

WL-TR-94-4008

DISCOVERY SYSTEMS FOR MANUFACTURING

AD-A276 216



DAN WOODS
JACK PARK

THINKALONG SOFTWARE, INCORPORATED
16740 WILLOW GLEN ROAD
BROWNSVILLE CA 95919

JANUARY 1994

FINAL REPORT FOR 12/01/91-12/01/93

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

S **DTIC**
ELECTE
FEB 28 1994
A

DTIC QUALITY

MATERIALS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7734

10880

94-06399



2 23 602

NOTICE

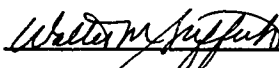
WHEN GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA ARE USED FOR ANY PURPOSE OTHER THAN IN CONNECTION WITH A DEFINITELY GOVERNMENT-RELATED PROCUREMENT, THE UNITED STATES GOVERNMENT INCURS NO RESPONSIBILITY OR ANY OBLIGATION WHATSOEVER. THE FACT THAT THE GOVERNMENT MAY HAVE FORMULATED OR IN ANY WAY SUPPLIED THE SAID DRAWINGS, SPECIFICATIONS, OR OTHER DATA, IS NOT TO BE REGARDED BY IMPLICATION, OR OTHERWISE IN ANY MANNER CONSTRUED, AS LICENSING THE HOLDER, OR ANY OTHER PERSON OR CORPORATION; OR AS CONVEYING ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY IN ANY WAY BE RELATED THERETO.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

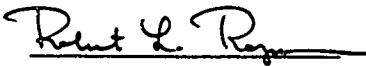
THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.



STEVEN R. LECLAIR
Technical Director, Manufacturing Research
Integration and Operations Division
Materials Directorate



WALTER M. GRIFFITH
Branch Chief, Manufacturing Research
Integration and Operations Division
Materials Directorate



ROBERT L. RAPSON
Chief, Integration and Operations Division
Materials Directorate

IF YOUR ADDRESS HAS CHANGED, IF YOU WISH TO BE REMOVED FROM OUR MAILING LIST, OR IF THE ADDRESSEE IS NO LONGER EMPLOYED BY YOUR ORGANIZATION PLEASE NOTIFY WL/MLIM, WRIGHT-PATTERSON AFB, OH 45433-7746 TO HELP MAINTAIN A CURRENT MAILING LIST.

COPIES OF THIS REPORT SHOULD NOT BE RETURNED UNLESS RETURN IS REQUIRED BY SECURITY CONSIDERATIONS, CONTRACTUAL OBLIGATIONS, OR NOTICE ON A SPECIFIC DOCUMENT.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE JAN 1994	3. REPORT TYPE AND DATES COVERED FINAL 12/01/91--12/01/93		
4. TITLE AND SUBTITLE DISCOVERY SYSTEMS FOR MANUFACTURING		5. FUNDING NUMBERS C F33615-92-C-5802 PE 65502 PR 3005 TA 05 WU 17		
6. AUTHOR(S) DAN WOODS JACK PARK				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) THINKALONG SOFTWARE, INCORPORATED 16740 WILLOW GLEN ROAD BROWNSVILLE CA 95919		8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) MATERIALS DIRECTORATE WRIGHT LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT PATTERSON AFB OH 45433-7734		10. SPONSORING/MONITORING AGENCY REPORT NUMBER WL-TR-94-4008		
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) This has been a three-part project: (1) enhancing generalized discovery tools on our existing computer platform, (2) building specialized discovery tools, and (3) applying generalized and specialized discovery tools to specific materials-related projects including protein studies and crystallography. During the work, we have had the opportunity to incorporate some of our other activities, including studies we have done in biomedical fields using our discovery system. We have also folded in some of our external work in design. Portions of this other work are included in this report because it serves to illustrate key points we wish to make. This report includes portions of contributions made by our consultants, W.B. Dress and A.G. Jackson. We begin our report with a review of the philosophical aspects of what we call computational theory formation. We then present the computer tool that has been extended as part of this work, the goal of which is to address the issues brought up in the first section. Finally, the three different applications mentioned above are introduced.				
14. SUBJECT TERMS ARTIFICIAL INTELLIGENCE, DISCOVERY SYSTEMS, POLYMER COMPOSITE CURING, PROTEIN STRUCTURES		15. NUMBER OF PAGES 108		
		16. PRICE CODE		
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

Chapter 1 Objectives

1.1	Overview	1-1
1.2	Discovery	1-1
1.2.1	Philosophical Background: What is Science?	1-1
1.2.1.1	Popper's Three Worlds	1-2
1.2.1.2	Dynamical Systems View	1-2
1.2.1.3	Doing Science: The Modeling Relation	1-3
1.2.1.3.1	Theoretical and Empirical Models	1-3
1.2.1.3.2	Computational Models	1-4
1.2.1.4	The Simulated Laboratory	1-4
1.2.2	The Scientist ↔ Model Relation	1-4
1.2.2.1	Modeling The Scientist	1-5
1.2.2.2	Modeling The Model	1-6
1.3	Tools	1-7
1.3.1	Qualitative Modeling	1-8
1.3.2	TSC's Exploratory Behaviors	1-8
1.3.2.1	Case-based Reasoning and Analogy	1-8
1.3.2.2	Directed Evolution	1-8
1.3.3	Rough Sets	1-9
1.3.4	Nearest Neighbor Analysis	1-9
1.4	Applications — Proteins	1-9
1.4.1	Prediction	1-9
1.4.2	Design	1-9
1.5	Applications — Crystallography	1-10

Chapter 2 Tool Building

2.1	The Scholar's Companion	2-1
2.1.1	Architectural Overview	2-1
2.1.2	The TSC Language: Statements	2-2
2.1.3	The TSC Knowledge Base Structure: Taxonomy	2-3
2.1.4	The TSC Knowledge: Rules	2-3
2.2	Model Building	2-4
2.3	Encyclopedia Behaviors	2-5
2.4	Exploratory Behaviors	2-5
2.4.1	Case-Based Reasoning	2-6
2.4.1.1	CBR on TSC	2-6
2.4.1.2	TSC Case-Based Design Approach vs. Conventional AI CBR	2-6
2.4.1.2.1	Similarities	2-7
2.4.1.2.2	Differences	2-7
2.4.2	Hypothesis Formation	2-8
2.4.3	Design	2-9
2.4.4	Directed Evolution	2-9
2.4.4.1	Directed Evolution vs. Genetic Algorithms	2-10
2.4.4.2	Algorithm	2-10
2.4.4.3	Optimization Strategy	2-11
2.4.4.4	Crossover	2-12
2.4.4.5	Mutation	2-12
2.4.5	Genetic Programming	2-13
2.4.5.1	Algorithm	2-13
2.4.5.2	Survival and Reproduction	2-14
2.4.5.2.1	Survival	2-14
2.4.5.2.2	Program Reproduction	2-14

Chapter 2 (continued)

2.4.5.3	Crossover and Mutation	2-15
2.4.5.3.1	Program Crossover	2-15
2.4.5.3.2	Program Mutation	2-15
2.5	Data Evaluation Tools	2-16
2.5.1	Rough Set Evaluation of Genetic Program Fitness	2-16
2.5.1.1	Definitions	2-16
2.5.1.2	Evaluating Attributes	2-17
2.5.2	Nearest-Neighbor Pattern Recognition	2-17

Chapter 3 Applications

3.1	Protein Structure Prediction	3-1
3.1.1	Methodology	3-1
3.1.2	Genetic If-Then Rule Generation	3-2
3.1.3	Prediction Program Generation	3-4
3.1.4	Testing Recall and Prediction	3-4
3.2	Protein Design	3-5
3.2.1	Case-Based Protein Design	3-5
3.2.2	Approach	3-6
3.2.3	Case-Based Design Algorithm	3-8
3.3	TEM Control	3-11
3.3.1	TEM Control Approach	3-11
3.3.2	The TSC Combined Analysis/Controller	3-11

Chapter 4 Results

4.1	Tools	
4.1.1	Discovery Tools	4-1
4.1.2	Design	4-1
4.2	Applications	4-2
4.2.1	Proteins	4-2
4.2.1.1	Prediction by GA Rule Building	4-2
4.2.1.2	Prediction by GA Program Building	4-5
4.2.1.3	Prediction by Nearest Neighbor and Protein Design	4-7
4.3	TEM	4-10
4.3.1	Launching the TEM Controller/Simulator	4-10
4.3.2	Launching TSC	4-11
4.3.3	Initialize Simulator	4-11
4.3.4	User Dialog With the TEM Simulator	4-13

Chapter 5 Summary and Conclusions

5.1	General Improvements	5-1
5.2	Prediction System Improvements	5-2
	Acknowledgments	5-4
	References	5-5

Appendices

A	"Nearest Neighbor" code	A-1
B	"Rough Sets" code	B-1
C	"Evolution" code	C-1

Chapter 1

Objectives

1.1 OVERVIEW

We discuss our ongoing experience in the development of tools for research in materials science and engineering. A measurable quantity of progress has been made during this SBIR 02 activity, but much remains to be accomplished before we can say that a "complete" set of materials discovery tools has been built. In our discussions here, it will be revealed that we have looked at several important aspects of materials. Key is the notion that we did not restrict our tool building to structural materials, but rather we branched out to explore biological materials as well. We looked also into aspects of design, with design of molecules a target. We see the tools we are building as being appropriate to the entire materials science and engineering activities, starting from concept discovery and ending with product manufacturing and support.

This has been and continues to be a three-part project: (1) enhancing generalized discovery tools on our existing computer platform, (2) building specialized discovery tools, and (3) applying generalized and specialized discovery tools to specific materials-related projects including protein studies and crystallography.

During the work, we have had the opportunity to incorporate some of our other activities, including studies we have done in biomedical fields using our discovery system. We have also folded in some of our external work in design. Portions of this other work are included in this report because it serves to illustrate key points we wish to make. This report includes portions of contributions made by our consultants, W.B. Dress and A.G. Jackson.

We begin our report with a review of the philosophical aspects of what we call *computational theory formation*. We then present the computer tool that has been extended as part of this work, the goal of which is to address the issues brought up in the first section. Finally, the three different applications mentioned above are introduced.

1.2 DISCOVERY

"There is nothing more practical than a good theory."

Hilbert

1.2.1 Philosophical Background: What is Science?

The ideas we wish to explore and the paths we wish to follow concern the world of scientific discovery and how such high-level human activities might be first simulated and then aided by computer-based tools. The goal is a tool-building one, having immediate utility for selected research programs, and long-term benefit for general scientific activities. This may be viewed as a next-generation "intelligence amplifier" which Ashby hinted at nearly forty years ago [Ashby, 1960].

To clearly present our case, we need to say a few words about science, the task and tools of science, and its relationship to the vast intellectual structure that make up its formal tools and color how we perceive and think about the world. For example, is the study of molecular structure really the study of molecules? Or is it rather the study of the recorded behavior of molecules as reported in the literature? Or is it the study of methods successfully used to study molecules? What level of abstraction do we deal with? When are we doing science and when are we studying how science is done?

These are not meaningless questions as their answers will determine how we go about finding new theories, how we go about verifying them, and how ensuing predictions relate to reality and hence new ideas, methods, and products.

1.2.1.1 Popper's Three Worlds

As a point of departure, imagine that reality consists of three distinct worlds as illustrated in figure 1-1. Following Karl Popper [Popper and Eccles, 1977], we take these worlds to be

1. The world of entities having existence: naturally occurring entities and human artifacts. Examples include molecules, automobiles, animals, and people.
2. The world of mental states and activities not directly perceptible as entities except by ourselves; yet we know that they must exist.
3. The world of mental constructs (whether "correct" or not). Here, we find political theories, plans of action, subjects of contemplation, and scientific theories.

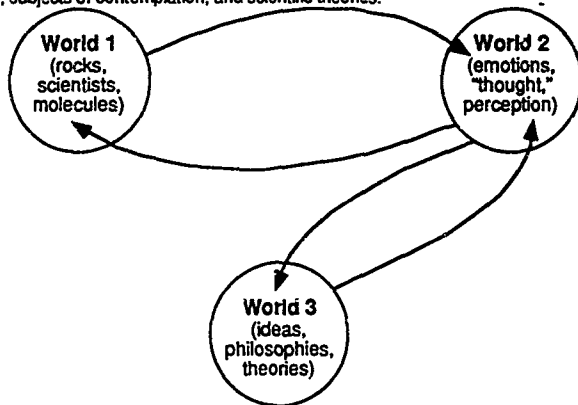


Figure 1-1: Possible relationships among Popper's three worlds.

Note that this scheme is a convenience for our discourse, and not meant to be a philosophical position to be debated or defended. It merely serves as a point of departure. We simply wish to talk about reality, its reflection in the modeling relation, and the several levels of models needed to describe that reality. Thus, the view of science as an activity of World 1 entities building World 3 constructs by means of World 2 activities for elucidating World 1 relationships is the real subject of our presentation.

We are not going to discuss the couplings between the worlds, except to note that they must exist.

1.2.1.2 Dynamical Systems View

We will assume the sufficiency of the Newtonian Paradigm that generalized forces, coordinates, and velocities are all that are necessary to describe the universe to any degree of accuracy. The higher derivatives are not needed. Thus, the model is

$$\dot{\mathbf{x}} = \mathbf{f}[\mathbf{x}, \alpha, t]$$

where x represents the generalized coordinates (i.e., coordinates and velocities in particle description), α represents any parameters (e.g., masses), t represents the time parameter, and dot denotes time derivative. The generalized coordinates are observables of the system and their range comprises the state space of the system. f is a vector of real-valued functions on the state space, and any other observable of the system is represented by some f . This is the most general dynamical system that we will consider; in fact, this viewpoint may be said to comprise what we mean by "science." Note that all of physics (even quantum mechanics), chemistry, and much of biology fits into this scheme. The proof is the unparalleled success of the scientific method over the last three centuries and the explosion in activity of the latter half of this one.

The study of dynamical systems then becomes a study of points and trajectories in the state space. Topology may also be studied when ensembles of state-space points are important (as in statistical mechanics).

Any of the standard models of science can be cast in the dynamical systems formalism, and the difference-equation representation can serve as a computationally efficient model of the formal dynamical system. We will call on this form of dynamical systems modeling exclusively when we explore the construction of models from collections of finite-state automata.

1.2.1.3 Doing Science: The Modeling Relation

The main activity of science is the search for correct and useful models of natural systems. Once a model has been established, it becomes a tool for further scientific exploration and discovery. Along the way, any given model will be revised, refined, and perhaps discarded. This is the normal progression of science.

As depicted in figure 1-2, a natural system is encoded into a formal system (the model) by means of observations and measurements. Inferences are made on the model using the formal rules (e.g., mathematics). Results are then decoded into predictions on the natural system.

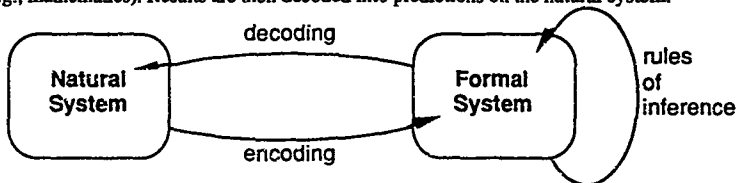


Figure 1-2: Schematic of the general modeling relation (after Rosen).

If we wish to study science with the goal of providing a set of advanced simulation tools for aiding the scientist in his quest, we must first study and understand the modeling relation. We will show later that this study must be expanded to include a model of the scientist and his/her activities as well as a model of the scientific theories and process.

1.2.1.3.1 Theoretical and Empirical Models

The standard model of science is theoretical in the sense that the model builder relies on a well-known theory of measurement and observation and uses standard mathematical techniques to describe and codify the results of the measurements. The next stages are likewise theoretically based as the scientist attempts to construct a deductive system whose goal is to provide predictions of behaviors in World 1.

Empirical models may result during the course of a laboratory experiment wherein certain parts and functions of the natural system under study are replaced by contrivances or held fixed while

other parts are observed and measured under controlled constraints. This tight interaction between theory and experiment is where most of today's science takes place.

1.2.1.3.2 Computational Models

It is only recently that the third type of model became possible. A computational model is a novel mixture of the theoretical and empirical models. It is theoretical in that it is based on the formal notions of mathematics and logic, and empirical in that the computation must actually be carried out in many cases. Here, we must distinguish computation that simulates a system under study from calculation that obtains a solution to a differential equation, for example. The results of calculations have been used for centuries prior to the invention of the computer, but complex simulations of certain systems need the computer to become feasible. After all, the closed-form or a numerical function as an answer is the exception. Certain simulations are then beyond the reach of the "standard" modeling techniques of differential equations and can only be carried out on a suitable simulator (e.g., a digital computer).

An example of a computational model that rarely has a solution obtainable by calculations in the usual sense is provided by boolean switching networks. Theory shows that limit cycles are to be expected, but which limit cycles under which initial conditions can only be determined by a computational model once a certain complexity of the network is surpassed. Such switching networks can provide a model of genetic behavior during organism growth (the differentiation process). The operon model, in particular, is amenable to the switching-network description. To obtain answers to questions of evolutionary behavior requires that the system be simulated since no known short-cut exists. This point has been formally discussed by Wolfram [1988].

1.2.1.4 The Simulated Laboratory

Doing science by computer must ultimately and intimately involve an integrated cybernetic system designing and conducting actual experiments in a real-world (World 1) laboratory. The system will be an extension of the human scientist—a cybernetic "graduate student" with instant access to sophisticated scientific databases, a suite of standard scientific methods, and a repertoire of laboratory techniques. Such a process requires a sophisticated set of effectors and affectors that we do not yet possess. The next best thing is to provide a simulated laboratory.

1.2.2 The Scientist ↔ Model Relation

Since we effectively have all three worlds at our disposal, we can look at and think about the relationship between the scientist and his world of study. This is exactly the system we need to study and model to achieve our goal of building a tool for doing science by computer. To carry out such an ambitious program, we will need to model the scientist (no mean task) and the systems under study (much easier, but certainly non-trivial).

To model the systems under study requires that we must obtain effective models of scientific models of natural systems. We are necessarily one step removed from the realm of the natural or physical scientist and must remain aware of that fact in all that follows. To forget will cause us to confuse our models and constructs with first-order scientific theories, when they are theories of theories and theories of behavior. In effect, what we are doing is studying relationships that map World 3 onto World 3, whereas the (first-order) scientist is studying relationships mapping World 1 onto World 1 by means of constructs in World 3. This activity is illustrated in figure 1-3, which shows how the scientist builds entities in World 3 from observations (World 2 entities) of World 1 behaviors.

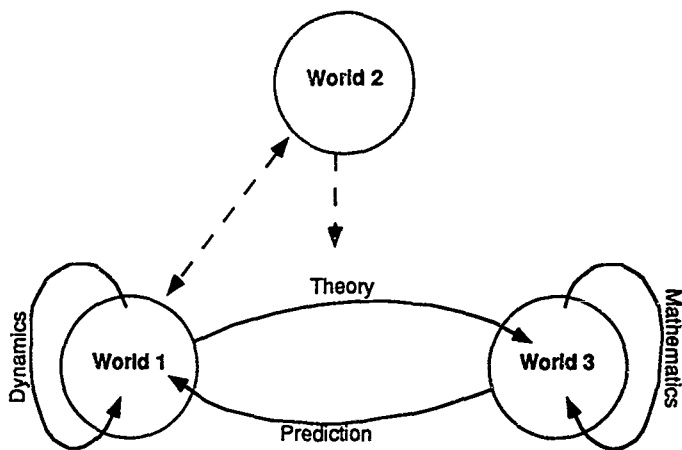


Figure 1-3: Schematic representation of the entities and relationships involved in the discovery process

The only sure guide we have in this undertaking is the successful models constructed by first-order science. These must be our touchstone to the utility of what we wish to accomplish.

1.2.2.1 Modeling The Scientist

This difficult and fascinating subject is at the core of a following discussion of the discovery system tools we call *The Scholar's Companion* (TSC): how to model scientific discovery. Note that numerous authors have been exploring such ideas over the last decade. Notable among such undertakings are studies in qualitative physics [Bobrow, 1985] and simulation of scientific discovery on specially crafted data sets [Langley, 1987; Thagard, 1988]. The goal of TSC is to provide the user with a functional model of the scientist—a scientist-computer that can make hypotheses and perform useful “computer” work such as database manipulation and certain calculations, as well as constructing and testing reasonable models germane to the problems posed by the human scientist.

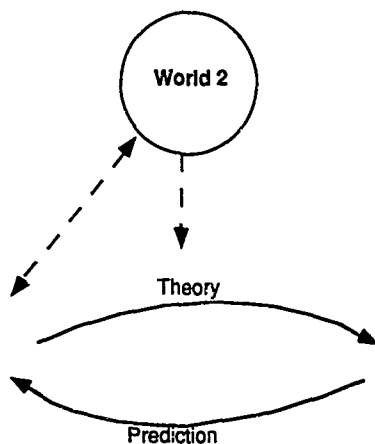


Figure 1-4: Abstraction of entities and relationships comprising the scientist and scientific activities.

Figure 1-4 represents that portion of the scientific discovery process that is to be modeled by TSC. This shows what The Scholar's Companion needs to model at its core. (The left dashed arrow represents some portion of scientific observations required to construct a theory or scientific model. Note that World 1 entities such as the scientist himself or the actual systems under study are not required for this abstraction.) Previous work, cited above, has concentrated only on the encoding-decoding relationships (labeled "Theory" and "Prediction" in the figure), and that on rather limited and contrived data sets. Our discussion in this presentation, based on the mappings among the three worlds, clearly shows that additional modeling is needed. Not only must some of the World 2 entities be taken into account, but the interfaces between World 1 and 2 and between World 2 and 3 are essential to any successful machine representation of scientific activity. Classic artificial intelligence (AI) has had some success in modeling perceptions and thought processes; much of the work discussed in Langley [1987] involves these constructs. The lower-level World 2 entities and the World 1—World 2 interface remain largely unexplored areas. As yet, no comprehensive system as depicted in figure 1-4 has been attempted.

1.2.2.2 Modeling The Model

Our concern in the remainder of this section will be with the computational models representing the natural system under study. We will need to look at the interface between these systems and the observing system, as well as possible formalisms for constructing the model systems. Figure 1-5 illustrates the relationships between the scientific discovery system and the scientist's model of the natural system. In the figure, the scientific theory is represented in the computational system by the Computational Model module and interacts with it by models of the encoding and decoding processes.

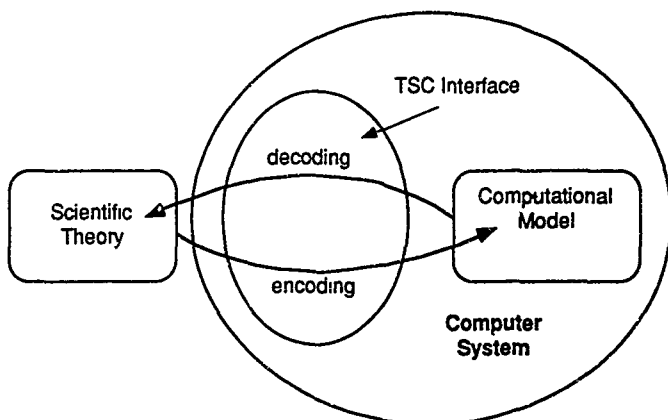


Figure 1-5: A modeling view of a computer-based system for doing science.

The relationships among the model, the theory, and the user are manifested in the interface between certain computer programs. This interface primarily consists of "measures of performance" of the computational model and interpretations of these measures analogous to the of effectors discussed above. An example would be a set of numbers returned to TSC representing a concentration of a certain molecular species that was the subject of the requested model. The effectors consist of a grammar allowing TSC to specify types of models to be created in the Computational Model module. This interface is shown in figure 1-5 as the "TSC Interface."

A realization of these ideas is idealized in figure 1-6. Our Ø2 project has been to develop tools suggested in figure 1-6, and apply those tools to specific problems.

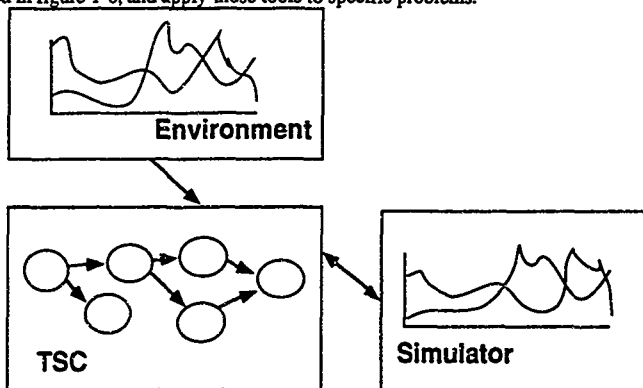


Figure 1-6: A realization of a toolbox for scientific and engineering discovery.

1.3 TOOLS

We implement the philosophical notions discussed above in a toolbox, introduced above, called

The Scholar's Companion (TSC). TSC is being developed to provide scientists and engineers with modeling, simulation, control, and discovery tools.

1.3.1 Qualitative Modeling

Tool building has been the main thrust of this Ø2 activity. Most important has been the refinement of our qualitative modeling tools, called QPD [Wood and Park, 1990]. Modeling, as we shall see, forms the backbone of most of the materials discovery activities we pursue. We call the internal structure of a qualitative model an *envisionment*.

A simple *envisionment* (discussed in detail in the next chapter) from the polymer curing domain that is illustrated in figure 1-7. The initial conditions are that a polymer is contained in an autoclave and heat is applied by the autoclave heater. The polymer eventually begins its condensation curing reaction at which point the IF-THEN rules suggest envisioning two different outcomes of the process: a cured component and a burnt component.

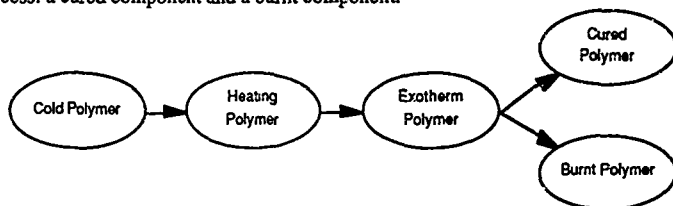


Figure 1-7. A simple envisionment.

Model building provides the opportunity to explore the possible outcomes of a physical process given some initial conditions. Process control discovery then involves finding a process algorithm which encourages the desirable outcome and discourages the undesirable outcome.

1.3.2 TSC's Exploratory Behaviors

We are able to take advantage of the model-building behaviors by asking "what if" questions. Such questions are posed in goal-oriented statements we give TSC. There are two exploratory behaviors we are developing and we discuss here. They are *case-based reasoning* and *analogy*, and *directed evolution*.

1.3.2.1 Case-based Reasoning and Analogy

Case-based reasoning involves searching a library database of cases *similar* to the case represented by the qualitative model TSC is presently considering. This search may find one or several cases in which portions of a case may, by *analogy*, be found to apply to the model presently being considered.

1.3.2.2 Directed Evolution

The primary tool of discovery in our work is an exploratory machine learning technique we call directed evolution. We have applied directed evolution in three different ways during this work: (1) genetic algorithm mutation to IF-THEN rules used to predict protein structures, (2) genetic algorithm mutation to lisp-like programs being designed to predict protein structures, and (3) heuristic mutations to design rules. The general approach to directed evolution is select some member(s) of a universe of rules or objects in a knowledge base, clone and mutate, then study the results.

1.3.3 Rough Sets

Determination of the relative importance of pieces of information and the refinement of knowledge to dense, accurate representation is at the core of much AI research. A relatively new approach to the analysis of large data sets is Rough Set Theory [Ziarko, 1989]. We have implemented a version of Rough Set Theory in the TSC toolbox, and have begun to apply it to the problem of protein structure analysis.

1.3.4 Nearest Neighbor Analysis

The study of molecule structures requires the ability to perform pattern recognition on sequences of components of the structure. Amino acid sequences are found in proteins, and the recognition of sequences which result in different structures such as helices and beta sheets is germane to the prediction of protein structures. We have implemented a variant of the nearest neighbor algorithm in the TSC toolbox, and have tested it on a variety of proteins of known conformation.

1.4 APPLICATIONS — PROTEINS

Nearly all of our tool-building activities in this Ø2 project have been directed specifically at the understanding of protein structures. Our task was to select some activity in the materials domain and apply our tool-building skills to that activity; our sponsors requested that we focus our efforts on protein structures.

Proteins are the building blocks of life itself, and it turns out these tiny molecules have many properties that are interesting and potentially useful in applications other than living tissue. For example, the electro-optical properties of specific protein structures suggest applications in optical filters. The study of these properties is of current interest to our sponsors, and the tool building of this Ø2 activity supports the sponsor's work. The overall flow of protein analysis starts with the analysis of amino acid sequences, predicts the final protein structure, and ultimately designs proteins with specific desired structures. These structures may be useful in the electro-optical domain, and they may also be useful in the biotechnology domain, as new disease-fighting drugs, for example.

