leu	Trp	Ile	Asp	Stp	Met	Lys	Val	Asp	Gly	Stp
c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}

Figure 10.2 – An example of a partial assignment to codon variables.

10.2.4 The Solution Constraint

Checking for the potential existence of an acceptable set of oligos for synthesis is more difficult. We'll use the algorithm from chapter 6 to find a solution, but without a full assignment we don't have a fully-defined sequence. The solution is to effectively ignore all nucleotides in the sequence that are undefined. We adopt the following rules:

1. Undefined characters never contribute to repeats.
2. Undefined characters never contribute to the formation of self-priming hairpins.
3. Overlaps containing undefined nucleotides are assumed to have the target melting temperature.

Modifying our algorithms to make these assumptions allow us to confidently state when a partial assignment A has no children $B \subset A$ that are solutions.

10.2.4.1 Repeats

We need to modify our repeat finding algorithm — if we simply ran it on a sequence full of ? characters as we did above, it would find many repeats containing those characters — exactly what we do not want. Instead, we replace each contiguous region of ? characters with a single unique character that appears nowhere else in the sequence (figure 10.3). Call these unique characters $?_1, ?_2, ?_3, \ldots$. They will serve the same purpose as the unique separator did in algorithm 6.2: they prevent repeats from nonsensically spanning two unrelated places in the sequence.

We use a modified version of algorithm 6.2 on this sequence to find all maximal repeats that do not include any unique characters $?_i$ (algorithm 10.1). The main addition is the use of the function Replace-Undefined-Regions in line 1. Because the mapping from indices in S to indices in S' is much more complicated

```
GTGAATGGGT??????ATT???TGAGTAACAGAGT??????????GGG???AAAAG
           ?1     ?2                  ?3        ?4
                          |
                          v
GTGAATGGGT?1ATT?2GTAACAG?3GGG?4AAAAG
```

Figure 10.3 – Replacing undefined regions with unique characters.

when using this function, we assume it returns a function g that implements that mapping. The rest of algorithm 10.1 is similar to algorithm 6.2.

Algorithm 10.1 Finding all maximal repeats in a partially undefined sequence.

```
FIND-REPEATS-WITH-UNDEFINED-REGIONS(S, R_min)
S', g ← REPLACE-UNDEFINED-REGIONS(S + SEPARATOR + S̄)
L ← (|S'| − 1)/2
P ← MAXIMAL-PAIRS(S', R_min)
R ← ∅
for all (i, j, l) ∈ P
    do if i ≤ L and j ≤ L
          then R ← R ∪ {(g(i), (g(j), l, +)}
       if i ≤ L and j ≥ L + 2
          then a ← i
               b ← 2L + 3 − j − l
               if a ≤ b
                  then R ← R ∪ {(g(a), (g(b), l, −)}
return R
```

Why does this approach work? Suppose we have two assignments: A and $B \supset A$. Further suppose we run algorithm 10.1 on S_A and S_B, the sequences corresponding to both assignments, yielding R_A and R_B. It is clear that every repeat $r_A \in R_A$ either appears in R_B or is a subsequence of some some repeat $r_B \in R_B$ — every character in S_A appears in S_B, so every repeat in S_A either occurs in S_B or becomes larger in S_B.

Since R_B contains every repeat in R_A (or a larger one), every oligo from S_A with repeats at its ends will also have repeats at its ends in S_B. Furthermore, because there could be larger repeats in R_B, additional oligos from S_B might be have repeats at their ends.

Now consider the graphs the algorithm from chapter 6 constructs for each

sequence, G_A and G_B. Each node in a graph corresponding to an oligo that could misprime is removed from that graph; since the set of mispriming oligos from S_B is a superset of the set of mispriming oligos in S_A, the set of nodes in G_B is a subset of the set of nodes in G_A.

10.2.4.2 Mispriming Oligos

We find mispriming oligos using algorithm 6.3, but we use Find-Repeats-With-Undefined-Regions instead of Find-Repeats.

