1989

Geneplanner: A Prototype of an expert system to assist with chemical DNA gene synthesis planning

Marilyn Daum

Rochester Institute of Technology
School of Computer Science and Technology

Geneplanner:
A Prototype of an Expert System to Assist with Chemical DNA Gene Synthesis Planning

by

Marilyn Daum

A master's thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

21 July 1989

Thesis Committee:

Dr. Walter Wolf

Professor John A. Biles

Dr. Robert Rothman

Abstract

Expert systems are a popular area of artificial intelligence. The development of an expert system involves the selection of an appropriate problem, acquisition of knowledge from the expert, selection of control mechanisms and knowledge representations, selection of tools, implementation, and testing. This thesis describes the development of a prototype expert system in the area of genetic engineering. The prototype system suggests the fragments of DNA to chemically synthesize and the steps for joining these fragments in order to make a gene. The system follows the hueristic rules of an expert to select the fragments and strategy for synthesis, backtracking where necessary. After reviewing expert systems and the problem area, the thesis focuses on the development process. Each of the steps is discussed, and the iterative nature of implementation, testing, and refinement is displayed. Results are reviewed, showing Geneplanner to handle simple to moderate cases fairly well. Finally, shortcomings are discussed and future enhancements are suggested.

Contents

List of Figures

Chapter 1

Introduction

Expert systems are a relatively new and popular area of artificial intelligence. The development of an expert system involves selecting an application problem, acquiring knowledge from an expert, representing that knowledge in an appropriate way, and implementing and testing a system that uses and explains this expertise. One of the areas that has provided several applications for expert systems (and computer systems in general) is molecular biology [1]. Within this area, chemical DNA gene synthesis planning presents a problem particularly well-suited to an expert system. In chemical gene synthesis, a long chain of DNA is made by synthesizing small fragments of the chain and then joining these fragments via enzymes. These fragments, or oligonucleotides, cannot be chosen indiscriminately; rather, there is a complex set of rules used to determine where to place the oligonucleotide boundaries.

This paper describes an expert system prototype called Geneplanner which will suggest the optimal oligonucleotides of DNA to chemically synthe-

size and the procedure for joining these fragments into the final gene. The knowledge base of the system contains the rules of an expert gene synthesizer. PROLOG, the implementation tool, provides the inference engine and applies the backtracking, which is an inherent part of the synthesis planning process. The general strategy follows the expert's approach: to proceed forward through the gene, fragment by fragment, adjusting boundaries as necessary to avoid critical problem areas. Test results show that Geneplanner handles simple to moderate cases fairly well.

Chapter 2 provides background on expert systems and PROLOG. Chapter 3 presents background on DNA and describes the chemical gene synthesis process. Chapter 4 provides background on the approach to developing Geneplanner. Chapter 5 describes the implementation process. Chapter 6 presents test results. Finally, Chapter 7 discusses potential future enhancements.

Chapter 2

Background on Expert Systems

2.1 Overview

The development of expert or knowledge-based systems has become a popular area within artificial intelligence over the past few years. These systems incorporate expert knowledge to intelligently solve problems within a narrow domain. The knowledge in an expert system is structured and represented symbolically; thus these systems are said to perform symbolic reasoning. Expert systems address difficult and complex problems which are solved via heuristics, that is, rules of thumb that simplify the search for solutions. Also, they are usually able to explain their own operation [WATE86].

An expert system is logically divided into two parts, the knowledge base and the inference engine: "The knowledge base ... contains facts (data) and rules (or other representations) that use those facts as the basis for decision making. The inference engine contains an interpreter that decides how to

apply the rules to infer new knowledge and a scheduler that decides the order in which the rule should be applied." [WATE86].

The success of an expert system depends largely on the knowledge it possesses; therefore, the representation of that knowledge is critical to the design of the system [RICH83]. Facts are declarative statements, e.g. "Adenine is a DNA base." The remaining knowledge in a knowledge base is usually represented in one of three ways: rules, frames, or semantic nets. Rules are usually expressed as if-then-else structures, e.g. "If complimentary DNA bases are exposed to each other in hybridization, they will bond together." Frames associate a set of features or attributes with objects or concepts. Semantic nets are networks describing relationships between objects or concepts.

Rules are applied by the inference engine in one of two ways: forward chaining or backward chaining. In forward chaining, the inference engine starts with the given knowledge and reasons in a forward direction, comparing each inference with the goal. In backward chaining, the inference engine starts with the goal and proceeds backwards trying to satisfy those rules necessary to prove the goal.

Expert systems are valuable in that they represent a permanent store of expert knowledge. Also, they are generally portable, affordable, consistent, and predictable. They can, therefore, be used where human expertise is scarce or diminishing, or where expertise is needed in numerous locations or in undesirable environments [WATE86].

However, there are still many problems with expert systems. They lack common-sense reasoning and do not recognize the limits of their ability. They

do not lend themselves to temporal or spatial knowledge, and they may allow inconsistencies in knowledge. Expert systems lack creativity and adaptability; they have a narrow focus. Finally, because expert systems require correctly representing the heuristic knowledge of another person, they take a long time to build [WATE86].

Expert systems, then, are best suited to problems which possess certain characteristics: the domain should be well-defined and of manageable size and complexity; the nature of the problem should involve hueristic reasoning; and, finally, experts must be willing and able to contribute to the development process. The general classifications of expert systems are interpretation, prediction, diagnosis, design, planning, monitoring, debugging, repair, instruction, and control [WATE86].

2.2 Expert System Tools: PROLOG

The types of tools available for expert system development can be divided into four categories: programming languages, skeletal systems, general purpose representation languages, and other expert system building aids [HAYE83, WATE86]. Programming languages are considered to be oriented either towards numerical procedures, like conventional languages, or towards symbol manipulation, such as LISP or PROLOG. Skeletal systems are stripped-down versions of completed expert systems. General purpose representation languages are designed specifically for expert systems development, but are less flexible than programming languages. The final set of tools includes knowledge acquisition aids, explanation aids, knowledge base editors, and other facilities

useful for particular aspects of expert systems development.

PROLOG is a logic-based symbol manipulation language. It is based on predicate calculus, and this logic is used to structure a program and guide its execution: "A PROLOG program is a database of logical assertions (facts and rules)describing a general problem space. When queried about a specific problem, PROLOG systematically searches this database for a logical justification of the query, i.e. a solution." [LUGE85].

The form of a rule in PROLOG is: $X\ if\ Y_1\ and\ Y_2\ and\ ...\ and\ Y_n$. A query attempts to prove X by satisfying Y_1, Y_2, ... Y_n; i.e. via backward chaining. A program may describe more than one way of proving X, or more than one way of satisfying Y_i $(1 \leq i \leq n)$: "The program clauses for a relation are always tried in the order in which they appear in the sequence of clauses. PROLOG programmers often exploit this order of use to give ordinary rules for a relation first, followed by a default rule that should only be used if the ordinary rules have failed." [CLAR]. This built-in search strategy is usually referred to as backtracking with depth-first search [LUGE85].

In its attempt to satisfy a clause, PROLOG employs a powerful pattern matching mechanism called unification. The unification algorithm determines when clauses or clause arguments match, and resolution is allowed. Pattern matching provides appropriate variable bindings. With backtracking, all possible bindings for variables can be found. [LUGE85].

PROLOG can be a good tool for certain kinds of artificial intelligence work. Its backtracking with depth-first search and built-in unification and pattern matching lend PROLOG well to theorem provers, production (rule-

based) systems, and expert systems that require this kind of processing. Its syntax requires little time to learn and code, lending it to rapid prototyping. Finally, PROLOG provides a database for storing facts and rules.

Several of PROLOG's criticisms lie around its built-in control structure. Problems that do not use backtracking with depth-first search or unification and pattern-matching can be difficult to implement in PROLOG. PROLOG programs operate under a closed-world assumption, and contradictions in the database are allowed; both of these may yield unexpected results. PROLOG code, while efficient, is very dense and, therefore, is not easy to read. Finally, PROLOG's user interface can be cumbersome and tedious to use.

2.3 Expert Systems Work in Biotechnology

Within the last two decades, revolutionary advances have been made in molecular biology which have called for the development of computational systems and have encouraged the application of AI techniques to this area [FRIE]. Current applications of computers to biotechnology can be grouped into five categories: (1) data collection, assembly, storage, and retrieval, (2) primary structure determination of nucleic acids and proteins, (3) simulation of molecular processes, (4) experiment planning and debugging, and (5) three-dimensional physical structure generation for biological molecules [FRIE, NUCL82, NUCL84].

As part of the search and review of the literature for this project, a principal researcher, a developer, and an end user were each queried to learn of current computer applications in the general area of gene synthesis planning

[2, 3, 4]. Applications to date have approached the problem from several different angles. The PEP program from IntelliGenetics, Inc., suggests a total base sequence for a desired amino acid sequence for a protein. The SEQ program, also from IntelliGenetics, finds various types of patterns within a DNA sequence which are critical to the synthesis process. Genex Inc.'s SYNTH program performs chemical calculations to minimize the cost and materials required for synthesis [LOMB83].

The specific problem of selecting the optimal oligonucleotides of DNA to synthesize and determining the strategy for joining them has not been addressed by available commercial packages or in publicized academic work. IntelliGenetics does have plans to develop such a package in the future, although it appears that their approach emphasizes enzymology concerns only and not chemical synthesis concerns [5]. It can be speculated that private companies whose research includes synthetic genes have in-house systems that address this issue [6]; however, information on this is, of course, proprietary and thus not available.

Existing expert systems applications and methods were reviewed to learn of approaches which could be used for the proposed system. Since the expert's approach to solving the problem uses depth-first search with backtracking, this paradigm was chosen as the basic model. PROLOG provided a natural tool for the control mechanism. However, no existing systems were found to serve as specific models. Information on the selection of detailed approaches to implementing the system can be found in Chapters 4 and 5.

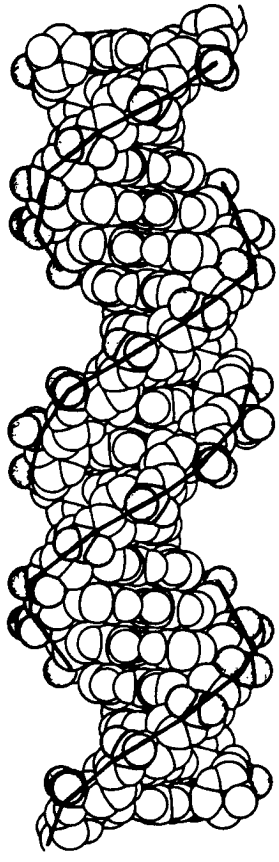
Chapter 3

Description of the Problem

3.1 General Background on DNA

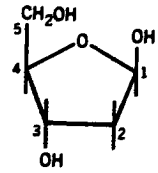
The most common structure of deoxyribonucleic acid (DNA) consists of two helical chains each coiled around the same axis, i.e. a double helix. Each chain is composed of sugar molecules joined to each other by phosphate groups. Attached to each sugar is a base. The bases from one chain form an electrostatic bond with the bases from the opposite chain. A segment of DNA can be modeled as given below in Figure 3.1.

Each of the chains in a segment of DNA can be oriented in space by the direction of the linkages between the sugars. A DNA sugar is depicted in Figure 3.2. The carbon atoms on the sugar are numbered from 1' through 5'. The linkage between two sugars runs from the 5' carbon of one sugar to the 3' carbon of the next. In a segment of DNA, the sugar at one end will have its 5' carbon free (unattached), and the sugar at the other end will have its



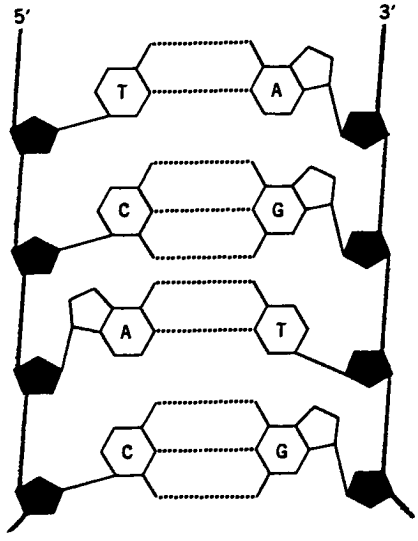
[LEWI85]

Figure 3.1: Model of a Segment of DNA



[LEWI85]

Figure 3.2: 2-deoxyribose (DNA sugar) with Numbered Carbon Atoms



[LEWI85]

Figure 3.3: An Uncoiled Segment of DNA

3' carbon free. The bases on one strand of DNA can be read in two different directions: (1) 5' to 3', where the first base listed has its 5' end carbon free and the last base listed has its 3' end carbon free; or (2) 3' to 5', where the first base listed has its 3' end carbon free and the last base has its 5'end carbon free. Each of the chains in a double helix runs in opposite directions. If a segment of DNA were uncoiled, it would appear as in Figure 3.3.