1.4.1 Prediction

Prediction of the secondary structure of a new sequence is performed by any of a variety of learning techniques. From the literature, approaches to the prediction of protein secondary structure have included the genetic algorithm [Unger and Moul, 1993], our own approach [LeClair et al., 1992], neural nets [Qian and Sejnowski, 1988], [Holley and Karplus, 1989], and statistical approaches which include conformational propensity parameters [Chou and Fasman, 1978].

We have explored two different approaches to the genetic algorithm, and have developed tools to perform studies based on an algorithm known as the nearest neighbor algorithm [Cost and Salzberg, 1993], [Salzberg and Cost, 1992].

1.4.2 Design

How amino acid sequences specify a protein's three-dimensional structure remains unanswered [DeGrado, et al., 1989]. One approach to gaining understanding is *de novo* design of model proteins. This approach has long been useful in designing small molecules. We have extended our protein study tools to use the process of analogy and analysis of proteins to design an experimental protein structure.

1.5 APPLICATIONS — CRYSTALLOGRAPHY

Synthesis of materials is a very old problem, as is processing them to produce a tool or an object of art. Although the limits of our knowledge about materials has increased tremendously in the last century, our means for exploring the possibilities of designing materials is only in the early stages of development.

Examples of materials problems of great interest are those associated with specific properties of biopolymers, semiconductors, and intermetallics. Optical properties are of particular interest for polymers and semiconductors, and strength and ductility are concerns for intermetallics.

The transmission electron microscope (TEM) is an important tool associated with the study of materials properties. We have explored the coupling of our discovery tools to the control of experiments with a TEM, and to automating detection of properties of materials with crystalline structures.

Chapter 2

Tool Building

The technical approach applied in ThinkAlong's research involves the coupling of computational tools to problems involving exploration of datasets. The core of our efforts centers around an artificial intelligence tool we developed in earlier work, in development since the mid-1980s, called The Scholar's Companion (TSC), as introduced in chapter 1. Its hardware and software architectures are discussed in further detail below.

In general, TSC is created to serve the user with several *behaviors*. These include model building by applying specific knowledge to some given initial conditions; general purpose encyclopedia behaviors such as using internal knowledge to answer user queries; exploratory behaviors including directed evolution, genetic programming, hypothesis formation, and assisting the user in creative design tasks; and data evaluation tools including rough set analysis and nearest neighbor analysis.

These behaviors are discussed after the introduction to TSC.

2.1 THE SCHOLAR'S COMPANION

2.1.1 Architectural Overview

TSC is constructed as a message-based object-oriented system, the architecture of which is illustrated in figure 2-1. The main data interaction with the system begins at the environment, flowing through encoders to a global message list. The knowledge base interacts with the global messages, deleting old messages and writing new ones. Some of the new messages are decoded and returned to the environment. This flow of information is an outgrowth of the "expert system" approach to artificial intelligence. TSC adds a variety of *learning* technologies to the expert system approach.

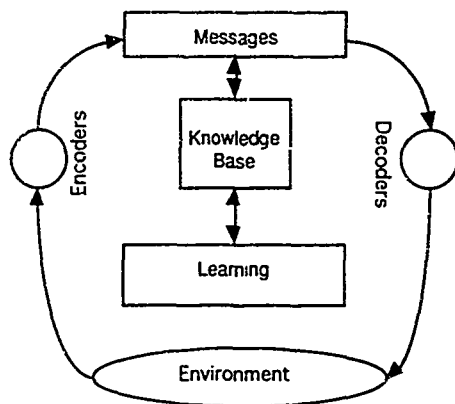


Figure 2-1: TSC Architecture.

TSC is intended to operate in networked computational environments, though it is suitable for stand-alone desktop application as well. The network approach permits application of a variety of simulation tools and large databases to the discovery process. This is illustrated in figure 2-2.

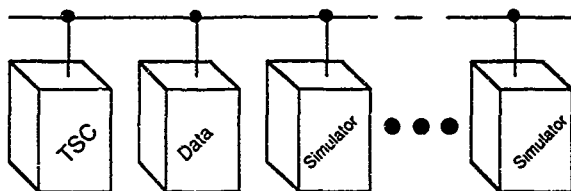


Figure 2-2: TSC on a network

Part of AI research centers around schemes for representing in a computer program the knowledge we carry in our heads, i.e. World 2 models of World 1. Another part of that research looks for algorithms which can apply the represented knowledge to some task. The following discussion of TSC's learning behaviors considers knowledge as represented as "actors," their relationships and states, and behaviors in some physical or chemical process.

TSC supports a knowledge base in a frame format. Frames represent concepts which include actors, relations, states, rules of behavior and rules which represent physical and chemical processes. Each frame is similar to a small relational database entry. For example, the following frame (adapted from [Karp, 1992]) contains much of the data on a particular molecule. It is read by forming "sentences" from its entries. For example, carbon monoxide belongs to all kingdoms. We can also see from the last line that carbon monoxide is a compound.

```
c: carbon-monoxide
display-coords-2d ((-0.77 0.0) (0.77 0.0))
structure-bonds ((2 1 3))
structure-atoms (o c)
priority 1
mesh-ids "D1.154.328" "D1.655.498.185" "D002248"
kingdoms all
chemical-formula ((c 1) (o 1))
roots carbon
sources lhcsdb mlmavro
cas-registry-numbers "630-08-0"
molecular-weight 28.01
synonym CO
atom-charges ((2 -1) (1 1))
sub.of compound
```

2.1.2 The TSC Language: Statements

TSC writes messages for use internally, and for use in communication with the user. The syntax of a typical TSC message is based on a sentence with a subject, predicate, and truth; a message can also involve a sentence with a subject, predicate, object, and truth. Variables are words which begin with an asterisk (*). These sentences are read as illustrated:

```
( predicate ( subject ) truth )
eg. ( ala ( *x ) true )
( predicate ( subject object ) truth )
eg. ( abuts ( *x *y ) true )
```

Actors are regarded as things which occupy space and have mass. Examples of statements TSC can process about actors include:


```
( thermal.mass ( body.01 ) true )
( heat.source ( autoclave.34 )true )
( b-cell ( b-cell.01 ) true )
( temperature.sensor ( t1 ) true )
```

Typical of the statements TSC can process about relationships include:

```
( hotter.than ( body.01 body.02 ) true )
( hotter.than ( autoclave.32 body.01 ) true )
( abuts ( body.01 body.02 ) true )
( inside ( t1 body.01 ) true )
( binds ( b-cell.01 antigen.04 ) true )
```

Typical statements TSC can process about states include:

```
( increasing ( t1 ) true )
( increasing ( t2 ) false )
```

2.1.3 The TSC Knowledge Base Structure: Taxonomy

The entirety of the knowledge base is stored in the form of *taxonomies*. A taxonomy contains information on a group of actors, the relationships they form with each other, and the states in which they may be detected. A fragment of a taxonomic structure for the domain of immunology is presented in figure 2-3. Here we illustrate structures related to a particular actor in immunology, the B-Cell. This cell may be detected as one of several subspecies, those behaving as activated, and those behaving as natural killer cells. The B-Cell actor is mentioned in several process rules and statements (as listed below), so the taxonomy includes this information.

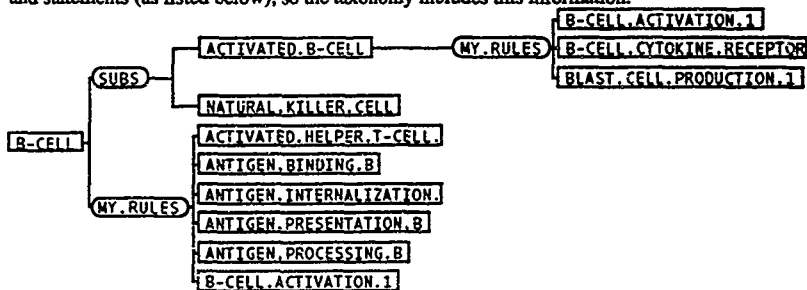


Figure 2-3. A fragment of the B-Cell taxonomy.

2.1.4 The TSC Knowledge: Rules

Physical and chemical processes (e.g. nucleation, evaporation, etc.) as well as TSC behaviors (discussed below, e.g. prediction, design, etc.) are represented as IF-THEN rules. A typical structural prediction process rule from our protein study exercise is illustrated here:

```
c: OBS.1
  worth      680
  IF actors are ALA and GLU
  AND ALA abuts GLU
  THEN predict helical structure
```

A typical process rule from a biomedical domain (immunology) looks something like:

¹Illustrated in an English-like format for readability.

```

C: ANTIGEN.BINDING.B
LEVEL          BASIC
SUB.OF         PHYS.PROCESS
INSTANCE.OF    RULE
CONTEXT        HUMAN.IMMUNOLOGY
IF.ACTORS      ( ( ANTIGEN ( *ANTIGEN ) TRUE )
                ( B-CELL ( *B-CELL ) TRUE ) )
IF.RELATES     ( ( ABUTS ( *B-CELL *ANTIGEN ) TRUE ) )
IF.NOT.RELATES ( ( BINDS ( *B-CELL *ANTIGEN ) TRUE ) )
THEN.RELATES   ( ( BINDS ( *B-CELL *ANTIGEN ) TRUE ) )

```

This rule says that if you have a B-Cell abutting Antigen, the B-Cell then binds the Antigen. Firing such rules is the process by which TSC builds a qualitative model of a process. We now turn our discussion to these models.

2.2 MODEL BUILDING

Knowledge bases, as discussed earlier, include IF-THEN rules which describe physical processes, and the taxonomic knowledge base which includes information on the actors, their relationships, and their states. These knowledge base entries are then applied to the construction of a qualitative model of some aspect of the domain represented by the knowledge base. The specific aspect of the domain is constrained by an entry supplied by the user, the initial conditions. TSC "fires" IF-THEN rules which the initial conditions enable, building an *envisionment*. A stylized envisionment for the immunology knowledge base looks something like that of figure 2-4.

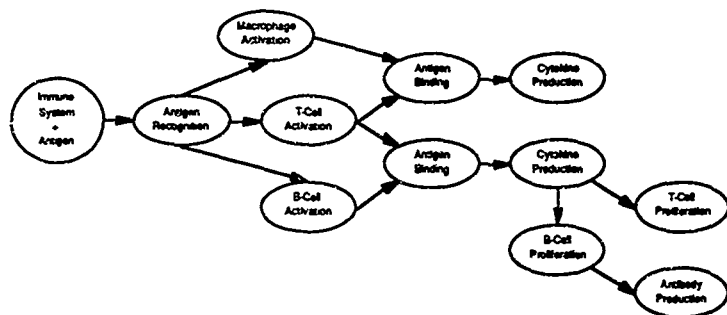


Figure 2-4: Immune system envisionment.

An envisionment is "grown" by TSC when some *initial conditions* are provided. Initial conditions represent statements about the initial actors, their initial relationships, and their starting states. A set of initial conditions may be provided by the user, or it may be supplied by TSC as it generates its own experiments. The notion of "growing" an envisionment follows on the observation that the envisionment is a tree-structured *directed graph*, also called a *digraph*.

Directed graphs consist of *nodes* connected by *arcs*, which are represented by the arrows in the figure. In our parlance, each node is called an *episode*, and each episode represents the envisioned new state of affairs in a process. Each episode is the result of a single process rule firing. Notice that alternate branches of the tree are formed, as illustrated in figure 2-4. Alternate branches mean that more than one process may occur given the same conditions.

Qualitative models in TSC provide the structure of *theories* the TSC uses to *explain* and to *predict* the outcomes of physical processes on given initial conditions. We illustrate this by discussion of the reasoning process called *abduction*, in which hypotheses may be formed from the qualitative model to explain some observation.

If a small digraph made from the observation of a physical process can be "matched" to an environment we can use abduction to hypothesize that the processes of the environment provide a *mechanism* to explain the causal relationships. For example, the digraph in figure 2-5 can be matched to a path in the environment in figure 2-4; thus we have reason to suspect that figure 2-4 describes a mechanism explaining figure 2-5.



Figure 2-5: Causal Digraph Example

Abduction in the context of TSC is a process of logic which relates consequences to their candidate antecedents. That is, if we know that some consequent B follows from antecedent A, and we observe B, then we can hypothesize that A is true. Specifically, applying abductive inference, if we know that introduction of antigen into the immune system will lead to antibody production via a mechanism that includes cytokine production (as in figure 2-4), and we "observe" causal relationships between introduction of antigen, increasing cytokine, and increasing antibodies (as in figure 2-5), then we hypothesize that the observed causality is explained by the known mechanism.

Applying this reasoning in a situation where the digraph of observations is not found to match the environment results in an important event in the use of TSC, an *expectation failure* is said to occur. Since the environment both explains and predicts, any time it is unable to explain or predict observations, the resulting expectation failure causes TSC to begin a set of tasks to deal with this new event.

Dealing with expectation failures evokes several behaviors in TSC. The initial behavior is to alert the user, and describe the nature of the expectation failure. Initial TSC behaviors are then guided by a diagnostic knowledge base which tries to determine if the expectation failure is caused by, for example, a sensor failure. More advanced behaviors involve TSC's exploration tools which allow it to conjecture the presence of an unknown (to the program) process which may be involved in the observations.

Model building involves specification of the observables. These form the initial conditions of an environment. The environment then characterizes the linkages between the observables, which TSC builds by firing process rules. Rule firing involves matching the "IF-side" of an IF-THEN rule to the current episode (the first one being the initial conditions). When a match is found, the "THEN-side" of the rule is used to build a new episode. The resulting environment is an expression of the model.

2.3 **ENCYCLOPEDIA BEHAVIORS**

TSC is designed for interactive, cooperative exploration of some environment with a user. Two aspects of the TSC system enable interaction with the user: 1) the program will read knowledge supplied by the user in a frame format or a "natural language" format, both from a text file, and 2) the program will accept knowledge supplied by the user in the "natural language" format when the user types the new knowledge in TSC's "conversation" window.

The software which parses sentences typed by the user in the conversation window will accept either statements (new knowledge) or questions. A user question causes the conversation software to attempt to generate an answer.

This conversation facility has been used to generate two knowledge bases, but has not been used in the Ø2 activity.

2.4 *EXPLORATORY BEHAVIORS*

2.4.1 Case-Based Reasoning

2.4.1.1 CBR on TSC

Case-based reasoning (CBR) is an outgrowth of artificial intelligence research. The approach enables machine learning of some environment by storing and indexing the experience provided in training exercises. This indexing builds "cases" by which the program may, during some later exercise, notice similarities between the current experience, and prior cases. Mapping a prior case to the current situation involves reasoning by analogy.

We have implemented a version of CBR in which all of the Brookhaven Database proteins serve as cases from which a new protein design may be generated. We now contrast our approach to conventional CBR techniques.

2.4.1.2 TSC Case-Based Design Approach vs. Conventional AI CBR

While the "dialect" of case-based reasoning (CBR) used to produce the proteins has similarities to traditional AI-based CBR [Riesbeck and Schank, 1989]; it also has some important differences. The comparisons are presented below. The traditional CBR approach is based on the needs of powerful reasoning systems used in story understanding, and creative activities such as design and authorship. The TSC approach is based rather strictly on the needs of molecule design. Thus, the TSC code may be considered a specialization of the traditional CBR approach. Traditional AI CBR is illustrated in figure 2-6, (after [Riesbeck and Schank, 1989]).

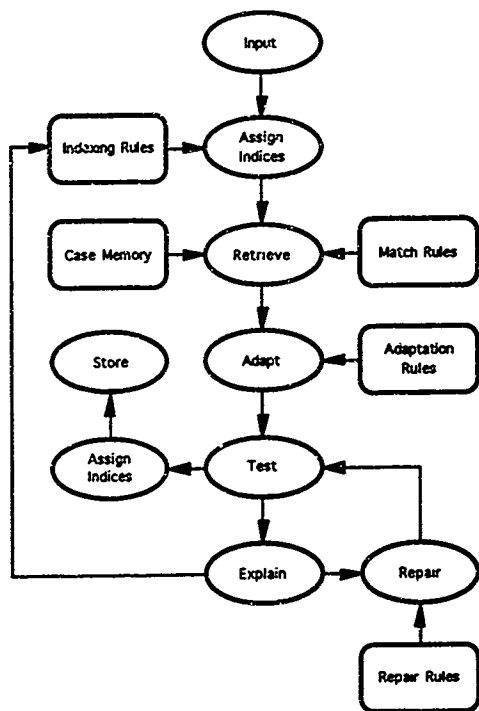


Figure 2-6: Traditional Case-based Reasoning Algorithm.

2.4.1.2.1 Similarities

- Case library used as knowledge base.

The majority of the knowledge of the protein design system is contained within the case library. The system doesn't know why a certain sequence of amino acids forms a helix, it just knows that it does. It has a set of examples of helices without ever knowing what a helix is, or how to create one from scratch.

- Solving problems includes searching a case library for examples.

When the system is asked to design a protein containing certain structures, it searches the case library for similar structures. This is analogous to searching a traditional case library for similar plans, or recipes, or events, etc. The TSC case-based design system doesn't need to know how to create a helix from scratch, because it's seen a helix before (in its case library), and knows something about what they are composed of.

2.4.1.2.2 Differences

- No adaptation of cases—a case is always used exactly as is.

A traditional CBR system will find a similar case and then adapt it to fit the current goal of the system. The TSC case-based design system will find a similar case and then plug it in to the solution without altering the original case at all.

- New proteins aren't stored as cases, because they aren't cases.

A traditional CBR system will generate new cases as it goes about its process, and will typically store them for later use. TSC presently does not generate a new case; it merely combines old cases in a new order to form a new protein.

- No sure near-term way to determine effectiveness of algorithm

Because the testing of the designs using a structure prediction algorithm is limited in its accuracy to the 50 to 70 percent range, there is no certain near-term way to determine if a design was successful or not. Therefore, there can be no immediate adaptation or learning from the system's failures. If TSC generates an incorrect protein given a certain case library and specification, it will generate that same incorrect protein every time.

2.4.2 Hypothesis Formation

In the following example, we show, drawing from the biomedical domain, that we can ask TSC to investigate the prevention of some outcome. TSC explores candidate approaches by forming conjectures on ways to prevent the outcome, and then presents those conjectures to the user.

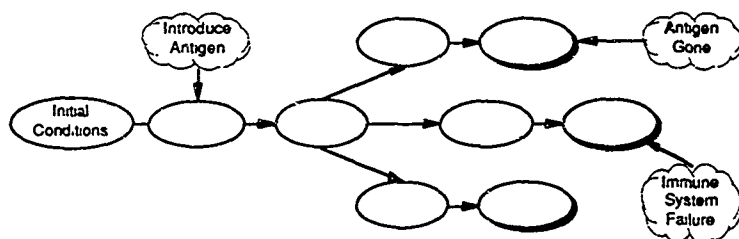


Figure 2-7 Hypothetical environment.

Figure 2-7 illustrates an immune system environment in which some antigen (disease causing agent) is introduced into the organism. We ask of TSC's exploratory behaviors how to prevent a particular outcome—immune system failure. The exploratory behaviors derive one or more conjectures in the form of proposed alternate branches of the environment tree structure, as illustrated in figure 2-8.

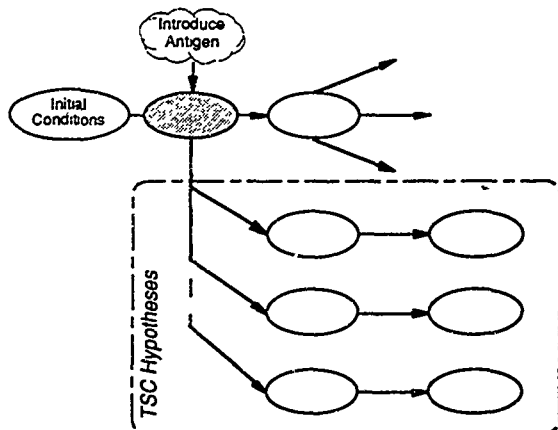


Figure 2-8: New branches on the environment.

TSC's hypotheses may include several different approaches to prevent the final outcome. For the immune example, *prophylaxis* (prevention of antigen introduction) may be proposed. Another branch may propose an "antibiotic" to inhibit the action of the antigen. These hypotheses are generated by TSC in a variety of ways. The first is to examine a database of "cases" which may be similar to the case represented by the current environment. There, the prophylaxis hypotheses may be found and applied as an alternate branch to the current environment. This is an example of case based reasoning as described above.

TSC may look through its collection of process rules looking for a process which might, by analogy, be *mapped* to the existing situation. The classic example of this (drawn from historical discussions, not from TSC's own experience) is the mapping of a battle strategy (divide the forces up and attack from different directions) to the problem of reducing or eliminating radiation burns when radiation therapy is indicated in tumor therapy.

2.4.3 Design

We have developed two different approaches to design, both of which are discussed elsewhere in this report. The first approach is our application of case-based reasoning to design a new protein by analogy to a case library of known proteins.

The second approach applies design rules to a "design environment" in which the initial conditions are a "seed" design, and design rules build an environment, always trying to improve the design. This approach requires that we supply TSC with a simulator which is capable of evaluating each new design.

2.4.4 Directed Evolution

Directed evolution (DE) is our name² for a computational approach to discovery pioneered by Douglas Lenat in his AM and Eurisko programs [Lenat, 1983]. The approach involves performing mutations to elements of a knowledge base and examining the results.

²The term is borrowed from molecular biology [Abelson, 1990].

2.4.4.1 Directed Evolution vs. Genetic Algorithms

One variant of directed evolution is based on "heuristically guided" machine learning. The other variant is based on random mutation. We have begun to develop both approaches. In the random mutation variant, TSC applies a machine learning technique adapted from the genetic algorithm (GA) [Goldberg, 1989]. Recent work has applied the GA to computational chemistry and chemometrics [Lucasius and Kateman, 1989]. Our system extends the GA to function in the traditional symbolic environments of AI.

John H. Holland seeded the creation of the GA and wrote the seminal works on the subject. (cf [Holland, 1992], [Holland et al., 1986], [Holland, 1986], [Holland, 1992], [Goldberg, 1989], [Farmer, Packard, & Perelson, 1986], and [Judson & Rabitz, 1992], and [Koza, 1992]). Holland and his work have been honored by the 1992 MacArthur Prize. His students have evolved the algorithm to its present level of power and generality. The DE algorithm is, roughly speaking, a slight generalization of the GA.

The GA serves as a guided optimization system in that, over a number of "generations," it selects elements and combines them into predictive rules similar to OBS.1 shown on page 3. Rules are then rated for their predictive accuracy during the learning exercise and successful outcomes are reinforced. Overall performance improves over time, since successful elements are allowed to survive through subsequent generations.

The select-combine algorithm mimics evolutionary processes such as crossover, point mutation, viral infection, and so forth. For example, a pair of "strong" rules (i.e. good predictors) may be selected as "parents" in a crossover breeding exercise. Actors or relations will be traded between them such that "child" rules are constructed from parts of each parent.

Directed evolution, like a genetic algorithm, is applied to a population of rules to evolve a more successful population. Success is defined externally, by way of a goal to reach. In contrast to Darwinian evolution, directed evolution and genetic algorithms employ goals such as finding a class of objects, e.g., rules, as applied to some task or problem. Evolution in the Darwinian sense has no such goal.

2.4.4.2 Algorithm

In applying the GA to any proposed activity, one maps the actors, relations, and states of the domain into classes or "gene pools." In a biochemistry study, actors might include atoms, substructures (molecule fragments), and entire molecules. Relations might include types of bonds and various structural features. Properties such as hydrophobicity may also be included. Thus, candidate descriptors covering topological, geometric, electronic, and physicochemical properties, and mode of toxic action are available for selection.

Figure 2-9 illustrates the program flow when running the GA. Given a database and an initial set of rules (typically generated at random), the system exercises rules on data, looping until all rules have been tried on all data. Following this, all rules are evaluated for their predictive capabilities: selection is made of successful candidate rules to be "parents" in a breeding exercise. By applying crossover and a variety of point mutations on individuals, a new body of rules is created which must "compete" with the prior body of rules. Rules gain or lose worth based on prediction accuracy following exercises with the data. Rules of higher worth have a higher probability of becoming parents in future trials.

This process repeats for many trials. A learning curve results from the accumulated experience of many cycles of GA evolution. Directed evolution continually seeks to improve the performance of the predictive rules. In most cases, more cycles of the learning system results in better predictive performance of the resulting rules.

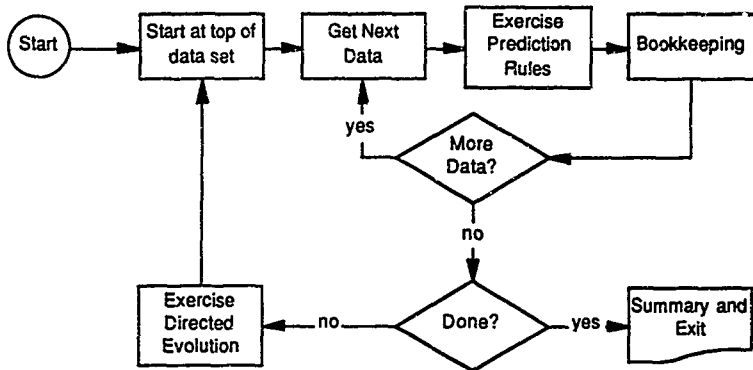


Figure 2-9: Flow of Activity in Directed Evolution.

Our "Exercise Directed Evolution" block in figure 2-9 applies heuristic guidance to the exercise of the genetic algorithm. Here, we are interested in "directing" the GA toward rapid improvement in its discovery of good predictive rules. A periodic evaluation of the various combining/mutating functions of the GA is conducted by TSC and heuristic rules in the knowledge base direct changes to the probability of occurrence of each GA function. Heuristic rules offer "suggestions"; the results from a given heuristic rule's firing may not result in anticipated improvement of the GA performance. The flow chart of figure 2-9 illustrates the cyclic nature of directed evolution: it is a search for improvements in toxicity prediction. By guiding the evolution, TSC eventually improves the performance of the GA during learning cycles. This heuristic discovery system is patterned after the Eurisko program of [Lenat, 1983] and the Hyppgene program of [Karp, 1989].

Starting with a Darwinian Evolution approach, implemented as a genetic algorithm, the system follows the guidance of a knowledge base. This coupling of heuristic guidance to a GA creates learning curves that achieve desired results in a reasonable amount of time. We present an example of this coupling later in the Mutation section.

2.4.4.3 Optimization Strategy

Because GAs are a family of iterative search algorithms, and therefore comparable to both linear and non-linear optimization techniques, it is important to understand what distinguishes GAs from conventional systems.

A universal problem associated with optimization is that, when applied, the methods are typically over-constrained by the numerous assumptions made to transform a dynamic real-world problem into a mathematical formalization. In general, optimization techniques have three difficulties: 1) depending on their search strategy they are sensitive to large or erratic noise in the data, 2) they are hampered by local performance peaks that may be unrelated to the overall maximum, and 3) their search strategy uses the slope of the function to select the next step in the search process. For the more complex problems, the local slope does not provide adequate information about the location of the maximum. This is particularly the case for nonlinear problems.

These difficulties vary and depend both on the problem and optimization method. The GA would be subject to these characteristics, but properties of the genetic algorithm mitigate them. For example, local peaks are escaped by the mutation operator (discussed below). As a consequence,

a GA is more analogous to complete enumeration than to any of the math model-based optimization techniques, i.e., a GA is largely a trial and error process involving multiple candidate solutions instead of a slope-guided process involving a single candidate solution. But in contrast to complete enumeration, the multiple candidates are only a small subset of the total number of solutions and they are evaluated in parallel (as a set of solutions).

A GA employs and combines qualitative and quantitative operators encoded as conditions in the search for a qualitative (i.e., find an instance of, find a class of, or find all) or quantitative goal (i.e., to maximize or minimize some numeric value). In comparison with other conventional optimization techniques, a GA has several advantages: 1) GAs encode the parameters which they have to optimize and base their procedure on the codes—not on the parameters themselves, 2) GAs work in parallel on a number of search points (potential solutions) and not on a unique solution, which means that the search method is not local in scope but rather looks globally at the search space, 3) GAs require from the environment only an objective function measuring the fitness score of a candidate solution, and 4) both selection and recombination steps (discussed below) are performed by using probabilistic rules rather than deterministic ones. [Renders & Norvik, 1992].

