1990

Knowledge-based support systems for statistical software

Michael Ray Carley
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Artificial Intelligence and Robotics Commons, and the Statistics and Probability Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/9423

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313 761-4700 800 521-0600
Knowledge-based support systems for statistical software

Carley, Michael Ray, Ph.D.
Iowa State University, 1990
Knowledge-based support systems
for statistical software

by

Michael Ray Carley

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Departments: Statistics
Economics
Co-majors: Statistics
Economics

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

Signature was redacted for privacy.

For the Major Departments

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1990
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER II. KNOWLEDGE-BASED SYSTEMS</td>
<td>4</td>
</tr>
<tr>
<td>Overview</td>
<td>4</td>
</tr>
<tr>
<td>Implementation of Knowledge-Based Systems</td>
<td>9</td>
</tr>
<tr>
<td>Knowledge Representation</td>
<td>11</td>
</tr>
<tr>
<td>Prolog - A Computer Language for Predicate Logic</td>
<td>24</td>
</tr>
<tr>
<td>CHAPTER III. KNOWLEDGE-BASED COMPUTER SUPPORT</td>
<td>26</td>
</tr>
<tr>
<td>Conventional Support</td>
<td>26</td>
</tr>
<tr>
<td>Question-Answering Systems</td>
<td>27</td>
</tr>
<tr>
<td>Alternate Approaches</td>
<td>34</td>
</tr>
<tr>
<td>CHAPTER IV. EG EXPERT: A PROTOTYPE KNOWLEDGE-BASED EXPERT SYSTEM TO SUPPORT THE USE OF STATISTICAL SOFTWARE</td>
<td>38</td>
</tr>
<tr>
<td>Background and Objectives</td>
<td>38</td>
</tr>
<tr>
<td>System Overview</td>
<td>40</td>
</tr>
<tr>
<td>Using EG Expert</td>
<td>44</td>
</tr>
<tr>
<td>Implementation and Knowledge Representation</td>
<td>46</td>
</tr>
<tr>
<td>Comments and Conclusions</td>
<td>52</td>
</tr>
<tr>
<td>CHAPTER V. EG NETWORK - A KNOWLEDGE-BASED INFORMATION SYSTEM FOR STATISTICAL PACKAGES</td>
<td>57</td>
</tr>
<tr>
<td>Overview</td>
<td>57</td>
</tr>
<tr>
<td>The Network of Examples</td>
<td>59</td>
</tr>
<tr>
<td>Prolog Representation of the Example Network</td>
<td>69</td>
</tr>
<tr>
<td>Traversing the Network of Examples</td>
<td>75</td>
</tr>
<tr>
<td>Comments</td>
<td>91</td>
</tr>
</tbody>
</table>
CHAPTER VI. SUMMARY  93
BIBLIOGRAPHY  96
ACKNOWLEDGEMENTS  101
APPENDIX A. EG EXPERT INFERENCE PROCESS  102
APPENDIX B. PROGRAM LISTING FOR EG NETWORK  104
APPENDIX C. USER BROWSING PATH ALGORITHM  118
APPENDIX D. REACTIVE BROWSING DECISION EXAMPLES  120
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.</td>
<td>Components of a knowledge-based system</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.</td>
<td>A typical rule statement</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.</td>
<td>Forward-chaining inference example</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4.</td>
<td>Backward-chaining inference example</td>
<td>16</td>
</tr>
<tr>
<td>Figure 5.</td>
<td>Example of a rule incorporating uncertainty</td>
<td>18</td>
</tr>
<tr>
<td>Figure 6.</td>
<td>Links and nodes in a semantic network</td>
<td>19</td>
</tr>
<tr>
<td>Figure 7.</td>
<td>Example of a Regression Frame Network</td>
<td>23</td>
</tr>
<tr>
<td>Figure 8.</td>
<td>Internal operations in question-answering systems</td>
<td>29</td>
</tr>
<tr>
<td>Figure 9.</td>
<td>Overview of EG Expert’s (external) operation</td>
<td>44</td>
</tr>
<tr>
<td>Figure 10.</td>
<td>Overview of EG Expert’s (internal) operation</td>
<td>47</td>
</tr>
<tr>
<td>Figure 11.</td>
<td>Example of a general solution</td>
<td>49</td>
</tr>
<tr>
<td>Figure 12.</td>
<td>Topic link</td>
<td>60</td>
</tr>
<tr>
<td>Figure 13.</td>
<td>Across-subject link</td>
<td>61</td>
</tr>
<tr>
<td>Figure 14.</td>
<td>Within-subject link</td>
<td>62</td>
</tr>
<tr>
<td>Figure 15.</td>
<td>Detail-of link</td>
<td>63</td>
</tr>
<tr>
<td>Figure 16.</td>
<td>Same-detail-of link</td>
<td>63</td>
</tr>
<tr>
<td>Figure 17.</td>
<td>Network of examples</td>
<td>66</td>
</tr>
<tr>
<td>Figure 18.</td>
<td>Language links</td>
<td>68</td>
</tr>
<tr>
<td>Figure 19.</td>
<td>Language link SAS to description</td>
<td>69</td>
</tr>
<tr>
<td>Figure 20.</td>
<td>Frame representation of an example program</td>
<td>70</td>
</tr>
<tr>
<td>Figure 21.</td>
<td>Browse screen with SAS example and Description</td>
<td>77</td>
</tr>
</tbody>
</table>
Figure 22. Browse screen with MINITAB and SAS windows 79