There are four kinds of bases in DNA: adenine, guanine, cytosine, and

thymine. These are abbreviated A, G, C, and T. A sequence of bases on a DNA chain is customarily read in the 5' to 3' direction. So the sequence for the segment in Figure 3.3 would be T C A C (for the chain on the left). It is this sequence of bases which represents the codes for genes, and it is genes which contain the information that enables living organisms to function and reproduce.

Each of the four bases has a complimentary base with which it bonds chemically: A pairs with T, and C pairs with G. Given a sequence of bases forming a single chain of DNA, the second chain in the double helix will consist of the sequence of complimentary bases. For example, the left chain in Figure 3.3 contains the base sequence T C A C, and the complimentary right chain contains the bases A G T G (in the 3' to 5' direction), and the chains base pair as shown in the diagram.

There are several types of patterns which occur commonly in DNA chains. For example, an area which contains a high concentration of G's and A's is said to be purine-rich. An area with mostly G's is said to be g-rich. Examples of both are given below:

Purine-rich:	A G G A C A G G G A A
G-rich:	G G G A G G G G

Another type of pattern is the repeat. This identifies a sequence of bases which occurs at more than one place within one piece of DNA. For example, the sequence C C G A T C may occur at two places in the same strand of DNA:

. . . C C G A T C . . . C C G A T C . . .

A more interesting pattern is the palindrome. In this case, a single DNA chain contains a sequence which, if folded back upon itself, would base pair with itself. Such a sequence is said to be self-complimentary. For example, the chain A G C C T G C A G G C T, if folded back upon itself, would base pair as follows:

A G C C T G
T C G G A C

3.2 Chemical Gene Synthesis Planning

Genes are very long sequences of DNA. Techniques have been developed to chemically synthesize genes which may contain rare or new base sequences. Given a desired gene, the biochemist first synthesizes a set of short single-stranded DNA segments called oligonucleotides, or oligos for short. The bases in each oligo represent a subset of the final sequence for one of the two chains. These segments are then joined together by enzymes to make the gene. Advances in synthesis techniques are enabling biochemists to synthesize longer oligos, but the overall strategy has not changed and is not expected to do so in the near future [6].

To synthesize a segment of DNA, the biochemist builds the oligo, one base at a time, on a silica or glass foundation. Starting at the 3' end, the first

base is attached to the silica or glass, the second is attached to the first, and so on. At the end of the oligo synthesis, the DNA strand is cleaved from the silica or glass support.

There are two methods of joining the synthesized oligos: (1) hybridization and ligation and (2) extension. Since hybridization and ligation requires less time and effort, it is the method of choice; however, it may not always be feasible due to patterns in the sequence of bases.

With hybridization and ligation, the full length of both DNA strands are synthesized in optimally chosen oligos. Then in a minimal number of steps, complimentary portions of opposing strands are joined in a process called hybridization. The enzyme ligase joins breaks in each chain, where one synthesized oligo stopped and another started. The entire process is depicted in Figure 3.4.

With extension, optimally selected oligos from alternating strands are synthesized. The 3' end of each oligo is an overlap area or "sticky end"; i.e., it must contain the bases that are the compliments to the bases at the 3' end of the next segment. These oligos are joined in sets of two at the sticky ends by hybridization. Next, the enzyme DNA polymerase is used to generate the complimentary portion of each single stranded area. Finally, ligase is used to join the double-stranded segments together. This process is depicted in Figure 3.5.

The selection of oligos to chemically synthesize is largely guided by two factors: patterns existing in the sequence of bases, and the method used to join the fragments. This latter factor is guided, in turn, by patterns in the

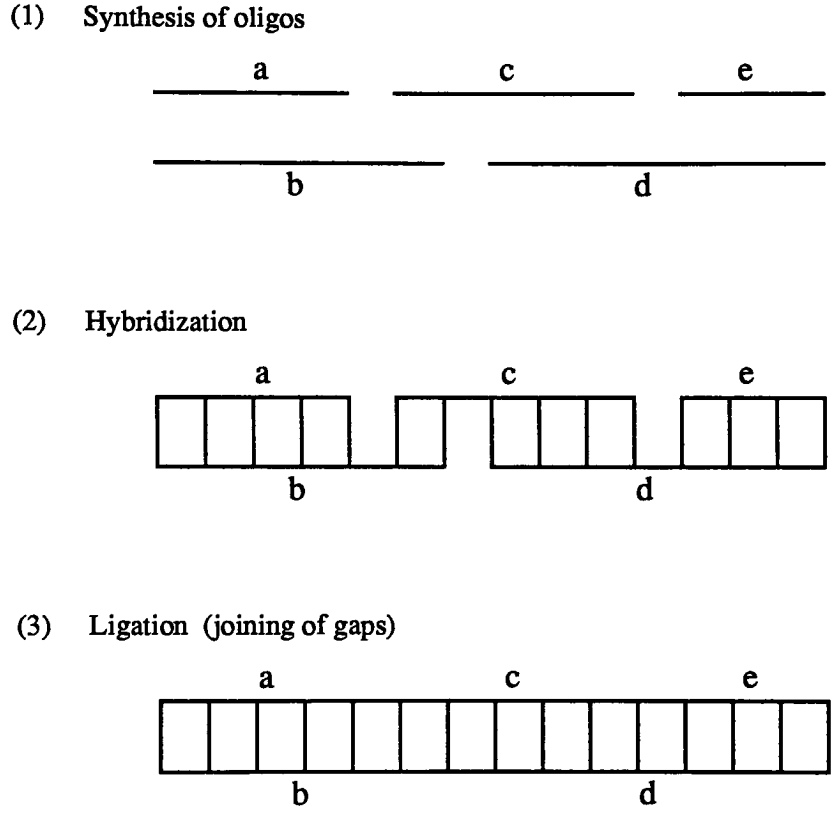
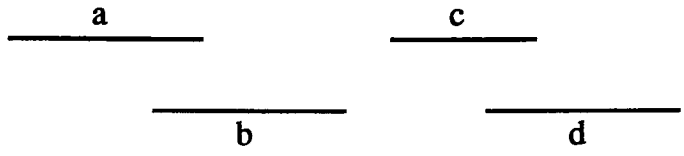
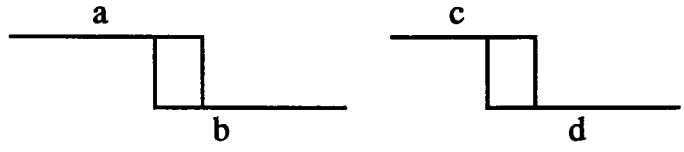


Figure 3.4: Gene Synthesis by Hybridization-Ligation

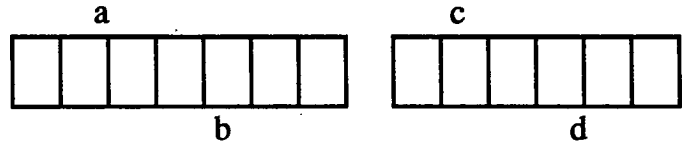
(1) Synthesis of oligos



(2) Hybridization



(3) Extension with DNA polymerase



(4) Ligation (joining of double-stranded segments)

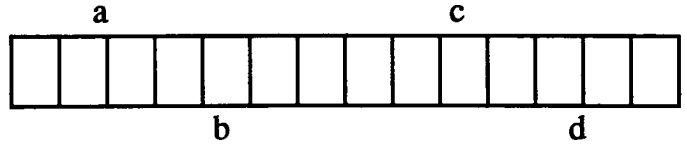


Figure 3.5: Gene Synthesis by Extension

base sequence.

Some of the rules of synthesis planning are general knowledge. For example, oligos should be about 50 bases long, and "there is the first basic requirement of the overlap of about five base pairs at the protruding ends" [KHOR79]. Other rules cover the avoidance of patterns at certain ends. For palindromes, "it is important from the standpoint of enzymatic work that complimentarity (self-structure) within the single-stranded segments as well as self-complementarity at the 5' protruding ends be avoided as far as possible" [KHOR79]. The reason for this is that self-complimentary sticky ends may base-pair with themselves, preventing hybridization with their intended complimentary oligos. Repeats must also be avoided at oligo ends: an oligo ending in the same sequence as another may base pair with the other's intended compliment instead of its own. The rules for purine avoidance are less well known. Purines should be avoided at the 3' end because they are more susceptible to depurination (falling off) during the acid wash that occurs after each base is added to the oligo. G-rich areas should be avoided anywhere since G's are particularly susceptible to depurination.

The basic rules can be summarized as follows: (1) oligos should be about 50 bases long; (2) there must be sufficient overlap between oligos for hybridization; and (3) patterns should not exist at certain oligo ends.

There are more complex rules covering what action to take to avoid patterns at the ends of an oligo. The proposed oligo of 50 bases in Figure 3.6 (a) would not be recommended, because it would leave part of a palindrome both at its end and at the start of the next oligo. The primary rules to avoid

situations such as this involve shortening or lengthening an oligo to get around the critical area. These two options are shown in Figure 3.6 (b) and (c). Should this strategy fail, less desirable alternatives can be pursued. For example, a palindrome may be split between two oligos if the maximum length of the strand on each oligo which could fold back on itself is shorter than the length of the sticky end (d). Also, if a palindrome contains a segment in the middle which is not self-complimentary, the oligo boundary may be placed here (e).

If an oligo boundary is changed, the neighboring and complimentary oligos may have to be adjusted as well. As shown in Figure 3.7 (a), shortening the oligo to avoid the palindrome may remove a necessary sticky end. If the complimentary oligo is now shortened, it must be checked to make sure it is not too short (b). If it is lengthened, it may now be too long or may interfere with the palindrome (c). The best approach may be to further adjust the original oligo, as shown in (d). Secondary rules handle this fine-tuning of oligo boundaries while maintaining the overall integrity of the solution. Many iterations of adjustments are possible.

In general, the problem is to determine an optimal selection of oligos to synthesize and join. The solution process can be described as (1) applying the basic rules to the gene to yield oligos of standard length and overlap, and (2) backtracking with the more advanced rules to adjust oligo boundaries as necessary around critical patterns.