As per their biological origins, GAs imply the use of mutation as a fundamental mechanism of innovative population variation, but instead of the usual genetic material, i.e., DNA in biology, problems encoded in the form of IF-THEN rules are addressed. In addition to mutation, GAs typically rely on two additional operators called reproduction and crossover for population variation. The system discussed here does not employ reproduction, but limits population variation to crossover and mutation. Components of "if-then" rules (i.e., antecedents and consequents) serve as the genome (biological domain), and the rules themselves serve as the phenotypes. Thus, a collection of rules serve as a "gene pool" for crossover and mutation operators. We discuss those two operators next.

2.4.4.4 Crossover

Crossover is regarded in the literature as the dominant operator when compared to mutation. Using crossover, two rules are selected to produce "offspring" by exchanging a portion of their rules: IF (antecedent) subjects, objects and relations; and/or THEN (consequent) subjects, objects and relations (analogous to gene splicing). The offspring replace weaker rules in the population. Crossover serves two complementary functions. First, it provides new points for further testing within the existing problem "subspaces" (represented by the parent rules). Secondly, it introduces representative members of "subspaces" not already existing through prior crossover. The DE variant of crossover alternates the type of selection of parents between randomly selecting parents and on the basis of strength (cross those rules with the highest predictive accuracy). Given one pair of parents, two children are produced by the process. Genetic material that comprises the antecedents (IF clauses of the rules) are spliced and exchanged to make two new children. The new rules are placed in the voting "pool" of rules.

2.4.4.5 Mutation

Mutation is a secondary operator in directed evolution, and is applied with very low probability of occurrence, typically less than a few percent of the time. Its purpose is to alter the encoded value of a random position (point) on a string. Examples of point mutation are insertion, deletion, or change of some rule component. In the TSC DE system a selected rule is copied and the mutation operator, selected at random, is applied to that copy. Source of a "mutant" DNA element for insertion or change is typically a random member of the rule set. In a "viral" mutation, a DNA element is selected at random from a source pool outside the rule set.

Mutations may be guided by individual heuristic IF-THEN rules, or they may take the more Darwinian flavor of random changes. A simple example of a heuristically-guided mutation is drawn from our work in design; a design rule which has contributed successfully to the design of a vehicle is selected for mutation. It is then cloned and mutated and the results are then studied.

A stylized initial rule is:

If you want to improve the performance of the vehicle
And the vehicle includes an aerodynamic structure (e.g. a wing)
Then consider increasing span by 10%

The mutated version of the rule is:

If you want to improve the performance of the vehicle
And the vehicle includes an aerodynamic structure (e.g. a wing)
Then consider increasing wingspan by 20%

The relatively simple mutation involved increasing the *rate* by which this rule applies its "mutations" to the evolving design. This mutation was suggested by a rule of the form:

If you have a design rule which is more than XX stronger than other rules
Then consider cloning the rule
And mutate the new clone by increasing/decreasing its rate parameter
And post a task to study the new rule's performance

Directed evolution involves a *hypercycle*, that is, the design exploration process is a cyclic evolution acting on the product being designed, while the tools of evolution are, themselves, subject to the forces of evolution.

2.4.5 Genetic Programming

In an approach to directed evolution different from the rule-building discussed above, we evolve lisp-like *programs* which perform some task. Our programs are intended to predict protein structure, as discussed in the next chapter. Here, we discuss this approach to genetic programming. Our implementation of genetic programming is based on Koza [1992]. Genetic programming is used to modify a population of programs which perform tests on the data. Programs in this study are constructed of boolean expressions. The terminal values of these expressions are generated by application-specific primitive functions.

2.4.5.1 Algorithm

Create an initial (random) population of programs.

1. Execute and evaluate programs to determine fitness.
2. Rank order programs according to fitness.
3. Generate a new population of programs by applying reproduction, crossover, and mutation to the best of the old programs.

Go to step 1.

Normally this algorithm chooses the best program to appear in any generation. In our version the result is the final population of programs with non-zero fitness. These programs will be used in concert for recall and prediction.

The process begins by applying a population of randomly-generated programs to the elements of a database. Program results are placed in a matrix and evaluated to obtain a measure of each program's fitness. These fitness values are used in selecting programs to breed into the next generation. This process is illustrated in figure 2-10.

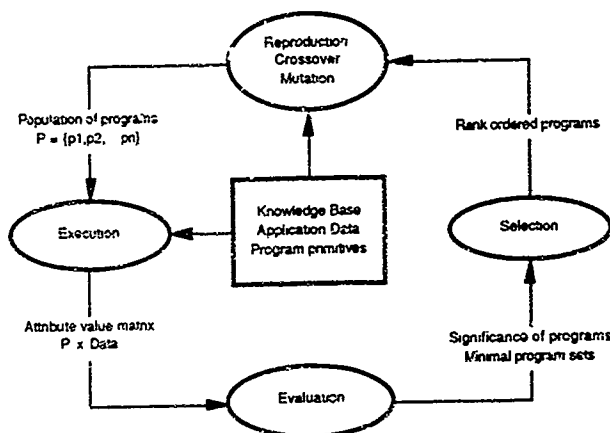


Figure 2-10: Genetic Programming.

Programs are constructed of boolean operators connecting application-specific primitives. Our implementation constructs programs using Scheme, a dialect of Lisp. This representation is easy to manipulate and directly executable in the Scheme environment.

An evaluation matrix is created by applying each program to each point in the training data. For each data point (i), and for each program (j), the program is executed and the result is stored at location (i,j) in the matrix. Thus each column of the matrix contains the values returned by a particular program for all the data points, and each row contains the values returned by all programs for a particular data point.

The evaluation matrix is used in evaluating the fitness of programs using rough set theory. We discuss rough set technology elsewhere in this report.

2.4.5.2 Survival and Reproduction

Because our implementation is searching for a good population of programs rather than a good individual program, it is convenient to separate the concepts of survival and reproduction.

2.4.5.2.1 Survival

We have experimented with three options for survival of programs from one generation to the next.

1. Retain a fixed number of the best programs.
2. Calculate the minimum SGF (significance) for all programs and retain all programs with an SGF greater than the minimum
3. Retain M, i.e. programs with $SGF > 0$. Note that M is a minimal set so it will by definition contain a diverse population

2.4.5.2.2 Program Reproduction

We have experimented with two options for production of new programs from one generation to the next.

1. Replace discarded programs
2. Supplement M by a constant number of programs. This allows P to increase or decrease according to the size of the minimal set M. Population size adapts to the problem and with progress toward a solution.

New programs are created by the genetic operators, crossover and mutation.

2.4.5.3 Crossover and Mutation

In this study the genetic operators modified only the boolean expressions. Crossover was implemented by selecting one sub-expression from each parent and swapping them to produce two new programs. Mutation was implemented by selecting a sub-expression within a single parent and replacing it with a randomly generated expression. Random replacement by the mutation operator was the only method implemented for modifying a terminal function.

2.4.5.3.1 Program Crossover

The crossover operator first selects two members of the current population of programs. A crossover point is selected within each program and the sub-expressions below these points are swapped to produce two new programs. Figure 2-11 shows two programs with sub-expressions selected.

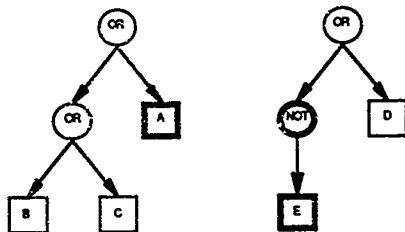


Figure 2-11: Sub-expressions selected for crossover in parent programs.

Figure 2-12 shows the two new programs created by the crossover operation.

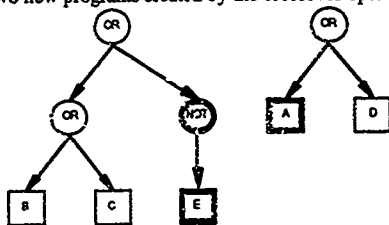


Figure 2-12: New programs created by crossover.

2.4.5.3.2 Program Mutation

The mutation operator first selects one member of the current population of programs. A mutation point is selected within the program and the sub-expression below this point is replaced with a randomly generated expression. Figure 2-13 shows an example of an original program and one possible result of applying the mutation operator.

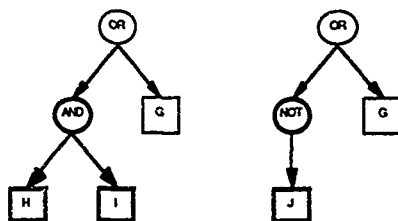


Figure 2-13: Original and new program created by mutation.

2.5 DATA EVALUATION TOOLS

2.5.1 Rough Set Evaluation of Genetic Program Fitness

Fitness evaluation in the genetic programming approach requires an *objective function* with which to perform the evaluation. As in the Darwinian algorithm, some critic—typically, the environment itself in the biological domain—is required to rate the performance of each individual. In our experimental work, we apply a *rough set* evaluation tool, based on rough classification described by Pawlak [1984,1991] and Ziarko [1989]. These methods allow the fitness of functions to be evaluated in a context with other functions. We expect this approach to promote population diversity by a natural tendency to assign lower fitness to redundant programs. To anticipate the results, our trials of rough set evaluation were used on small tasks such as developing rules which identify certain cancers or flowers. The method discussed here turns out to be far too compute-intensive to be appropriate to protein studies on our workstation-based TSC. We do believe that these tools, when ported to a super-computer, will provide useful results.

These properties are particularly important for our applications where we are trying to find a population of programs that perform well in concert. This is in contrast to typical genetic programming applications where the goal is to find a single program which performs well.

2.5.1.1 Definitions

$S = (U, A, V, f)$ is an information system consisting of a set of data objects (U), a set of attributes (A), a set of possible attribute values (V), and a function that maps data objects and attributes to attribute values (f).

U is the set of all data objects. In our insect application it is the set of all iris examples in our database.

A is the set of all attributes. In our implementation it is the set of concepts measured or tested by the attribute programs. The set A also corresponds to the set of all columns in the evaluation matrix.

V is the domain of attribute values. In our insect application it is {setosa, virginica, versicolor, true, false}.

f is the description function mapping $U \times A \rightarrow V$. In our implementation it corresponds to the evaluation matrix.

N is the set of predicted attributes or columns of the evaluation matrix. In our application it is a single column containing the species name for the data objects.

P is the set of predictor attributes or columns of the evaluation matrix. These correspond to the set of programs to be determined.

M is a minimal subset of P which retains the full ability of P to discern elements of N .

N' is the set of elementary sets or equivalence classes based on predicted attributes. It corresponds to the set of sets of rows which have matching values in all the N columns.

P' is the set of elementary sets or equivalence classes based on all predictor attributes. It corresponds to the set of sets of rows which have matching values in all the P columns.

M' is a set of elementary sets or equivalence classes based on a minimal set of predictor attributes. It corresponds to the set of sets of rows which have matching values in all the M columns.

$\text{ind}(P)n'$, is the union of all elementary sets in P' which are subsets of the i th element in N' . These are the lower approximations of the sets in N' .

$\text{POS}(P,N)$ is the union of $\text{ind}(P)n'$, for all elements n' , in N' . This is the subset of U for which P is sufficient for discerning membership in the equivalence classes of N' .

$k(P,N) = \text{card}(\text{POS}(P,N)) / \text{card}(U)$

This is the fraction of the set U for which P is sufficient for discerning membership in the equivalence classes of N' .

$\text{SGF}(P,N,p) = (k(P,N) - k(P-p,N)) / k(P,N)$.

This is the significance value; i.e., the relative change in $k(P,N)$ resulting from deletion of p from P . SGF has the advantage that it rewards programs for their contribution to recall or prediction in the context of all other programs.

2.5.1.2 Evaluating Attributes

The sets and measures above are used to determine the significance of members of P for classifying members of U into the equivalence classes of N' . One goal is to find a minimal subset of attributes that retains the capability of all P for discerning elements of N' . Another goal is to assign a significance value to each attribute. This value will be used in calculating program fitness for the genetic programming process.

The following method calculates significance factors for each program while determining a minimal set. Programs with zero significance are not included in the minimal set, M .

Initialize: $M_1 = P$.

For each pk in P :

Calculate $\text{SGF}(M_k, N, pk)$

If $\text{SGF}(M_k, N, pk) \neq 0$

then $M_{k+1} = M_k - pk$

else $M_{k+1} = M_k$

There may be several minimal sets. The order of selecting pk during calculation of M will affect the final contents of M . We have chosen to select the pk beginning with the lowest-SGF members of the previous generation. This encourages turnover in the population by allowing an older program to be replaced by an equivalent set of new programs.

2.5.2 Nearest-Neighbor Pattern Recognition

The technology of pattern recognition with the *nearest neighbor* technique is primarily applied to our protein structure prediction project. Briefly, a large body of exemplars is created based on a library of proteins and is stored in computer memory. Each exemplar represents a window of data which slides along the amino acid sequence of any given protein. The windowed data is combined, in the exemplar, with a prediction of the structure, derived directly from the given protein. This collection of exemplars is then applied to the prediction of structure when presented with a window of data from the new protein. The exemplar with a window of data nearest to the window of data from the new protein "wins" the right to offer its prediction. This prediction process is repeated for all data windows in the new protein.

The approach used here is to collect the exemplars from a group of proteins which are not selected from the design case library. In general, we develop the algorithm by training on a large collection of proteins, then test the algorithm on a different collection of known proteins. Once the algorithm has been "tuned," it is next applied to the testing of new proteins.

The TSC code for structure prediction, which is applied to the designed protein, is based on an algorithm documented in [Cost and Salzberg, 1993], [Salzberg and Cost, 1992]. The program PEBLS (Parallel Exemplar-based Learning System) is reported to achieve accuracies in prediction of protein secondary structure as high as 71%. The algorithm is closely related to the memory-based reasoning approach of [Zhang et al., 1992].

The PEBLS approach (after [Salzberg and Cost, 1992]) as implemented in this work is as follows: given a sequence of residues from a fixed length window from a protein chain, classify the central residue in the window as helix, sheet, or coil. The table below (after [Salzberg and Cost, 1992]) compares the correlation coefficients from PEBLS and a variety of other algorithms.

Algorithm	%correct	C α	C β	Ccoil
PEBLS	71.0	0.47	0.45	0.40
Zhang et al. 1992	66.4	0.47	0.387	0.429
Qian & Sejnowski 1988	64.3	0.41	0.31	0.41
Holley & Karpus 1989	63.2	0.41	0.32	0.46

The TSC code applies an algorithm patterned after PEBLS, but does not presently include the weighting scheme used by PEBLS. Our simplified approach requires a single pass through a training set of 66 to 91 proteins selected for non-intersection of training proteins with the protein set used for case-based design. During this first pass, tables are constructed that contain the distances between amino acid values. As explained in [Salzberg and Cost, 1992], distance is estimated statistically; distance between amino acids is a sum over three classes (helix, sheet, coil). The sum is based on the number of times residue 1 was classified into a particular category, the total number of times that residue occurred, and the number of times residue 2 was classified into a particular category, and the total number of times that residue occurred. Values should be similar if they occur with the same relative frequency for all classes.

A different table is built for each position in the data window. Thus, if window length is specified as, say, 17, then 17 tables are constructed. Exemplars are constructed from each window presented by the training set. Each exemplar is stored in the form: ((*window*) *prediction*). An example follows: ((ALA PRO LYS...) A) where "A" is the prediction. If there are, say, 500 windows of residue sequence data in a training set, there will be 500 exemplars created. [Cost and Salzberg, 1993] report that the performance of their algorithm improves with increasing window length to a peak at a length of 19. The TSC code has typically applied a window length of 17.

3.1 *PROTEIN STRUCTURE PREDICTION*

Our work has focused on prediction of protein structures from a given amino acid sequence, and to design a protein sequence that will yield a given structure. We use our directed evolution and genetic programming techniques, along with the nearest neighbor analysis. We then use case-based reasoning to design a protein.

3.1.1 Methodology

Consider the evolution of a set of rules which are successful at predicting the structure of a protein when given certain information about that protein. The application is quite similar to the application of a genetic algorithm in chemometrics [Lucasius, 1989], in which prediction of the conformational analysis of DNA molecules was studied. Unlike Lucasius and Kateman, the gene information is much broader than just a few parameters such as the order of the nucleotides in the DNA sequence, bonding distances and bonding angles. The approach taken here acknowledges the need for many more parameters as suggested by Lozano-Perez in an article by Erickson [1992] as follows: "We're finding you need more like 100 data points to characterize a molecule properly." Attributes such as molecular charge and hydrophobicity add dimensionality that is difficult for humans but simple for computers to consider.

We apply two variants of genetic algorithms in our DE tools: (1) mutations guided by random selection, and (2) mutations guided by heuristic rules. Both were illustrated above. We then apply DE to DE itself. This enhancement uses heuristics to notice current performance levels of predictive rules and alter the breeding and/or mutation methods to allow more successful populations of rules to "gain a foothold" and begin performing. Once performance achieves a predetermined level, breeding/mutation methods are again altered by DE heuristics to either breed more generalized or more specialized rules. In both cases, we are building a set of rules which perform prediction.

As noted earlier concerning Darwinian Evolution, typically a GA evolves rules in a dynamic (continually changing) environment—an environment Holland describes as a generator of "perpetual novelty" or concept drift. The TSC DE algorithm is also capable of dealing with the concept of drift; however the protein structure problem is a static environment—an environment in which the protein training sets do not change with each learning cycle.

Directed evolution develops a population of rules intended to predict the presence of helical structures in a protein when given the amino acid sequence. Initially, at system startup, the DE randomly produces many general rules for predicting a helix, attempting to fill every (candidate solution) niche in the environment. When the environment is sufficiently seeded, the DE begins evaluating those rules. Subsequently, the randomly generated startup rules are bred based on rule performance. A number of search parameters can be adjusted by the DE rules; changes to these "knobs" may change the size and performance of the gene pool, and may alter the probability of any given GA strategy.

A typical protein represented in a TSC-readable form looks like the following frame:

```

C: PDB3B5C
name CYTOCHROME.B5
instance.of protein
functionality electron.transport
tertiary.structure small.ss.rich.or.metal.rich-met-
al.rich-up.down.ligand.cages
source bovine.liver
MY.DATA ( SER LYS ALA VAL LYS TYR TYR THR LEU GLU GLU ILE GLN
LYS HIS ASN ASN SER LYS SER THR TRP LEU ILE LEU HIS TYR LYS VAL TYR ASP LEU
THR LYS PHE LEU GLU GLU HIS PRO GLY GLY GLU GLU VAL LEU ARG GLU GLN ALA GLY
GLY ASP ALA THR GLU ASN PHE GLU ASP VAL GLY HIS SER THR ASP ALA ARG GLU LEU
SER LYS THR PHE ILE ILE GLY GLU LEU HIS PRO ASP ASP ARG SER LYS ILE THR LYS
PRO SER GLU SER )
HELIX.POSITION ( ( 9 12 ) ( 33 38 ) ( 44 47 ) ( 53 60 ) ( 65 71 ) (
81 86 ) )
SHEET.POSITION ( ( 5 7 ) ( 21 25 ) ( 27 32 ) ( 51 54 ) ( 74 80 ) )
TURN.POSITION ( ( 17 21 ) ( 24 28 ) ( 39 42 ) ( 49 52 ) )

```

3.1.2 Genetic If-Then Rule Generation

The problem is to predict the presence of helical structures in a protein when given its amino acid sequence as illustrated. One of TSC's methods is to build a population of IF-THEN rules when given a "genome." The genome is built from three "chromosomes," the working components of an observer rule. A pair of observer rules follows:

```

c: OBS.1
worth 680
if.actors ( ( ala ( *x ) true ) ( glu ( *y ) true ) )
if.relations ( ( abuts ( *x *y ) true ) )
then.predict ( ( helix ( window ) true ) )

c: OBS.2
worth 560
if.actors ( ( leu ( *x ) true ) ( glu ( *y ) true ) )
if.relations ( ( abuts ( *x *y ) true ) )
then.predict ( ( helix ( window ) true ) )

```

The three chromosomes are:

- actors
- relations
- predictions

Actors are comprised of the twenty natural amino acids:

Alanine	Arginine	Asparagine	Aspartate
Cystine			
Glutamine	Glutamate	Glycine	
Histidine			
Isoleucine			
Leucine	Lysine		
Methionine			
Phenylalanine	Proline		
Serine			
Threonine	Tryptophan	Tyrosine	
Valine			

Relations are primarily structural or spatial in this example:

```

abuts      A*B          (abuts (A B) true)
abuts-1    A*X*B        (abuts-1 (A B) true)
abuts-2    A*X*Y*B      (abuts-2 (A B) true)

```

The two predictions, used as votes by a population of rules, are:

```

( helix ( window ) true )
( helix ( window ) false )

```

An example sequence for the protein ACYLTRANSFERASE (after [Gibbs & Leslie, 1990]) looks like the following:

```

MET ASN TYR THR LYS PHE ASP VAL LYS ASN TRP VAL ARG
ARG GLU HIS PHE GLU PHE TYR ARG HIS ARG LEU PRO CYS
GLY PHE SER LEU THR SER LYS ILE ASP ILE THR THR LEU
LYS LYS SER LEU ASP ASP SER ALA TYR LYS PHE TYR PRO
VAL MET ILE TYR LEU ILE ALA GLN ALA VAL ASN GLN PHE
ASP GLU LEU ARG MET ALA ILE LYS ASP ASP GLU LEU ILE
VAL TRP ASP SER VAL ASP PRO GLN PHE THR VAL PHE HIS
GLN GLU THR GLU THR PHE SER ALA LEU SER CYS PRO TYR
SER SER ASP ILE ASP GLN PHE MET VAL ASN TYR LEU SER
VAL MET GLU ARG TYR LYS SER ASP THR LYS LEU PHE PRO
GLN GLY VAL THR PRO GLU ASN HIS LEU ASN ILE ALA ALA
LEU PRO TRP VAL ASN PHE ASP SER PHE ASN LEU ASN VAL
ALA ASN PHE THR ASP TYR PHE ALA PRO ILE ILE THR MET
ALA LYS TYR GLN GLN GLU GLY ASP ARG LEU LEU LEU PRO
LEU SER VAL GLN VAL HIS HIS ALA VAL CYS ASP GLY PHE
HIS VAL ALA ARG PHE ILE ASN ARG LEU GLN GLU LEU CYS
ASN SER LYS LEU LYS

```

Observer rules are exercised on segments of a natural system as read from a data base called windows of sequence data. During learning, the window is shifted along the data. An example window of data, with a window size of five amino acids, looks like:

```
[ ala arg gly ala pro ]
```

TSC encoders write a body of statements about the window:

```

(ala (ala.1) true) (arg (arg.1) true) (gly (gly.1) true)
(ala (ala.2) true) (pro (pro.1) true)
(abuts (ala.1 arg.1) true) (abuts-1 (ala.1 gly.1) true)
(abuts-2 (ala.1 ala.2) true) (abuts (arg.1 gly.1) true)
...

```

All rules are then exercised (allowed to vote) on this encoded window. This voting is repeated as the window is "slid" along the entire data set. A reward/punishment algorithm—part of the directed evolution component of TSC—then examines the performance of the individual rules which cast a vote. Following the "bucket brigade" algorithm of John Holland [Holland, 1986], those rules which participate in the vote, and which vote correctly, get a reward (their worth is increased). Thinking of a given rule and the source (parents) of that rule as a "bloodline," additional reward is bestowed upon the source of the successful rules.

Once rewards have been passed to appropriate rules, a small decay (reduction of worth) of all rules is computed. This has the effect of punishing those rules which do not participate in the vote, or which vote incorrectly. Rules whose worth falls below a specific value are eliminated.

At this point the directed evolution component, with its genetic algorithm, mutates the rule population and conducts a search for the optimum rule set. For example, using as parents OBS.1 and OBS.2 listed above, "sexual recombination" builds a child that looks like the following:

```

c: OBS.3
my.source      obs.1 obs.2
my.creator     crossover.1
worth          200
if.actors      ( ( leu ( *x ) true ) ( ala ( *y ) true ) )
if.relations   ( ( abuts ( *x *y ) true ) )
then.predict   ( ( helix ( window ) true ) )

```

This "child" rule is added to the population of rules and given a starting worth value. Now, consider the effect of a point mutation on the rule OBS.3 to make a new rule OBS.4.

```

c: OBS.4
my.source      obs.3
my.creator     point.mutate.2
worth          200
if.actors      ( ( leu ( *x ) true ) ( ala ( *y ) true ) )
if.relations   ( ( abuts ( *x *y ) true ) )
then.predict   ( ( helix ( window ) false ) )

```

This rule is essentially the same rule as its source, OBS.3, except that it votes a different way. If the rule is successful, it will eventually replace its source in the rule population.

To summarize directed evolution, using a biological metaphor, we see that:

- The strongest rules get to breed
- Successful rules get fed well
- Parents of successful rules get treats
- All rules age

3.1.3 Prediction Program Generation

We turn our discussion from generation of IF-THEN rules capable of predicting the conformation of a protein from its amino acid sequence, to the generation of lisp programs which are capable of the same predictions. In this, we use the genetic algorithm to evolve programs. As it turns out, the approach we explore is vastly too compute-intensive for protein studies. Our preliminary efforts centered, instead, on the development of the approach, which we discuss now. The approach has sufficient merit that it should be ported to a super-computer and tested on proteins.

3.1.4 Testing Recall and Prediction

We have tested our implementation with a relatively small database of iris flowers (Fisher's iris data reproduced in [Salzberg, 1990]). Each entry includes the name of the species and four values for sepal length, sepal width, petal length, and petal width.

We tested for both recall and prediction. The data was partitioned into two disjoint sets for training and prediction testing. Recall was tested using a subset of the training data. Prediction was tested using data which was not used in training. Testing was accomplished as follows:

1. Each program in the minimal set, M , is applied to the test data point.
2. The resulting list of values is matched against the corresponding values in each row of the evaluation matrix. We use an analog of hamming distance to select a matching row for prediction. We calculate the distance as the sum of $SGF(M,N,m)$ for columns m that do not match the corresponding test value. The row with the minimum distance from the list of test values is selected. If several rows have the same distance measure then the first of these is arbitrarily

selected.

3. The attribute values to be recalled or predicted are retrieved from the N columns of the selected row. Success is measured as the fraction of the test data that is correctly recalled or predicted.

3.2 PROTEIN DESIGN

Protein design generally refers to *de novo* approaches for new proteins. The process starts with first principles and attempts to design model proteins from scratch. Because the design process starts from scratch, small homology with native sequences is expected in the *de novo* approach. The approach critically tests the designer's understanding of protein structure.

A number of recent experiments have explored variants along the *de novo* theme, including [DeGrado, et al., 1989], [Regan and DeGrado, 1988], [Wendoloski and Salemme, 1992], [Fedorov, et al., 1992], [Hecht, et al., 1990].

For example, the design strategy of [Hecht, et al., 1990] is to use natural structural motifs to design sequences that are native-like in pattern and composition, are locally non-repetitive, and are not homologous to any known protein. This is a kind of "design by analogy." The protein created by this strategy is called "Felix," a four helix bundle protein. We present a sketch of that protein later, together with a discussion of an approach to the re-creation of the secondary structure of Felix using our TSC system for case-based design, also referred to as design by analogy. We now discuss that effort.

3.2.1 Case-Based Protein Design

We have selected a case-based (c.f. [Riesbeck and Shank, 1989]) approach to protein design. This approach suggests the ability to "learn" directly from the database supplied by nature. The alternative is *de novo* design approaches, which require a knowledge base strong in protein folding first principles. Coupling of the case-based approach with the discovery of *de novo* design rules is suggested as an important extension of this work. We apply our nearest neighbor prediction algorithm to analyze the designed proteins. Our approach to design, analysis, and evaluation of proteins is diagrammed in figure 3-1.

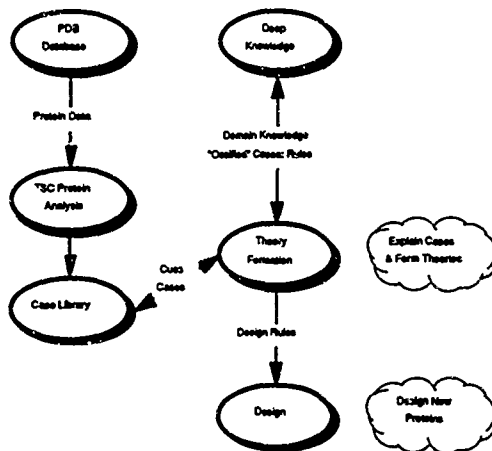


Figure 3-1: General TSC Case-Based Protein Design.