10.2.4.3 Hairpins

We find self-priming hairpins using algorithm 6.4, but we use Find-Repeats-With-Undefined-Regions instead of Find-Repeats.

10.2.4.4 Overlaps

The modification for overlaps is simple: when constructing the graph, an edge that corresponds to an overlap that contains undefined nucleotides is given a weight of zero. We've already seen that, if $B \supseteq A$, then the set of nodes of G_B is a subset of the set of nodes of G_A. Now consider the sets of edges of G_A and G_B, E_A and E_B. Because $B \supset A$, there are as many or fewer undefined nucleotides in S_B as there are in S_A. Therefore, any edge appearing in E_B will also appear in E_A with the same or lower weight. Furthermore, because the nodes of G_B are a subset of the nodes of G_A, we know that $|E_B| \leq |E_A|$.

10.2.4.5 Putting Them Together

Suppose we have two assignments, A and $B \supset A$, and that we use the algorithms given above on them. We know a few things:

- There are at least as many nodes in G_A as in G_B.
- Every edge in G_B also appears in G_A, possibly with a lower weight.

Going from G_A to G_B means deleting nodes (thereby deleting some edges), deleting some other edges, and increasing the weight of yet other edges. In other words, it means removing some paths through the graph and increasing the cost

of others. Therefore, the shortest path through G_B is at least as long as the shortest path through G_A, so if no solution exists for A, no solution can exist for B.

10.2.5 Being Smart

We now have all the tools we need to use branch and bound to solve the codon optimization problem. However, there are some things we can do to make the algorithm more effective.

The first is to assign codons in an intelligent order. Our search will be fastest if the subtrees we prune are as large as possible. We therefore want our search tree to branch as little as possible at the top — that way, when we find a subtree to prune, we'll eliminate a large number of solutions. It is best to start by assigning the most restricted codons (those with the smallest domains) first.

The second thing we can do is to always explore the best solutions first. The higher our upper bound, the more often we can prune subtrees. Because our COP exhibits MPI, it is easy to search in best-first order — we simply pick the best codon at each node.

10.3 Using Conflict-directed A*

This section describes how to use conflict-directed A* to solve the codon optimization problem. There is a single hurdle to jump: we need to be able to extract conflicts from a complete assignment that does not satisfy some constraint (line 7 in algorithm 9.2). This section examines the problem of conflict extraction and shows how to use CDA*.

10.3.1 Extracting Conflicts

Recall that we have two constraints: the sequence constraint and the oligo set constraint. We'll need different methods to extract minimal conflicts from complete assignments that violate each.

10.3.1.1 *Extracting Minimal Conflicts for Sequence Constraints*

The problem of extracting conflicts from an assignment that violates sequence constraints is easily solved. Violations of sequence constraints are caused by the presence of forbidden subsequences in the full sequence; the only variables participating in such violations are those that correspond to nucleotides participating in the formation of forbidden subsequences.

To find the smallest conflicts possible, we treat each instance of a forbidden subsequence individually and construct a conflict consisting only of the variable assignments contributing to the formation of that instance. Algorithm 10.2 shows this process in detail.

Algorithm 10.2 Extracting conflicts for sequence constraints.

```
EXTRACT-SEQUENCE-MIN-CONFLICTS(⟨PROBLEM⟩, A)
1  K ← ∅
2  for every instance z of a sequence x ∈ X
3      do k ← ∅
4          for every assignment c_i = v_i for which c_i is part of z
5              do k ← k ∪ {c_i = v_i}
6          K ← K ∪ k
7  return K
```

10.3.1.2 *Extracting Minimal Conflicts for the Solution Constraint*

Many forms of CSP admit algorithms for extracting minimal conflicts from complete assignments like that above; unfortunately, our graph formulation does not. We therefore use algorithm 10.3, a more general method of minimal conflicts.

This algorithm take a complete assignment A that contains a conflict and constructs a minimal conflict $B \supseteq A$. It does so by initializing B to A and walking through each individual assignment to a variable, deleting those assignments that don't make B feasible when removed.