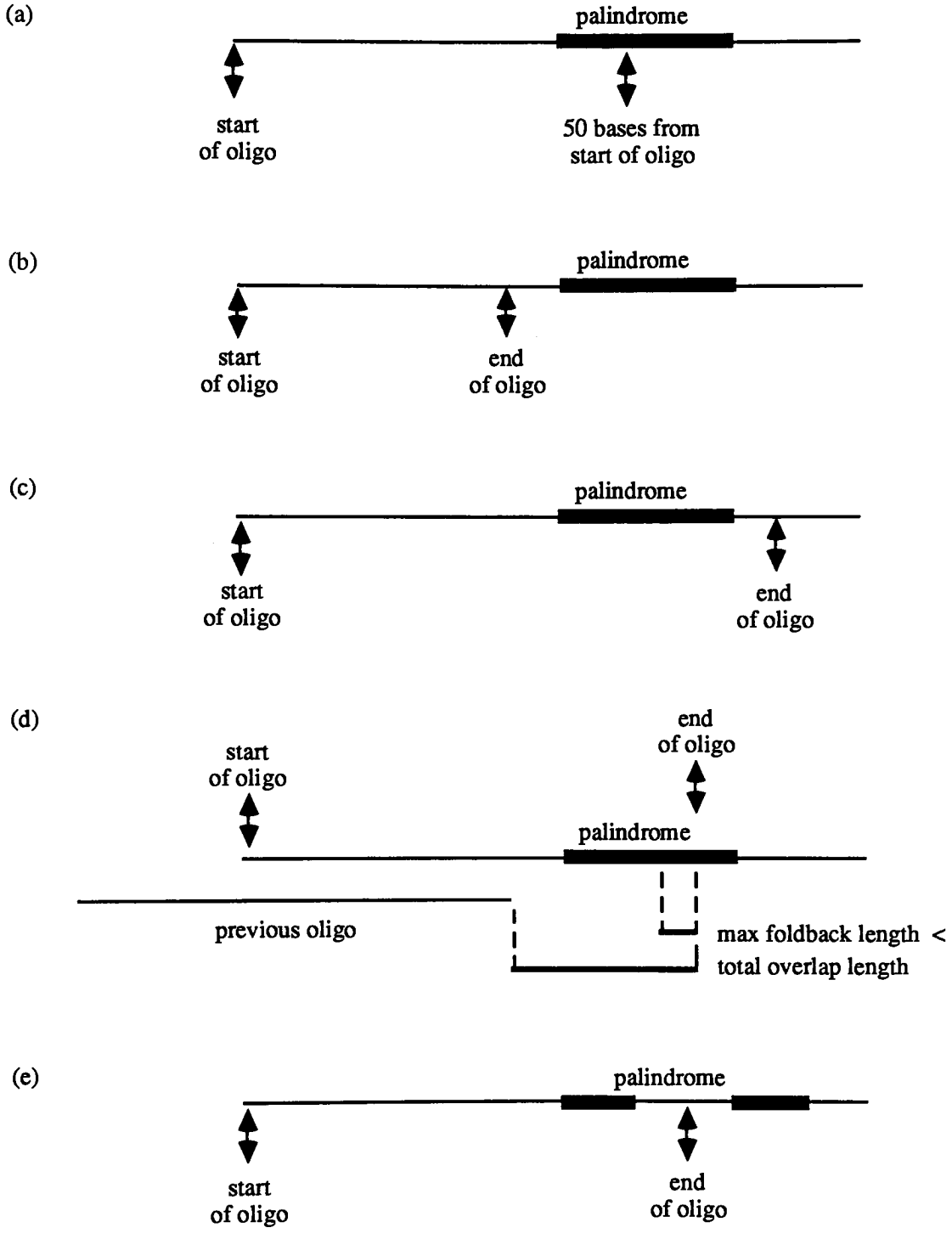


Figure 3.6: Palindrome Avoidance Strategies

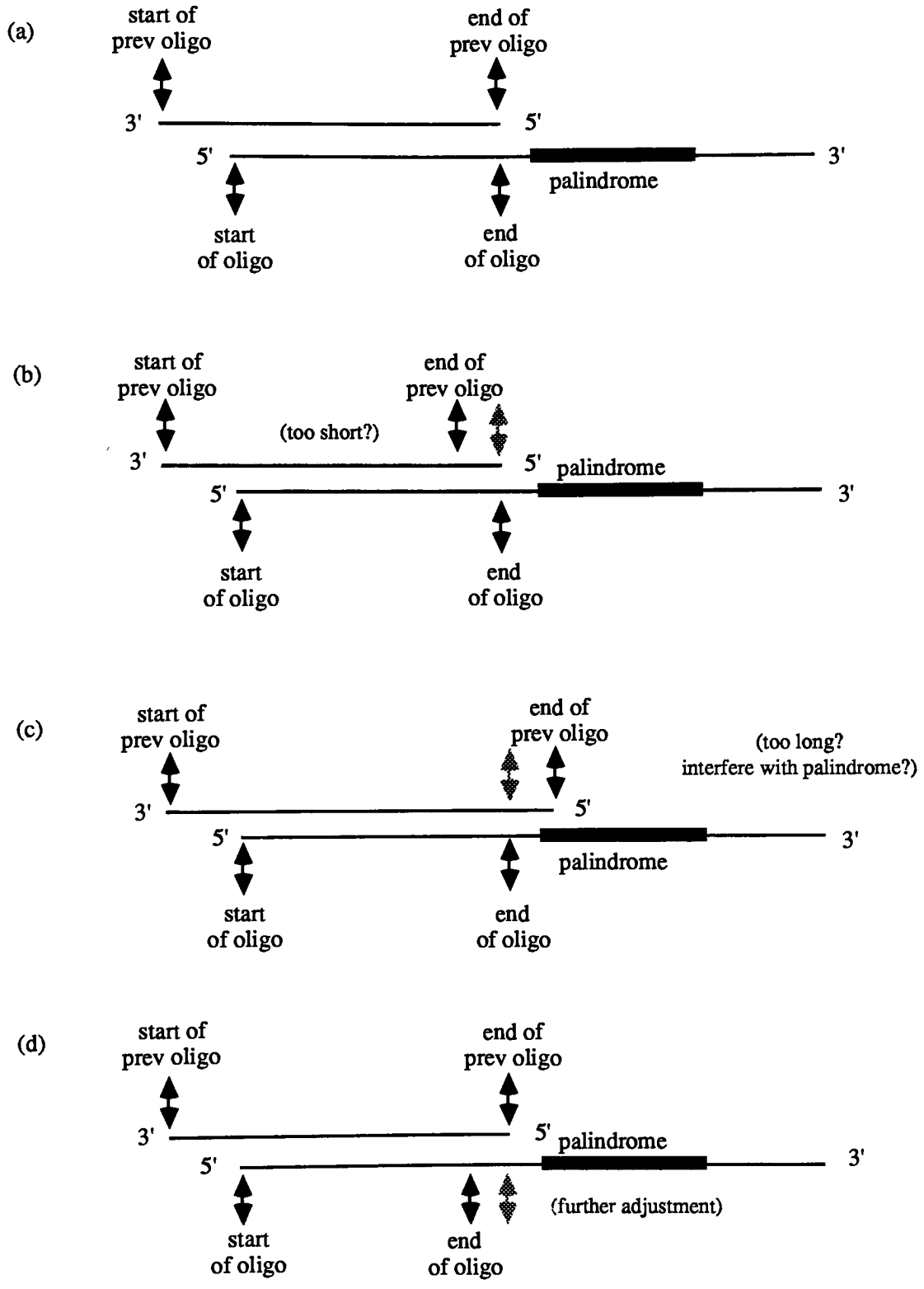


Figure 3.7: Further Adjustments to Avoid a Palindrome

Chapter 4

Approach to the Development of Geneplanner

4.1 Motivation

The main reason for pursuing this project was to go through the process of developing an expert system. This knowledge engineering process, as depicted in Figure 4.1, involves identification of the problem, conceptualization, formalization, implementation, and testing. All steps may be repeated until a production-quality system is achieved. For Geneplanner, the same basic steps have been followed: selecting an appropriate problem; refining the scope, acquiring knowledge, and selecting tools; selecting control mechanisms and knowledge representations; implementing; and testing. Implementation and testing were repeated as the system was refined, to the point of achieving a working prototype. Further refinement, implementation, and testing would be

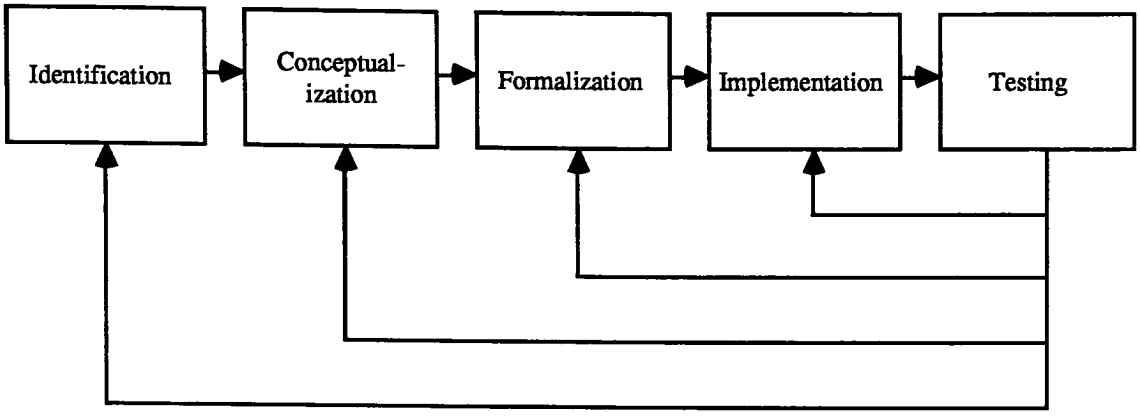


Figure 4.1: Steps in the Development of an Expert System

necessary to achieve a production-quality system.

4.2 Choosing an Appropriate Problem

Two domains were initially considered: law (in particular, conflicts of interest) and genetic engineering. Both contained problems potentially applicable to expert systems development, and both had experts available for consultation. The domain of genetic engineering was selected because the application area was of particular interest and there was an expert who was particularly interested and willing to help with the development of an expert system.

First, potential application areas were discussed with the expert. The expert was told what types of applications were well-suited to expert systems. The expert then presented several application possibilities. Each was reviewed to see if part or all of the problem would be an appropriate application for a

small expert system. Some possibilities were dismissed because of the problem size (too large, too small) or because the nature of the problem was not appropriate for an expert system (too algorithmic, too unknown). An example of an application that was eliminated was that of analyzing a gene for the presence of certain base sequence patterns that would indicate potential problems in the health of the carrier. This was eliminated because it appeared to be a relatively straight-forward algorithmic pattern-matching problem.

The particular problem of chemical DNA gene synthesis planning was ultimately selected as the application. It appeared well suited to a small expert system for several reasons. Fragment selection and joining strategy is a narrow, well defined, but non-trivial problem. The approach to solving the problem is hueristic in nature. Finally, the expert was particularly knowledgeable in this area.

4.3 The Expert

The expert for Geneplanner is Cheryl Heiner, a research biochemist with Applied Biosystems, Inc. of Foster City, California. Applied Biosystems manufactures automated DNA synthesizers. Ms. Heiner has been synthesizing genes and advising others in this work for several years.

Some aspects of working with the expert became apparent very early in the project. The expert was very enthusiastic about the sharing her knowledge of the application domain and of her process for solving the particular problem of gene synthesis planning. Also, the expert could easily verbalize her

knowledge in simple terms.

The main problem in working with the expert was recognized as the distance between her and the development site. She is located in California, while the development work was done in Rochester, New York. Several visits were planned and made to California to work on the project, and numerous phone calls were made to answer questions and check results, but most of the work was done without the expert's presence. This proved to be detrimental in two ways: (1) Some decisions were implemented before checking with the expert, and some small decisions were never checked with the expert; (2) Due to both the distance and the time lags, the expert was unable to maintain a continual interest in the workings of the system or a sense of ownership for the system. The phone conversations were not a sufficient substitute for hands-on experimentation. The experience with Geneplanner indicates that it is important for an expert to spend a significant amount of time with the developer during implementation of an expert system.

4.4 Tool Selection

C-PROLOG was chosen as the implementation language for the proposed project. First, a low-level tool such as PROLOG was appropriate due to the educational nature of the project. It provided exposure to all aspects of building a prototype system. Secondly, the problem inherently involved backtracking with depth-first search, and this is the inference mechanism automatically built into PROLOG. Also, the language's sophisticated pattern-matching and unification features also would lend themselves well to the pattern-dependent rules

in gene synthesis planning. Finally, PROLOG was available and known, so there would be no learning curve. In general, C-PROLOG was an appropriate tool for the task.

Chapter 5

The Implementation Process

5.1 Overview of the Implementation Process

The implementation process followed the general steps of expert systems development as described in Section 4.1: preliminary knowledge acquisition, tool selection, selection of control mechanisms and knowledge representations, implementation of the rules, testing, and refinement. As stated earlier, testing and refinement were conducted only through completion of a working prototype; significant further refinement would be necessary to achieve a production-quality system.

As much as possible, generic structured design and programming techniques were followed throughout the implementation. These included top-down design, modularization of detail design and code, hiding (as much as possible) implementation details, meaningful variable names, etc. While this ensured that the code could be modified without major difficulty, it did not

ensure that the initial strategies were correct.

The actual steps in the implementation of Geneplanner are discussed in detail in the following sections. These steps are as follows:

- Initial Knowledge Acquisition
- Determining High-Level Control Mechanisms and Knowledge Representations
- Development of High-level Rules
- Implementation of Pattern Information Storage and Retrieval Rules
- Implementation of Default Oligo Boundary Rules
- Implementation of Critical Pattern Check and Retry Initiation Rules
- Development of Retry Strategy and Low-Level Retry Rules
- Evolution of the User Interface
- Adding Explanation Facilities
- Adding the Complexity Check
- Testing and Refinement

5.2 Initial Knowledge Acquisition

The first step in knowledge acquisition was to become familiar with the terms and concepts in the application domain. This was achieved by reading an introductory book recommended by the expert [ROSE83]. The material was reviewed with the expert so that a common frame of reference was established.

Next, sessions were conducted with the expert in-person in which she described the particular problem of chemical gene synthesis and the way to solve this problem. This was done verbally and with pictures. Many details

were presented by the expert over a short period of time. The intensity of these sessions was partially due to their limited duration. These discussions included relevant details as well as details on parts of the problem which would sooner or later be excluded from the scope. All information nevertheless provided good background material.

After initial exposure to the problem, the effort focused on structuring the expert's knowledge, i.e. developing rules. This involved clarifying the problem and the solution with the expert. Most of the interaction took place over the telephone. Clarifying questions were answered and possible scenarios were discussed step by step. Questions were of the sort "What if...?", "What would you do next?", and "Why?". This process resulted in restructuring the model of expert's knowledge, i.e. prioritizing the rules. The scope was cut back and refined: rules were simplified, and some exception rules were excluded. The final result of this step was a high-level model of expert's knowledge that would be implementable in a prototype expert system.

During this phase, the expert received her first impression that Geneplanner would not be as robust as initially hoped. This dampened her enthusiasm temporarily. After reviewing the purpose of the project, though, progress continued enthusiastically.