3.2.2 Approach

In a big picture, our approach is:

Given a library of proteins and a design declaration:

Search for all candidate structures similar to structures in the desired design

Find the combinations of structures with a "best fit"

Publish the results

The TSC case-based code is a program that will design proteins of a given secondary structure, using a model from case-based reasoning. An overview of this process is depicted in figure 3-2. By starting with a database of proteins (a case library) whose structure is known, the system finds, by indexing and analogy, appropriate sequences of amino acids needed to produce desired structures. The design process then becomes a process of merely "putting together the pieces."

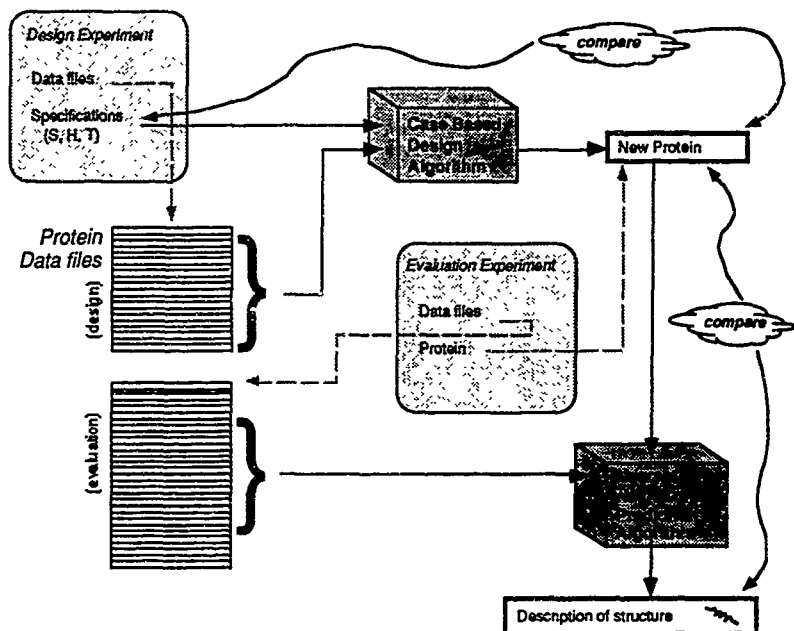


Figure 3-2: Architecture for protein design and evaluation

Although the TSC system uses a moderately extensive database of amino acids and their properties, this database is used only to refine the design process. The system currently uses a simple template matching process in order to do its first-pass design, and then uses the amino acid database to select options from that first design. To accomplish this task, the system needs to be able to derive answers to the following questions:

1. What structures have I seen next to each other in a protein?
2. What was the size of those structures?
3. What amino acids were involved in creating those structures?
4. What are the structures, neighbors and sizes involved in the protein to be designed?

Furthermore, the system should be able to accommodate proteins that have multiple subunits. A protein is said to have multiple subunits when it contains two or more disconnected sequences. The system needs to be able to access all of these sequences, but also to know that they are physically disconnected. To accomplish these goals, the system searches protein database frames (database entries), each of which holds several important components. Each protein frame contains (at least) the following pieces of information: an ordered list of the amino acids that compose it, and, for each possible structure, an indication of where (if at all) the structure occurs in this particular protein.

To more accurately illustrate this, we present an example of a protein database frame here, adapted from the Brookhaven PDB. Note the list of amino acids, and the indications of where helices, sheets, and turns are found.

```

C: PDB385C
   name          CYTOCHROME.B5
   instance.of   protein
   functionality electron.transport
   tertiary.structure small.ss.rich.or.metal.rich-met-
al.rich-up.down.ligand.cages
   source        bovine.liver
   MY.DATA       ( SER LYS ALA VAL LYS TYR TYR THR LEU GLU GLU ILE GLN
LYS HIS ASN ASN SER LYS SER THR TRP LEU ILE LEU HIS TYR LYS VAL TYR ASP LEU
THR LYS PHE LEU GLU GLU HIS PRO GLY GLY GLU GLU VAL LEU ARG GLU GLN ALA GLY
GLY ASP ALA THR GLU ASN PHE GLU ASP VAL GLY HIS SER THR ASP ALA ARG GLU LEU
SER LYS THR PHE ILE ILE GLY GLU LEU HIS PRO ASP ASP ARG SER LYS ILE THR LYS
PRO SER GLU SER )
   HELIX.POSITION ( ( 9 12 ) ( 33 38 ) ( 44 47 ) ( 53 60 ) ( 65 71 ) (
81 86 ) )
   SHEET.POSITION ( ( 5 7 ) ( 21 25 ) ( 27 32 ) ( 51 54 ) ( 74 80 ) )
   TURN.POSITION  ( ( 17 21 ) ( 24 28 ) ( 39 42 ) ( 49 52 ) )

```

In addition, a target protein is needed; in order to create a new design, TSC needs to have a design specification. The user creates an experiment frame which contains the description of the protein along with several parameters used by the system. The following is an example of an experiment frame used in this design exercise.

```

C: EXPERIMENT.42
   INSTANCE.OF   EXPERIMENT
   WORTH         500
   CONTEXT       PROTEIN
   DATA.SOURCE  PDB.DATA
   DATA.FILES   ( PDB1HDS PDB2LTH PDB3HHB PDB2DHB PDB1FDH PCB1LDS
PDB1PYP PDB1FC2 PDB1TGS PDB2CCY PDB2CA2 PDB2CAB PDB3SGB PDB1SGT PDB1PPD
PDB1HNE PDB2STV PDB1GCR PDB1MBO PDB1MBS PDB2CDV PDB1CY3 PDB1LZT PDB1FX1
PDB1CC5 )
   ACTOR.SOURCE  NATURAL.AMINO.ACID
   TEST.ATTRIBUTES ( POLARITY MOLECULAR.WEIGHT SIZE.OF.SIDE.CHAIN
SIDE.CHAIN.MUTABILITY HYDROPATHY )
   TEST.WEIGHTS  ( .25.10.20.10.35 )
   OVERLAP       1
   SIZE          65
   2.STRUCT      ( ( 4 15 ) ( 41 49 ) ) ( H ( 16 28 ) ( 32 40 ) ) ( T ( 49 54 ) )

```

3.2.3 Case-Based Design Algorithm

With all this data in hand, the target protein can be designed. The structure of the case-based design algorithm is as follows:

1. Initialize the system by loading the protein database, and analyzing it to produce the information used by the design algorithms
2. Analyze the target protein to produce the data structures necessary for design.
3. For each structure in the target list, do the following:
 - a. Search for a known structure(s) that has the same structures both before and after, and differs in length by fewer than 3 amino acids. If several can be found that are equally close in length, store all of them, giving preference to longer structures.

- b. If none can be found with the same neighboring structures, try to find a known structure that has similar structures both before and after, and that differs in length by fewer than 3 amino acids. Again, keep all that are equally close, with preference to longer ones.
 - c. If none can be found with similar neighbors, try to find a known structure of closest length with any neighboring structures. Keep all that are equally close, with preference to longer ones.
 - d. Get the data for each structure from its native protein, including one amino acid on either side. If there were several possibilities, get the data for all of them.
4. At this point, we have a list of possible choices for each structure in the protein. For each possible neighboring pair of structures, compare the overlapping amino acids, and determine their difference score, as explained in section 2.1.4. Store this data.
 5. Find the combination of possible structures that has the lowest total difference, compile the list of structures into a protein, and fill in the slots of the new protein frame appropriately.

For the new protein, it may be true that the system could not find an exact match for a given structure, and resorted to a structure of slightly different length. In the resulting protein frame, the system notes the position of all structures as they are placed by the design algorithm. These positions may not be the same as those in the target specification.

In placing structures, the system compares the similarity of the ends of the structure with the ends of its neighbors. For instance, if the system is trying to place a helix between two random coils, it will remember not only the amino acids that compose the helix, but also the first amino acid out of the helix on each end. Then, the last residue in the helix and the first residue in the coil from each grabbed helix are compared with the last residue in the helix and the first residue in the coil from each grabbed coil. This comparison is repeated for every junction between structures, and the overall configuration with the lowest total difference is selected for the final design.

For example, consider that we want to design a protein containing a helix that connects to a turn, and we have helices with the following endings:

... GLY TRP ALA, ... VAL CYS VAL, ... ALA LEU VAL, ... LEU VAL THR, ... ARG THR
GLY

The last amino acid in each group above is actually the first amino acid of the turn. We also have one turn, with the following beginning structure: ALA ILE THR..., where the alanine is the last amino acid of the helix. The system will compare the overlapping region for each possible combination (TRP-ALA and ALA-ILE, CYS-ALA and VAL-ILE, LEU-ALA and VAL-ILE, etc.), and note the difference score for each choice. It will then look at every possible combination of structures to make up the entire protein, and then choose the structure that has the fewest differences between neighbors. Consider the following illustration of this configuration:

<-helix> <-turn>			
... GLY	TRP	ALA	possible helix
... VAL	CYS	VAL	.
... ALA	LEU	VAL	.
... LEU	VAL	THR	.
... ARG	THR	GLY	.

ALA	ILE	THR...	possible turn
-----	-----	--------	---------------

The use of only one extra residue at each end of a structure is arbitrary, and might be more

effective were there to be two or more compared. An interesting direction for future work might be to compare the effectiveness of using only one extra residue with using more than one.

To determine the difference score of adjacent structures, the following algorithm is applied:

1. Determine which amino acid attributes (molecular weight, polarity, etc.) will be used to determine the difference, and gather their possible values. Each attribute will have a list of discrete possible values. (The attributes are supplied by the user, but the algorithm gathers the values.)
2. Determine the weight of each attribute as given by the user.
3. For each amino acid in the overlapping region, for each attribute under test, calculate the difference value as shown below
4. Sum the difference for all amino acids in the overlapping region to determine the difference score.

For example, the possible values for molecular weight are: very light, light, medium, heavy, and very heavy. We interpret these as evenly spaced numerical values (e.g. 1 through 5) and define the individual attribute difference as:

$$\text{Dist}_{\text{attr}} \equiv \frac{|\text{Val}_1 - \text{Val}_2|}{\text{Max} - \text{Min}}$$

We then define the amino acid difference value as:

$$\text{Dist}_{\text{total}} \equiv \sum_{\text{attributes}} \text{Dist}_{\text{attr}} \times \text{Weight}_{\text{attr}}$$

The particular attributes and weights used in our initial evaluation (which can be seen in the description of the EXPERIMENT.42 experiment shown on page 8) were chosen "seat-of-the-pants," and have no particular theoretical justification. It would be an interesting future project to model the effects of different choices of these parameters on the accuracy of the resulting design.

Consider this example of the frame of a designed protein. This example is derived, by design, from the EXPERIMENT.42 task listed above.

```
C: CON_1
  INSTANCE.OF      PROTEIN
  SOURCE           BBT.CREATE.PRO
  SHEET.POSITION   ( ( 5 16 ) ( 41 48 ) )
  HELIX.POSITION   ( ( 17 29 ) ( 32 40 ) )
  TURN.POSITION    ( ( 49 54 ) )
  MY.DATA          ( ILE PRO GLU TYR ARG GLY SER THR THR GLY THR HIS SER
GLY SER VAL GLY PHE VAL GLY ALA SER TYR VAL PHE ALA LEU MET ASN ASP PHE LEU
PHE PRO PRO LYS PRO LYS ASP THR LEU LYS ALA ASN VAL PRO PHE VAL ASP TRP ARG
GLN LYS GLY PRO PRO ALA SER PRO LYS ALA ASP ALA PRO ILE )
```

The structures found in CON_1 can be compared to the specifications in EXPERIMENT.42 (the contents of 2.STRUCT). Note that the structures are not in exactly the positions intended by the designer, but they are. TSC operates under the assumption that random coils are the least important structure. . . . cause of that, they can be shrunk or expanded to make up for errors introduced in the positioning of previous structures. This is why the second helix and sheet are in

the proper position, even though the first helix and sheet were not.

3.3 TEM CONTROL

3.3.1 TEM Control Approach

The general architecture of our approach is illustrated in figure 3-3. In this approach, TSC, with its discovery behaviors and a new knowledge base created for design and control of TEM experiments, is coupled to both a scope, and to a scope simulator. Work to date has coupled TSC only to a scope simulator. Future work may complete the coupling to a live scope.

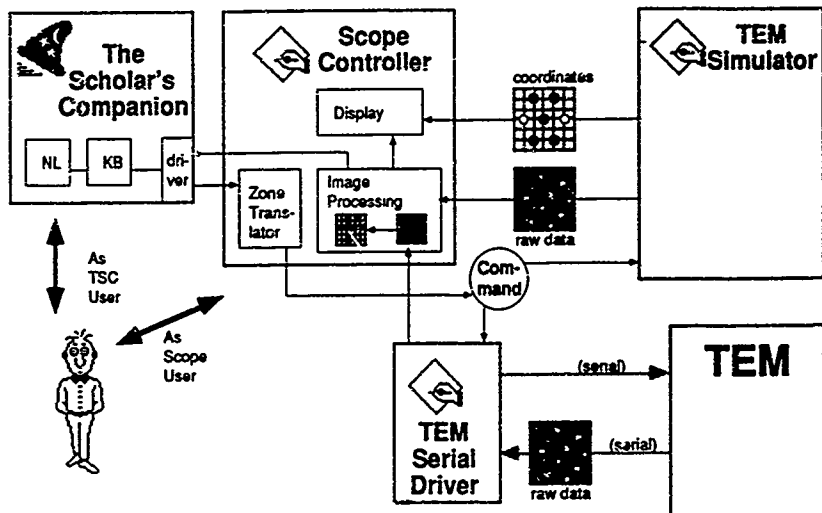


Figure 3-3: Overview of scope control.

3.3.2 The TSC Combined Analysis/Controller

The generalized approach to our TEM controller involves analysis of crystal patterns yielded by either a scope or a simulator. Figure 3-4 illustrates that our approach will combine both analytical algorithms and case-based studies of crystals. We believe that this will enable the TSC system to more-rapidly and accurately identify crystal structures by considering cases from its experience, and relying on industry-standard analytical techniques when cases fail to explain a detected crystal structure.

Figure 3-5 is a flowchart of the analytical approach taken by the crystal analysis routines written for TSC.

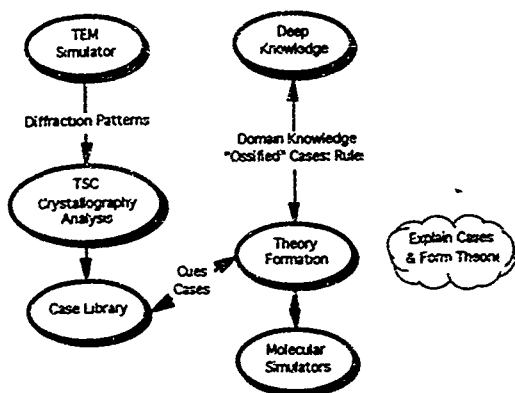


Figure 3-4: Approaches to studying crystals.

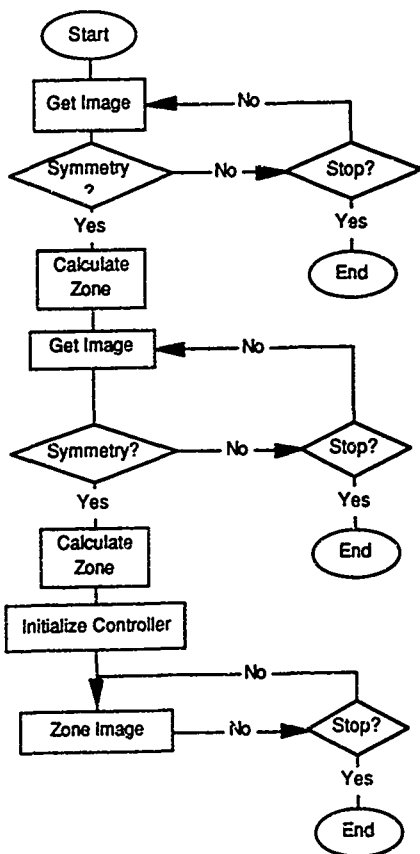


Figure 3-5: Flowchart of analysis

Chapter 4

Results

4.1 TOOLS

4.1.1 Discovery Tools

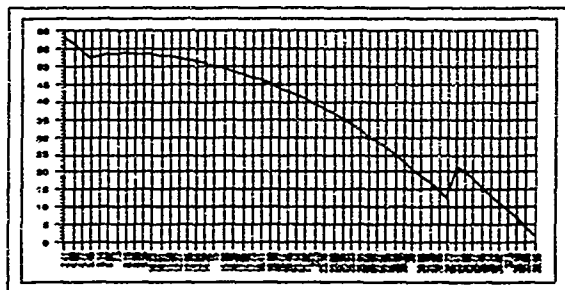
Source codes for the nearest neighbor pattern recognition and rough set evaluation packages are included in an appendix to this report. Source code for the genetic rule builder is also included.

4.1.2 Design

Our directed evolution work with TSC inspired a design activity as a means to develop and extend the capabilities of the directed evolution approach. It makes sense that design be explored since all of materials science and engineering involves design. We shall return to materials design in a discussion of protein results below; for now, we illustrate a different exercise in design using the TSC directed evolution approach. This project enabled the development of the directed evolution approach to design.

The task was to design a very fast sailboat [Park, 1993], one with an unconventional configuration which would maintain contact with the water, and use a wind to hit speeds greater than 60 miles per hour. To do so, we applied directed evolution to the task of evolving a design, given an initial design, and we applied directed evolution to the task of evolving the design rules themselves. This was illustrated earlier.

The approach was to use the envisionment building tools to evolve the design—an envisionment of possible designs grew out of the exercise. Periodically, a mutation rule fires and mutates a design rule. A number of design episodes are created, some applying the new rule. All rules are evaluated according to their contribution to the design, and the best design is studied. The graph below illustrates one of the designs explored by TSC. The plot shows the relationship between net forward thrust and boat speed for a 20 mph wind speed. The concave downward curve satisfies our intuitions that the faster the boat travels, the less surplus thrust it will have to accelerate. Ideally, one reads the maximum speed as the point where the curve crosses the x-axis.



The curve offers a pair of interesting points worth pondering. Two discontinuities are noted. The upper left—occurring at low speed—is easily explained by the boat lifting a balance ski out of the water when it is no longer needed to maintain, as sailors would say, an even keel. The second discontinuity, occurring at much higher speed, is a bit more interesting. In fact, we found this second point an inspiration for discovery: design rules to keep the boat in the water. In fact, the

problem of the boat lifting clear of the water remains a partially unsolved problem at this writing; the effective maximum speed of the boat is therefore the speed at which it tries to fly. The problem suggests a counter-intuitive solution: the heavier the boat, the faster it can go.

4.2 APPLICATIONS

4.2.1 Proteins

4.2.1.1 Prediction by GA Rule Building

The problem is to predict the presence of helical structures in a protein when given its amino acid sequence. The DE approach is to build a population of rules when given a "genome." The genome is built from three "chromosomes," the working components of an observer rule. A pair of observer rules ("englishized" for readability) follow:

```
RULE: OBS.1
IF you have the actors: ALA and GLU
AND the relation: ALA abuts GLU
THEN predict a helix
RULE: OBS.2
IF you have the actors: LEU and GLU
AND the relation LEU: abuts GLU
THEN predict a helix
```

The three chromosomes are: actors, relations, and predictions. Actors are comprised of the twenty natural amino acids (e.g.: Alanine Arginine Asparagine etc.)

Relations are primarily structural or spatial in this example:

```
abuts-(LEU is followed by ALA)
precedes.1-(LEU is preceded by ALA with one amino acid between them)
precedes.2-(LEU is preceded by ALA with two amino acids between them)
```

The types of predictions available are based on the structure to be predicted. Helix, Sheet, and Turn are the typical protein structures to be predicted.

A TSC experiment begins with the observer rules being exercised on segments of a protein database. These segments are called windows of data, i.e., a sequence of amino acids. During learning, the window is shifted along the sequence from start to finish. An example window of data, with a window size of five (5) amino acids, looks like:

```
... PHE GLN THR [ ala arg gly ala pro ] HIS ILE VAL...
```

Shifting of the window involves moving the window to the right by one amino acid:

```
... PHE GLN THR ALA [ arg gly ala pro his ] ILE VAL...
```

All rules are exercised (allowed to vote) on each window. Voting is repeated as the window is "slid" along the entire data set. A reward/punishment algorithm then examines the performance of the individual rules which cast a vote. Following the "bucket brigade" algorithm of John Holland [Holland, 1986], those rules which participate in the vote, and which vote correctly, get a reward (their worth is increased).

Once rewards have been passed to appropriate rules, a small decay (reduction of worth) of all rules is computed. This has the effect of punishing those rules which do not participate in the vote, or vote incorrectly. All rules are used for breeding until their worth falls below a specified value, at which time they are eliminated from the gene pool.

Directed evolution exercises a genetic algorithm on the rule population to conduct a search for

the optimum rule set. For example, using as parents OBS.1 and OBS.2 listed above, crossover builds one child or constructed rule that looks like the following:

```
RULE: CON.3
IF you have the actors LEU and ALA
AND the relation LEU abuts ALA
THEN predict a helix
```

This "child" rule is added to the population of rules and given a starting worth value. Using OBS.3, point mutation may build CON.4:

```
RULE: CON.4
IF you have the actors LEU and MET
AND the relation LEU abuts MET
THEN predict a helix
```

We have developed a test knowledge base comprised of 70 observation rules (random combinations of actors and relations) and exercised the DE on this knowledge base. Training was conducted with ten proteins from the Brookhaven Protein Database. Results are illustrated in figure 4-1 but represent only the initial performance of the DE system, and provide some early indication of the make-up of rules which address the objective of the project, i.e., the discovery of rules which successfully predict helices in proteins with fair accuracy.

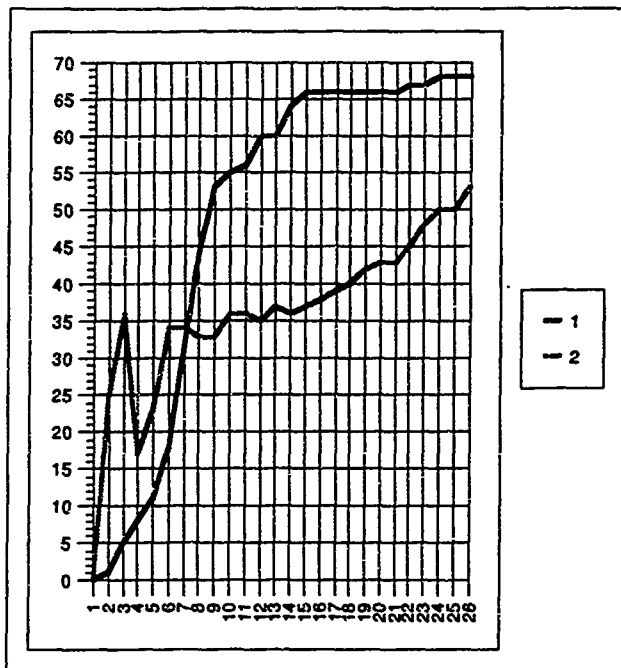


Figure 4-1: System Performance (trial KB) 11 Sep 92.

Legend:

error of omission- % of helices with no prediction,
 (100% = no errors-value of line #1)
 error of commission- % of correct predictions,
 (100% = no errors-value of line #2)

Figure 4-2 shows two curves, the percentage of helices discovered by the DE (line #1) and the accuracy with which the rules fired (line #2). The DE attempts to find most of the helices before it attempts to improve the accuracy of the rules. The graphs also indicates a few missed opportunities during the experiment. For example, upon finding fifty percent of the helices, the DE began refining the rules through "viral" mutation. A lost opportunity was caused by ending the experiment after five hundred cycles; predictive accuracy was still increasing at a useful rate.

Interestingly, the votes may be expanded to include positive and negative predictions, e.g.: predict helix and predict no helix. Empirically, because of the large number of "negative" training examples, false rules begin to dominate and performance deteriorates, i.e., there are typically 5 to 10 times as many non-helix as helix windows in a protein file. As a result, we have learned that FALSE predictions are detrimental to the DE system's performance, and thus only TRUE predictions are pursued. Prediction, in this research, is restricted to the helix structure.

The following graph illustrates the generally upward slope of accuracy in our experimental work with the GA approach to building protein structure prediction rules. As work wound down on

this project, the slope continued to point upward. Our experience indicates that this is a rather computational-intensive approach.

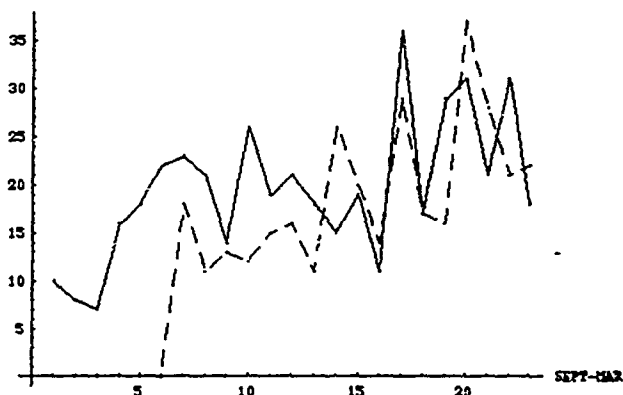


Figure 4-2: Directed Evolution.

4.2.1.2 Prediction by GA Program Building

Testing of the genetic programming approach, coupled to rough set evaluation was conducted during this 82 exercise. As mentioned before, testing was limited to small database predictions. Our testing confirmed the ability of this approach to recall stored patterns and to predict from unseen patterns. We achieved 100% accuracy on recall and 96% accuracy for prediction. This is consistent with the 93% to 100% accuracy reported by Salzberg [1990]. Our results for the iris database are illustrated below. This approach turns out to be sufficiently compute-intensive that the protein application was not explored in this project.

Figure 4-3 shows the $k(P,N)$ values obtained during training.

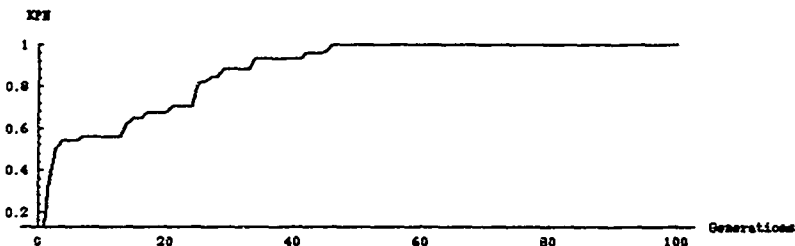


Figure 4-3: $k(P,N)$ during training.

Figure 4-4 shows results of testing for pattern recall during training. This data was obtained by testing on a subset of the data used for training.

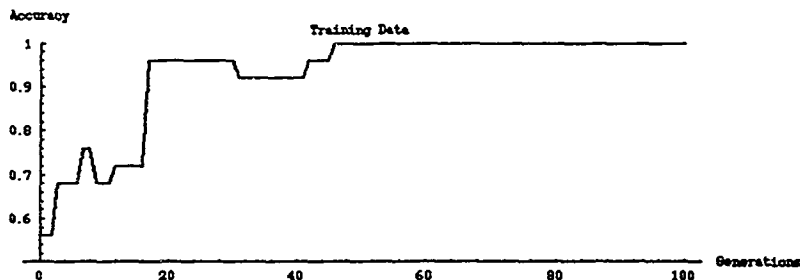


Figure 4-4: Recall accuracy during training

Figure 4-5 shows results of testing for predictive ability during training. This data was obtained by testing on a subset of the data disjoint from that used for training.

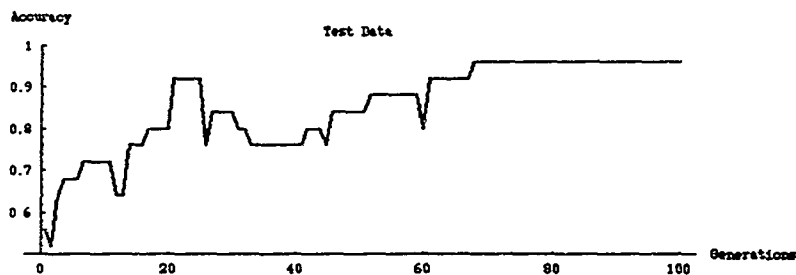


Figure 4-5: Prediction accuracy during training

Figure 4-6 shows changes in the size of the minimal set, M , during training.



Figure 4-6: Number of programs in minimal set during training

Our test results suggest there may be two distinct aspects to learning in this approach. The first is the development of a population of programs sufficient to recall members of the training set. We see this in Figures 4-3 and 4-4 where $k(P,N)$ and the recall accuracy proceed to the maximum value of 1.0. At this point we might expect learning to stop. However, in a number of trials we

found that prediction accuracy continued to improve, albeit sometimes erratically.