Figure 23. EG Network after executing next section move 82

Figure 24. EG Network screen in mapped-traversal mode 85

Figure 25. Selection of Other Subjects 86

Figure 26. Selection of Other Details 88
CHAPTER I. INTRODUCTION

A popular research topic in statistical computing involves the application of methodology and programming techniques from the field of artificial intelligence to problems in statistics. Most efforts thus far have involved research into developing knowledge-based expert computer systems which emulate some of the activities of the expert statistical consultant - most involve either guiding a user through a correct statistical analysis or assisting a user in choosing an appropriate statistical technique. The motivation behind such systems has been the recognition that statistical software has become widely available and is being used more and more by those with little statistical training. This has opened the door for "much uninformed, unguided, and simply incorrect data analysis" (Chambers, 1981). Statistically naive users need guidance in the application of the data analysis techniques supported by the statistical software. Knowledge-based systems are one vehicle by which users can be provided with this guidance via explicit software implementations of a statistical consultant's strategy and expertise. For more information on this topic, the reader is directed to Chambers (1981), Hand (1984), Gale and Pregibon (1984), Hahn (1985), Gale (1986), and Streitberg (1989).
A more basic need for most users of statistical software is support (training and guidance) in using the software from a programming perspective versus an analysis perspective. Large general-purpose statistical software packages like SAS, SPSSX, and BMDP are extremely powerful but program development can often prove time consuming and frustrating, especially for inexperienced or occasional users. In general, people have trouble using and learning to use statistical packages and often seek guidance of one form or another. Conventional sources of computer software support (statistical software and otherwise) include short courses, paper manuals, computer-based simulations and tutorials. However, Leigh et al. (1987) note that short courses are rarely available at a user's convenience, paper manuals are often more general and abstract than might be best suited for a specific user and his problem, and computer-based tutorials are expensive and rarely developed at deeper than an overview level. Furthermore, research has shown (Lang et al., 1982 and O'Malley, 1986) that when users of computer software are in need of assistance, they prefer to consult other people (e.g., the "local expert") rather than to use manuals (on or off line) or other types of help available to them. As such, a new approach to provision of support for computer software systems has been the development of computer-based support systems which
capture the characteristics and expertise of the human software consultant. Again, knowledge-based systems are one vehicle by which users can be provided with this guidance via explicit software implementations of a software consultant's strategy, knowledge, and expertise.

The specific goal of this research is to study the design and development of computer-based systems that help people use and learn to use statistical software by providing them with workable example programs. Knowledge-based ideas and programming techniques will be used to develop these systems. In Chapter II, we first discuss the area of knowledge-based expert systems in general including the topics of knowledge representation and inference. Chapter III describes some applications of knowledge-based systems for other types of computer software including operating systems like UNIX. EG Expert, described in Chapter IV, is an example of a traditional expert system design which emphasizes the problem solving role of the human expert. EG Network, described in Chapter V, emphasizes the human expert's ability to provide people with information and to teach them.
CHAPTER II. KNOWLEDGE-BASED SYSTEMS

In this chapter, we provide a general overview of knowledge-based systems and discuss their implementation. Various knowledge representation schemes are discussed including rules, semantic networks, and frames. Finally, the Prolog programming language is introduced and examples of knowledge representations implemented in Prolog are given.