Knowledge acquisition continued throughout the implementation process. Interactions with the expert when refining control mechanisms, implementing rules, and reviewing test cases resulted in further learning of the chemical DNA gene synthesis planning process.

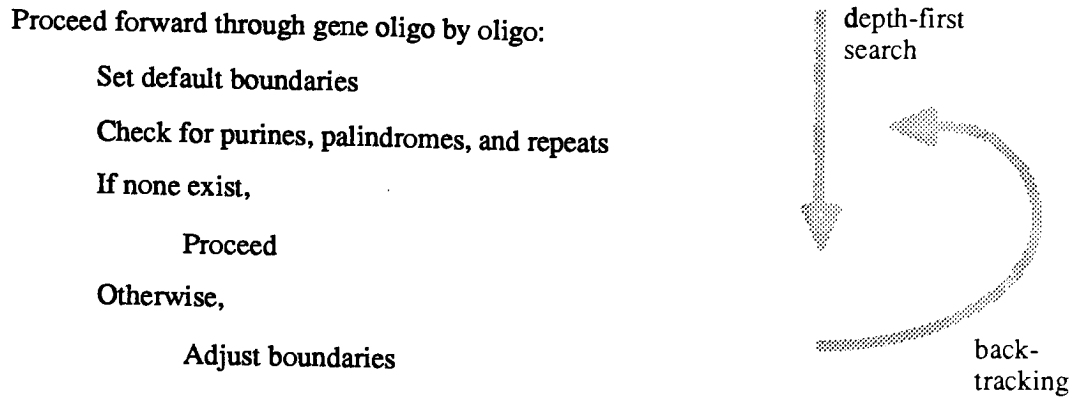


Figure 5.1: Depth-First Search and Backtracking in Geneplanner's General Strategy

5.3 Determining High-Level Control Mechanisms and Knowledge Representations

The general strategy in designing Geneplanner was to solve a gene planning problem in the same way as the expert, as described by the expert. That process is described, at a high level, as follows: Starting at the beginning of the gene, proceed forward oligo by oligo. For each oligo: (1) set default boundaries and (2) check for purines, palindromes, repeats. If any of these patterns are found in critical areas, adjust the oligo boundaries as necessary. The strategy of using default boundaries first and then adjusting as necessary fit neatly into PROLOG's depth-first search with backtracking, as in Figure 5.1.

The actual implementation contained both PROLOG's built-in backtracking and recursion capabilities to control adjusting boundaries. Adjustment of oligo boundaries was implemented by retracting the old boundaries, asserting new boundaries, and then checking these new boundaries for specific

criteria. If the criteria checks failed, fine tuning adjustments would be made by recursively calling the routine to retract the previous boundaries, assert the newer boundaries, and then check these new boundaries for the same criteria. For example, if a new boundary which was proposed to avoid a purine failed the sufficient-overlap check, then recursion would be initiated to propose another new oligo. This could continue for many levels of recursion until a satisfactory oligo is found or, theoretically, all alternatives have been exhausted.

The knowledge representation for patterns and oligos went through several changes throughout the implementation process. The basic information needed to represent patterns in the knowledge base was pattern type (purine, g-rich, palindrome, or repeat), chain (5'- 3' or 3'- 5'), starting position (base number), and ending position (base number). Two pieces of data were added due to their use in the basic pattern-checking rules: concentration and length. Concentration is only necessary for purines and g-rich patterns; it is set to a default value of 100% for other patterns. Length is only necessary for palindromes: it represents the length of one of the two complimentary parts of the pattern. For example, the palindrome A G A C C T G T C T, with complimentary portions A G A C and G T C T, would have a length of 4 (the intervening C T would form a hairpin loop in the middle of the palindrome). By definition, this length must be at most half the distance between the start and the end of the palindrome. For all other patterns, this piece of data was calculated by the system as the length between the starting and ending positions. The final structure to represent patterns is shown in PROLOG format in Figure 5.2. Two examples are also given. As indicated in the examples, the chain direction was implemented using a simple 53 or 35 to represent 5'- 3' or

Final structure: (pattern (Type,
Chain,
Starting position,
Ending position,
Length,
Concentration))

Examples: (pattern (purine, 53, 105, 130, 26, 85))
(pattern (palindrome, 35, 220, 240, 8, 100))

Figure 5.2: Knowledge Representation for Patterns

3'- 5', respectively. This was done because PROLOG does not accept the 5'- 3' format.

For oligos, the first decision was whether to represent them using an argument list, passed throughout the program, or a database structure. An argument list would provide automatic deletion of elements when backtracking occurred; however, it is more resource intensive and prohibits keeping rejected fragments for reference. For these last two reasons, the final decision called for storing oligos in the knowledge base, keeping track of good oligos as well as the rejected ones. Storing rejected oligos proved beneficial not only for tracing and debugging, but also for preventing the proposal of a previously rejected oligo.

The basic information needed to represent oligos in the knowledge base was status (good or rejected/no-good, the latter signified by "ng"), chain orientation (5'- 3' or 3'- 5'), starting position(base number of the first base in the

Final structure: (frag (good/ng,
Orientation,
Starting position,
Ending position,
Join step,
Reason))

Examples: (frag (good, 35, 131, 160, 2, pal1))
(frag (ng, 53, 51, 100, 1, arb))

Figure 5.3: Knowledge Representation for Oligos

oligo), and ending position (base number of the last base in the oligo). Join step was added for those oligos to be joined in multiple steps under extension. For hybridization-ligation, the join step will always be 1 unless multiple steps are needed due to repeats. Oligo length was also added for completeness; however, it was later deleted since it was never used. A reason code was added later during implementation to support the explanation facilities. The code pointed to an explanation as to how the starting and ending positions were determined. Two other parameters representing the criticality of the starting and ending positions were added to assist with retry adjustments. These, however, were soon deleted. The reasons behind this are discussed in Section 5.8. The final structure to represent oligos is shown in PROLOG format in Figure 5.3. Two examples are also given.

5.4 Development of High-Level Rules

The next step in the implementation was to completely document and verify the high-level rules. These rules were written by the developer and reviewed and modified by the expert. They were written in structured English in a way that the expert could understand and the developer could translate into code with relative ease. As an example, Figure 5.4 gives the high-level rules for purine checks and adjustments. The initial set of rules contained the high-level control strategy and details for the critical pattern checks. The adjustment and retry rules were detailed later in the implementation. The full set of high-level rules is given in Appendix A.

The knowledge engineer had to do a fair amount of translating and processing to bridge the gap between the expert's world and the formalisms and structures of the systems development world. The problem was accentuated by the distance between the expert and the knowledge engineer. Most communication here was verbal (via mail and telephone); graphical expressions would have been preferable.

The expert wanted system to work perfectly from the point of view of an actual user in a production environment. She always gave her rules from this point of view. By putting Geneplanner's rules on paper, the actual scope of the system sank into the expert as well as the developer. This diminished the hopes of both a bit. However, after reviewing of the purpose of the project once again, a substantial amount of the enthusiasm was regained and progress continued.

Purine checks

If length of fragment > 50,
then check for purine patterns within 20 bases of 3' end;
if first 20 bases are > 75% purine or
first 10 bases are > 60% purine,
then adjust for purines.

If length of fragment is > 30,
then if fragment is > 30% G's
then adjust for purines.

If we've passed the checks to this point,
then fragment passes purine checks.

Purine adjustments

If fragment is on the 3' to 5' strand,
1st: retry with fragment boundary beginning at the end of the purine area (i.e. extend the previous fragment).
2nd: retry with fragment boundary ending at the end of the purine area and starting further back (i.e. shorten the previous fragment).

If fragment is on the 5' to 3' strand,
1st: retry with fragment boundary ending right before the start of the purine area (i.e. shorten the fragment).
2nd: retry with fragment boundary beginning at the start of the purine area and extending further out (i.e. extend the previous fragment).

Figure 5.4: High-Level Rules for Purine Checks and Adjustments

For all patterns:	pattern type starting position ending position
Additional information for purines:	chain concentration
Additional information for palindromes:	foldback length

Figure 5.5: Pattern Information Provided by the User

5.5 Implementation of Pattern Information Storage and Retrieval Rules

In the problem selection and knowledge acquisition phases of Geneplanner, it was decided that the system would not include the actual pattern search processing to determine the location and other data for purines, g-rich areas, palindromes, and repeats. This is a problem related to but distinct from the determination of oligo boundaries, and it is addressed by existing systems, such as the SEQ program mentioned in Section 2.3. Geneplanner receives its pattern information from the user and stores it in the knowledge base. The information needed for each pattern type is presented in Figure 5.5.

The implementation of the rules for acquiring, storing, and retrieving pattern information underwent several changes during the development of Geneplanner. The original design implemented the expert's approach directly by asking the user for pattern information every time a specific region needed to be examined. For example, the user would be asked for any patterns in the

first 20 bases of the 3' end during the checks for purines/g-rich areas and then for any patterns in the first and last 10 bases during the checks for palindromes and repeats. Clearly, it was necessary only to ask for additional patterns at the 5' end during the palindrome and repeat check. As it was, the system appeared to have no memory of patterns input. If a previously checked area had to be re-checked during a retry adjustment, the user would be asked again for this information. This procedure involved too many questions, and it was therefore abandoned.

In the second approach, the user was asked for all patterns at the beginning of the session. This reduced the number of questions for the user substantially. Once the patterns were known, the system went into a black-box mode while it determined the solution. This prevented the user from following the development of the solution. It also deviated substantially from the expert's approach. For these two reasons, it was ultimately abandoned in favor of a modified version of the first approach.

The third and final method queried the user for specific patterns in specific regions of the gene, but only if the user had not already been asked for this information. This method required keeping track of what the user had been asked in addition to the pattern existence information. As could be expected, this still resulted in a lot of questions for the user. To reduce the questions, a feature was added to expand the area under question beyond the 10 or 20 bases of concern. Initially, the system asked for patterns in an area 10 bases longer on each end than the area of immediate concern. For example, if bases 150 to 170 were being searched for purines, the system would ask the user for purines anywhere between bases 140 and 180. The result approximated

asking the user for patterns in all areas of the gene, segment by segment. The expert was consulted, and she indicated that she looked only at the 10 or 20 bases of immediate concern. A compromise was finally implemented. Geneplanner asks for patterns in an area 3 bases wider on each end than the area of immediate concern. The figure of three bases was chosen because this is the amount by which oligos are adjusted to achieve minimum overlap.

In retrospect, the second approach is preferable to the current implementation, since it asks the user by far the least number of questions. A more sophisticated explanation system could be implemented to provide the inquisitive user with an explanation of how the solution was determined.

5.6 Implementation of Default Oligo Boundary Rules

The next rules implemented were those for setting default oligo boundaries. These are the rules for determining where the next oligo should start and end and for asserting the appropriate structure in the knowledge base. This involved a basically straight forward translation of the expert's rules into PROLOG. The strategy for setting the boundaries is dependent on the join method used. For hybridization-ligation, the set of oligos synthesized must cover the entire gene, and there must be overlap at both ends of each oligo. For extension, there must be overlap between the oligos within a step, but no overlap between steps. These two strategies are shown in Figure 5.6.

The expert's rules called for oligos of 50 bases and overlaps of 10 bases,

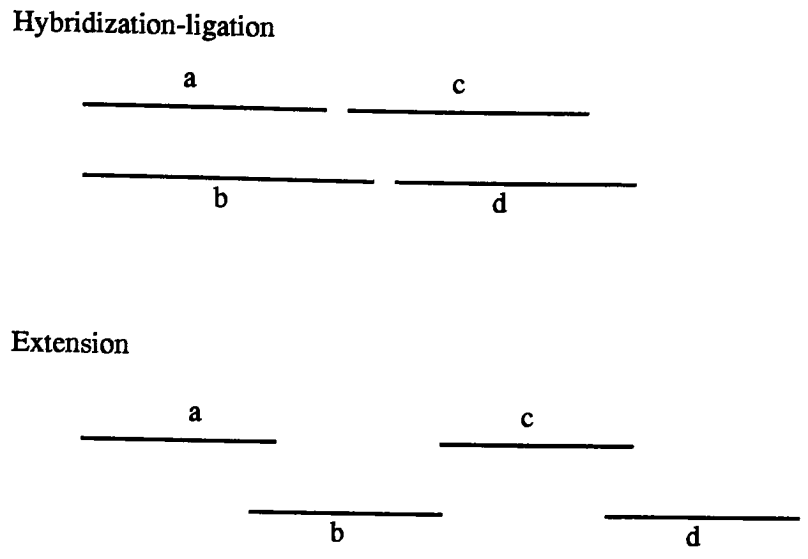


Figure 5.6: Overlap Needs for Hybridization-Ligation and Extension

if possible. In the initial implementation, if a length of 50 was found to provide insufficient overlap, the system tried extending the oligo by one base at a time. This yielded successive attempts until the oligo was 10 bases longer than its compliment. For example, if Oligo 1 ended at base 50, the attempted ending positions for Oligo 2 were: 50 (rejected), 51 (rejected), ..., 59 (rejected), and 60 (accepted). This was soon made more sophisticated by immediately setting the oligo boundary to 10 bases short of or beyond the end of its compliment.