Looking at Figure 4-5 we see that prediction accuracy continued to increase for some time after k(P,N) reached its maximum. We suggest a explanation may be found in the size of the minimal set shown in Figure 4-6. At about the same time as prediction accuracy began its rise to a final maximum of 96% the size of the minimal set began to decrease from a high of 10 programs to a range of 6-8 programs. This is not inconsistent with other machine-learning techniques in which smaller representations tend to have a greater ability to generalize.

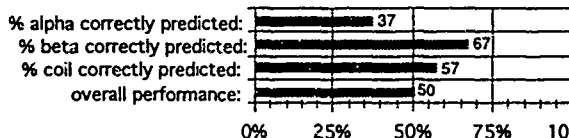
The biggest drawback to this approach is the computational cost. Our implementation performed well on the database of iris flowers, but the computational burden was a significant problem in preliminary tests on the much larger task of discovering regularities between the primary and secondary structure of proteins.

4.2.1.3 Prediction by Nearest Neighbor and Protein Design

In a typical experiment, a protein was designed by the TSC case-based design code. The resulting design frame from one run, known as CON_3, is:

```
C: CON_3
name          TESTPRO3
instance.of   protein
functionality  none
source        design.pro
HELIX.POSITION ( ( 10 21 ) ( 51 56 ) ( 59 66 ) ( 71 80 ) )
SHEET.POSITION ( ( 27 35 ) ( 41 46 ) )
MY.DATA       ( HIS TRP GLY TYR GLY LYS HIS ASN GLY GLU VAL THR CYS
VAL VAL VAL ASP VAL SER HIS GLU PRO SER SER LEU ASP CYS SER LEU GLY PHE ASN
VAL GLY ASP SER LEU VAL THR PHE THR VAL ALA GLY GLU ALA ASN SER CYS VAL GLY
CYS HIS LEU GLY ASP GLY ASP ASP VAL VAL ALA LYS TYR GLY LEU ASP GLY LEU LYS
PRO LEU ALA GLN SER HIS ALA THR GLY PHE HIS GLY )
```

In early developmental trials of the nearest neighbor code, a training set of 66 proteins from the Brookhaven PDB collection was selected, which generated 9,794 exemplars. With a window length of 17, we ran the nearest neighbor algorithm on the designed protein, and evaluated the predictions of the nearest neighbor as compared to the design specification. Consider these results (where overall performance = total # correct predictions + total # predictions available) from a trial on CON_3 and compare them to results presented this table:



The output of the nearest-neighbor structure prediction algorithm is a list comprising the sequence, indicating whether the amino acid is part of a helix (A), sheet (B), or coil (C). The trial resulted in the following predicted structure:

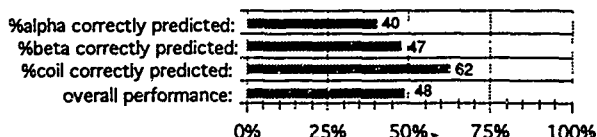
```
( C C B A A B B B C C C C C C C B B B B B B C C C C C C B B B B B
C C C A A A A C C C A C C A A A A C C C C C C A A A A A )
```

The CON_3 protein was intended to look like the following:

```
( C A A A A A A A A A A A C C C C C B B B B B B B C C C C C B B B B B
B C C C C A A A A A C C A A A A A A A C C C C A A A A )
```

The test was then repeated with a training set of 91 proteins (25 additional). The test applied

15,704 exemplars. The test, on CON_3, yielded these results:



This trial resulted in the following predicted structure:

```
( C B A B B B B B A A A A B C C C C B B B B B B A C C C C B C C C C C
C C A A A A C C C C C C A C A A B C C C C B C A A A A A )
```

A later trial was generated to involve a "redesign" of a particular de novo protein documented in the literature. The creators of Felix describe the protein as a de novo antiparallel four-helix bundle designed for a specific topology [Hecht, et al., 1990]. Its designers intended to choose an amino acid sequence unrelated to any native sequence, but which will fold into a desired three-dimensional structure.

We have chosen to closely follow the design specifications of Felix, and apply the TSC case-based design code. Note that our case-based design does not duplicate the specific residue sequence of Felix, but does duplicate the secondary structure of that protein. Figure 4-7 illustrates the TSC clone of Felix, which has the same shape as the original Hecht et al. sequence.

The case-based design program created an amino acid sequence and named the protein CON_4T. Consider the following design frame developed to duplicate the secondary structure of Felix, with the application of analogy rather than de novo rules:

```
C: CON_4T
instance.of      protein
source          bbt.create.pro
HELIX.POSITION  ( ( 1 19 ) ( 22 37 ) ( 40 58 ) ( 63 78 ) )
MY.DATA         ( PRO ILE LYS TYR LEU GLU PHE ILE SER GLU ALA ILE ILE
HIS VAL LEU HIS SER LYS ASP PHE SER ASP GLY GLU TRP HIS LEU VAL LEU ASN VAL
TRP GLY LYS VAL GLU ASP PHE PRO ILE LYS TYR LEU GLU PHE ILE SER GLU ALA ILE
ILE HIS VAL LEU HIS SER ARG LYS HIS LYS ILE TYR PRO GLY GLN ILE THR SER ASN
MET PHE CYS ALA GLY TYR LEU GLU )
```

The intended structural configuration is illustrated in figure 4-7, below.

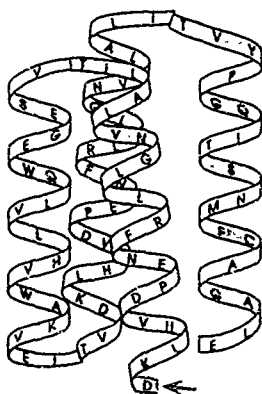


Figure 4-7: Felix—following [Hecht, et al., 1990].

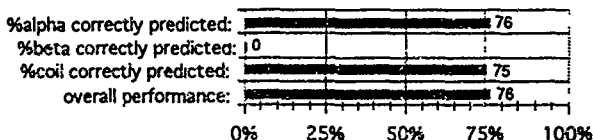
The structure represented by the TSC-designed sequence was intended to look like the following:

```
( A A A A A A A A A A A A C C A A A A A A A A A A A A A A A A C C A A A A A A
A A A A A A A A A A A A A C C C C A A A A A A A A A )
```

The nearest neighbor structure prediction code, trained on 91 proteins from the Brookhaven PDB collection (none of which are available to the design algorithm), predicted the following structure from the sequence:

```
( A A A A A A A A A A A A C C C C C C A A C A A A A A A A A C C B C C C A A A A A A
A A A A A A A A A A A A C A A A C C A A A A A C C C )
```

The final results of the prediction were:



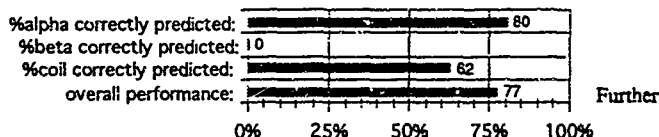
The agreement with helix and turn (coil) design is interesting; the results are higher than published predictions for any protein based strictly on native sequences. There is no agreement on beta sheet prediction since none were included in the design, and only a single instance of B showed up in the prediction.

[Zhang et al., 1992] report that the variation in performance of a single algorithm from one test set to another can be quite large. A fair measure of accuracy of an algorithm is the average of several different tests. Indeed, the [Salzberg and Cost, 1992] results in the first table reflect a result averaged over 10 tests. The results reported here are not averaged over a number of tests; that remains for future work.

Our results illustrated here suggest that accuracy improves slightly by decreasing the prediction training set size. To examine this accuracy behavior, a trial was conducted with the original 66 proteins serving as the training set for the nearest neighbor code. The designed protein is CON_4T, the TSC Felix clone. The prediction improved, and produced the following predicted structure:

(A A A A A A A A A C C C C C A A A A A A A A C C B A B C C A A A A
A A A A A A A A A A A A A B A C A A A A A C C C)

The prediction made three errors by including "B" beta sheets, but appears to have improved its alpha helix prediction. Consider the results:



characterization of this behavior will require a population study of the selected training proteins. Preliminary indications are that certain training proteins from the additional set of 25 generate exemplars which may be "nearer" to the testing window than are exemplars generated from the original 66 proteins, but which offer an improper prediction. Factors involved in this prediction performance include window length, and training set homology. These, and other factors, remain a topic of continued research. An interesting approach to prediction improvement, as suggested in section 3.2, is to enable the exemplar weighting scheme in the TSC nearest neighbor code.

The correlation coefficients Ca, Cb, and Ccoil reported by [Salzberg and Cost, 1992] illustrated above in Table 1 are computed with an algorithm due to [Mathews, 1975], and may reflect slightly higher values than those reported here by us. The differences, if any, are the subject of continued study; direct comparison of results is problematic since the [Salzberg and Cost, 1992] results are based on tests conducted on proteins from the Brookhaven database, and our results are based on tests conducted on proteins designed by analogy to proteins in the Brookhaven database. In addition, PEBLS applied a weighting factor to the exemplars which was reported to improve its performance significantly over the unweighted version; our work has not yet applied the weighting scheme. Finally, PEBLS includes a post-processing step based on the minimum sequence length restrictions used by [Holley and Karplus, 1989]. This restricts beta sheets to a minimum contiguous sequence of two residues, and alpha helix no fewer than four residues. This is reported to improve PEBLS performance [Cost and Salzberg, 1993].

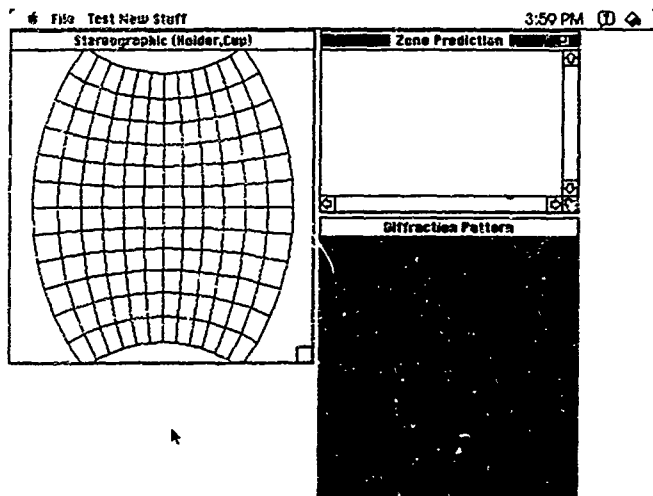
4.3 TEM

Our results to date on the crystallography task have been limited by budget considerations in the other tasks. We have built the TEM simulator with our consultant A.G. Jackson, and have integrated it with a knowledge base for TSC which operates the simulator as though it were a live TEM. As a Q3 activity, we met with an interested scope maker and exhibited our tool. Discussions on further development in a cooperative venture with this manufacturer were limited due to the fact that they do not have a large enough market to justify R&D in this domain. They did, however, offer to us their entire clientele by way of their research newsletter. Dr. Jackson has authored a paper for that journal.

We now sketch the demonstration, as it has been conducted.

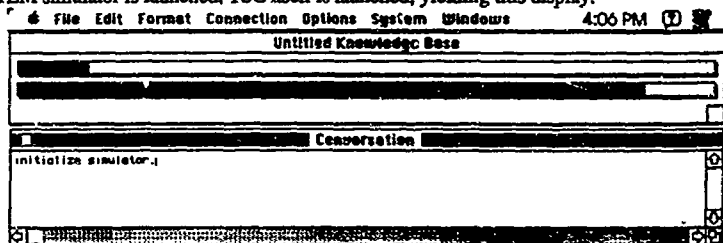
4.3.1 Launching the TEM Controller/Simulator

The following illustrates the display when the TEM simulator is loaded.



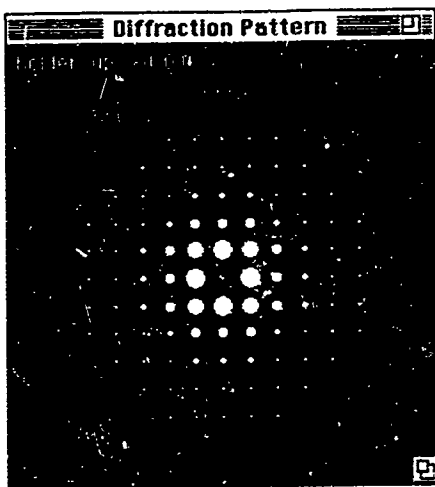
4.3.2 Launching TSC

Once the TEM simulator is launched, TSC itself is launched, yielding this display.

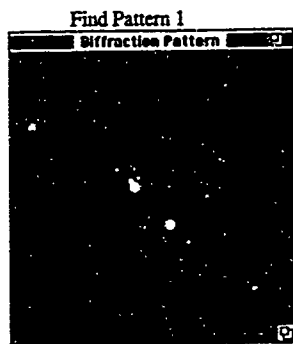


4.3.3 Initialize Simulator

With the simulator and TSC in memory, the simulator is then initialized by TSC. This initialization calibrates the simulator to enable further analysis of zones and crystal structures. Initialization involves TSC calculating the zone on each of two different crystal images generated by the simulator. The first image follows.



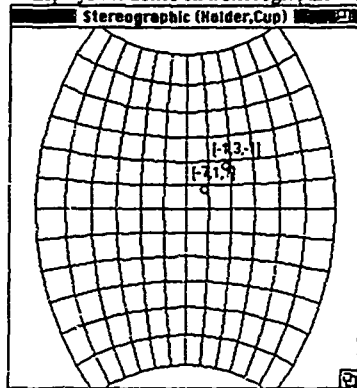
This is the first image requested by TSC.



Next, the second image requested by TSC.



With both images analyzed, the TSC TEM controller is now ready to analyze the crystal further. It displays the zones on a stereographic image obtained from the two calibration images.



4.3.4 User Dialog With the TEM Simulator

Following initialization, the user is free to use dialogs and the command-line interface to request different images from the simulator. On a live TEM, the same dialogs and command-line references would cause the scope to produce images. A typical command-line request is:

go to zone [1 -4 1]

This request would cause TSC to respond with an image appropriate to that zone. The following images illustrate using the dialog windows to explore a crystal.

Initialize Simulator

crystal type
(t, i, o, m, h, r or a)

indices
(t, h3 or h4)

a
b
c

alpha
beta
gamma

Set Holder Cup

holder
cup

Go To Zone

u
v
w
t

Initialize Controller

u1 <input type="text" value="0.00"/>	u2 <input type="text" value="0.00"/>
v1 <input type="text" value="0.00"/>	v2 <input type="text" value="1.00"/>
w1 <input type="text" value="1.00"/>	w2 <input type="text" value="1.00"/>
t1 <input type="text" value="0.00"/>	t2 <input type="text" value="-1.00"/>
h1 <input type="text" value="0.00"/>	h2 <input type="text" value="0.00"/>
c1 <input type="text" value="0.00"/>	c2 <input type="text" value="45.00"/>

As mentioned earlier, the simulator has been coded and demonstrated, but project requirements in other—especially the Protein—tasks prevented completion of the TEM controller activity.

Chapter 5 Summary and Conclusions

The totality of the work performed thus far supports and envisions a unified tool approach to materials science and engineering. We summarize the architecture of such a system in figure 5-1.

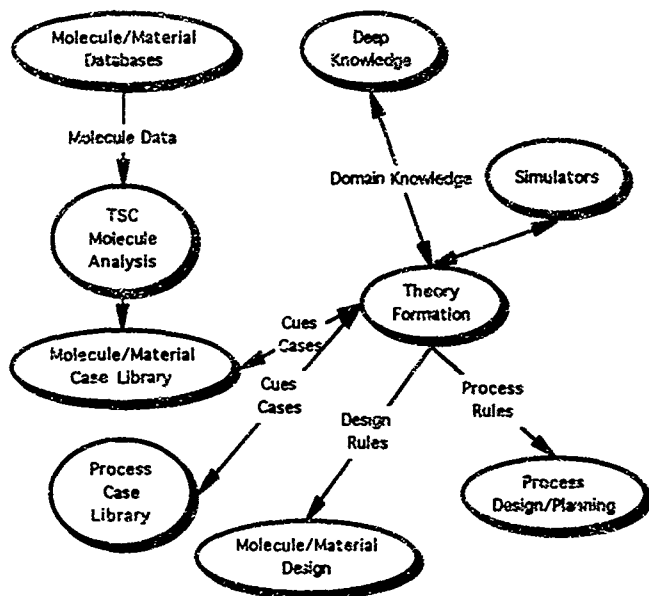


Figure 5-1: Overall project summary.

To achieve such an architecture, much work remains. We list many of the suggestions for improvement and further work which have emerged from our research. We suspect that a tool very useful to the materials fields will emerge when this research yields a successful Ø3 activity.

5.1 GENERAL IMPROVEMENTS

Improvements to the TSC case-based design system include:

- Ability to have user generated overlap parameters.

As discussed above, the TSC case-based design system compares adjacent structures by selecting one additional residue at each end, and makes a judgement about the similarity of the potential new overlap with the overlap from the original sequence. This choice of only one residue is arbitrary, and it may prove interesting to vary the amount of overlap used and see how the accuracy of the final design varies. While this poses a few minor additional computational problems (such as what to do when a chosen structure is so near the end of its native protein that there aren't enough extra residues to make up the overlap), none of these computational problems are beyond the capability of TSC.

- Testing of different parameter values for amino acid comparison.

The method used to make judgements about the similarity of neighboring amino acids is arbitrary. It might very well be the case that a different method (a different set of attributes and weights, or possibly an analysis of the molecular structure of the residues) could produce much better results.

- Ability to generate new cases through adaptation of old cases.

At present, the system finds all the potential residue fragments, compares them, and connects the least different to form the output protein. There is no provision for modifying the individual structures in any way to build a better protein. It may be useful to apply a heuristically guided method for changing the newly created protein.

- Validation of new proteins and inclusion in case library.

Validating proteins by X-ray crystallography (or other method) and placing the results back into the case libraries would create an external feedback loop as described above. This, in combination with the following, may allow the system to eventually discover new details of the first principles of protein folding.

- Checking a new sequence against library for unwanted structure matches.

The program could check the case library to see if it has incorporated a sequence of amino acids which is identical to a known example of an unwanted structure. While this would not guarantee that the protein would form the desired structure, it would reduce the likelihood that the protein would fold to some other structure.

- Characterize the effects of larger case-based libraries

The case libraries used in the protein design algorithm are at this point a small subset of the Brookhaven PDB. It is anticipated that providing more cases will improve the performance, though there may be a trade-off in accuracy in comparison to design time. Further study will be needed to find the best mix of speed and accuracy.

- Long-term improvements

Beyond each of these immediate improvements to the TSC system, a long-term extension is to combine case-based design with aspects of *de novo* design. This would require that the system discover new protein "first principles" from both its design-prediction cycles, and from database mining.

A further exercise would organize the design and the training proteins into a taxonomy as suggested above. Design experiments may then be conducted by specifying the branch of the taxonomy to be used in both design and prediction; the set selected for an exercise must then be partitioned into design and prediction training sets.

Finally, it will be useful to construct a family of designed proteins, characterize them, and apply appropriate feedback to TSC and to build a library of protein designs exhibiting certain (e.g. electro-optical) properties. With this feedback, the family of proteins designed may be classified as accurate or inaccurate, and the details of errors generated noted in the design frames. With this feedback mechanism, TSC would be able to modify its own memory (akin to traditional dynamic memory algorithms [Schank, 1982]).

5.2 PREDICTION SYSTEM IMPROVEMENTS

Improvements to the TSC nearest neighbor prediction system include:

- Characterize increasing of prediction database

Currently, only a small subset of the Brookhaven PDB is used as a basis for the prediction algorithm. Preliminary results have shown that increases in the database size may improve the accuracy of the nearest neighbor technique in characterizing proteins the system has designed. If some implementational speedups are introduced to the algorithms, it may be practical to introduce larger protein databases than are currently used.

- Training exemplar weights to improve the prediction performance.

PEBLs [Cost and Salzberg, 1993] applies a weighting scheme to predictions offered by exemplars. The weights require a second training pass through the training proteins to adjust the weights. As mentioned in section 3.1, [Cost and Salzberg, 1993] report the weighting improves prediction performance of PEBLS.

When PEBLS selects exemplars for prediction, the distance of the exemplar window from the testing window is calculated, and that distance is multiplied by the weight value of the particular exemplar. Smaller weight values imply smaller distance values; the lowest corrected distance determines the "winning" exemplar. Weights represent a kind of statistical property of the exemplars. Lower weight in a given exemplar (over the rest of the population) implies that exemplar is more reliable at formation of valuable predictions. The current implementation defaults to a weight equal to 1.0 for every exemplar.

The TSC nearest neighbor code presently allows for weight training, but the weighting algorithm is not enabled for the experiments reported here. It will be useful and interesting to enable the code and measure changes to prediction performance.

Overall, it is fair to comment that this Ø2 project evolved over time to emphasize the directed evolution aspect of protein evaluation, largely to the detriment to the other aspects of this research originally envisioned. However, the work continues. We see a reasonable Ø3 extension into the pedagogical applications of our TEM simulator. We further see a reasonable Ø3 extension of our nearest neighbor, rough set, and qualitative modeling tools.

Acknowledgments

This work has benefited greatly from the many suggestions and contributions of the following individuals: W.B. Dress, A.G. Jackson, S.R. LeClair, F.L. Abrams, C.W. Lee, J.B. Rose II, and J.B. Rose III. ThinkAlong employees involved in some aspect of this work have included: F. Behr, S. King, D. Foster, B. Towle, and J. Bishop.

References

- Abbas, Abul K., Andrew H. Lichtman, and Jordan S. Pober, *Cellular and Molecular Immunology*, Philadelphia, W.B. Saunders Company, 1991.
- Abelson, J. Directed Evolution of Nucleic Acids by Independent Replication and Selection. *Science*, 249, 488-489, 1990.
- Abrams, F., P. Garrett, T. Lagnese, S. LeClair, C. W. Lee, and J. Park, "Qualitative Process Automation for Autoclave Curing of Composites," AFWAL-TR-87-4083, Wright-Patterson AFB, Ohio, 1987.
- Anfinsen, C. B., Haber, E., Sela, M., & White, F. H. *Proceedings of National Academy of Science, USA*, 47, 1309, 1961.
- Ashby W. Ross, *Design for a Brain*, 2nd ed., Wiley, New York, 1960.
- Barker, Winona C., David G. George, and Lois T. Hunt, "Protein Sequence Database," *Methods Enzymol.*, 183, 31-49, 1990.
- Beltrami, Edward, *Mathematics for Dynamic Modelling*, Academic Press, New York, 1987.
- Bobrow Daniel G., editor, *Qualitative Reasoning About Physical Systems*, The MIT Press, Cambridge, Mass., 1985.
- Casti, John L., *Alternate Realities: Mathematical Models of Nature and Man.*, New York, John Wiley & Sons, 1989.
- _____. *Searching For Certainty: What Scientists Can Know About The Future*. New York: William Morrow and Company, 1990.
- Chou, Peter Y., and Gerald D. Fasman, "Prediction of the Secondary Structure of Proteins from their Amino Acid Sequences," *Adv. Enzymology*, 47, 45-148, 1978.
- Cost, Scott, and Steven Salzberg, "A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features," *Machine Learning*, 10, 57-78, 1993.
- DeGrado, William F., Zelda R. Wasserman, and James D. Lear, "Protein Design, a Minimalist Approach," *Science* 243, 622-243, 3 February, 1989.
- DeLisi, Charles, "Computers in Molecular Biology: Current Applications and Emerging Trends," *Science*, 240, 47-52, 1988.
- Dress, W.B., "On Doing Science by Computer—Part I: Philosophical Underpinnings," ThinkAlong Technical Report, Brownsville, ThinkAlong Software Inc, 1992a.
- _____, "On Doing Science by Computer—Part III: Modelling the Model," ThinkAlong Technical Report, Brownsville, ThinkAlong Software Inc, 1992b.
- Erickson, D. (1992). Intuitive Design. *Scientific American*, 267 (6), 124-125.
- Erickson, Michael D., and Jan M. Zytkow, "Utilizing Experience for Improving the Tactical Manager," in *Proceedings of the Fifth International Conference on Machine Learning*, San Mateo, Morgan Kaufmann Publishers, pp 444-450, 1989.
- Falkenhainer, B., "A Unified Approach to Explanation and Theory Formation," in [Shrager and Langley, 1990].

Farmer, J. D., N. H. Packard, and A. S. Perelson, "The Immune System, Adaptation, and Machine Learning," in Dooyne Farmer, Alan Lapedes, Norman Packard, and Burton Wendroff, editors: *Evolution, Games, and Learning: Models for Adaption in Machines and Nature*, Amsterdam, North-Holland, pp 187-204, 1986.

Fasman, Gerald D., "The prediction of the secondary structure of proteins: Fact or Fiction," *Current Science*, 59, Nos. 17 & 18, 839-845, 25 September, 1990.

Fedorov, A.N., D.A. Dolgikh, V.V. Chemeris, B.K. Chernov, A.V. Finkelstein, A.A. Schulga, Yu. B. Alakhov, M.P. Kirpichnikov, and O.B. Pitsyn, "De Novo Design, Synthesis and Study of Albebetin, a Polypeptide with a Predetermined Three-dimensional Structure," *J. Mol. Biol* 225, 927-931, 1992.

Forbus, Kenneth D., *Qualitative Process Theory*, Ph.D. Thesis, MIT Technical Report 789, 1984.

Foster, David H., W. James Bishop, Scott A. King, Jack Park, "Knowledge Discovery Using Genetic Programming with Rough Set Evaluation"; in Piatetsky-Shapiro, Gregory, Editor, *Knowledge Discovery in Databases—Proceedings, KDD 93 Workshop*, AAAI, Palo Alto CA, 1993.

Franks, Nigel R. "Army Ants: a Collective Intelligence," *American Scientist*, 77, March-April 1989, pp 139-145.

Gardner, M., *Wheels, Life and Other Mathematical Amusements*, San Francisco, Freeman, 1983.

Alan E. Gelfand and Crayton C. Walker, *Ensemble Modeling*, Marcel Dekker, New York, 1984.

Gibbs, M.R., and A.G.W. Leslie, in the Brookhaven Protein Data Base, 1990.

Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Menlo Park, Addison-Wesley Publishing Company, 1989.

Hecht, Michael H., Jane S. Richardson, David C. Richardson, and Richard C. Ogden, "De Novo Design, Expression, and Characterization of Felix: A Four-Helix Bundle Protein of Native-Like Sequence," *Science* 249, 884-891, 1990.

Holland, J. H. Escaping Brintleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallell Rule-Based Systems. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (pp. 593-624). Los Altos: Morgan Kaufmann, 1986.

_____, *Adaptation in Natural and Artificial Systems*. Boston: MIT Press, 1992.

_____, "Using Classifier Systems to Study Adaptive Nonlinear Networks," 1989, in [Stein, 1989], pp 463-500.

Holland, J.H., K.J. Holyoak, R.E. Nisbett, & P.R. Thagard. *Induction: Processes of Inference, Learning, and Discovery*. Boston: MIT Press, 1986.

Holley, L. and M. Karplus, "Protein secondary structure prediction with a neural network," *Proc. Nat. Acad. Sci., U.S.A.*, 86, 152-156, 1989.

Jackson, A.G., "The Role of Genetic and Self-Learning Algorithms in Research from a User Scientist's Viewpoint," ThinkAlong Technical Report, Brownsville, ThinkAlong Software Inc, 1992.

Jen, Erica, "Limit Cycles in One-Dimensional Cellular Automata," in [Stein, 1989], pp 743-758,

1989.

Jerne, N. K., "The Immune System," *Scientific American*, 229, 1973, pp 52-60.

Judson, Richard S. and Herschel Rabitz, "Teaching Lasers to Control Molecules," *Phys. Rev. Letters*, 68, Number 10, pp 1500-1503, 1992.

Karp, Peter D., *Hypothesis Formation and Qualitative Reasoning in Molecular Biology*, Ph.D. Thesis, STAN-CS-89-1263, Stanford University, 1989.

_____, *CABIOS*, 8:347-357, 1992.

King, Scott A., "Discovery in Protein Structure with a Genetic Algorithm, and TSC," manuscript, 1993.

Kougiass, Ch. F., and J. Schulte, "Simulating the Immune Response to the HIV-1 Virus with Cellular Automata," *J. Statistical Physics*, 60, Nos. 1/2, 1990, pp 263-273.

Koza, John R. *Genetic Programming: on the programming of computers by means of natural selection*, Cambridge MA, MIT Press, 1992.

Langley Pat, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow, *Scientific Discovery*, The MIT Press, Cambridge, MA, 1987.

LeClair, Steven R., Allen G. Jackson, Scott King, Dan Wood, and Jack Park, "A Heuristically Guided Genetic Algorithm for Predicting Protein Secondary Structure," manuscript, 1992.