Overview

Knowledge-based systems are generally associated with an area of computer science called artificial intelligence. Artificial intelligence is a vast field covering topics from cognitive modeling, knowledge representation, and problem solving to robotics, machine learning, and natural language processing. The vastness of the field makes a general definition difficult, but Barr and Feigenbaum (1981) offer a suitable definition for our purposes: "AI is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior." Intelligence is difficult to define but it certainly involves the ability to acquire and apply knowledge. Consequently, much of the focus in applied
artificial intelligence research has been on the study of so-called knowledge-based systems. Literature concerned with this topic is extensive and includes Bobrow and Stefic (1986), Davis (1986), Waterman (1986), Kowalik (1986), Walker (1986), and Fisher (1986).

A knowledge-based system (KBS) is a computer program. The fundamental difference between a KBS and a traditional computer program is the explicit representation of the knowledge required to solve a problem—knowledge is not built into program code but rather exists as a separate entity referred to in some structures as the knowledge base. In this regard, the knowledge in a KBS is declarative as opposed to procedural, the emphasis being on the expression of "what to know" as opposed to "what to do". Implicit in this configuration is the feature that knowledge itself can be represented with symbolic forms suitable for computer manipulation. Furthermore, a KBS, through an inference mechanism, is able to reason with such symbolic forms in order to apply knowledge to the task at hand and thus appear to act intelligently. As Forsyth (1984) comments, the traditional viewpoint of DATA + ALGORITHM = PROGRAM is replaced with the alternative viewpoint of KNOWLEDGE + INFERENCE = SYSTEM (that is, intelligent system). Various knowledge representation structures and inference mechanisms have been studied by
researchers in artificial intelligence and are discussed in a later section.

The knowledge-based approach is often used to develop computer programs that model the behavior of a human expert. Such programs are commonly referred to as knowledge-based expert systems or simply expert systems. The knowledge structures in an expert system serve to capture human expertise in such a way that the expertise can be generally applied to problems within the system's domain. Current expert systems are designed to solve problems only within a narrowly defined domain. This idea is in contrast to early work in artificial intelligence which concentrated on the design of general, non-domain-specific problem solvers (Newell and Simon, 1963). In addition, expert systems are aimed at solving problems where the expertise involved is not algorithmic but rather more of an "art", based on experience and heuristic reasoning (rules-of-thumb). Gottinger (1988) gives an excellent example:

... fitting a curve through a cloud of data by nonparametric smoothing does not qualify as expert behavior - fitting is described by a well-defined algorithm. Choosing the most appropriate smoothing technique is expert behavior - it requires heuristic knowledge about what properties of the data are displayed by each technique, and which are important for the data set at hand.

Note that knowledge-based expert systems attack the types
of problems that are not easily handled using traditional procedural programming techniques. Examples include problems involving diagnosis, interpretation, evaluation, and planning. Expert systems have been successfully applied in many of these areas. MYCIN (Buchanan and Shortliffe, 1984), for example, is an expert system which assists physicians in diagnosing and treating antimicrobial infections. PROSPECTOR (Duda et al., 1979) aids geologists in the evaluation of mineral sites for potential ore deposits. R1 (McDermott, 1982) is an expert system for configuring large computer systems. TAXADVISOR (Michaelsen and Michie, 1983) is an expert system for tax and estate planning. The range of applications has been quite large and literature addressing expert systems is extensive. See, for example, Frost (1986), Forsyth (1984), Johnson and Keravnou (1985), Buchanan and Duda (1983), and Coombs (1984).

The term expert system has typically been associated with knowledge-based systems that emphasize the problem-solving or diagnostic role of the human expert. As Coombs and Alty (1984) note, this is due to the fact that most expert systems have been designed with the primary objective of finding a known solution to a well-circumscribed problem. Furthermore, the goals of most expert systems remain the same each time they are used.
MYCIN for example, is designed to identify the most likely infectious organisms based on patient information supplied by an attending physician - the only thing that changes with each use is the particular patient data. Coombs and Alty recognize that in real life an expert is more often called upon to provide conceptual guidance to associates and help them solve problems for themselves rather than simply provide them with an answer to a well-defined problem. Knowledge-based expert systems designed more toward this end are often more appropriately referred to as advice-giving systems or consulting systems, although the nomenclature is not well established. An example of such a system is KENS, developed by Hand (1987) within the domain of nonparametric statistics. KENS is described as a "flexible computer program for providing a user with information about nonparametric statistics" and was designed "not to solve problems for its user, but to assist the user to solve problems and to improve the user's understanding of nonparametric statistics." Hand coined the term knowledge enhancement system as a more appropriate description of the aims and capabilities of KENS (Knowledge Enhancement system for Nonparametric Statistics), thus emphasizing the more didactic role the system is intended to play in the problem solving process.
Implementation of Knowledge-Based Systems

The implementation of knowledge-based systems in artificial intelligence is a vast subject which has been treated extensively in many texts referenced earlier. Knowledge-based systems are generally composed of four major components as shown in Figure 1.