Later a major enhancement was added to these rules to look first for any new boundaries proposed during retry adjustments. These new adjustments had to override the default boundaries. This was easy to accomplish in PROLOG, by placing these new rules ahead of the default rules.

Rules:

If length of fragment is > 30,
then if fragment is > 30% G's
then do purine retry.

If we've passed the checks to this point,
then fragment passes purine checks.

Implementation:

```
do_purine_checks (Chain, Start, End, Length):-
        Length > 30,
        check_pattern (g_rich, Chain, Start, End, 30, entire),
        !, fail.

do_purine_checks (_, _, _, _).
        /* succeed */
```

Figure 5.7: High-Level Rule for G-Rich Pattern Avoidance and its Implementation

5.7 Implementation of Critical Pattern Check and Retry Initiation Rules

This step covers the implementation of those rules which check the critical areas of an oligo for specific patterns. This also includes those rules that determine the initial adjustment of boundaries should a pattern be found. These rules were the easiest to implement and no major changes were ever made. These are the rules at the heart of the synthesis problem. The expert spent most of her time on this part, and the developer spent a good deal of time understanding this process and verifying the rules. An example of a high-level rule and its straight-forward implementation are given in Figure 5.7.

In incorporating these rules into the control process, the program had to follow one of two disjoint paths depending on whether or not patterns were found in a critical region. This was done by a set of rules for each pattern-check of the following nature:

Check for patterns of type x

If none exist, continue ...
    Check for patterns of type y

If checks failed,
    Reject the oligo
    Determine new boundaries
    Start checks for new oligo

One problem with this design and implementation was that critical patterns were located twice: once during the initial checks to see if any critical patterns existed, and once more in the adjustment logic to determine where to place the new boundaries. A possible enhancement would be to mark a pattern discovered during the first check as the "key" pattern, and then retrieve this key pattern directly when choosing the new boundaries. This was not implemented because the prototype's performance was sufficient under the original design.

5.8 Development of Retry Strategy and Low-Level Retry Rules

A fair amount of design effort was put into the logic for adjusting oligo boundaries, checking these adjustments, and then proceeding with the new bound-

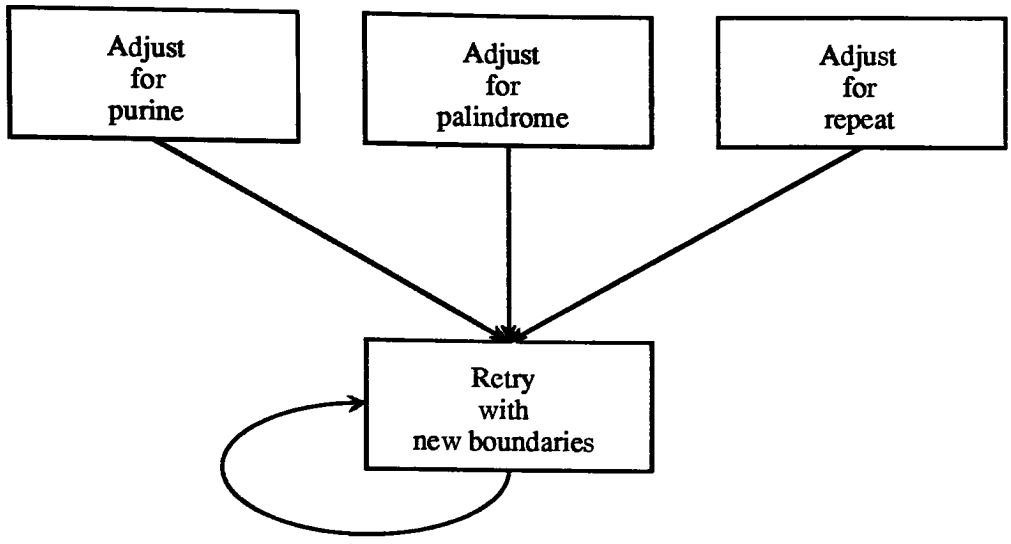


Figure 5.8: Common Retry Strategy

aries. This was done largely by the knowledge engineer, with occasional checks by the expert. The basic strategy was to have the rules which discovered purines, palindromes, or repeats in critical areas all invoke a common rule with one suggested adjustment to the oligo. This design is depicted in Figure 5.8. The common rule would set up the actual adjustment of the oligo boundaries, perform low-level checks on these new proposed boundaries, fine-tune the adjustment if necessary, and tell the system to continue on from this new point. The overall control mechanism for the common retry rule is described in Figure 5.9.

The difficulty in this strategy lay in determining how to adjust the oligo boundaries possibly several times and then tell the system to continue on from this point. PROLOG's backtracking and recursion were useful for this, although some effort was required to shape the problem into a form easily executed in PROLOG. If a fine-tuning adjustment is necessary, the com-

Retract all fragments whose boundaries are changing
Assert new proposed boundaries
For each new proposed fragment:
    Check for minimum length
    Check for maximum length
    Check for sufficient overlap with complementary oligos

If any check fails, fine-tune the boundaries and try again.

Figure 5.9: Control Mechanism for Retry Logic

mon retry rule is called recursively with the modification. If the adjustment fails, backtracking points to the next alternative fine-tuning modification. If all fine-tuning adjustments fail, back-tracking takes the system to the point immediately after the pattern was found and the first retry adjustment was attempted. The next alternative adjustment, if one exists, is then attempted. When a successful oligo plan is finally achieved, the system pops out of the (possibly recursive) retry logic and proceeds immediately to the main rule. The next action here is to determine the boundaries for the next fragment. At this point, the most recently proposed new boundaries are retrieved, and the flow continues as normal.

Different versions of the basic retry control mechanism were needed for each possible adjustment scenario. The criteria for each scenario are join method (hybridization-ligation vs. extension), where the adjustment is being made (the start vs. the end of the oligo), and the presence or absence of neighboring or complimentary oligos. For each case, the steps in Figure 5.9 were defined and implemented.

Final structure: (next_boundary (good/ng/used,
Orientation,
Starting position,
Ending position,
Join step,
Reason))

Examples: (next_boundary (good, 35, 131, 160, 2, pal1))

Figure 5.10: Knowledge Representation for Proposed New Boundaries

5.8.1 Proposed New Boundary Knowledge Representation

The retry logic introduced a new structure in the knowledge base, the proposed new boundary. This representation is similar to the oligo structure. The basic information is status (good, ng, or used), chain orientation (5'- 3' or 3'- 5'), starting position, ending position, and join step. A status of used indicates that a proposed boundary has been used to establish a new fragment; this prevents the proposed information from being used more than once. As with the oligo structure, parameters were added and then deleted for the criticality of the starting and ending positions. Also, a reason code parameter was added for the explanation system. The final knowledge representation and an example are shown in PROLOG format in Figure 5.10.

Check for maximum length:

If proposed oligo is $>$ 90 bases,
then adjust for maximum length.

If proposed oligo is $\leq$ 90 bases,
then proceed.

Adjust for maximum length:

If proposed oligo was adjusted at its beginning,
1st: move the ending boundary of this oligo back to 90 bases from the start
2nd: move the starting boundary of this oligo to 90 bases back from the end

If proposed oligo was adjusted at its end,
1st: move the starting boundary of this oligo to 90 bases back from the end
2nd: move the ending boundary of this oligo back to 90 bases from the start

Figure 5.11: Low-level Rule for Maximum Length

5.8.2 Implementing the Low-Level Checks

The low-level checks and fine-tuning adjustments were developed in a manner similar to the pattern check and adjustment rules. English-like rules were developed and reviewed with the expert and then implemented using PROLOG's depth-first search with backtracking. The low-level checks are for a minimum length of 20 bases, a maximum length of 90 bases, a minimum overlap of 3 bases if an overlap is required (always required under hybridization-ligation, only required within the same join step under extension), and a separation of join steps if the join method is extension. Figure 5.11 shows an example of a low-level rule.

In coding these rules, the need for lower-level support rules became apparent. These included simple rules for operating upon knowledge base structures and more complex rules such as determining the relative position of an oligo in an extension join step. It later proved advantageous to have the simple rules which operated upon the knowledge base directly: when the knowledge base structures underwent field-level changes, it was usually necessary to modify only these simple rules.

Two of the low-level retry rules underwent changes due to changes in the expert's rules. During initial knowledge acquisition, the expert had specified a minimum oligo length of 10 and a maximum length of 120. These rules remained in place through initial implementation. However, during a later review, the expert specified a minimum length of 20 and a maximum length of 90. The expert felt that these revised limits were more appropriate for the system, given its approach and its level of sophistication. The system was changed to reflect these new rules.

5.8.3 Problems and Attempts at Resolution

In general, the retry adjustment design works well, although it required a fair amount of debugging. Two problems became apparent during testing. First, fine-tuning adjustments occasionally moved oligo boundaries back into critical pattern areas. Second, after a low-level check caused a fine-tuning adjustment, the remaining retry checks on the originally proposed boundaries would proceed. Each of these problems and the attempts at their resolution are discussed below.

To attack the problem of fine-tuning a boundary back into a critical pattern area, parameters referred to as criticality factors were added to the structures for oligos and proposed boundaries. For example, if the starting position of an otherwise standard oligo was adjusted to avoid a purine, the oligo structure would appear as follows:

(frag (... , start-is-critical, end-is-arbitrary, ...))

Now if the end needed to be adjusted to avoid a palindrome, the new oligo would appear:

(next_boundary (... , start-is-critical, end-is-critical, ...))

With this new structure, fine-tuning into the critical pattern areas is prevented. The problem with this solution is that no fine-tuning is allowed at all. One answer to this is to incorporate various levels of criticality. Each level would indicate the extent to which a boundary could be adjusted and, possibly, the direction it should be moved. The final solution, implemented in Geneplanner, is to inform the low-level checks which end of the oligo has just been changed; fine-tuning adjustments are then ordered in such a way as to minimize further modifications in this area. While this is not as sophisticated as the multi-level criticality factor approach, test results showed it to be sufficient for the prototype system.

The second problem involved the continuation of low-level checks on a rejected proposed boundary. This is caused by the continuation of normal

processing after returning from a recursive call to the common retry rule. For example, given the proposed fragment below sent to the common retry rule from the checks for purines:

(... , 3' 5', start = 150, new-end = 220, ...)

If the complimentary oligo stops at base 219, an adjustment will be necessary to provide sufficient overlap. So a newer fragment is proposed with the following boundaries:

(... , 3' 5', start = 150, new-end = 222, ...)

After this new fragment passes all the low-level checks, processing will resume with checks on the original proposed boundary of 220. The problem becomes acute if a further low-level check on the 220 boundary proposes yet another new fragment covering this same area.

To attack this problem, continuation indicators were incorporated into the sequential low-level checks as described in Figure 5.12. If the check for maximum length caused an adjustment, continuation indicator C3 would be set to skip the subsequent overlap check. However, this proved to be an incorrect approach. Most fine-tuning adjustments are made at the opposite end of the oligo from the original pattern-avoidance modifications. Therefore, after a fine-tuning adjustment has been made at one end of a proposed oligo, low-level checks should continue at its other end. An example of this is displayed

Retry logic:

. . .
check_min_length (. . , C1, C2, . .),
check_max_length (. . , C2, C3, . .),
check_overlap (. . , C3, C4, . .),
. . .

For check_max_length:

Proceed only if C2 is set to "continue"
If max length checks pass, then set C3 to "continue"
If max length checks fail and adjustment done,
then set C3 to "do not continue"

Figure 5.12: Continuation Indicators

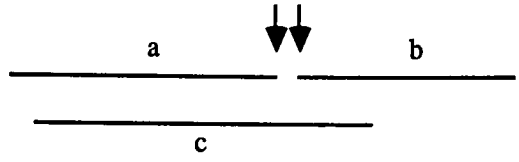
in Figure 5.13. The final solution was to order the low-level checks in a way that minimizes the impact of possible fine-tuning adjustments. Test results show this approach to work in most cases.

5.9 Evolution of the User Interface

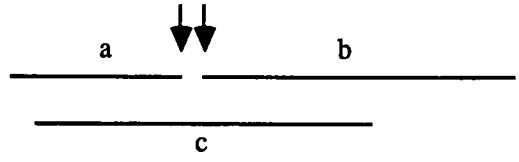
Geneplanner's user interface can be divided into two parts: the question/answer interactions and the presentation of final results to the user. Both of these aspects evolved during the development of the system. The basic format for input and output of information was determined early in the formalization phase. However, features to make the user interface more flexible were not added until later in the implementation phase.