Lenat, Douglas B., "The Role of Heuristics in Learning by Discovery: Three Case Studies," in Michalski, R. S., J. G. Carbonell, and T. M. Mitchell, editors: *Machine Learning, an Artificial Intelligence Approach*, Palo Alto, Tioga Publishing Company, 1983.

Lucasius, C.B., & Kateman, G. Application of Genetic Algorithms in Chemometrics. 3rd International Conference on Genetic Algorithms, 1989.

Maggiora, G.M., B.Mao, K.C. Chou, and S. L. Narasimhan, "Theoretical and Empirical Approaches to Protein-Structure Prediction and Analysis," *Meth. Biochem. Anal.*, 35, 1-86, 1991.

Mathews, B.W., "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta*, 405, 442-451, 1975.

Michalski, Ryszard S., Jaime G. Carbonell, and Tom M. Mitchell, editors: *Machine Learning: Volume II*, Los Altos, Morgan Kaufmann Publishers, Inc, 1986.

Morrison Foster, *The Art of Modeling Dynamic Systems*, John Wiley & Sons, New York, 1991.

Muskal, Steven M. and Sung-Hou Kim, "Predicting Protein Secondary Structure Content: A Tandem Neural Network Approach," *J.Mol.Biol.*, 225, 713-727, 1992.

Nishikawa, Ken, Yasushi Kubota, and Tatsuo Ooi, "Classification of Proteins into Groups Based on Amino Acid Composition and Other Characters. I. Angular Distribution," *J. Biochem.*, 94, 981-995, 1983.

Orlowska, Ewa and Pawlak, Zdzislaw. "Expressive power of knowledge representation systems," *Int. J. Man-Machine Studies*, vol. 20 pp. 485-500, London, Academic Press Inc, 1984.

O'Rorke, P., S. Morris, D. Schulenburg, "Theory Formation by Abduction: A Case Study Based on the Chemical Revolution," in [Shrager and Langley, 1990].

Park, Jack. "On an Approach to Index Discovery." 1993, submitted to KDD 1993.

_____, and Dan Wood. *User's Manual—The Scholar's Companion.*, Brownsville, CA, ThinkAlong Software, 1993.

_____, "Creative Sailboat Design"; in *Proceedings AAAI 93 Spring Symposium—Creativity and AI*, AAAI, Palo Alto CA, March 1993.

Pawlak, Zdzislaw. "Rough Classification," *Int. J. Man-Machine Studies*, vol. 20 pp. 469-483, London, Academic Press Inc, 1984.

Pauling, L., & Corey, R. B. *Proceedings of National Academy of Sciences, USA*, 37, 729-735, 1951.

Pawlak, Zdzislaw. *Rough Sets: Theoretical Aspects of Reasoning about Data.*, Boston, Kluwer Academic Publishers, 1991.

Popper Karl R. and John C. Eccles, *The Self and Its Brain*, Routledge & Kegan Paul, London, 1977.

Qian, N. and T. Sejnowski, "Predicting the secondary structure of globular proteins using neural network models," *J. Mol. Biol.*, 202, 865-884, 1988.

Regan, Lynne, and William F. DeGrado, "Characterization of a Helical Protein Designed from First Principles," *Science* 241, 976-978, 19 August, 1988.

Riesbeck, Christopher K., and Roger C. Schank, *Inside Case-Based Reasoning*, New Jersey: Lawrence Erlbaum Associates, 1989.

Renders, J. M., & Norvik, J. P. (1992). Genetic Algorithms for Process Control: A Survey. In *IFAC/IFIP/IMACS International Symposium on Artificial Intelligence in Real-Time Control*, (pp. 579-584). Delft, the Netherlands: IFAC/IFIP/IMACS, 1992.

Rosen Robert, *Anticipatory Systems*, Pergamon Press, New York, 1985.

Salzberg, Steven, and Scott Cost, "Predicting Protein Secondary Structure with a Nearest-neighbor Algorithm," *J. Mol. Biol.* 227, 371-374, 1992.

Salzberg, Steven L., *Learning with nested generalized exemplars.*, Boston, Kluwer Academic Publishers, 1990.

Schank, R.C., *Dynamic Memory: A Theory of Learning in Computers and People*, Cambridge University Press, 1982.

Shrager, J., and P. Langley, Editors, *Computational Models of Scientific Discovery and Theory Formation*, San Mateo, Morgan Kaufmann, 1990.

Sieburg, H., H. McCutchan, O. Clay, L. Caballero, and J. Ostlund, "Simulation of HIV-infection in Artificial Immune Systems," in Howard Gutowitz, Ed.: *Cellular Automata: Theory and Experience*, Cambridge, MA, MIT Press, 1991, pp 208-228.

Souile Françoise Fogelman, Yves Robert, Maurice Tchuente, *Automata networks in computer science*, Princeton University Press, Princeton, NJ, 1987.

Stein, Daniel L., Editor, *Lectures in the Sciences of Complexity*, Redwood City, Addison-Wesley Publishing Company, 1989.

- Thagard Paul, *Computational Philosophy of Science*, The MIT Press, Cambridge, Mass., 1988.
- Toffoli Tommaso and Norman Margolus, *Cellular Automata Machines*, The MIT Press, Cambridge, Mass., 1987.
- Unger, Ron, and John Moult, "Genetic Algorithms for Protein Folding Simulations," *J. Mol. Biol* 231, 75-81, 1993.
- Wendoloski, J.J. and F.R. Salemme, "PROBIT: A Statistical Approach to Modeling Proteins from Partial Coordinate Data Using Substructure Libraries," *J. Mol. Graphics*, 10, 124-126, June 1992.
- Wolfram Steven, "Complex Systems Theory," in *Emerging Syntheses In Science*, David Pines, editor, Addison-Wesley, Redwood City, Cal., 1988.
- _____, *Mathematica*, 2nd Ed, Addison-Wesley, Redwood City, Cal., 1991.
- Wood, Dan, and Jack Park, *Qualitative Process Discovery: The Next Step Toward Intelligent Systems*, WRDC-TR-90-4129, Dayton, Wright Research Laboratories—United States Air Force, 1990.
- Zhang, X., J. Mesirov, and D. Waltz, "A hybrid system for protein secondary structure prediction," *J. Mol. Biol*, 225, 1049-1063, 1992.
- Ziarko, Wojciech. "A Technique for Discovering and Analysis of Cause-Effect Relationships in Empirical Data," *Knowledge Discovery in Databases—Proceedings, KDD 89*, 1989.

{ GENERALIZED.NEAR.NEIGHB.T

to do. consider implementing a "tree" of exemplars, perhaps anchored in the AA database such that you take each AA in a window and walk the tree. If the tree no longer supports your next AA, then you must compare your window to all the REST of the tree exemplars
This should cut down on some of the searching needed to compare exemplars to windows.

dates

05/20/93 jp2: first cut, made from NEAR.NEIGHB.T to generalize
away from proteins
05/21/93 jp2: added weights, other minor changes
05/24/93 jp2: minor fixes to get weights working properly
06/01/93 jp2: added correctness totalizers

\----- GLOBALS

(" *flag *exemplar *window *classification *win.rad \These should not be global
*table.list *exemplar.list *train# *test# *train.list *test.list *self.exemplar
*cta *ctb *ctc *cor.a *cor.b *cor.c *mix")
&global.list union to-anchor &global.list

\----- MISC SUPPORT

{ LIST.REPLACE

description: Given a list, a number, and a sx, return the list with that position element
replaced by the given sx
example input: (A B C D E) 3 X
example output: (A B X D E)
notes: X

c: LIST.REPLACE

instance of list.func
\ my.creator bbt
i.take list number sx
i.give list
arguments *my.list *position *new.element
my.vars *pos *new.list *max.pos
algorithm (do
(cond
((equal? *position 1)
(bindq *new.list (cons *new.element (rest *my.list))))
(T
(bindq *new.list (concat (concat (grab.first.n *my.list (sub1 *position))
(list *new.element))
(clip.list *my.list (add1 *position))))))
(return *new.list))

c: SUBUNIT OF

sub of information.slist

c: HAS.SUBUNITS

instance of flow.func
my.creator wjb

```

i.take      nonc
i.give      list
my.vars     *protein.list *pro.name ~list *data
algorithm   (do
  (bindq *protein.list (get *PROTEIN *SUBS))
  (loop.until (null? *protein.list) \for each protein
    (do (bindq *pro.name (first *protein.list))
      (bindq *data (reverse (get *pro.name MY.DATA)))
      (if.true (greater.than? (length *data) 1)
        (bindq *list (cons *pro.name *list)))
      (bindq *protein.list (rest *protein.list))))
  (return *list;))

=====
SET.USE

description: install an exemplar use value in an exemplar--the third value
}

c: SET.USE
instance.of  flow.func
my.creator   jp2
i.take       list number
i.give       list
arguments    *exemp *use
my.vars      *cor
algorithm    (list.replace *exemp 3 *use)

comment:
algorithm    (do
  (bindq *exemp (reverse *exemp))
  (bindq *cor (first *exemp))
  (bindq *exemp (rest (rest *exemp)))
  (bindq *exemp (cons *use *exemp))
  (bindq *exemp (reverse (cons *cor *exemp))))
  (return *exemp))
comment;

c: GET.USE
instance.of  flow.func
my.creator   jp2
i.take       list
i.give       number
arguments    *exemp
algorithm    (third *exemp)

comment:
algorithm    (second (reverse *exemp))
comment;

=====
INC.USE

description: increment an exemplar correct use value in an exemplar--the third value
}

c: INC.USE
instance.of  flow.func

```

```

my.creator      jp2
i.take         list
i.give         list
arguments      *exemp
my.vars        *use
algorithm      ( list.replace *exemp 3 ( add1 ( thurd *exemp )))

comment:
algorithm      ( do
  ( bindq *exemp ( reverse *exemp ))
  ( bindq *use ( second *exemp ))
  ( bindq *use ( add1 *use ))
  ( return ( set.use ( reverse *exemp ) *use )))
comment;

( =====
  SET.CORRECT

  description: install an exemplar correct use value in an exemplar--the fourth value
)

c: SET.CORRECT
instance.of    flow.func
my.creator     jp2
i.take         list number
i.give         list
arguments      *exemp *cor
algorithm      ( list.replace *exemp 4 *cor )

comment:
algorithm      ( do
  ( bindq *exemp ( reverse *exemp ))
  ( bindq *exemp ( rest *exemp ))
  ( bindq *exemp ( reverse ( cons *cor *exemp )))
  ( return *exemp ))
comment;

c: GET.CORRECT
instance.of    flow.func
my.creator     jp2
i.take         list
i.give         number
arguments      *exemp
algorithm      ( fourth *exemp )

comment:
algorithm      ( first ( reverse *exemp ))
comment;

( =====
  INC.CORRECT

  description: increment an exemplar correct use value in an exemplar--the fourth value
)

c: INC.CORRECT
instance.of    flow.func
my.creator     jp2

```

```

i.take      list
i.give      list
arguments   *exemp
my.vars     *cor
algorithm   (list.replace *exemp 4 (add1 (fourth *exemp)))

comment:
algorithm   { do
              (bindq *exemp (reverse *exemp))
              (bindq *cor (first *exemp))
              (bindq *cor (add1 *cor))
              (return (set.correct (reverse *exemp) *cor)))
}

=====
GET.WEIGHT

description: returns weight from an exemplar
              weight = #uses/#correctuses
              smaller = better
}

c: GET.WEIGHT
instance.of  flow.func
my.creator   jp2
i.take       list
i.give       number
arguments    *exemp
algorithm    (quotient (third *exemp) (float (fourth *exemp)))

=====
SORT.ON.WEIGHT

description: sort exemplar list on ascending weights
}

COMMENT:

c: SORT.ON.WEIGHT
instance.of  flow.func
my.creator   jp2
i.take       list
i.give       list
arguments    *list

COMMENT:

c: INC.ACTUAL
instance.of  flow.func
my.creator   jp2
i.take       symbol
i.give       none
argument     *act
algorithm    { cond ((same? *act 'A)
                     (setq *cta (add1 *cta)))
               ((same? *act 'B)
                 (setq *ctb (add1 *ctb)))
               (if (setq *ctc (add1 *ctc)))
}

```


c: INC.PREDICTED

```
instance.of    flow.func
my.creator     jp2
i.take         symbol
i.give         none
arguments      *pred          \ only when correct
algorithm      ( cond        (( same? *pred 'A )
                             (setq *cor.a ( add1 *cor.a )))
                             (( same? *pred 'B )
                             (setq *cor.b ( add1 *cor.b )))
                             ( T (setq *cor.c ( add1 *cor.c ))))
```

----- LIST SUPPORT

DELETE.ELEMENT

description: Deletes the element at position *pos of list *list returning the result;
numbering begins with 1.

}

c: DELETE.ELEMENT

```
instance.of    flow.func
my.creator     wjb
i.take         list number
i.give         list
arguments      *list *pos
my.vars        *temp *count
algorithm      ( do
                ( if.true ( equal? *pos 0 )
                  ( display "DELETE.ELEMENT: Cannot take 0 as position argument." error )
                )
                ( bindq *count 1 )
                ( bindq *temp nil )
                ( loop.until ( equal? *count *pos )
                  ( do ( bindq *temp ( cons ( first *list ) *temp ) )
                      ( bindq *list ( rest *list ) )
                      ( bindq *count ( add1 *count ) ) )
                  ( bindq *temp ( reverse *temp ) )
                  ( return ( concat *temp ( rest *list ) ) ) )
```

INSERT.ELEMENT

description: Insert *element at position *pos of list *list returning the result;
numbering begins with 1.

}

c: INSERT.ELEMENT

```
instance.of    flow.func
my.creator     wjb
i.take         list number sx
i.give         list
arguments      *list *pos *element
my.vars        *temp *count
algorithm      ( do
                ( if.true ( equal? *pos 0 )
```

```

        (display "INSERT.ELEMENT: Cannot take 0 as position argument." error)
      )
      (bindq *count 1)
      (bindq *temp nil)
      (loop.until (equal? *count *pos)
        (do (bindq *temp (cons (first *list) *temp))
          (bindq *list (rest *list))
          (bindq *count (add1 *count))))
      (bindq *temp (cons *element *temp))
      (bindq *temp (reverse *temp))
      (return (concat *temp *list)))

```

REPLACE.ELEMENT

description: Replaces the element at position *pos of list *list with *element
returning the result; numbering begins with 1.

c: REPLACE.ELEMENT

```

instance.of  flow.func
my.creator   wjb
i.take       list number sx
i.give       list
arguments    *list *pos *element
my.vars      *temp *count
algorithm    (do
  (if.true (equal? *pos 0)
    (display "REPLACE.ELEMENT: Cannot take 0 as position argument." error)
  )
  (bindq *count 1)
  (bindq *temp nil)
  (loop.until (equal? *count *pos)
    (do (bindq *temp (cons (first *list) *temp))
      (bindq *list (rest *list))
      (bindq *count (add1 *count))))
  (bindq *temp (cons *element *temp))
  (bindq *temp (reverse *temp))
  (return (concat *temp (rest *list))))

```

ALGORITHM SUPPORT

COUNT.CLASSES

description: takes the list of exemplars
typical exemplar looks like: ((gly glu pro ...) B 1 1)
returns a list e.g. (2869 1091 576 1202 .38 .20 .42)
which is: Total #alpha #beta #coil %alpha %beta %coil
MUST BE MODIFIED TO INC CLASSES OTHER THAN A, B, and C &&&&&&&&

c: COUNT.CLASSES

```

instance.of  flow.func
my.creator   wjb
i.take       list \exemplars
i.give       list
arguments    *list

```

```

my.vars      *a.count *b.count *c.count *length *exemp
algorithm    (do
  (bindq *a.count 0)
  (bindq *b.count 0)
  (bindq *c.count 0)
  (bindq *length (length *list))
  (loop.until (null? *list)
    (do (bindq *exemp (first *list))      \get an exemplar
      (if.true (list? *exemp)
        (bindq *exemp (second *exemp))) \get classification
      (cond ((same? *exemp 'A)
        (bindq *a.count (add1 *a.count)))
        ((same? *exemp 'B)
        (bindq *b.count (add1 *b.count)))
        (bindq *c.count (add1 *c.count))))
    (bindq *list (rest *list)))
  (return (list *length *a.count *b.count *c.count
    (quotient *a.count (float *length))
    (quotient *b.count (float *length))
    (quotient *c.count (float *length)))))

```

FOCUS.IN.CLASS?

description: Tests to see if the AA at position *pos (focus) in the protein given by *pro.name lies in class (A, B, C) *struct. Struct values are ALPHA and BETA.
This is the TRAINING feedback routine.

}

c: FOCUS.IN.CLASS?

```

instance.of  flow.pred
i.take       symbol symbol integer
i.give       flag
arguments    *struct *pro.name *pos
my.vars      *truth *position *first
algorithm    (do
  (bindq *truth F)
  (cond ((same? *struct 'ALPHA)
    \typical helix position slot val: ((33 38) (53 60) (65 71) (81 86))
    (bindq *position (value.of *pro.name 'HELIX.POSITION)))
    ((same? *struct 'BETA)
    \typical sheet position slot val: ((5 7) (21 25) (27 32) (51 54) (74 80))
    (bindq *position (value.of *pro.name 'SHEET.POSITION)))
    (T (bindq *position nil))) \NOT alpha, NOT beta
  (loop.until (or? *truth (null? *position)) \for each helix/sheet position pair
    (do (bindq *first (first *position))
      (bindq *truth (and? (not? (less.than? *pos (first *first)))
        (not? (greater.than? *pos (second *first)))))
      (bindq *position (rest *position))))
  (return *truth))

```

PARTITION.DATA

description: Assigns training and testing data from *protein.list to g1c.bals *train.list and *test.list according to the values of *train# and *test#.
Just takes a list of proteins and cuts it into two parts

}

c: PARTITION.DATA

```
instance.of  flow.func
my.creator   wjb
i.take       list
i.give       none
arguments    *protein.list
my.vars      *length *train.clip *test.clip *temp
algorithm    (do
  (bindq *length (length *protein.list))
  (if.true (greater.than? (plus *train# *test#) *length)
    (display> "Attempted to partition more data than exists!" error))
  (bindq *train.clip (minus *length *train#))
  (bindq *test.clip (minus (minus *length *test#) *train#))
  (bindq *temp (reverse *protein.list))
  (setq *train.list (reverse (clip.list *temp *train.clip)))
  (bindq *temp (reverse *temp))
  (bindq *temp (clip.list *temp *train#))
  (bindq *temp (reverse *temp))
  (setq *test.list (reverse (clip.list *temp *test.clip))))
```

SHIFT.DIST

description: Subtracts *distance from each of the entries of each pair in *pairs.

c: SHIFT.DIST

```
instance.of  flow.func
my.creator   wjb
i.take       list number
i.give       list
arguments    *pairs *distance
my.vars      *in *out *shifted.prs
algorithm    (do
  (bindq *shifted.prs nil)
  (loop.until (null? *pairs)
    (do (bindq *out nil)
      (bindq *in (first *pairs))
      (bindq *out (cons (minus (second *in) *distance) *out))
      (bindq *out (cons (minus (first *in) *distance) *out))
      (bindq *shifted.prs (cons *out *shifted.prs))
      (bindq *pairs (rest *pairs)))))
  (return (reverse *shifted.prs)))
```

INTERNAL-PAIRS

description: find pairs from a given *pairs.list which are within c.term & u.term

)

c: INTERNAL-PAIRS

```
instance.of  flow.func
my.creator   wjb
i.take       number number list
i.give       list
arguments    *lft.end *rt.end *pairs.list
my.vars      *pair *intern.pairs
algorithm    (do
```

```

(bindq *intern.pairs nil)
(loop.until (null? *pairs.list)
  (do (bindq *pair (first *pairs.list))
    (if.true (and? (not? (less.than? (first *pair) *lft.end))
      (not? (greater.than? (second *pair) *rt.end))))
      (bindq *intern.pairs (cons *pair *intern.pairs)))
    (bindq *pairs.list (rest *pairs.list))))
  (return (reverse *intern.pairs)))

```

=====

CREATE.SUBUNIT

description: make a new concept as a subunit of a given concept

e.g. hemoglobin has 4 subunits

}

c: CREATE.SUBUNIT

```

instance.of    flow.func
my.creator     wjb
i.take        symbol list list number list
i.give        symbol
arguments      *pro.name *helices *sheets *prev.subs.length *sub.data
my.vars       *con *c.term *n.term *sub.helices *sub.sheets
algorithm      (do
  (bindq *con (new.atom))
  (bindq *c.term (add1 *prev.subs.length))
  (bindq *n.term (plus *prev.subs.length (length *sub.data)))
  (bindq *sub.helices (internal.pairs *c.term *n.term *helices))
  (bindq *sub.helices (shift.dist *sub.helices *prev.subs.length))
  (bindq *sub.sheets (internal.pairs *c.term *n.term *sheets))
  (bindq *sub.sheets (shift.dist *sub.sheets *prev.subs.length))
  (set.value *con 'SUBUNIT.OF *pro.name)
  (set.value *con 'MY.DATA *sub.data)
  (set.value *con 'HELIX.POSITION *sub.helices)
  (set.value *con 'SHEET.POSITION *sub.sheets)
  (return *con))

```

=====

ORGANIZE.PROTEINS

description: examines each protein to determine whether it has subunits. If it does, concepts are created for each subunit from sequence, helix and sheet data and inserted in place of the protein name in the protein list. The protein list is then partitioned into testing and training data.

NEEDS TO BE REPLACED WITH A GENERAL THINGY

}

c: ORGANIZE.PROTEINS

```

instance.of    flow.func
my.creator     jp2
i.take        list
i.give        list
arguments      *protein.list

```

```

my.vars      *pos *temp.pro.list *pro.name *data *#subunits *helices *sheets
my.vars      *prev.subs.length *sub.data *con
algorithm    (do (display> " Organizing Proteins" print)
              (display> *protein.list print)
              (bindq *pos 1)                                \ keeps track of position of current protein
              (bindq *temp.pro.list *protein.list)
              (loop.until (null? *temp.pro.list)            \ for each protein
                (do (bindq *pro.name (first *temp.pro.list))
                    (bindq *data (reverse (get *pro.name 'MY.DATA)))
                    (bindq *#subunits (length *data))
                    \this assumes subunits are in nested lists from 'my.data
                    (if.true (greater.than? *#subunits 1) \ if more than one subunit ...
                      (do (bindq *protein.list (delete.element *protein.list *pos))
                          (bindq *helices (value.of *pro.name 'HELIX.POSITION))

                          (bindq *sheets (value.of *pro.name 'SHEET.POSITION))

                          (bindq *prev.subs.length 0)
                          (loop.until (null? *data) \ for each subunit of protein data
                            (do (bindq *sub.data (first *data)) \ get next subunit
                                (bindq *con (create.subunit *pro.name *helices *sheets

*prev.subs.length *sub.data)))
                                \ is this smart enough not to count same protein again?
                                (bindq *protein.list (insert.element *protein.list *pos *con))
                                (bindq *prev.subs.length (plus (length *sub.data) *prev.subs.length))
                                (bindq *data (rest *data))))
                                (bindq *pos (add1 *pos))))
                                (bindq *pos (add1 *pos))
                                (bindq *temp.pro.list (rest *temp.pro.list))))
              (return *protein.list))

```

{ ===== }

GET.PROTEINS

description: Gets the list of proteins,
 RANDOMIZES them and examines each to determine
 whether it has subunits.
 If it does, concepts are created for each subunit from
 sequence, helix and sheet data and
 inserted in place of the protein name in the
 protein list.
 The protein list is then partitioned into
 testing and training data.

NEEDS TO BE REPLACED WITH A GENERAL THINGY

}

c: GET.PROTEINS

```

instance.of  flow.func
my.creator   jp2
i.take       none
i.give       list
my.vars      *protein.list
algorithm    (do
              (bindq *protein.list (get 'PROTEIN 'SUBS))
              (bindq *protein.list (randomize.list *protein.list))
              (bindq *protein.list (organize.proteins *protein.list))

```

```
(partition.data *protein.list)
(return *train.list))
```

```
(=====
INIT-AA.SLOTS
description: Initializes the slots
              FEATURE.COUNT, ALPHA.COUNT, BETA.COUNT & COIL.COUNT
              of each AA to (0 0 ... 0) where the length of this list is given by *win.length.
)
```

```
c: INIT-AA.SLOTS
instance.of      flow.func
my.creator       wjb
i.take           number
i.give           none
arguments        *win.length
my.vars          *count *initial *AA.list *AA
algorithm        (do
  (bindq *count 1)
  (bindq *initial nil)
  (bindq *AA.list (get 'NATURAL-AMINO-ACID 'SUBS))
  (loop.until (greater.than? *count *win.length)
    (do (bindq *initial (cons 0 *initial))
      (bindq *count (add1 *count))))
  (loop.until (null? *AA.list)
    (do (bindq *AA (first *AA.list))
      (set.value *AA 'FEATURE.COUNT *initial)
      (set.value *AA 'ALPHA.COUNT *initial)
      (set.value *AA 'BETA.COUNT *initial)
      (set.value *AA 'COIL.COUNT *initial)
      (bindq *AA.list (rest *AA.list))))))
```

```
(=====
COUNT-AA'S
description: Call INIT-AA.SLOTS to initialize the
              FEATURE.COUNT, ALPHA.COUNT, BETA.COUNT & COIL.COUNT
              slots of each AA to (0 0 ... 0)
              where the length of this list is given by *win.length.
              Then for each AA in each of the proteins used for training
              examine the window determined by this AA and *win.rad:
              1. Increment the ith entry of the FEATURE.COUNT slot of the AA in window
                 position i.
              2. If the window is centered on an AA which is part of a particular class,
                 increment the ith entry of the corresponding slot of the AA in window position
                 i and set the classification flag. (A, B, C for proteins)
              3. Create an exemplar from the window, the current value of classification and the
                 exemplar weight (initially set to 1) and place it on *exemplar.list.
              e.g.: (window classification #use #correct)
                     ((gly glu pro ...) B 1 1)
)
```

```
c: COUNT-AA'S
instance.of      flow.func
my.creator       jp2 wjb
i.take           number
i.give           none
arguments        *win.rad
my.vars          *protein.list *pro.name *pro.array *pro.length *win.length *focus *
```

```

my.vars      *pos *disp *AA *feature.list *list *temp
            \ *flag *exemplar.list *exemplar *window *classification - Globals which should be locals
algorithm    (do (display> "Counting AAs" print)
(setq *exemplar.list nil)
(cond (*mix
      (bindq *protein.list (get.proteins)))
      (T (do (bindq *protein.list (get 'PROTEIN 'SUBS))
              (bindq *protein.list (organize.proteins *protein.list))
              (setq *train.list *protein.list))))
(bindq *win.length (add1 (times *win.rad 2)))
(init.aa.slots *win.length)
(loop.until (null? *protein.list)
  (do (bindq *pro.name (first *protein.list))
      (display> "pro.name:" print) (display *pro.name print)
      (bindq *pro.array (create.protein.array *pro.name))
      (bindq *pro.length (array1@ *pro.array 0))
      (bindq *focus 1)
      (loop.until (greater.than? *focus *pro.length) \For each focus in protein ...
        (do (setq *exemplar nil)
            (setq *window nil)
            (bindq *i 1)
            (setq *flag F)
            (loop.until (greater.than? *i *win.length) \For each element of window
              (do (bindq *disp (sub1 (minus *i *win.rad)))
                  (bindq *pos (plus *focus *disp))
                  (if.true (and? (greater.than? *pos 0)
                                  (not? (greater.than? *pos *pro.length)))
                    (do (bindq *AA (array1@ *pro.array *pos))
                        (if.true (and? (equal? *i 1)
                                        (and? (greater.than? *focus *win.rad)
                                              (less.than? *focus (add1 (minus *pro.length *win.rad)))))
                          (setq *flag T))
                    (if.true *flag
                      (setq *window (cons *AA *window)))
                    (bindq *feature.list (value.of *AA 'FEATURE.COUNT))
                    (if.true (null? *feature.list)
                      (do (display> *i print)
                          (display *aa print)
                          (display *feature.list print)
                          ))
                    (bindq *temp (add1 (nth *feature.list (sub1 *i))))
                    \ (display *temp print)
                    (bindq *feature.list (replace.element *feature.list *i *temp))
                    (set.value *AA 'FEATURE.COUNT *feature.list)
                    \ (display> "Count AA 4" print)
                    (cond ((focus.in.class? 'ALPHA *pro.name *focus)
                          (do (bindq *list (value.of *AA 'ALPHA.COUNT))
                              (bindq *temp (add1 (nth *list (sub1 *i))))
                              (bindq *list (replace.element *list *i *temp))
                              (set.value *AA 'ALPHA.COUNT *list)
                              (setq *classification 'A)))
                          ((focus.in.class? 'BETA *pro.name *focus)
                          (do (bindq *list (value.of *AA 'BETA.COUNT))
                              (bindq *temp (add1 (nth *list (sub1 *i))))
                              (bindq *list (replace.element *list *i *temp))
                              (set.value *AA 'BETA.COUNT *list)
                              (setq *classification 'B)))
                          (T (do (bindq *list (value.of *AA 'COIL.COUNT))

```



```

        (setq *temp (add1 (nth *list (sub1 *i))))
        (bindq *list (replace.element *list *i *temp))
        (set.value *AA 'COIL.COUNT *list)
        (setq *classification 'C)))
    ))
    (bindq *i (add1 *i))
  ))
  (if true *flag
    (do (setq *exemplar (cons 1 *exemplar))
      (setq *exemplar (cons 1 *exemplar))
      (setq *exemplar (cons *classification *exemplar)) \ (classification #use #correct)
      (setq *window (reverse *window)))

    (setq *exemplar (cons *window *exemplar))
    \ (window classification #use #correct)
    (setq *exemplar.list (cons *exemplar *exemplar.list)))
    (bindq *focus (add1 *focus))
  ))
  (bindq *protein.list (rest *protein.list)))
)

\ could sort exemplar list on weight ...

(setq *exemplar.list (reverse *exemplar.list))
(display> "***** AA slots set *****" print)
(display> "***** Exemplars created *****" print)
(display> (first *exemplar.list) print)
(display> "FIRST EXEMPLARS CREATED:" log)
(display> (first *exemplar.list) log)
)

```

--- **MAKE.TABLE**

description: Given a list of ratios build the table (2D array) of the SW-VDM values of this list.

```

c: MAKE.TABLE
instance.of flow.func
my.creator wjb
i.take list list
i.give symbol
arguments *ratio.list *label.list
my.vars *dum *table *i *j *trio.i *trio.j *ratio.i *ratio.j *delta *temp *next
algorithm (do
  (bindq *dim (add1 (length *ratio.list)))
  (bindq *table (create.array2 *dum *dum))
  (bindq *j 1)
  (bindq *temp *label.list)
  (loop.until (equal? *j *dum) \ fill row 0 with labels (AA's)
    (do (bindq *next (first *temp))
      (array2! *table 0 *j *next)
      (bindq *temp (rest *temp))
      (bindq *j (add1 *j))))
  (bindq *i 1)
  (bindq *temp *label.list)
  (loop.until (equal? *i *dim) \ fill column 0 with labels (AA's)
    (do (bindq *next (first *temp))
      (array2! *table *i 0 *next)

```

```

        (bindq *temp (rest *temp))
        (bindq *i (add1 *i)))
(bindq *i 1)
(loop.until (equal? *i *dim)
  (do (bindq *j 1)
    (loop.until (equal? *j *dim)
      (do (bindq *trio.i (nth *ratio.list (sub1 *i)))
        (bindq *trio.j (nth *ratio.list (sub1 *j)))
        (bindq *delta 0)
        (loop.until (null? *trio.i)
          (do (bindq *ratio.i (first *trio.i))
            (bindq *ratio.j (first *trio.j))
            (bindq *delta (plus (abs (minus *ratio.i *ratio.j)) *delta))
            (bindq *trio.i (rest *trio.i))
            (bindq *trio.j (rest *trio.j))))
          (array2! *table *i *j *delta ,
            (bindq *j (add1 *j))))
        (bindq *i (add1 *i))))
  (return *table))

```

CREATE.TABLES

description: For each window position i construct a list of ratio trnos.