Why does this algorithm work? At each step, we form Q by removing the next individual variable assignment $v \in A$ from K, our set so far. At this point in the iteration, K is infeasible and so must contain some minimal conflict κ. If removing v does not fix the problem, then v cannot be a part of κ, so we leave it

Algorithm 10.3 Extracting a minimal conflict from a complete assignment.

```
EXTRACT-MIN-CONFLICT(⟨X, D, C⟩, A)
1  K ← A
2  for each assignment x_i = v_i in A
3      do Q ← K ∖ {x_i = v_i}
4         if ⟨X, D, C⟩ is not feasible for Q
5            then K ← Q
6  return K
```

out. If removing v does fix the problem, then it must be a part of κ, so we put it back in. After all variable assignments have been examined, $K = \kappa$, the minimal conflict.

10.3.2 Our Conflict Extraction Algorithm

We now have the tools we need to use conflict-directed A* to solve the codon optimization problem. We use the algorithm as given in algorithm 9.2. The particular form of our implementation of EXTRACT-CONFLICTS is shown in algorithm 10.4.

Algorithm 10.4 Extracting minimal conflicts.

```
EXTRACT-CONFLICTS(⟨PROBLEM⟩, A)
1  K_seq ← ∅
2  K_sol ← ∅
3  if A violates the sequence constraint
4     then K_seq ← EXTRACT-SEQUENCE-MIN-CONFLICTS(⟨PROBLEM⟩, A)
5  if A violates the oligo set constraint
6     then K_sol ← {EXTRACT-SOLUTION-MIN-CONFLICT(⟨PROBLEM⟩, A)}
7  return K_seq ∪ K_sol
```

10.4 Branch and Bound vs. Conflict-directed A*

We now address the question of which of the two COP algorithms we've discussed is superior. Both work by excluding large portions of the search space, but their methods differ: branch and bound evaluates a heuristic function on partial assignments; conflict-directed A* extracts conflicts from complete assignments. Branch

and bound finds the best solution by proving that no unexplored assignments could have a score higher than the optimal solution; conflict-directed A* examines assignments in best-first order.

We must answer two questions:

1. Do most sequences have an acceptable set of oligonucleotides?
2. Can the algorithm from chapter 6 be run quickly?

If the answer to both questions is yes, then conflict-directed A* is the superior choice. If most sequences have solutions, then CDA* won't have to explore many states before finding the best. If the algorithm from chapter 6 can be run quickly, then exploring those few states will not take long.

If, on the other hand, the answer to both questions is no, branch and bound may be superior. Algorithm 10.3 calls the algorithm from chapter 6 many times; if that algorithm is slow, conflict extraction for CDA* will be slow. If most sequences don't have an acceptable set of oligos, we'll have to try many sequences before finding a solution, and CDA* will be even slower. Branch and bound makes fewer calls to the algorithm from chapter 6 and, because it tries partial assignments, might rule out subtrees quickly.

In other situations, it is difficult to predict how the algorithms will compare.

11 Conclusion & Future Work

This thesis has introduced gene synthesis, explained its importance, and shown that the software tools available to address it are inadequate. It has formalized the problem of oligonucleotide design for PCR-based gene synthesis, given an algorithm that solves that problem, and presented Mason — a software tool employing that algorithm. Finally, it has formalized the problem of codon optimization with oligonucleotide design and shown that existing constraint optimization algorithms could theoretically be combined with a modified version of the oligonucleotide design algorithm to solve that problem.

Moving forward, the first step is to begin synthesizing genes using the existing implementation of Mason. This will provide validation that the constraints described here are valid, and may suggest new constraints to be added.

The next step is to extend Mason to solve the problem of codon optimization with oligonucleotide design. This requires a number of new or expanded algorithms to be implemented, and requires integration with an existing constraint solver, or implementation of a new one.

Eventually the combination of Mason and a COP-based codon optimization framework could become a powerful tool for synthetic biologists, and one that they are sorely lacking.