User-tolerant features should be present in any expert system. For example, a user should be able to recover gracefully from erroneous input.

Original boundaries:

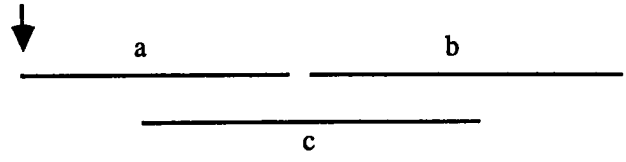


Proposed new boundaries:



1. check [a] for min length (will fail)
2. check [b] for max length
3. check [a] & [b] for overlap with [c]

When [a] fails the check for minimum length, a further adjustment changes its starting position:



Low-level checks 2 and 3 should continue.

Figure 5.13: Shortcomings of Continuation Indicators

Write the question
Read the user's input
Process the input as follows:

    If the user entered "exit", "quit", or "end", then quit.

    If the user entered "why", then output reason and
        invoke the generic routine again.

    If the user entered data, audit this data.
        If the data is valid, return it's translated value
            as the response.
        If the data is invalid, output the question's error message,
            read the new input, and process this new input.

Figure 5.14: Framework for Questions

He/she should also be able to abbreviate input or exit from any point in a session. To implement these features in Geneplanner, a standard question/answer framework was established. A common rule was developed for processing all questions. This rule takes as its input a question number and any necessary parameters and returns the user's response, validated and optionally translated. The logic for this common rule is shown below in Figure 5.14.

All questions in Geneplanner utilize this framework. For each, the following information is established:

- rule to output the question
- rule to output a response to "why"
- rule(s) to validate and translate the input
- rule to output an error message

Since most of the questions are yes/no questions, one set of default validate/translate rules was established to handle these. Appendix B.2 contains examples of some of the features described here.

Geneplanner outputs the final list of good fragments to the user at the end of each consultation. This list, however, is presented in an order which is the reverse of the order in which the user was queried for patterns and the oligos were planned. This is because as oligos are planned, they are added to the top of the knowledge base, and when they are retrieved, the top oligo is retrieved first. In this way it is efficient for PROLOG to access the most recently planned oligo(s) as needed. The list is frequently not in a perfect descending order because of the recursive adjustments possible in the retry logic.

Consideration was given to sorting the final set of oligos before output. This was not implemented, though, because the existing format was acceptable to the expert, and because it would have required either tabling and sorting the fragments or developing a sorting strategy which would not require tabling. Tabling and sorting was ruled out due to its large resource requirement, and alternative sorting strategies fell outside the scope of this project. For a production-quality system, sorting the results should be investigated and implemented. A possible approach for this would be to output the results to an external file and sort this file for the user.

Further enhancement of the user interface was limited because of C-PROLOG's weak support in this area. In a production quality system, it would be preferable to have a user interface which, minimally, does not require

the user to input a period at the end of each input term and, optimally, uses menus and graphics for all user interfacing.

5.10 Adding Explanation Facilities

The explanation system in Geneplanner was added after the design and early implementation. Whenever an oligo is asserted, the system asserts along with it a reason as to how the boundaries were set. The reason applies to either the starting or the ending boundary – whichever boundary was determined last. For oligo boundaries set by the default length of 50 bases, the reason code is "arb". For oligos whose boundaries were proposed during retry adjustments, the reason for the adjustment is asserted with the proposed boundary. A minor problem exists with this strategy: if an oligo is adjusted first to avoid a purine and again for minimum length, the reason from the last adjustment is the one recorded, i.e. the adjustment for minimum length. Reasons are stored in code form for conservation of space. A table of code translations is consulted during the output of reasons to the user. Figure 5.15 shows examples of reason codes and their translations.

This explanation system is fairly minimal in scope. It could be enhanced by tracing the adjustments and storing more data at each step. This information could then be presented for specific oligos or portions of the gene, as requested by the user.

An additional explanation feature was attempted but abandoned. As the explanation system currently works, reasons for final fragments are pre-

Code	Translation
arb1	Close to the end of the gene
arb2	Default fragment length
arb3	Default fragment length adjusted for overlap
pal1	Back up to avoid palindrome near start of proposed fragment
pur2	Move 3' end of proposed fragment forward to avoid purine
min1	Extend end of proposed fragment to make fragment at least 20 bases long
over2	Extend end of proposed fragment to assure overlap with complimentary fragment

Figure 5.15: Sample Reason Codes and Explanations

sented to the user upon request. Afterwards, the system asks, "Are these results okay?" The initial intent was to allow the user to respond "no", which would result in a request for user comments. These comments would then be stored in a file for the knowledge engineer. This strategy was abandoned because no method could be found to input continuous text into PROLOG from the terminal.

5.11 Adding the Complexity Check

The last feature added to Geneplanner was a self-check facility to prevent the system from aborting while trying to solve a problem beyond its scope. This happens when an area on a gene contains several patterns close to each

other. Geneplanner iterates infinitely while attempting to fine-tune around the problem area. In order to prevent this, a self-check was implemented. The systems checks each pattern input by the user: if the pattern is within 20 bases of another pattern or the start or end of the gene, the system will not accept it. The user is given the option of bypassing that pattern or ending the session. In a production-quality system, such a check would be used to catch only those extreme pattern situations which would signal special action for an expert chemical gene synthesizer, such as a large area with extremely high G concentration or many consecutive repeats.

5.12 Testing and Refinement

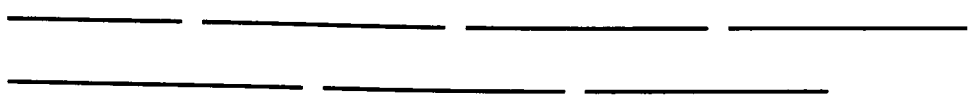
Testing in Geneplanner began as soon as the first set of rules was coded. It proceeded throughout development in a modular fashion. Each set of rules was first tested to remove bugs and ensure initial compliance with the expert's rules. This was followed by integrated testing and ultimately system testing. The testing plan called for the expert to utilize remote access facilities for testing and verification. Although instructions were given several times and the existence of the necessary equipment was verified, the expert never implemented this plan. It appeared that the expert was wary of using an unfamiliar testing station by herself. Ultimately, test scripts were used to demonstrate to the expert how the system worked and to serve as a focal point for feedback. Still, the expert's hands-on involvement in testing would likely have yielded a better system as well as a richer development environment.

The first pass at testing yielded changes which were primarily initiated

by the knowledge engineer. These involved changes for improved performance and areas where the knowledge engineer questioned the correctness of the system. Changes for performance were generally developed by the knowledge engineer alone. As an example, a check was added to the retry and adjustment logic to prevent proposing boundaries which had already been tried. For questions of conceptual correctness, there was substantial verbal communication with the expert to answer specific questions, to verify the correctness of answers, and to review how the system would behave under specific conditions. During this process, the knowledge engineer had to translate the system's behavior between PROLOG and the expert's language. Had a high-level tool been used, and had the expert been at the development site, the expert could possibly have reviewed the system's architecture directly and suggested explicit modifications.

Reviewing the test results with the expert revealed an interesting misunderstanding. The expert initially intended the system to suggest oligo boundaries which would fall in the middle of their complimentary oligos, as shown in Figure 5.16 (a) below. However, this was never made explicit. The system was developed, instead, to place the boundaries of one oligo as close as possible to the boundaries of its complimentary oligo. This strategy allows for minimal overlap, as shown in (b). Before the expert saw the results of the system, however, she was introduced to an oligo planning strategy different from her own. She adopted the new strategy as a valid alternative. This new strategy was exactly in line with Geneplanner's. When this was discovered, the expert suggested that Geneplanner keep its strategy. If the outcome had not been so fortunate, a change in Geneplanner's strategy would have been

(a) Expert's original intent:



(b) Geneplanner's approach:

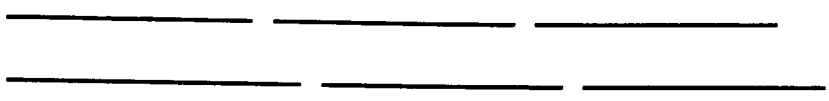


Figure 5.16: Relative Oligo Placement: Expert's Intention -vs- Geneplanner's Results

necessary.

After testing and modifications, Geneplanner was able to handle simple test cases with a few scattered patterns quite well and moderate cases which involved low-level refinement satisfactorily (these are reviewed in detail in Section 6). However, in cases which contained several patterns in very close proximity, the system was unable to come up with a solution. This is because the rules focus on one pattern at a time; they do not consider proposing oligos to cover multiple critical patterns.

At this point, the knowledge engineer and expert examined the system's overall correctness and performance. It was decided that the next step would be to approach the problem from a different angle: to have the system take a more global view before setting default boundaries. This option and other possible future enhancements are described in Section 7.

Chapter 6

Test Results

This section presents test results from Geneplanner. Simple to moderate test cases are presented and discussed. The entire script from one consultation is included. The final case shows some of Geneplanner's user interface features.

6.1 Case #1 - Simple Extension

Characteristics of gene:

gene length = 200 bases

g-rich area

from base 60 to base 75 on the 5' 3' strand

100% concentration

palindrome

from base 140 to base 155 (both strands)

single foldback length = 6

Results (oligos to synthesize):

strand	start base	end base	step	explanation
5'- 3'	1	50	1	default size
3'- 5'	41	90	1	default size
5'- 3'	91	124	2	avoid palindrome
3'- 5'	105	124	2	avoid palindrome
5'- 3'	125	174	3	default size
3'- 5'	165	200	3	at end of chain

Oligos must be joined in 3 steps using extension because of the presence of g-rich areas.

This first case is a simple extension problem. The g-rich area forced the use of extension. This area would be extremely difficult to chemically synthesize and it was therefore avoided completely. The palindrome in the gene caused an adjustment of the oligos near it. The results show one oligo from base 125 to 174 on the 5'- 3' strand encompassing the palindrome completely. The oligos in the previous join step were shortened to avoid this pattern.

The actual run for this test is given in Appendix B. It shows the user interface including introductory instructions.

6.2 Case #2 - Simple Hybridization-Ligation

Characteristics of gene:

gene length = 200 bases

purine-rich area

from base 38 to base 50 on the 3' 5' strand

90% concentration

palindrome

from base 135 to base 160 (both strands)

single foldback length = 10

Results (oligos to synthesize):

oligo	strand	start base	end base	explanation
1	5'- 3'	1	50	default size
2	3'- 5'	1	60	default size w/overlap
3	5'- 3'	51	96	shortened for min size of neighboring oligo
4	3'- 5'	61	100	end extended for overlap
5	5'- 3'	97	116	min size
6	3'- 5'	101	172	avoid palindrome
7	5'- 3'	117	175	avoid palindrome, end extended for overlap
8	3'- 5'	173	200	at end of chain
9	5'- 3'	176	200	at end of chain

All oligos can be joined at once with hybridization-ligation.

This case is a fairly simple hybridization-ligation problem with two widely separated patterns. As shown in the results, the palindrome caused an adjustment of the oligos near it. The final solution includes oligos from base 101 to 172 on the 3'- 5' strand and from base 117 to 175 on the 5'- 3' strand, both encompassing the palindrome completely, leaving no self-complimentary sequences at the ends of the oligos. The purine in this example did not cause

any adjustments.

These results show several of the low-level refinement rules at work. For example, after oligo #7 was extended back to position 117 to avoid the palindrome, its previous neighbor, oligo #5, had to be shortened to base 116. But this oligo was then too short; its starting position had to be pushed back to base 97. This, in turn, affected the ending position of its previous neighbor, #3. The complimentary oligo, #4, then had to be adjusted to base 100 to provide sufficient overlap.

6.3 Case #3 - Moderate Pattern Conditions

Characteristics of gene:

gene length = 300 bases

purine-rich area

from base 30 to base 50 on the 3' 5' strand

85% concentration

repeat

from base 105 to 110 and

from base 135 to 140 (both strands)

g-rich area

from base 200 to base 215 on the 3' - 5' strand

83% concentration

palindrome

from base 250 to base 265 (both strands)

single foldback length = 6

Results (oligos to synthesize):

strand	start base	end base	explanation
5'- 3'	1	52	end extended for overlap
3'- 5'	1	47	shortened for min size of neighboring oligo
3'- 5'	48	67	min size
3'- 5'	68	87	min size
5'- 3'	53	89	avoid repeat
3'- 5'	88	175	start extended for overlap
5'- 3'	90	152	avoid repeat
5'- 3'	153	202	default length
3'- 5'	176	225	default length
5'- 3'	203	234	avoid palindrome
3'- 5'	226	300	close to end
5'- 3'	235	300	close to end

All oligos can be joined at once with hybridization-ligation.