The trnos are obtained by dividing the i:th element of each of the structure count slots by the i:th element of the FEATURE.COUNT slot for each of the 20 AA's.

These lists are then passed to MAKE.TABLE and the resulting handels are returned in a list.

```

)

c: CREATE.TABLES
instance.of      flow.func
my.creator       wjb
i.take          number
i give          list
arguments        *win.rad
my.vars          *AA.list *feature *win.length *temp *ratio.list *AA
my.vars          *num *denom *ratio *trno *tables
algorithm        (do
  (display> "Creating tables ..." print)
  (bindq *AA.list (get NATURAL.AMINO.ACID 'SUBS))
  (bindq *tables nil)
  (bindq *feature 1)
  (bindq *win.length (add1 (times *win.rad 2)))
  (loop.until (greater.than? *feature *win.length) \for each feature in a window
    (do (bindq *temp *AA.list)
      (bindq *ratio.list nil)
      (loop.until (null? *temp)
        \for each amino acie &&&& generalize
        (do (bindq *AA (first *temp))
          (bindq *trno nil)
          (bindq *denom (nth (value.of *AA FEATURE.COUNT) (sub1 *feature)))
          \ (display> "denom=" debug) (display *denom debug)

          (bindq *num (nth (value.of *AA ALPHA.COUNT) (sub1 *feature)))
          \ (display> "num1=" debug) (display *num debug)

```

```

      (cond ((equal? *denom 0)
              (bindq *ratio 0))
            (T (bindq *ratio (quotient *num (float *denom))))))
      (bindq *tno (cons *ratio *tno))
      (bindq *num (nth (value.of *AA BETA.COUNT) (sub1 *feature)))
\ (display> "num2:" debug) (display *num debug)

```

```

      (cond ((equal? *denom 0)
              (bindq *ratio 0))
            (T (bindq *ratio (quotient *num (float *denom))))))
      (bindq *tno (cons *ratio *tno))
      (bindq *num (nth (value.of *AA COIL.COUNT) (sub1 *feature)))
\ (display> "num3:" debug) (display *num debug)

```

```

      (cond ((equal? *denom 0)
              (bindq *ratio 0))
            (T (bindq *ratio (quotient *num (float *denom))))))
      (bindq *tno (cons *ratio *tno))
\ (display> "tno:" debug) (display *tno debug)

```

```

      (bindq *ratio.list (cons *ratio *ratio.list))
      (bindq *temp (rest *temp)))
      (bindq *ratio.list (reverse *ratio.list))
      (bindq *tables (cons (make.table *ratio.list *AA.list) *tables))
      (bindq *feature (add1 *feature)))
(display> "Table handles:" print) (display (reverse *tables) print)
      (return (reverse *tables))

```

```

GET.WINDOW

```

description: Returns the *data window centered on *focus with radius *win.rad. An error is reported if the requested window extends outside the data.

1

c: GET.WINDOW

```

instance.of  flow.func
my.creator   wjb
i.take       number list number
i.give       list
arguments    *win.rad *data *focus
mv.vars      *length *temp
algorithm    (do
              (bindq *length (length *data))
              (if.true (or? (not? (greater.than? *focus *win.rad))
                             (greater.than? *focus (minus *length *win.rad))))
              (display "GET.WINDOW: Requested window outside data!" error)
              )
              (bindq *temp (clip.list *data (sub1 (minus *focus *win.rad))))
              (bindq *temp (reverse *temp))
              (bindq *temp (clip.list *temp (minus *length (plus *focus *win.rad))))
              (return (reverse *temp)))

```

```

DELTA.AA

```

description: Returns the entry at row *AA_i and column *AA_j of table *k.

entry is Distance between *AAi and *AAj

```

)

c: DELTA.AA
instance.of    flow.func
my.creator     wjb
i.take         symbol symbol number
i.give         number
arguments      *AAi *AAj *k
my.vars        *AA.list *i *j
algorithm      (do
  (bindq *AA.list (get 'NATURALAMINOACID 'SUBS))
  \POSITION returns 0 if X not found in list ...
  (bindq *i (position *AAi *AA.list))
  (bindq *j (position *AAj *AA.list))
  (return (array2@ (nth *table.list (sub1 *k)) *i *j)))

```

DELTA.WINDOW

description: Returns the distance between the given windows using
 Salzberg's method with $r = 1$ and weight = 1.
 When $r = 1$, yields "manhattan" distance
 When $r = 2$, yields "euclidian" distance
 Salzberg uses 2 typically, but 1 on protein problem
 Ripple along window of given length, comparing "features"; sum up similarities, then
 multiply by weights

```

)

c: DELTA.WINDOW
instance.of    flow.func
my.creator     wjb
i.take         list list
i.give         number
arguments      *wind1 *exemp
my.vars        *k *sum *AAi *AAj *delta *wind2
algorithm      (do
  (bindq *k 1)
  (bindq *sum 0)
  (bindq *wind2 (first *exemp)) \ get window from exemplar
  (loop.until (null? *wind1)
    (do (bindq *AAi (first *wind1))
        (bindq *AAj (first *wind2))
        (bindq *delta (delta.AA *AAi *AAj *k)) \ distance between 2 feature values
        \ if r=2, you would square *delta here *****
        (bindq *sum (plus *sum *delta))
        (bindq *k (add1 *k))
        (bindq *wind2 (rest *wind2))
        (bindq *wind1 (rest *wind1))))
  \ now that you have sum, multiply by Wx times Wy -- the weights *****
  \ assume Wy is always 1.0 (for now) *****
  \ (display> " SUM=" print) (display *sum print)
  \ (display> " WT=" print) (display (get.weight *exemp) print)
  (bindq *sum (times *sum (get.weight *exemp)))
  \ (display *sum print)
  (return *sum))

```

GET.PREDICTION

description: Locates the exemplar closest to window and returns it
*delta.best is the distance metric of the closest find -- smaller=better
exemplars are stored as a triple:
e.g.: (window classification #use #correct)
((gly glu pro ...) B 1 1) -- A=helix, B=sheet, C=coil

}

c: GET.PREDICTION

```
instance.of    flow.func
my.creator     jp2 wjo
i.take         'list list
i.give         list
arguments      *window *exemplars
my.var:        *delta.best *exemp *delta *best
algorithm      (do
                (bindq *delta.best 1000)
                (loop.until (null? *exemplars) \For each exemplar
                            (do (bindq *exemp (first *exemplars))
                                (bindq *delta (delta.window *window *exemp))
                                \delta is adjusted for exemplar weight, if use.weights=T
                                \NOTE: if *delta = 0, exit loop -- you're done *****
                                (cond ((equal? *delta 0)
                                        (do (bindq *best *exemp)
                                            (bindq *delta.best *delta)
                                            (bindq *exemplars nil)))
                                        ((less.than? *delta *delta.best)
                                         (do (bindq *best *exemp)
                                             (bindq *delta.best *delta))
                                         (T nil))
                                        (bindq *exemplars (rest *exemplars))))))
                \kill loop
                \watch for best
                (display> " *BEST:" print) (display *best print)
                \ (display *delta.best print)
                (return *best))
```

RECALL.PREDICTION

description: Locates the exemplar closest to window and returns it
*delta.best is the distance metric of the closest find -- smaller=better
exemplars are stored as a triple:
e.g.: (window classification #use #correct)
((gly glu pro ...) B 1 1) -- A=helix, B=sheet, C=coil

}

c: RECALL.PREDICTION

```
instance.of    flow.func
my.creator     jp2
i.take         list list
i.give         list
arguments      *window *exemplars
```

```

my.vars      *delta.best *exemp *delta *best
algorithm    (do (setq *self.exemp nil)
              (bindq *delta.best 1000)
              (loop.until (null? *exemplars) \For each exemplar
                (do (bindq *exemp (first *exemplars))
                    (bindq *delta (delta.window *window *exemp))
                    \delta is adjusted for exemplar weight, if use.weights=T
                    (if.true (equal? *delta 0)
                        (setq *self.exemp *exemp)))

\remember "self"
              (if.true (not? (equal? *delta 0))

\skip "self"
              (do (if.true (less.than? *delta *delta.best)

\watch for next best
              (do
                (bindq *best *exemp)

                (bindq *delta.best *delta))))))
              (bindq *exemplars (rest *exemplars))))
              \ (display> " *BEST:" print) (display *best print) (display *delta.best print)
              (return *best ))

```

--- COLLECT.RESULTS

description: Give a lists of predicted and actual classes
determine the percent accuracy of alpha, beta, coil and overall prediction
return them in a list e.g. (.52 .17 .56 .50)

```

c: COLLECT.RESULTS
instance.of  flow.func
my.creator   wjb
i.take       list list
i.give       list
arguments    *actual *predicted
my.vars      *firstactual *first.predicted *%correct.A *%correct.B *%correct.C *%correct
my.vars      *total.A *correct.A *total.B *correct.B *total.C *correct.C
algorithm    (do
              (bindq *total.A 0) (bindq *correct.A 0)
              (bindq *total.B 0) (bindq *correct.B 0)
              (bindq *total.C 0) (bindq *correct.C 0)
              (loop.until (null? *actual)
                (do (bindq *firstactual (first *actual))
                    (bindq *first.predicted (first *predicted))
                    (cond ((same? *firstactual 'A)
                        (do (bindq *total.A (add1 *total.A))
                            (if.true (same? *firstactual *first.predicted)
                                (bindq *correct.A (add1 *correct.A))))))
                        ((same? *firstactual 'B)
                        (do (bindq *total.B (add1 *total.B))
                            (if.true (same? *firstactual *first.predicted)
                                (bindq *correct.B (add1 *correct.B))))))
                        ((same? *firstactual 'C)
                        (do (bindq *total.C (add1 *total.C))
                            (if.true (same? *firstactual *first.predicted)
                                (bindq *correct.C (add1 *correct.C))))))
                    (T (display "TEST.CLASSES: Bug# 'actual list' error"))

```

```

    )
    (bindq *actual (rest *actual))
    (bindq *predicted (rest *predicted)))
  (cond ((greater-than? *total.A 0)
    (bindq *%correct.A (quotient *correct.A (float *total.A))))
    (T (bindq *%correct.A 0))) \avoid dividing by 0
  (cond ((greater-than? *total.B 0)
    (bindq *%correct.B (quotient *correct.B (float *total.B))))
    (T (bindq *%correct.B 0))) \avoid dividing by 0
  (cond ((greater-than? *total.C 0)
    (bindq *%correct.C (quotient *correct.C (float *total.C))))
    (T (bindq *%correct.C 0))) \avoid dividing by 0
  (bindq *%correct (quotient (plus (plus *correct.A *correct.B) *correct.C)
    (float (plus (plus *total.A *total.B) *total.C))))
  (return (list *%correct.A *%correct.B *%correct.C *%correct)))

```

c: COLLECT.FINAL.RESULTS

```

instance.of flow.func
my.creator jp2
i.tskc none
i.give list
my.vars *%a *%b *%c
algorithm (do
  (cond ((greater-than? *cl.A 0)
    (bindq *%A (quotient *cor.A (float *cl.A))))
    (T (bindq *%A 0))) \avoid dividing by 0
  (cond ((greater-than? *cl.B 0)
    (bindq *%B (quotient *cor.B (float *cl.B))))
    (T (bindq *%B 0))) \avoid dividing by 0
  (cond ((greater-than? *cl.C 0)
    (bindq *%C (quotient *cor.C (float *cl.C))))
    (T (bindq *%C 0))) \avoid dividing by 0
  (return (list *%a *%b *%c)))

```

{ ===== }

CHECK.CLASSES

description: check to see which class is involved -- return the class
MUST BE GENERALIZED TO OTHER CLASSES

;

c: CHECK.CLASSES

```

instance.of flow.func
my.creator jp2
i.tskc symbol integer
i.give symbol
arguments *pro *focus
my.vars *%rslt
algorithm (do (cond ((focus.in.class? 'ALPHA *pro *focus)
  \generalize &&&&&
    (bindq *rslt 'A))
    ((focus.in.class? 'BETA *pro *focus)
    (bindq *rslt 'B))
    ((focus.in.class? 'GAMMA *pro *focus)
    (bindq *rslt 'C))
  (return *rslt)))

```

{ ===== }

TEST.CLASSES

description: Uses tables, exemplars, and test proteins to construct lists of actual and predicted classes which are used to generate test results.

Used for testing the training set to adjust weights, and for running test set.
MUST BE GENERALIZED TO OTHER CLASSES

c: TEST.CLASSES

```
instance.of      flow.func
my.creator       jp2 wjb
i.take          list list list
i.give          list
arguments        *tables *exemplars *proteins
my.vars          *actual *predicted *pro *pro.data *pro.length *focus *window *results *exemp
algorithm        (do
  (display> "Testing Results ..." print)
  (loop.until (null? *proteins)
    \for each test protein
    (do (setq *cla 0) (setq *clb 0) (setq *clc 0)
      (setq *cor.a 0) (setq *cor.b 0) (setq *cor.c 0)
      (bindq *actual nil) \clear the lists
      (bindq *predicted nil)
      (bindq *pro (first *proteins))
      (display> "Next Protein:" print) (display *pro print)
      (display> "Next Protein:" log) (display *pro log)
      (bindq *pro.data (value.of *pro MY.DATA))
      (bindq *pro.length (length *pro.data))
      (display> "pro.length:" print) (display *pro.length print)
      (display> "Window Radius:" print) (display *win.rad print)
      (bindq *focus (add1 *win.rad))
      (loop.until (greater-than? *focus (minus *pro.length *win.rad))
        \for each focus along sequence
        (do (bindq *actual (cons (check-classes *pro *focus) *actual))
          (display> "focus:" print) (display *focus print)
          \*actual grows as this loop runs
          \ (display> "actual:" print) (display *actual print)
          (bindq *window (get-window *win.rad *pro.data *focus))
          \ (display> "window:" print) (display *window print)
          (bindq *exemp (get-prediction *window *exemplars))
          (bindq *predicted (cons (second *exemp) *predicted))
          \NOW: update globals to keep track of how we are doing
          (inc-actual (first *actual))
          (if true? (first *predicted) (first *actual))
              (inc-predicted (first *predicted)))
          \*predicted grows as this loop runs
          \ (display> "predicted:" print) (display *predicted print)
          (if true? (equal? (mod *focus 10) 0)
            (do (bindq *results (collect-final-results))
              (display> "Intermediate Results:" print) (display *results print)
              (display> "Intermediate Results:" log) (display *results log)))
            (bindq *focus (add1 *focus))))
        (display> "predicted:" print) (display *pro print)
        (display> (reverse *predicted) print)
        (display> "PROTEIN LOOKS LIKE THE FOLLOWING:" log)
        (display> *pro log)
        (display> "ACTUAL:" log)
        (display> (reverse *actual) log)
```



```

        (display> " PREDICTED:" log)
        (display> ( reverse *predicted ) log )
        (bindq *results ( collect.final.results ))
        (display> " Final Results:" print ) (display *results print)
        (display> " Final Results:" log ) (display> *results log)
        (bindq *results ( collect.results ( reverse *actual ) ( reverse *predicted )))
        (display> " List Comparison Results:" log)
        (display> *results log)
        (bindq *proteins ( rest *proteins )))
    (return *results))

\ ( collect.results ( reverse *actual ) ( reverse *predicted )))

=====
TRAIN.WEIGHTS

description: Used for testing the training set to adjust weights.
            MUST BE GENERALIZED TO OTHER CLASSES
!

c: TRAIN.WEIGHTS
instance.of    flow.func
my.creator     jp2
i.take         iist list list
i.give         list
arguments      *tables *exemplars *proteins
my.vars        *actual *predicted *pro *pro.data *pro.length *focus *window *exemp *exemps
algorithm      (do
    (display> "Training Weights ..." print)
    (bindq *exemps nil) \ holds growing list of exemplars
    (bindq *actual nil)
    (bindq *predicted nil)
    (loop.until ( null? *proteins )
        \for each protein
        (do (bindq *pro ( first *proteins ))
            (display> "pro:" print) (display *pro print)
            (bindq *pro.data ( value.of *pro MY.DATA ))
            (bindq *pro.length ( length *pro.data ))
            (display> "pro.length:" print) (display *pro.length print)
            (display> " Window Radix:" print) (display *win.rad print)
            (bindq *focus ( add1 *win.rad ))
            (loop.until ( greater.than? *focus ( minus *pro.length *win.rad ))
                \for each focus on sequence
                (do (bindq *actual ( check.classes *pro *focus ))
                    (bindq *window ( get.window *win.rad *pro.data *focus ))
                    (bindq *exemp ( recall.prediction *window *exemplars ))
                    \get nearest neighbor
                    (bindq *predicted ( second *exemp ))
                    (if.true ( same? *actual *predicted )
                        { bindq *exemp ( inc.correct *exemp ) })
                    (bindq *exemp ( inc.use *exemp ))
                    \ ( display> "TRAINED" print ) (display *exemp print)
                    (bindq *exemps ( cons *exemp *exemps ))
                    (bindq *focus ( add1 *focus )))
                    (bindq *proteins ( rest *proteins )))
                (return *exemps))

\===== THE ALGORITHM

```

c: RUN.NEAREST.NEIGHBOR

```

instance.of      flow.func
my.creator       jp2 wjb
i.take          number number number number symbol
i.give          list
arguments        *win.radius *train *test *max.win.rad *wt
my.vars         *results *output *temp
algorithm        (do
  (display> "Is this repeated?" print)
  (display> "Window Radius=" log) (display> *win.radius log)
  (setq *seed 2.0) \not used &&&&
  (setq *win.rad *win.radius)
  (setq *train# *train)
  (setq *test# *test)
  (setq *ct.a 0) (setq *ct.b 0) (setq *ct.c 0)
  (setq *cor.a 0) (setq *cor.b 0) (setq *cor.c 0)
  (loop.until (greater-than? *win.rad *max.win.rad)
    (do (count.aa's *win.rad) \Sets globals *test.list *train.list and *exemplar.list
      (display> "train.list" log) (display> *train.list log)
      (display> "#training proteins:" log) (display (length *train.list) log)
      (display> "use weights:" log) (display> *wt log)
      (display> "#Exemplars =" log) (display (length *exemplar.list) log)
      (bindq *output (count.classes *exemplar.list))
      (display> "Exemplars: Tot #alpha #beta #coil %alpha %beta %coil:" log)
      (display> *output log)
      (display> "test.list" log) (display> *test.list log)
      \first, we build the feature value difference tables
      (setq *table.list (create.tables *win.rad))
      \TRAIN
      \now, we should test them on the training set and set weights
      (if true (same? *wt T)
        (setq *exemplar.list
          (train.weights *table.list *exemplar.list *train.list)))
      \TRAIN WEIGHTS
      (display> "#Exemplars after weight training =" log) (display (length *exemplar.list) log)
      \now, we test them on a test set
      (bindq *results (test.classes *table.list *exemplar.list *test.list))
      \TEST
      \ (display> "%alpha predicted by nearest neighbor:" print) (display (first *results) print)
      \ (display> "%alpha predicted by nearest neighbor:" log) (display (first *results) log)
      \ (display> "%beta predicted by nearest neighbor:" print) (display (second *results) print)
      \ (display> "%beta predicted by nearest neighbor:" log) (display (second *results) log)
      \ (display> "%coil predicted by nearest neighbor:" print) (display (third *results) print)
      \ (display> "%coil predicted by nearest neighbor:" log) (display (third *results) log)
      \ (display> "% overall predicted by nearest neighbor:" print) (display (fourth *results) print)
      \ (display> "% overall predicted by nearest neighbor:" log) (display (fourth *results) log)

      (setq *win.rad (plus *win.rad 2)) \inc window radius by 2
      (setq *seed 2.0) \not used &&&&
    ))

```

```
\( display> "FINAL RESULTS: " print ) ( display ( collect.final.results ) print )  
\( display> "Final Results:" log ) ( display> ( collect.final.results ) log )  
( display> "ALL DONE..." print ) )
```

{
ROUGHSET.SAK

description: basic Rough Sets modified by Scott King
based on RX-4.TST.T

need to do: rank ordering of attributes
generalize to multiple slot data moving
trim universe length to shortest data list length

to improve: X

PROBLEMS: X

notes: x

CHANGES:

2/09/93 first cut, NewCentury 12point, TabStops=4, LineWrap=100.
2/10/93 jp2: running on car data; generalizations for Browning study; first Browning run done
2/11/93 jp2: adding rankordering cols
}

{ ALMOST.EQUAL?

description: checks to see if two numbers are within some distance of each other
example input: 2.5 2.6 .15
example output: T
notes:

}

c: ALMOST.EQUAL?
sub.of predicate
i.take number number number
i.give flag
arguments *num1 *num2 *error
algorithm (between? *num2 (minus *num1 *error) (plus *num1 *error))

\=====
{ CALC.BETA

description: check degree of intersect of A in B
example input: (1 3 4) (1 3 4 6 7)
example output: 0 6
notes: x

}

c: CALC.BETA
instance.of flow.pred
i.take list list
i.give number
arguments *set1 *set2
my.vars *count *length
algorithm (do (bindq *length (length *set2))
(bindq *count 0)
(loop.until (null? *set1) \for every set1 member
(do (if.true { member? (first *set1) *set2)
(bindq *count (plus1 *count))
(bindq *set1 (rest *set1))))

```
( return ( quotient *count *length)))
```

```
{ SORT.COLS.ON.WORTH
```

```
description:  sort to decending worth-- "bubble sort"
example input:  list of concepts
example output: sorted list of concepts
notes:         made from SORT.ON.WORTH
```

```
}
```

```
c: GET.COL.WORTH
```

```
instance of  flow.func
i.take      number symbol
i.give      number
arguments   *col *array
algorithm   ( value.of ( array2@ *array 0 *col ) 'WORTH )
```

```
c: SORT.COLS.ON.WORTH
```

```
\ seems to work
```

```
instance of  flow.func
i.take      list symbol
i.give      list
arguments   *list *array
my.vars     *templist *moved
algorithm   (
  do (bindq *templist nil)
    (bindq *moved T)
    (loop.until (not? *moved)
      (do (bindq *moved F)
        (loop.until (null? *list)
          (do (\(display> "list debug") (display> "##" debug) (display *templist debug)
            (cond ((equal? (length *list) 1)
              (do (bindq *templist (cons (first *list) *templist))
                (bindq *list (rest *list))))
              ((greater.than?
                (get.col.worth (second *list) *array)
                (get.col.worth (first *list) *array))
              (do (bindq *templist (cons (second *list) *templist))
                (bindq *list (cons (first *list) (rest (rest *list))))
                (bindq *moved T))))
              (T (do (bindq *templist (cons (first *list) *templist))
                (bindq *list (rest *list)))))))
          (bindq *list (reverse *templist))
          (bindq *templist nil)))
    (return *list)
    \ (display> "sort.on.worth >>" debug) (display *list debug)
  )
```

```
COUNTING
```

```
COMMENT.
```

```
ALL ARRAY VALUES ARE SYMBOLIC
ALL ARRAY X&Y VALUES START AT 00
ROW 00 is reserved for NAMES
```

```
COMMENT:
```

{ COMPARE.ROWS?

description. see if currow (for each P col) is same as given row
 example input: x
 example output: x
 notes: THIS SHOULD ELIMINATE ANY ROWS
 WHICH ALREADY COMPARE -- to save some compute time
 eliminate by simply putting the row number on a list, then check
 the list before comparing all the columns in a row

}

c: COMPARE.ROWS?

instance.of flow.pred
 i take number list symbol list
 i.give flag
 arguments *row *cols *array *given \given is a given row of P data
 my.vars *truth
 algorithm (do (bindq *truth T)
 (loop.until (or? (null? *cols)
 \for every P col (not? *truth))
 \or mismatch found (do (bindq *truth (same? (array2@ *array *row (first *cols))
 (first *given))))
 (bindq *given (rest *given))
 (bindq *cols (rest *cols))))
 (return *truth))

{ COUNT.X.PRIME

description: collect partition of rows which look the same based on given set
 example input: x
 example output: list of lists of rows in array which look the same
 notes: x