![Figure 1. Components of a knowledge-based system](image)

The knowledge base contains the symbolic constructs representing the system’s knowledge about a particular domain. We will see later that these constructs can take several forms - there are several different ways of representing knowledge in a computer program. The knowledge itself can be classified roughly into two general
categories: deep knowledge and surface knowledge. Deep knowledge refers to theories and accepted principles as well as causal models, abstractions, analogies, and so forth. Surface knowledge, on the other hand, can be thought of as that which is "compiled" from an understanding of deep knowledge. That is, surface knowledge is that acquired by experience and involves "know-how" and rules-of-thumb. Such knowledge is often referred to as heuristic knowledge and usually consists of empirical as opposed to theoretical associations. Knowledge-based expert systems generally contain more surface knowledge than deep knowledge.

The inference engine encapsulates the mechanisms for inference and control. Inference involves using the knowledge in the knowledge base to make deductions or perform tasks necessary to complete the goal of the system. At a lower level, inference involves manipulating the symbolic representations of knowledge in a meaningful way. Control is concerned with the overall operation of the system. This typically involves agenda control, that is, the control of what is to be done in what order. Control also involves how the knowledge is accessed and manipulated. Often, meta-knowledge (i.e., knowledge about knowledge) is employed to assist the system in deciding what rules (for example) are applicable to the problem at
hand, in what order they should be examined, and how conflict-resolution (multiple rules applicable) should be accomplished.

The user interface handles interaction with the user including dialogue and input/output. The user interface often involves a natural language processor which allows the user to communicate with the system in natural language format, although this capability is still very limited.

Another important element of any knowledge-based system is the knowledge acquisition module (KAM). In the case of a knowledge-based expert system, the KAM simplifies the transfer of knowledge from the human expert to the expert system and allows for the updating or modification of existing knowledge. In addition, the KAM often tries to verify that the information it receives is consistent with the existing store of knowledge. The KAM gives the knowledge-based system a rudimentary capability of learning by being told.

Knowledge Representation

The power of knowledge-based systems is in their ability to reason with explicitly declared knowledge. As a result, effective representation of such knowledge is essential. Winston (1984) lists several characteristics of good representations including the following:
- Good representations make the important things explicit,
- they expose natural constraints,
- they are complete, saying all that needs to be said,
- they are concise and efficient,
- they are transparent in that one can understand what is being said,
- they suppress detail, keeping rarely used information hidden until needed,
- they facilitate computation and manipulation.

None of the existing knowledge representation schemes fulfills all of these criteria nor is any completely satisfactory for all applications. Certain representation structures are more suitable for certain types of knowledge and it is not uncommon for more than one type of representation to be utilized. Regardless of the representation scheme(s) employed, a knowledge-based system must be able to make effective use of its knowledge. This capability involves broader subjects of inference and control. In the following sections, we give brief outlines of the major knowledge representation structures which have been studied by researchers in artificial intelligence. In particular we discuss rules, semantic networks, and frames. In addition, we comment on the inference and control structures associated with these types of structures and later discuss their Prolog implementation. More detailed
information on these representation schemes and their corresponding inference and control structures can be found in the following: Davis and Lenat (1982), Winston (1984), Hand (1985), Harmon and King (1985), Michaelsen et al. (1985), Fikes and Kehler (1985), Generseth and Ginsberg (1985), Johnson and Keravnou (1985), Hayes-Roth (1985), O'Hare and Bell (1985), and Walker (1986).

Rules

Rules (Newell and Simon, 1972) are perhaps the simplest form of knowledge representation available. Expert systems that use rules to capture expert knowledge are often referred to as rule-based expert systems. Rules are statements of the form IF <antecedent> THEN <consequent> as shown in Figure 2. The example rule represents a "chunk" of expert knowledge concerning the identification of infectious organisms. MYCIN's knowledge base is composed of hundreds of these types of rules.