This case shows a moderate set of pattern conditions. As indicated, multiple levels of adjustments were made to avoid placing the oligo boundaries in critical regions. In general, the results are satisfactory. The major improvement could be accomplished by combining the two minimally sized oligos (from 48 to 67 and from 68 to 87 on the 3'- 5' chain) into one oligo.

6.4 Case #4 - Other Features

The final case shows examples of situations too complex for Geneplanner as well as some of the miscellaneous interaction features of the system. The actual run for this test is given in Appendix B. Situations to note include:

1. Patterns too close to each other or to the end are caught; the user can instruct Geneplanner to ignore these patterns or quit.
2. When the user asks "why" in response to a question, the system provides an explanatory response.
3. The system checks for reasonable input for patterns, such as foldback length of palindromes, patterns extending beyond end of gene, and misspellings.
4. The user can enter an entire word or a one-letter abbreviation for input.

Chapter 7

Issues for Future Enhancement

7.1 Finding Patterns in the Gene

An important element of DNA synthesis planning is knowledge of the patterns within a given sequence. There exist algorithms to search DNA sequences for a variety of patterns [NUCL82, NUCL84]. One example is the SEQ program of IntelliGenetics, Inc. The pattern matching algorithms are non-trivial in that the pattern length is unfixed and a minimal number of non-matches within the matched area are acceptable. Due to the complexity of these algorithms and their existence in production packages, it was decided not to include pattern searching in this project. The location of patterns within a given chain is provided by the user.

If this system were to be fully developed, a pattern-finder would have to be included. As Geneplanner currently exists, the user is continually asked for pattern information in various areas of the gene. This questioning consumes

by far most of the elapsed system run time, and it quickly becomes annoying to the user. Also, gene sequences to be synthesized are generally stored in machine-readable form, providing ready input for a pattern-finding program.

7.2 Initial Oligo Boundary Selection

Geneplanner was designed to tackle the problem of selecting where to place an oligo boundary as it was described by the expert: count out about 50 bases from the start of the oligo, look around for nearby patterns and other oligo boundaries, and then determine the exact position for the end of this oligo, adjusting other oligos as necessary. At a very basic level, this generated the initial rule: If 50 bases out is okay, set the oligo boundary here.

This logic in action yielded good results in very simple cases, but it yielded suboptimal results when patterns occurred in close proximity to each other or to the gene ends, as in test case #3, in section 6.3. Some refinements were made to the design, but the basic strategy was left intact. After further observation, it appeared the best strategy would be to have approached oligo boundary selection, and indeed much of the high level organization of the program, differently.

The alternative approach would be to look at patterns and previously established oligos in the chain from about 25 bases to about 100 bases out from the oligo start and then select the optimal placement of the oligo boundary. This would have required redeveloping the rules in line with this approach. For example, one rule under this new approach might be as follows: If P1

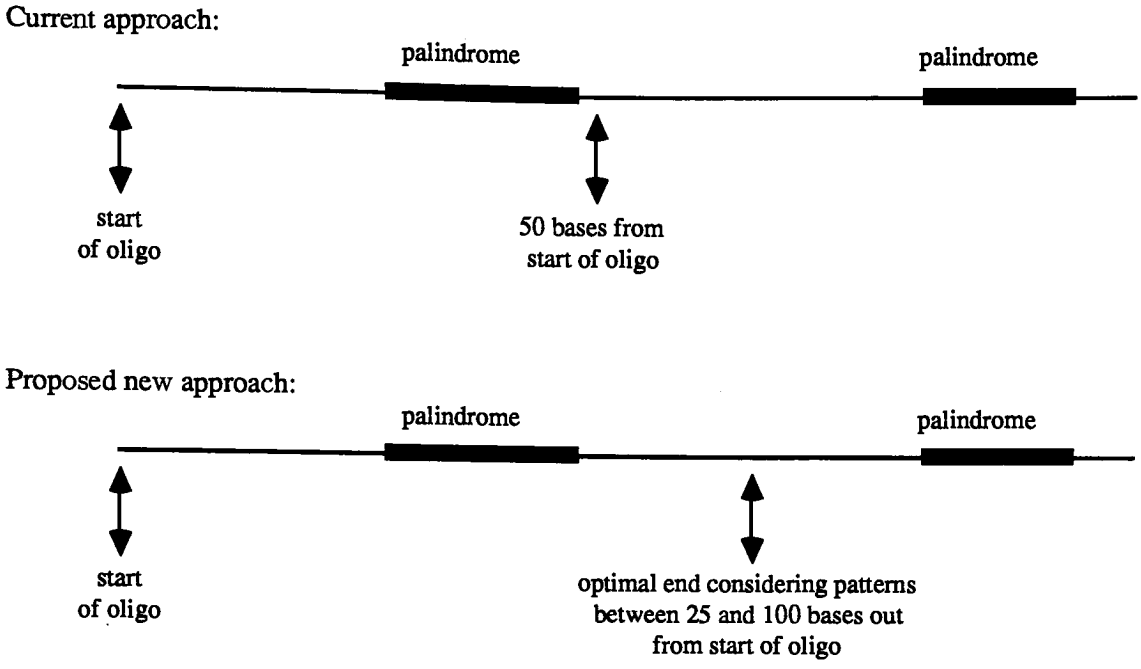


Figure 7.1: Alternative Initial Boundary Selection Strategy

and P2 are the only patterns that exist between bases (oligo start + 25) and (oligo start + 100), and the distance between the patterns is greater than X, then place the end of the oligo halfway between where P1 ends and where P2 begins. This approach is contrasted with the current implementation in Figure 7.1. The next step in the development of Geneplanner would be to implement this alternative strategy.

Using this alternative approach has an interesting implication. More logic would lie in selecting the optimal spot for oligo boundary placement, and less logic would lie in backtracking to adjust oligo boundaries. The backtracking, however, was a fundamental part of the problem as it was initially stated. In fact, PROLOG was chosen as the tool largely because of its built-in support for backtracking. With the fundamental problem strategy changing, the need for PROLOG must be re-evaluated. Another tool may well be more appropri-

ate, such as LISP (for low-level development) or an expert system shell (for more rapid and convenient development).

This design issue raises a point relative to modeling the system on the expert's description of the problem solving process. Although this expert is very knowledgeable and was quite definite about the process of chemical gene synthesis planning, her description may not have been accurate for the purpose of developing an expert system. This phenomenon of the difficulty on the expert's part to describe his or her knowledge in a way that can be easily modeled is not uncommon [WATE86]. So, in order to develop a functional expert system, one must be prepared not only to set aside the first prototype but also the expert's initially stated approach to the problem.

7.3 Retry Strategy

As mentioned above, some refinement was done to the strategy to improve performance. However, this refinement was relatively minor. Major refinement to the existing strategy may have yielded better performance, although probably not nearly as significant as would be achieved by implementing the alternative strategy for initial oligo boundary selection described above.

One possible major refinement is in the are of the retry logic. This is the logic which directs the adjustment of oligo or neighboring oligo boundaries when a pattern has been detected in a critical area. As described in the detail design, there are several levels of retry strategy. The top level includes strategies such as moving the oligo beyond the critical pattern or backing it up

to end before the pattern. The low level strategies include adjusting an oligo length for minimum or maximum number of bases or to ensure an overlap with a complimentary oligo. The problem is that the system tends to get stuck in the low-level strategies; it rarely pops back up to the higher level strategies.

This problem could be alleviated by incorporating meta-level knowledge in the low-level strategy logic. The system could monitor how many low-level adjustments it had tried without success. After a certain number of adjustments, the system could be forced back up to the higher level strategies. To carry this further, the number of possible low-level adjustments could vary depending on the likelihood that alternative high-level adjustments would be successful. For example, an extremely long pattern might encourage more low-level adjustments than a short one.

7.4 Final Oligo Sizes

The suggested oligos to synthesize which are output from the system sometimes include numerous oligos of short length next to each other on the same chain (as shown in Section 6.3). This is a result of shortening oligos several times to avoid a pattern. These results are not in line with what the expert would suggest. Consecutive short oligos can be combined into fewer oligos of more optimal length.

This problem could be addressed by implementing a second-pass optimizer. This optimizer would analyze short oligos and combine them if possible. A second-pass optimizer could also be used to change oligo boundaries rela-

tive to patterns and possibly adjust them to yield a more optimal solution in general. The rules for this would be similar to those described in section 7.2, but they would be applied after-the-fact. This is more of a patch-up approach to solving the fundamental problem, and, therefore, is less desirable than changing the initial oligo boundary selection strategy.

7.5 Explanation System

Geneplanner's explanation system is fairly shallow. It offers optional explanations as to how the system determined the one of the two boundaries for each oligo (the explanation applies to the boundary determined last by the system). The explanations are recorded in code form as each oligo is determined. In this process, the system has very limited vision, e.g. if the system is adjusting boundaries within the low-level retry logic, the explanation attached to the oligos in focus will refer to the low-level refinements, not to the high-level retry strategy.

This shortcoming could be alleviated by storing more data for each oligo during processing and/or by storing adjustment explanations separate from the individual oligos. For example, as the system proceeds through the levels of retry logic to adjust an oligo boundary, it could store a trace of its flow in the knowledge base. This trace could then be deleted if the path proved fruitless or kept and displayed along with its resulting oligos.

Another shortcoming of the explanation system is that Geneplanner does not inform the user of its progress-to-date during the consultation. The

user may interrupt a question by asking "why", but the system gives only a shallow reason as to why the current question is being asked. The system could be enhanced to provide a picture of the solution-to-date when asked for by the user.

Chapter 8

Conclusion

The development of Geneplanner provided good exposure to the issues inherent in expert systems development. The experience and insight gained were in line with the experience of others in this field [WATE86]. First of all, the expert's presence is extremely valuable during the entire development process. Also, the knowledge acquisition process is an extremely important phase of the project, and it is not easy. Many hours of communication with the expert are required, and what appears to be the proper approach to the problem in the early stages may not be the most accurate or optimal strategy. Thus the developer should be prepared to throw away the first prototype. Control structures, knowledge representations, and even tools may have to change. But, if the developer and expert have commitment and enthusiasm, the process can be enlightening and, even, fun.

This project also provided exposure to various designs and strategies applicable to chemical DNA gene synthesis planning. More is known in terms

of alternatives to avoid and pursue for this and similar problems. In general, PROLOG is a good tool for problems requiring depth-first search with backtracking, but a blind depth-first approach to this problem may not be optimal. It would be desirable to incorporate more intelligence in the selection of the initial oligonucleotide boundaries. Next, the rules which look for critical patterns function well, following the expert's approach and utilizing PROLOG's control mechanisms; however, the rules which record problem-causing patterns should be made more efficient. Geneplanner's retry strategy works satisfactorily, as long as only a few low-level adjustments are necessary. With a few optimizations, it should work well in conjunction with a more optimal approach to initial oligo boundary selection. Finally, the user input should be minimized, and the explanation system should contain information regarding the development of the solution which the user would consider valuable.

Appendix A

Detail Rules for the System

1. Main procedure

```
determine initial join method;
set initial starting position to base 1 on 5' to 3' chain;
do the following until we've reached the end of the chain:
        determine initial length of the next fragment;
        assume a fragment of initial length;
        do purine checks;
        do palindrome checks;
        do repeat checks;
        determine starting position for next fragment;
output the resulting list of fragments.
```

2. Determine initial join method

```
ask for g-rich patterns anywhere on chain;
if there are 10 or more G's in a row anywhere on chain,
then  join method is extension;
else  join method is hybridization-ligation.
```

3. Determine initial length of next fragment

```
if fragment begins within 75 bases of the end of the entire chain,
then  set length to the distance to the end of the chain;
      return;

if this is the first fragment or the join method is extension,
```

then set length to 50 bases;
return;

if a length of 50 bases would leave a sticky end of at least 10 bases
(i.e. if the difference between the end of this fragment and the
end of the last fragment on the opposite strand is at least 10)
then set length to 50 bases;
return.

determine the distance between the start of this fragment and
the end of the last fragment on the opposite strand;
if distance $\leq$ 50 bases,
then set length to distance + 10 bases,
else set length to distance − 10 bases.

4. Assume a fragment of length (n)

n must be greater than 20 and less than 90;
if starting position + n > total length of synthesized chain,
then set fragment length to be the length of the chain;
else set fragment length based on starting position + n;
add fragment to the list of good fragments.

5. Determine starting position for next fragment

if we're using hybridization-ligation,
then find shortest strand (5' to 3' or 3' to 5') we've "synthesized"
so far;
next starting position is 1 beyond end of last fragment on this strand;
return.

if we're using extension,
then if next fragment should start with sticky end,
then next starting position is 10 bases before end of complimentary
fragment;
else next starting position is next base after end of complimentary
fragment.