}

c: COUNT.X.PRIME

instance.of math func
 i take list number symbol
 i.give list \ list of lists
 arguments *cols *max.y *array
 my.vars *row *checked *result *temp.v *holder *temp.s *xx
 algorithm (do (\display> "Checking X" on "print") (display *cols print) (display *max.y print)
 (bindq *result nil) \holds list of subsets
 (bindq *checked nil) \holds N values that have been counted
 (bindq *row *max.y)
 (loop.until (equal? *row 0) \for: every row except 0
 (do \take specified contents of row
 (bindq *temp.v nil)
 (bindq *xx *cols)
 (loop.until (null? *xx))
 \for this row, make a list of Pcol entries
 (do (bindq *temp.s (array2@ *array *row (first *xx)))
 (bindq *temp.v (cons *temp.s *temp.v))
 (bindq *xx (rest *xx))))

```

\ now have a list (in reverse order) of a given row
\ see if it has already been checked
(bindq *temp.v (reverse *temp.v))
(bindq *temp.s nil) \ holds subset of rows
(if.true (not? (member? *temp.v *checked))
  (do \ not checked yet, keep track of it
    (bindq *checked (cons *temp.v *checked))
    (bindq *holder *row)
    (loop.until (equal? *holder 0)

\ for every row above this row, skipping row 0
  (do \ gather this row
    (if.true

(compare.rows?

  *holder *cols *array *temp.v )

(bindq *temp.s

  (cons *holder *temp.s)))

  (bindq *holder (sub1 *holder))))))
  (if.true (notnull? *temp.s)
    (bindq *result (cons (reverse *temp.s) *result)))
  (bindq *row (sub1 *row))))
  (return (reverse *result)))

=====
{ IND.P.PRIME

  description: collect partition of rows which look the same based on P (given)
  example input: x
  example output: list of lists of rows in array which look the same
  notes: P is a list of columns
)

c: IND.X.PRIME
instance of math.func
i.take list number symbol
i.give list \ list of lists
arguments *pset *max.y *array
algorithm (count.x.prime *pset *max.y *array)

=====
{ N.PRIME

  description: collect partition based on N (given)
  example input: list of cols called N, max#rows in array, array
  example output: list of lists of rows in array which have same N value
  notes: find all members of N which have the same value
)

c: N.PRIME
instance of math.func
i.take list number symbol
i.give list \ list of lists
arguments *cols *max.y *array
algorithm (count.x.prime *cols *max.y *array)

```

{ LOWER.IND.P

description: collect partition based on subset of N in ind.p prime
example input: x
example output: list of rows in array
notes: RESULT IS NO LONGER ORDERED

}

c: LOWER.IND.P

instance of math.func
i take list list number \N.PRIME subset, IND.P.PRIME list
i give list \list of rows
arguments *n.prime.s *ind.prime *beta
my.vars *result
algorithm (do (bindq *result nil)
 (loop until (null? *ind.prime) \for every n.prime
 (do (if true (greater.than?
 (calc.beta (first *ind.prime) *n.prime.s)
 *beta)
 (bindq *result (concat (copy.list (first *ind.prime))
 *result))))))
 (bindq *ind.prime (rest *ind.prime))))
 \ (display> "LOWER.IND(P)" log)
 \ (display> *n.prime.s log) (display> *result log)
 (return *result))

{ POS.P.N

description: x
example input: x
example output: x
notes: RESULT IS NO LONGER ORDERED

}

c: POS.P.N

instance of math.func
i take list list number \N.PRIME list, IND.P.PRIME list
i give list \union
arguments *n.prime *ind.prime *beta
my.vars *result
algorithm (do (bindq *result nil)
 (loop until (null? *n.prime)
 (do (bindq *result
 (list union (lower.ind.p (first *n.prime) *ind.prime *beta)
 *result))
 (bindq *n.prime (rest *n.prime))))))
 \ (display> "POS(P,N)" log) (display> *result log)
 (return *result))

{ K.P.N

description: dependency of N on P
example input: x


```

example output  x
notes:          x
}

: xfi( (sym sym -- sym)
integer> float
integer> float fswap
f/ >floating;

c: FQUOTIENT
instance of  math.func
i.take      number number
i.give      number
forth      %xf/

c: K.P.N
instance of  math.func
i.take      list list number number
i.give      number
arguments   *n.prime *ind.prime *universe *beta
my.vars     *result
algorithm   (do (bindq *result 0.0)
              (bindq *result (fquotient (length (pos.p.n *n.prime *ind.prime *beta))
                                         *universe))
              (return *result))

=====
{ DO.ROUGHSETS

description:  the big routine
example input: x
example output x
notes:      x
}

c: DO.ROUGHSETS
instance of  flow.func
i.take      list list number symbol number
i.give      number \k(p,n)
arguments   *pset *nset *universe *beta
my.vars     *n.prime *ind.prime *kpn
algorithm   (do\ (display> "DOING ROUGH SETS" print)
              \ (display> "#####DOING ROUGH SETS" log)
              \ (display> *pset log)
              (bindq *n.prime (n.prime *nset *universe *array))
              \ (display> "N=" print) (display *n.prime print)
              \ (display> "N=" log) (display> *n.prime log)
              (bindq *ind.prime (ind.x.prime *pset *universe *array))
              \ (display> "IND(P)=" print) (display *ind.prime print)
              \ (display> "IND(P)=" log) / (display> *ind.prime log)
              (bindq *kpn (k.p.n *n.prime *ind.prime *universe *beta))
              (display> "K(P,N)=" print) (display *kpn print)
              (display> "K(P,N)=" log) (display *kpn log)
              (return *kpn)
            )

```

```

c: MAKE.PLIST
sub.of      function
my.creator  sak
i.take      number
i.give      list
arguments   *num
my.vars     *list *count
algorithm   (do (bindq *list nil)
              (bindq *count 1)
              (loop.until (greater.than? *count *num)
                (do (bindq *list (cons *count *list))
                  (bindq *count (add1 *count))))
              (return (reverse *list)))

c: FIND.MIN.SET
instance.of  flow.func
i.take      list list list number number number
i.give      list
arguments    *pset *nset *universe *curkpn *beta
\*cop
my.vars     *pset *new.kpn *sig
algorithm   (do \ (display> "FIND.MIN.SET" print)
              (loop.until (null? *cop)
                (do (display> "Deleted member" print)
                  (display (value.of (first *cop) 'my.col) print)
                  (bindq *pset (delete (value.of (first *cop) 'MY.COL) (copy.list *pset)))
                  (display> "PSET" log) (display> *pset log)
                  (bindq *new.kpn (do.rough.sets *pset *nset *universe *beta))
                  (bindq *sig (quotient (float (minus *curkpn *new.kpn)) *curkpn))
                  (if.true (almost.equal? *sig 0.005)
                    (do (display> "Toss that one away" print)
                      (bindq *pset (delete (value.of (first *cop) 'MY.COL) *pset))))
                  \ (display> "PSET" print) (display *pset print)
                  (array2! *array 0 (value.of (first *cop) 'MY.COL) *sig)
                  (bindq *cop (rest *cop))))
              (return *pset))

c: FIND.SIG.VALS
instance.of  flow.func
i.take      list list number symbol number
i.give      list \k(p,n)
arguments    *pset *nset *universe *array *curkpn
my.vars     *count *test *new.kpn *sig
algorithm   (do (display> "FIND.SIG.VALS" print)
              (bindq *count *pset)
              (bindq *test *pset)
              (loop.until (null? *count)
                (do (display> "Deleted member" print)
                  (display (first *count) print)
                  (bindq *test (delete (first *count) (copy.list *pset)))
                  (display *test print)
                  (bindq *new.kpn (do.rough.sets *test *nset *universe *array))
                  (bindq *sig (quotient (float (minus *curkpn *new.kpn)) *curkpn))
                  (array2! *array 0 (first *count) *sig)
                  (bindq *count (rest *count))))
              (return *pset))

```

```
( return *pset )
```

c: SORT&EVAL.RULES

```
instance.of  flow.func
i.take      list list number symbol
i.give      list      \k(p,n)
arguments   *rules *nset *universe *array
my.vars     *cop *pset *curkpn *psetp *new.kpn *sig
algorithm   ( do ( display> "SORT&EVAL.RULES" print )
              ( bindq *cop *rules )
              ( display *cop print )
              ( bindq *pset ( make.plist ( length *rules ) ) )
              \ ( display> "pset, nset, universe, array" print )
              \ ( display *pset print ) ( display *nset print ) ( display *universe print ) ( display *array print )
              ( bindq *curkpn ( do.rough.sets *pset *nset *universe *array ) )
              ( if.true ( not? ( equal? *curkpn 0 ) )
                ( do ( bindq *pset ( find.min.set *cop *pset *nset *universe *array *curkpn )
                      \ ( bindq *pset ( find.sig.vals *pset *nset *universe *array *curkpn )
                        ) )
                  ( display> "MINIMAL SET" print ) ( display *pset print ) ( display *curkpn print )
                  ( display> "MINIMAL SET" log ) ( display> *pset log )
                  ( display> "H's K.P.N. of" log ) ( display> *curkpn log )
                  ( return *pset ) )
              )
```

COMMENT:

BIG CODE THAT NEEDS TO BE BROKEN DOWN

```
\ =====
( FIND.CORE.SET

  description:  an even bigger routine
  example input: x
  example output: x
  notes:       x
)
```

c: FIND.CORE.SET

```
instance.of  flow.func
i.take      list list number symbol
i.give      none      \k(p,n)
arguments   *pset *nset *universe *array
my.vars     *curkpn *core *xpset *plen *xkpn *del *len *core *minset
algorithm   ( do ( display> "FINDING CORE SET" print )
                \ pset and nset are the entire universe on first pass
                ( bindq *curkpn ( do.rough.sets *pset *nset *universe *array ) )
                \ now, one at a time, pick vbls out of pset
                ( bindq *core nil )
                ( bindq *plen 0 )
                ( bindq *len ( length *pset ) )
                ( loop.until ( equal? *plen *len )
                  ( do ( bindq *xpset ( copy.list *pset )
                        ( bindq *del ( nth *pset *plen ) )
                        \ ( display> "ABOUT TO DELETE " print ) ( display *del print )
                        \ ( display> "from " print ) ( display *xpset print )
```

```

(bindq *xkpn ( do.rough.sets
  ( delete *del *xpset ) *nset

*universe *array ))

\ now, tell about this column's "worth"
(display> "COL=" log ) ( display ( array2@ *array 0 *del ) log )
(display ( minus *curkpn *xkpn ) log )
\ now save this difference in the frame of the col
\ bigger is better!
(set.value ( array2@ *array 0 *del ) "WORTH
  ( minus *curkpn *xkpn ))
\ if the col is needed to maintain *curkpn, it's part of "core" set
(if.true ( less.than? *xkpn *curkpn )
  ( bindq *core ( cons *del *core )))
(display> "CORE=" print ) ( display *plen print ) ( display *core print )
(display *pset print )
(bindq *plen ( add1 *plen )))
(display> "SORTING CORE SET:" print )
(bindq *core ( sort.cols.on.worth *core *array ))
(display *core print )
(display> "CORE SET=" log ) ( display> *core log )
\ now, test core set
\ core set is intersect of all minimum sets
(bindq *minset *pset ) \ the minimum default set
(if.true ( notnull? *core )
  ( do ( display> "NOW TESTING CORE SET" log )
    ( bindq *xkpn ( do.rough.sets *core *nset *universe *array ))
    ( display *xkpn print )
    ( display> *xkpn log )
    \ now off to find minimum sets
    \ but...
    \ if the core set gets a full k(pn), then the core set is the minset
    ( cond (( less.than? *xkpn *curkpn )
      ( bindq *minset
        ( find.min.sets *pset *nset *universe *array *core *curkpn )))
      ( T ( bindq *minset *core ))) )
  )
(display> "MIN SETS=" print ) ( display *minset print )
(display> "MIN SETS=" log ) ( display> *minset log )
)

```

COMMENT:

```

(
  EVOLVE.T
  build a populanon of protein evaluation rules using the GA approach.
)

(
  CHOOSE.SET.BY.WORTH
  description:   Given the current population, randomly choose a member, weighted by the current worth
                  value of every member of the population
  example input: ( con_101 con_202 con_303 ... )
  example output: con_4242
  notes:        uses rnd.float instead of rnd.mod to avoid problems caused by calling rnd.mod with large arguments
)

```

```

c: CHOOSE.SET.BY.WORTH
  sub.of   function
  i.take   list
  i.give    symbol
  arguments *population
  my.vars  *my.pop *total.val *num *set *val *this.worth
  algorithm ( do
    ( bindq *my.pop *population )
    ( bindq *total.val 0 )
    ( bindq *set nil )
    ( loop.until ( null? *my.pop )      \@@@@ get total worth of population
      ( do
        ( bindq *total.val ( plus *total.val ( value.of ( first *my.pop ) 'WORTH ) ) )
        ( bindq *my.pop ( rest *my.pop ) ) )
      ( bindq *num ( int ( times ( rnd.float ) *total.val ) ) )
    \@@@@ choose random value
    ( bindq *val 0 )
    ( if.true ( equal? *total.val 0 )      \@@@@ if no sets with worth, grab one at random
      ( bindq *set ( member *population ) ) )
    ( loop.until ( notnull? *set )        \@@@@ wait till got one
      ( do
        ( bindq *this.worth ( value.of ( first *population ) 'WORTH ) )
        ( bindq *val ( plus *val *this.worth ) )
      \@@@@ add worths till over random val
      ( if.true ( greater.than? *val *num )
        ( bindq *set ( first *population ) ) )
      ( bindq *population ( rest *population ) ) )
    ( return *set ) )
  )

```

```

(
  CROSS.RULES
  description:   Given a rule list, randomly choose two different rules from it, and switch a portion of their
                  RHS slots, returning the new rules stuck into the old rule list
  example input: ( con_1 con_2 con_3 ... )
  example output: ( con_1 con_3 con_4 ... )
  notes:        does not change the then.predict slot - maybe it could/should? NO
)

```

c: CROSS.RULES

```

sub.of    function
i.take    list
i.give    list
arguments *rule.list
my.vars   *father *mother *size *cross.pt *holder *where *slot *son *daughter *position
algorithm (do
  (bindq *father (member *rule.list))
  (bindq *mother (member *rule.list))
  (loop.until (not? (same? *father *mother)))
  \@@@ get two different rules
    (bindq *mother (member *rule.list))
    (delete *mother *rule.list)
    (delete *father *rule.list)
    \ (bindq *rule.list (remove.all (list *mother *father) *rule.list))
      (bindq *son (new.rule))
      (bindq *daughter (new.rule))
      (bindq *size (value.of (@cur.concept) *WINDOW.SIZE))
      (bindq *cross.pt (add1 (rnd.mod (sub1 *size))))
      (bindq *where (add1 *cross.pt))
      (bindq *position 1)
      (loop.until (equal? *position *where))
  \@@@ copy up to switch point
    (do
      (bindq *slot (change.to.symbol *position))
      (set.value *son *slot (value.of *father *slot))
      (set.value *daughter *slot (value.of *mother *slot))
      (bindq *position (add1 *position)))
    (loop.until (greater.than? *position *size))
  \@@@ switch after that
    (do
      (bindq *slot (change.to.symbol *position))
      (set.value *son *slot (value.of *mother *slot))
      (set.value *daughter *slot (value.of *father *slot))
      (bindq *position (add1 *position)))
      (set.value *son THEN.PREDICT (value.of *father THEN.PREDICT))
      (set.value *daughter THEN.PREDICT (value.of *mother THEN.PREDICT))
      (bindq *rule.list (concat (list *son *daughter) *rule.list))
      (return *rule.list))

```

(MUTATE.ONE.RULE

description: Given a rule list, randomly choose one rule, mutate it, and return the new list

example input: (con_1 con_2 con_3 ...)

example output: (con_1 con_2 con_4 ...)

notes:

)

c: MUTATE.ONE.RULE

```

sub.of    function
i.take    list
i.give    list
arguments *rule.list
my.vars   *mutatee *slot *amino *aa.list *num *new.rule *times
algorithm (do
  (bindq *mutatee (member *rule.list))
  (delete *mutatee *rule.list)
  (bindq *new.rule (new.rule))

```

```

(copy.p.list *mutatee *new.rule)
(bindq *times (add1 (rnd.mod (quotient (value.of (@cur.concept) 'WINDOW.SIZE) 2))))
(loop.until (equal? *times 0)
  (do
    (bindq *slot (rnd.mod (add1 (value.of (@cur.concept) 'WINDOW.SIZE))))
    (cond
      ((equal? *slot 0)
        (set.value *new.rule 'THEN.PREDICT (create.prediction)))
      (T
        (do
          (bindq *slot (change.to.symbol *slot))
          (bindq *aa.list nil)
          (bindq *num (add1 (rnd.mod 9)))
          (loop.until (equal? *num 0)
            (do
              (bindq *aa.list (union (get.amino) *aa.list))
              (bindq *num (sub1 *num))))
          (rnd.do
            (set.value *new.rule *slot *aa.list)
            (set.value *new.rule *slot (cons 'X *aa.list))))))
        (bindq *times (sub1 *times))))
    (bindq *rule.list (cons *new.rule *rule.list))
    (return *rule.list))

```

(CROSS.PAIR.SETS

description: Given the current population of rule-sets, choose two sets and cross-breed them, switching rules around
 example input: (con_101 con_202 con_303 ...)
 example output: (con_101 con_303)
 notes: tries to find sets that have different talents

}

c: CROSS.PAIR.SETS

```

sub.of      function
itake       list
give        list
arguments   *curr.pop
my.vars     *father *mother *father.loc *mother.loc *son *daughter *father.rules *mother.rules *son.rules
my.vars     *daughter.rules *this.rule *tries
algorithm   (do
  (bindq *father (choose.set.by.worth *curr.pop))
  (bindq *mother (choose.set.by.worth *curr.pop))
  (bindq *tries 0)
  (loop.until (or? (not? (same? (value.of *father 'BEST) (value.of *mother 'BEST)))
    (greater.than? *tries 3))
    (do
      (bindq *mother (choose.set.by.worth *curr.pop))
      (bindq *tries (add1 *tries)))
    (if.true (greater.than? *tries 3)
      (do
        (bindq *tries 0)
        (loop.until (or? (not? (same? (value.of *father 'WORST) (value.of *mother 'WORST)))
          (greater.than? *tries 3))
          (do
            (bindq *mother (choose.set.by.worth *curr.pop))
            (bindq *tries (add1 *tries))))))
        (loop.until (not? (same? *father *mother))

```

```

(bindq *mother (choose.set.by.worth *curr.pop)))
(display> "Crossing " debug) (display *father debug)
(display " and " debug) (display *mother debug)
(bindq *mother.rules (copy.list (value.of *mother RULE.LIST)))
(bindq *father.rules (copy.list (value.of *father RULE.LIST)))
(bindq *son (new.set))
(bindq *daughter (new.set))
(bindq *son.rules nil)
(bindq *daughter.rules nil)
(bindq *father.loc (add1 (rnd.mod (sub1 (length *father.rules)))))
(bindq *mother.loc (add1 (rnd.mod (sub1 (length *mother.rules)))))
(bindq *son.rules (concat (grab.first.n *father.rules *father.loc)
                          (clip.list *mother.rules *mother.loc)))
(bindq *daughter.rules (concat (grab.first.n *mother.rules *mother.loc)
                                (clip.list *father.rules *father.loc)))
(set.value *son RULE.LIST *son.rules)
(set.value *daughter RULE.LIST *daughter.rules)
(set.value *son ACCURACY nil)
(set.value *daughter ACCURACY nil)
(set.value *son CREATOR CROSS.PAIR.SETS)
(set.value *daughter CREATOR CROSS.PAIR.SETS)
(set.value *son CREATED *cycles)
(set.value *daughter CREATED *cycles)
(display "to form " debug) (display *son debug)
(display " and " debug) (display *daughter debug)
(return (list *son *daughter))

```

(MUTATE.ONE.SET

description: Given the current population of rule sets, choose one and mutate a percentage of its rules, as determined by the experiment frame

example input: (con_101 con_202 con_303 ...)

example output: con_404

notes:

}

c: MUTATE.ONE.SET

```

sub.of      function
i.take      list
i.give      symbol
arguments   *curr.pop
my.vars     *mutatee *mutation *range *percent *#rules *the.rules *chance *min
algorithm   (do
              (bindq *mutatee (choose.set.by.worth *curr.pop))
              (display> "Mutating " debug) (display *mutatee debug)
              (bindq *the.rules (copy.list (value.of *mutatee RULE.LIST)))
              (bindq *range (get 'BREED.INFO 'MUTATE.RANGE))
              (bindq *min (value.of 'BREED.INFO 'MUTATE.MIN))
              (bindq *percent (int (rnd.normal (first *range) (second *range))))
              (bindq *#rules (quotient (times (length *the.rules) *percent) 100))
              (if.true (less.than? *#rules *min)
                        (bindq *#rules *min))
              (loop.until (equal? *#rules 0)
                          (do
                            (bindq *the.rules (rnd.do
                                                  (cross.rules *the.rules)
                                                  (mutate.one.rule *the.rules)

```



```

      (cons (invent.rule) *the.rules)
    \ (delete (member *the.rules) *the.rules)
  ))
  (bindq *#rules (sub1 *#rules)))
(bindq *mutation (new.set))
(set.value *mutation 'RULE.LIST *the.rules)
(set.value *mutation 'ACCURACY nil)
(set.value *mutation 'CREATOR 'MUTATE.ONE.SET)
(set.value *mutation 'CREATED *cycles)
(display "to form " debug) (display *mutation debug)
(return *mutation))

```

{ REPRODUCE.ONE.SET

description: Given the current population of rule sets, choose one
 example input: (con_101 con_202 con_303 ...)
 example output: con_101
 notes:

}

c: REPRODUCE.ONE.SET

```

sub.of    function
i.take    list
i.give     symbol
arguments *curr.pop
my.vars    *parent *rules *new.rules *this.rule *chance
algorithm  (do
  (bindq *parent (choose.set.by.worth *curr.pop))
  (display> "Reproducing " debug) (display *parent debug)
  \ (bindq *new.rules nil)
  \ (bindq *rules (value.of *parent 'RULE.LIST))
  \ (bindq *chance (rnd.float))
  \ (if.true (less.than? *chance 0.50)
    \ (bindq *rules (reverse *rules)))
  \ (bindq *spot (rnd.mod (length *rules)))
  \ (bindq *new.rules (concat (clip.list *rules *spot) (grab.first.n *rules *spot)))
  \ (loop.until (null? *rules) \@@@ scramble order of rules for more variety
    \ (do \@@@ in cross-breeding
      \ (bindq *this.rule (member *rules))
      \ (bindq *new.rules (cons *this.rule *new.rules))
      \ (delete *this.rule *rules))
    \ (set.value *parent 'RULE.LIST *rules)
  (return *parent))

```

{ REPRODUCE.BEST.SET

description: Create a copy of the best set ever, and return it
 example input:
 example output: con_5454
 notes: x

}

c: REPRODUCE.BEST.SET

```

sub.of    function
i.take     none

```

```

i.give      symbol
my.vars     *parent *rules *new.rules *this.rule
algorithm   ( do
              ( bindq *parent ( new.set ) )
              ( bindq *rules ( value.of *best.set.ever 'RULELIST' ) )
              ( bindq *new.rules nil )
              ( loop.until ( null? *rules )
                ( do
                  ( bindq *this.rule ( new.rule ) )
                  ( copy.p.list ( first *rules ) *this.rule )
                  ( bindq *new.rules ( cons *this.rule *new.rules ) )
                  ( bindq *rules ( rest *rules ) ) ) )
              ( set.value *parent 'RULELIST *new.rules )
              ( set.value *parent 'ACCURACY ( value.of *best.set.ever 'ACCURACY ) )
              ( set.value *parent 'CREATOR ( value.of *best.set.ever 'CREATOR ) )
              ( set.value *parent 'CREATED ( value.of *best.set.ever 'CREATED ) )
              ( set.value *parent 'HELIX.MATT ( value.of *best.set.ever 'HELIX.MATT ) )
              ( set.value *parent 'SHEET.MATT ( value.of *best.set.ever 'SHEET.MATT ) )
              ( set.value *parent 'RAN.MATT ( value.of *best.set.ever 'RAN.MATT ) )
              ( set.value *parent 'TOTAL.MATT ( value.of *best.set.ever 'TOTAL.MATT ) )
              ( set.value *parent 'PERC.PRED ( value.of *best.set.ever 'PERC.PRED ) )
              ( set.value *parent 'INDIV.ACCURACY ( value.of *best.set.ever 'INDIV.ACCURACY ) )
              ( set.value *parent 'BEST ( value.of *best.set.ever 'BEST ) )
              ( set.value *parent 'WORST ( value.of *best.set.ever 'WORST ) )
              ( return *parent ) )

```

(REPRODUCE.BEST.WORTH

```

description:  Create a copy of the best worth set ever, and return it
example input:
example output:  con_5454
notes:         x

```

c: REPRODUCE.BEST.WORTH

```

sub.of      function
i.take      none
i.give      symbol
my.vars     *parent *rules *new.rules *this.rule
algorithm   ( do
              ( bindq *parent ( new.set ) )
              ( bindq *rules ( value.of *best.worth.set.ever 'RULELIST' ) )
              ( bindq *new.rules nil )
              ( loop.until ( null? *rules )
                ( do
                  ( bindq *this.rule ( new.rule ) )
                  ( copy.p.list ( first *rules ) *this.rule )
                  ( bindq *new.rules ( cons *this.rule *new.rules ) )
                  ( bindq *rules ( rest *rules ) ) ) )
              ( set.value *parent 'RULELIST *new.rules )
              ( set.value *parent 'ACCURACY ( value.of *best.worth.set.ever 'ACCURACY ) )
              ( set.value *parent 'CREATOR ( value.of *best.worth.set.ever 'CREATOR ) )
              ( set.value *parent 'CREATED ( value.of *best.worth.set.ever 'CREATED ) )
              ( set.value *parent 'HELIX.MATT ( value.of *best.worth.set.ever 'HELIX.MATT ) )
              ( set.value *parent 'SHEET.MATT ( value.of *best.worth.set.ever 'SHEET.MATT ) )
              ( set.value *parent 'RAN.MATT ( value.of *best.worth.set.ever 'RAN.MATT ) )
              ( set.value *parent 'TOTAL.MATT ( value.of *best.worth.set.ever 'TOTAL.MATT ) )
              ( set.value *parent 'PERC.PRED ( value.of *best.worth.set.ever 'PERC.PRED ) )

```

```
(set.value *parent 'INDIV.ACCURACY (value.of *best.worth.set.ever 'INDIV.ACCURACY))
(set.value *parent 'BEST (value.of *best.worth.set.ever 'BEST))
(set.value *parent 'WORST (value.of *best.worth.set.ever 'WORST))
(return *parent))
```

[EVOLVE.POP

description: Given the current population, and the best score ever, evolve the population, resurrecting the best set ever (accuracy and/or worth) if we seem to be on the wrong track

example input: (con_101 con_202 con_303 ...) .55

example output: (con_101 con_202 con_303 ...)

notes: what should the divine intervention criteria be?
updates the dead.set.list

c: EVOLVE.POP

```
sub.of      function
i.take     list number number
i.give     list
arguments  *curr.pop *best.score *best.worth
my.vars    *random *crossover *reproduce *mutate *new.set *dead.sets *new.pop *dead.list *help
algorithm  (do
```

```
(bindq *random (value.of 'BREED.INFO 'RANDOM))
(bindq *crossover (value.of 'BREED.INFO 'CROSSOVER))
(bindq *reproduce (value.of 'BREED.INFO 'REPRODUCE))
(bindq *mutate (value.of 'BREED.INFO 'MUTATE))
(bindq *help 'T)
(bindq *new.pop nil)
(if.true (less.than? (quotient *best.worth (float *best.worth.ever)) .85)
  (do
    (bindq *new.pop (cons (reproduce.best.worth) *new.pop))
    (bindq *reproduce (sub1 *reproduce))
    (bindq *help 'F)))
(if.true (and? (less.than? (quotient *best.score *best.ever) .80) *help)
  (do
    (bindq *new.pop (cons (reproduce.best.set) *new.pop))
    (bindq *reproduce (sub1 *reproduce))))
(loop.until (equal? *random 0)
  (do
    (bindq *new.pop (cons (create.new.set) *new.pop))
    (bindq *random (sub1 *random))))
(loop.until (equal? *crossover 0)
  (do
    (bindq *new.pop (concat (cross.pair.sets *curr.pop) *new.pop))
    (bindq *crossover (minus *crossover 2))))
(loop.until (equal? *mutate 0)
  (do
    (bindq *new.pop (cons (mutate.one.set *curr.pop) *new.pop))
    (bindq *mutate (sub1 *mutate))))
(loop.until (equal? *reproduce 0)
  (do
    (bindq *new.set (reproduce.one.set *curr.pop))
    (bindq *curr.pop (remove.all (list *new.set) *curr.pop))
    (bindq *new.pop (cons *new.set *new.pop))
    (bindq *reproduce (sub1 *reproduce))))
(bindq *dead.sets (remove.all (cons *best.worth.set.ever (cons *best.set.ever *new.pop))
```

```

                (value.of 'RULE.SET 'SUBS)))
(bindq *dead.list (value.of 'KILL.INFO 'DEAD.SET.LIST))
(set.value 'KILL.INFO 'DEAD.SET.LIST (list.union *dead.list *dead.sets))
(bindq *dead.sets (remove.all *dead.list *dead.sets))
(loop.until (null? *dead.sets)
  (do
    (kill.frame (first *dead.sets))
    (bindq *dead.sets (rest *dead.sets))))
(return *new.pop))

```