IF the gram stain of the organism is gram negative
AND the morphology of the organism is rod
AND the aerobicity of the organism is anaerobic

THEN the identity of the organism is Bacteroides

Figure 2. A typical rule statement
Rules are the most widely used form of knowledge representation. The simplicity of the rule structure makes this approach appealing in many ways. For one thing, rules are both simple and homogeneous. In this respect, rules offer a relatively easy method by which knowledge can be encoded into a formal structure. In addition, individual rules are transparent; that is, it is easy to look at a single rule and know what it is saying. Rules are also modular in structure. This characteristic allows for incremental collection of knowledge through the addition of more and more rules - the more rules a system contains, the more "intelligent" or "expert" the system is. Modularity also implies that rules are independent "chunks" of knowledge having no direct links with one another. The deletion and modification of rules can thus be accomplished individually - the entire knowledge base need not be changed.

Rule-based systems are either antecedent driven (forward chaining) or consequent driven (backward chaining). Forward chaining systems operate as follows. Current information about the task at hand is kept in so-called "working memory". The system then uses this information to identify rules whose antecedents are satisfied - such rules are said to "fire". The consequents of fired rules are then executed accordingly. Such
execution often involves the placement of additional facts into working memory or the execution of some procedure which does so. An example of inference in a forward-chaining system is described next.

<table>
<thead>
<tr>
<th>Knowledge-base</th>
<th>Working Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) IF A and B THEN C</td>
<td>A, B, D</td>
</tr>
<tr>
<td>2) IF C and D THEN E</td>
<td></td>
</tr>
<tr>
<td>3) IF E and F THEN G</td>
<td></td>
</tr>
<tr>
<td>4) IF H THEN F</td>
<td></td>
</tr>
<tr>
<td>5) IF A THEN H</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Forward-chaining inference example

For this example, assume that the goal of the system is to establish that fact G is true, given the knowledge base and the initial contents of working memory (WM) shown in Figure 3. The forward-chaining inference process follows the following steps:

1) Initial state, WM:{A,B,D}
2) Rule 1 fires, infer fact C, WM:{A,B,D,C}
3) Rule 2 fires, infer fact E, WM:{A,B,D,C,E}
4) Rule 5 fires, infer fact H, WM:{A,B,D,C,E,H}
5) Rule 4 fires, infer fact F, WM:{A,B,D,C,E,F,H}
6) Rule 3 fires, infer fact G.

Forward chaining has established that fact G is true, given the initial contents of working memory and the rule base.
Backward-chaining rule systems are consequent driven and thus operate in a different manner. The backward-chaining process applied to the above example is depicted below in Figure 4.

![Diagram of backward-chaining inference example](image)

Figure 4. Backward-chaining inference example

Again, assume that the goal of the system is to establish that fact G is true, given the initial contents of working memory (WM). The backward-chaining procedure starts by identifying a rule which has as its consequent the fact G and then proceeds to establish that the corresponding antecedent holds true. In the above example, to establish fact G, facts E and F must be established.
Here, the process becomes recursive - to establish fact E, facts C and D must be established and so on. The process continues until either G is shown to be true (as it is the above example) or G is shown to be false (that is, G cannot be established by data and rules).

While both inference procedures seem straightforward enough, in real applications complexities arise. A typical knowledge base may consist of hundreds of rules and working memory will often contain a great amount of problem information. In attempting to match working memory information to rule antecedents, the entire rule base must be scanned and each rule's antecedent must be evaluated individually. Such scanning of the rule base can prove to be very inefficient for large rule bases. Furthermore, any number of rules should be able to fire subsequently. When they do, the system must be able to decide in what order these rules should be executed. Execution of the first fired rule may in turn induce changes in working memory. These changes may disengage certain rules and fire others. In many cases, the process can get very complicated and thus the transparency evident in individual rules is lost for the rules as a whole. In general, control mechanisms are necessary to handle the types of problems discussed above. As is apparent, the first-glance simplicity of rule
systems does not necessarily hold true for actual application problems.

In closing, we note that human experts do not always operate with perfect information. As a result, certain types of expert systems need the ability to make inferences under uncertainty. In rule-based expert systems, there can be uncertainty both in the knowledge base (rules and facts) and in the information obtained from the user by the system (data). The uncertainty in data, facts, and rules can be attributed to many sources. Frost (1986) lists some of these. Rule-based expert systems incorporate uncertainty by adding a qualifier to a rule statement as follows: If E then H with p. In this format, E represents evidence, H the resulting hypothesis, and p some measure of the strength of the relationship. Figure 5 gives an example of an actual rule used in the MYCIN expert system.