6. Do purine checks

if length of fragment > 50,
then check for purine patterns within 20 bases of 3' end;
if first 20 bases are > 75% purine or
first 10 bases are > 60% purine,
then do purine retry.

if length of fragment is > 30,
then if fragment is > 30% G's
then do purine retry.

if we've passed the checks to this point,
then fragment passes purine checks.

7. Do purine retry

if fragment is on the 3' to 5' strand,
1st: retry with fragment boundary beginning at the end of the purine area (i.e. extend the previous fragment).
2nd: retry with fragment boundary ending at the end of the purine area and starting further back (i.e. shorten the previous fragment).

if fragment is on the 5' to 3' strand,
1st: retry with fragment boundary ending right before the start of the purine area (i.e. shorten the fragment).
2nd: retry with fragment boundary beginning at the start of the purine area and extending further out (i.e. extend the previous fragment).

if all retries fail,
then program cannot determine fragments to synthesize.

8. Do palindrome checks

check for palindrome patterns within first or last 10 bases of the fragment;
if part/all of a palindrome falls within first or last 10 bases,
then do palindrome retry.

if no palindromes were found,
then fragment passes palindrome checks.

9. Do palindrome retry

for palindrome at start of fragment:
1st: retry with fragment boundary beginning 10 bases back from start of palindrome.
2nd: retry with fragment boundary beginning 10 bases beyond end of palindrome.

for palindrome at end of fragment:
1st: retry with fragment boundary ending 10 bases back from start of palindrome.
2nd: retry with fragment boundary ending 10 bases beyond end of palindrome.

final retry:
determine max foldback area length;
determine overlap area length;
if max foldback area length < overlap area,
then proceed
else program cannot determine fragments to synthesize.

10. Do repeat checks

check for repeat patterns within first or last 10 bases of the fragment (copy must exist within this join step);
if part/all of a repeat falls within first or last 10 bases,
then do repeat retry.

if no repeats were found,
then fragment passes repeat checks.

11. Do repeat retry

for repeat at start of fragment:
1st: retry with fragment boundary beginning 10 bases back from start of repeat.
2nd: retry with fragment boundary beginning 10 bases beyond end of repeat.

for repeat at end of fragment:
1st: retry with fragment boundary ending 10 bases back from start

of repeat.
2nd: retry with fragment boundary ending 10 bases beyond end of repeat.

final retry:
if join method is hybridization-ligation,
then start a new join step;
proceed as normal.
if join method is extension,
then retry with fragment boundaries such that pattern only
occurs once (i.e. set fragment length to pattern length).
if all retries fail,
then program cannot determine fragments to synthesize.

12. Retry

retract all fragments whose boundaries are changing;
assert new proposed boundaries;
for all fragments whose boundaries are changing:
fragment length must be > 20
fragment length must be < 90
fragment must provide sufficient overlap with complimentary fragments
(minimum 3 bases if sticky end is required)
fragment must be new (not previously tried)
if any of these checks fail,
then adjust boundaries of fragments as necessary (lengthen/shorten
fragments, attempting first to adjust a boundary which
is not in the critical area that generated this retry)
retry again.
if all checks pass,
then proceed with new proposed boundaries.

Appendix B

System Test Results

B.1 Test Results for case #1

The following is the script from test case #1, described in Chapter 6, section 1.

```
C-Prolog version 1.4
| ?- [geneplanner].
geneplanner consulted 624 bytes 0.183333 sec.

yes
| ?- geneplanner.

Please be patient while the files are loaded...

main consulted 1394 bytes 0.533333 sec.
output consulted 4644 bytes 1.56667 sec.
next_frag consulted 3228 bytes 0.966668 sec.
checks consulted 5808 bytes 1.96667 sec.
retry consulted 7120 bytes 2.55 sec.
support consulted 3104 bytes 1.3 sec.
purine consulted 2164 bytes 0.716667 sec.
palindrome consulted 1328 bytes 0.5 sec.
repeat consulted 1300 bytes 0.5 sec.
check_pattern consulted 3996 bytes 1.53333 sec.
questions consulted 8776 bytes 3.3 sec.
misc consulted 1056 bytes 0.450012 sec.
```

Welcome to Geneplanner, an assistant for chemical DNA gene synthesis planning.

Would you like to review instructions on how to use this system? y.

Geneplanner is a prototype of an expert system for planning what fragments (oligos) to synthesize when chemically manufacturing a gene. This system will ask for information on the existence of patterns in certain regions of the DNA. Using this information, Geneplanner will determine the starting and ending positions of the fragments to chemically synthesize. The system will also determine how to join these fragments. This information will be output to the user at the end of the consultation.

Questions can be answered as follows:
Yes/No questions can be answered with "yes.", "y.", "no.", or "n."
Questions requiring numeric input can be answered with the appropriate number followed by a period.
Note: chain directions are represented as numbers:
5^ - 3^ is represented 53
3^ - 5^ is represented 35
Pattern types can be spelled out in full or abbreviated to one character. They must be followed by a period.
"Why." can be input at any time to see why a particular question is being asked.
To exit the consultation at any time, enter "exit.", "quit.", or "end.".

Type any character followed by a period to proceed... c.

After the results have been displayed, the system asks if the results [illegible]
A response of "how.", "explain.", or "reasons." will produce a display of the results with reasons as to how fragment positions were determined and what join method should be used.

Please remember that Geneplanner is a prototype only. It cannot handle extremely complex or long genes very well. Please limit your input to minimal length (750 bases or less) and complexity (scattered patterns).

****

What is the total length of the gene? 200.

Are there any areas of 10 or more solid G's? y.

Starting position? 60.

Ending position? 75.

53 or 35 direction? 53.

Asserting g_rich on 53 strand,
from position 60 to position 75
of length 16 and 100% concentration.

Are there any more patterns of this type? n.

Are there any unknown palindrome or repeat patterns between bases 1 and 13? n.

Are there any unknown palindrome or repeat patterns between bases 38 and 53? n.

Are there any unknown purine patterns between bases 28 and 53 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 78 and 93? n.

Are there any unknown purine patterns between bases 38 and 63 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 88 and 103? n.

Are there any unknown palindrome or repeat patterns between bases 128 and 143? y.

Pattern type (palindrome/repeat) ? p.

Starting position? 140.

Ending position? 155.

```
Foldback length? 6.

Asserting palindrome on 53 strand,
        from position 140 to position 155
        of foldback length 6.

Asserting palindrome on 35' strand,
        from position 140 to position 155
        of foldback length 6.

Are there any more patterns of this type? n.

Are there any unknown purine patterns between bases 118 and 143 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 112 and 127? n.

Are there any unknown purine patterns between bases 102 and 127 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 152 and 167? n.

Are there any unknown purine patterns between bases 112 and 137 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 102 and 117? n.

Are there any unknown palindrome or repeat patterns between bases 122 and 137? n.

Are there any unknown palindrome or repeat patterns between bases 162 and 177? n.

Are there any unknown purine patterns between bases 152 and 177 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 168 and 2007 n.

Are there any unknown purine patterns between bases 162 and 187 on the 35 chain? n.
```

The fragments to synthesize are:

Direction	Start Pos	End Pos	Join Step
35	165	200	3
53	125	174	3
35	105	124	2
53	91	124	2
35	41	90	1
53	1	50	1

Are these results okay? how.

The fragments to synthesize are:

Direction	Start Pos	End Pos	Join Step	Reason
35	165	200	3	Close to end of chain
53	125	174	3	Default fragment length
35	105	124	2	Back up start of proposed fragment to make fragment at least 20 bases long
53	91	124	2	Back up to avoid palindrome near end of proposed fragment
35	41	90	1	Default fragment length
53	1	50	1	Default fragment length

Extension must be used to join the fragments because of the presence of g-rich areas.

Are these results okay? y.

B.2 Test Results for case #4

The following is the script from test case #4, described in Chapter 6, section 4.

```
Do you want to plan another gene? y.

What is the total length of the gene? 150.

Are there any areas of 10 or more solid G's? why.
To determine what join method to assume.
Are there any areas of 10 or more solid G's? y.

Starting position? 67.

Ending position? 80.

53 or 35 direction? 35.

Asserting g_rich on 35 strand,
        from position 67 to position 80
        of length 14 and 100% concentration.

Are there any more patterns of this type? n.

Are there any unknown palindrome or repeat patterns between bases 1 and 13? no.

Are there any unknown palindrome or repeat patterns between bases 38 and 53? yes.

Pattern type (palindrome/repeat) ? pallindrome.
Type must be "palindrome.", "p.", "repeat.", or "r."  Please try again. palindrome.
```

Starting position? 50.

Ending position? 60.

Foldback length? 9.

Pattern length must be a number less than or equal to 5.
  Please try again. 4.

This pattern creates a situation too difficult for the Geneplanner prototype. Wou
pass/quit)? bypass.

Are there any more patterns of this type? n.

Are there any unknown purine patterns between bases 28 and 53 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 78 and 93? n.

Are there any unknown purine patterns between bases 38 and 63 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 88 and 103? n.

Are there any unknown purine patterns between bases 78 and 103 on the 35 chain? n.

Are there any unknown palindrome or repeat patterns between bases 98 and 113? n.

Are there any unknown palindrome or repeat patterns between bases 138 and 150? y.

Pattern type (palindrome/repeat) ? p.

Starting position? 140.

Ending position? 155.

Pattern end must be a number between 140 and 150. Please try again. 150.

```
Foldback length? 4.

This pattern creates a situation too difficult for the Geneplanner prototype.  Wou
pass/quit)? b.

Are there any more patterns of this type? n.

Are there any unknown purine patterns between bases 128 and 150 on the 35 chain? n.

Are there any unknown purine patterns between bases 138 and 150 on the 35 chain? n.

The fragments to synthesize are:

Direc-  Start   End     Join
tion    Pos     Pos     Step

35      141     150     2
53      101     150     2
35      81      100     1
53      1       50      1

Are these results okay? how.

The fragments to synthesize are:

Direc-  Start   End     Join
tion    Pos     Pos     Step    Reason

35      141     150     2       Close to end of chain
53      101     150     2       Close to end of chain
35      81      100     1       Extend end of proposed fragment to make
                                fragment at least 20 bases long
53      1       50      1       Default fragment length

Extension must be used to join the fragments because of the presence of
g-rich areas.

Are these results okay? y.

Do you want to plan another gene? n.

yes
| ?- ^D
[ Prolog execution halted ]
```

Appendix C

Bibliography

[CLAR] K. L. Clark and F. G. McCabe, "PROLOG: A Language for Implementing Expert Systems", Department of Computing, Imperial College, London.

[FRIE] Peter E. Friedland, Paul Armstrong, and Thomas Kehler, "The Role of Computers in Biotechnology", Stanford University, Department of Computer Science.

[HAYE83] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat, Building Expert Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[KHOR79] H. G. Khorana, "Total Synthesis of a Gene", Science, Vol. 203, 16 February 1979, pp. 614-625.

[LEWI85] Benjamin Lewin, Genes: Second Edition, John Wiley & Sons, New York, 1985.

[LOMB83] Stephen Lombardi, Hollis Seidell, and John Hachmann, "The evaluation of synthetic strategies for oligonucleotides of defined sequence", Computer Programs in Biomedicine, Vol. 16, 1983, pp. 71-76.

[LUGE85] George Luger, Knowledge-Based Systems and AI, for Integrated Computer Systems, Integrated Computer Systems Publishing Co., 1985.

[NUCL82] Nucleic Acids Research, Vol. 10, Num. 1, January 1982.

[NUCL84] Nucleic Acids Research, Vol. 12, Num. 1, January 1984.

[RICH83] Elaine Rich, Artificial Intelligence, McGraw-Hill Inc., New York, 1983.

[ROSE83] Israel Rosenfield, Edward Ziff, and Borin van Loon, DNA for Beginners, Writers and Readers Publishing Cooperative Ltd., London, 1983.

[WATE86] Donald A. Waterman, A Guide to Expert Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

Appendix D

Footnotes

[1] For more information on current work in these areas, refer to [FRIE], [NUCL82], and [NUCL84].

[2] Dr. Peter E. Friedland, Stanford University, Computer Science Dept., Coordinator of MOLGEN project, Knowledge Systems Laboratory; personal communication.

[3] Dr. Dennis Smith, IntelliGenetics, Inc., Director of Research and Development; personal communication.

[4] Cheryl Heiner, Applied Biosystems, Inc., Research Biochemist.

[5] Per discussion of Cheryl Heiner, Applied Biosystems, Inc., with representative of IntelliGenetics, Inc., at trade conference.

[6] Per Cheryl Heiner, Applied Biosystems, Inc.