References

Marcus Bode, Samuel Khor, Hongye Ye, Mo-Huang Li, and Jackie Y. Ying. Tmprime: fast, flexible oligonucleotide design software for gene synthesis. *Nucleic Acids Research*, 37(Web Server Issue):W214–W221, 2009.

Marvin H. Caruthers. Gene Synthesis Machines: DNA Chemistry and Its Uses. *Science*, 230(4723):281–285, 1985.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

M. Gouy and C. Gautier. Codon usage in bacteria: correlation with gene expressivity. *Nucleic Acids Research*, 10(22):7055–7074, 1982.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

Guide to DNAWorks version 3.1. Helix Systems, March 2009. URL `http://helixweb.nih.gov/dnaworks/dnaworks_help.html`.

David M. Hoover and Jacek Lubkowski. DNAWorks: an automated method for designing oligonucleotides for PCR-based gene synthesis. *Nucleic Acids Research*, 30 (10):e43, 2002.

Keiichi Itakura, Tadaaki Hirose, Roberto Crea, Arthur D. Riggs, Herbert L. Heyneker, Francisco Bolivar, and Herbert W. Boyer. Expression in *Escherichia coli* of a Chemically Synthesized Gene for the Hormone Somatostatin. *Science*, 198(4321): 1056–1063, 1977.

Sebastian Jayaraj, Ralph Reid, and Daniel V. Santi. GeMS: an advanced software package for designing synthetic genes. *Nucleic Acids Research*, 33(9):3011–3016, 2005.

Tom Knight, Randall Rettberg, Leon Chan, Drew Endy, Reshma Shetty, and Austin Che. Idempotent Vector Design for the Standard Assembly of Biobricks. RFC 9, The BioBricks Foundation, 2003.

Don Lorimer, Amy Raymond, John Walchli, Mark Mixon, Adrienne Barrow, Ellen Wallace, Rena Grice, Alex Burgin, and Lance Stewart. Gene composer: database software for protein construct design, codon engineering, and gene synthesis. *BMC Biotechnology*, 9, 2009.

Sarah M. Richardson, Sarah J. Wheelan, Robert M. Yarrington, and Jef D. Boeke. Genedesign: Rapid, automated design of multikilobase synthetic genes. *Genome Research*, 16:550–556, 2006.

Jean-Marie Rouillard, Woonghee Lee, Gilles Truan, Xiaolian Gao, Xiaochuan Zhou, and Erdogan Gulari. Gene2Oligo: oligonucleotide design for in vitro gene synthesis. *Nucleic Acids Research*, 32(Web Server Issue):W176–W180, 2004.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

John SantaLucia, Jr. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. *Proc. Natl. Acad. Sci.*, 95(4):1460–1465, 1998.

John SantaLucia, Jr. and Donald Hicks. The Thermodynamics of DNA Structural Motifs. *Annu. Rev. Biophys. Biomol. Struct.*, 33:415–440, 2004.

William P. C. Stemmer, Andreas Crameri, Kim D. Ha, Thomas M. Brennan, and Herbert L. Heyneker. Single-step assembly of a gene and entire plasmid from large numbers of oligodeoxyribonucleotides. *Gene*, 164(1):49–53, 1995.

Lance Stewart and Alex B. Burgin. Whole Gene Synthesis: A Gene-O-Matic Future. *Frontiers in Drug Design & Discovery*, 1(1):297–341, 2005.

Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 1993.

James D. Watson, Nancy H. Hopkins, Jeggrey W. Roberts, Joan Argetsinger Steitz, and Alan M. Weiner. *Molecular Biology of the Gene*. The Benjamin/Cummings Publishing Company, 4th edition, 1987.

Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

Amanda V. Wozniak. A Systematic and Extensible Approach to DNA Primer Design for Whole Gene Synthesis. Master's thesis, Massachusetts Institute of Technology, 2005.

Gang Wu, Julie B. Wolf, Ameer F. Ibrahim, Stephanie Vadasz, Muditha Gunasinghe, and Stephen J. Freeland. Simplified gene synthesis: A one-step approach to PCR-based gene construction. *Journal of Biotechnology*, 124(3):496–503, 2006.