IF  site of culture is blood
AND organism was able to grow aerobically
AND organism was able to grow anaerobically
THEN  there is evidence that the aerobicity of the organism is facultative (.8) or anaerobic (.2)

Figure 5. Example of a rule incorporating uncertainty

The measure p may have many interpretations depending on what uncertainty approach is taken. The p may, for
example, be a probability, a probability range, or simply some ad hoc measure of certainty or belief. Whatever the interpretation, an expert system needs to be able to somehow use these measures in a meaningful way as rules are combined and inferences are made.

**Semantic Networks**

The semantic network (Quillian, 1968) is a traditional and very general knowledge representation scheme. Based on the idea that memory is composed of associations between concepts, semantic networks (also called associative networks) represent knowledge through a net structure composed of nodes and links. The nodes represent objects, concepts, events, situations, descriptions, ideas, and so on. The links (or semantics) express associations between the various nodes. Knowledge is then represented as a collection of nodes and links as illustrated in Figure 6.

![Figure 6. Links and nodes in a semantic network](image-url)
An advantage of semantic networks is the idea of inheritance. The REGRESSION TECHNIQUE node in Figure 6 may, for example, include the information that regression techniques can be used for prediction. Now, the network tells us that OLS is a type of regression technique. The inheritance property allows us to infer that OLS can be used for prediction - the OLS node need not contain this information. In general, "is_a" links imply property inheritance; that is, individual members of a class are assumed to possess the properties associated with the more general class to which they belong. The inheritance property allows us to reduce duplication of information and thus avoid redundancy.

Another advantage of the semantic network for knowledge representation is the inherent flexibility in the structure. To add additional knowledge or information, new nodes and links can be created as necessary. The flexibility of semantic networks allow the representation of more diverse types of knowledge.

Unfortunately, the flexibility of semantic networks also means more complex inference and control mechanisms. There is in fact no generally accepted set of links (or semantics) with which semantic nets are formed and thus the structure to some degree lacks formalism. As Hand (1985) notes, problems can occur if steps are not taken to prevent
the growth of arbitrarily large sets of links. For example, to use the knowledge encompassed in Figure 6, the system must know what meanings and implications are attached to a "property" or "condition" link. If we allow too large a set of links to be utilized, a separate knowledge base may be necessary to describe the meanings of the links themselves. Clearly, such a situation is not desirable. In general, the inference and control mechanisms needed to scan and draw conclusions from semantic networks are quite complex and are invariably problem specific (since arbitrary links can be used).

**Frames**

Frames (Minsky, 1975) offer an alternative but related method for representing knowledge. A frame consists of a set of slots which serve to capture all important information about an object, event, or procedure. A frame can thus be viewed as a chunk of a semantic network, that is, a construct which brings together all links and nodes which are associated with some particular item of interest. Knowledge about a particular subject is then represented as a collection of relevant frames. The entries in frame slots often contain not only specific values, but also procedures for obtaining those values, actions necessary
given that a certain value appears, pointers to additional frames, and other more detailed information.

A common example of frame usage involves creating a "blank" frame and then filling in the slots. For example, a frame for OLS regression could contain, among other things, slots for parameter values, test statistics, and assumptions specific to OLS. When an OLS regression problem is encountered by the system, a blank OLS frame could be created and filled in. The slots for parameter values might have associated with them procedures for calculating the parameter estimates. The slots for test statistics may point to other frames which describe the calculation and interpretation of such test statistics. Finally, the assumption slots may have procedural attachments which invoke certain actions given that an assumption is violated.

Frames, like semantic networks provide for inheritance. A frame will typically be of a certain general type and will inherit the characteristics associated with that general type. Frames will also contain additional slots which specify characteristics unique to that particular frame. In this regard, OLS, for example, can be described as a Linear_Regression_Technique (LRT) plus a set of properties which distinguish OLS from other LRTs. Likewise, a LRT can be described as a
Regression Technique (RgT) plus a set of properties which distinguish LRTs from other RgTs. Figure 7 illustrates these ideas.

Figure 7. Example of a Regression Frame Network

An entire knowledge framework can be built up by combining and expanding these types of frame systems. With frames, one can obtain very powerful and complete representations of knowledge. Frame systems, however, can get very complicated and the inference and control structures are generally more difficult to develop and implement than for other representation schemes.
Prolog - A Computer Language for Predicate Logic

PROLOG (PROgramming in LOGic), developed in the early seventies by Alain Colmerauer and others at the University of Marseilles, is a logic-based programming language that implements elements of first-order predicate calculus. PROLOG is a declarative programming language as opposed to a procedural one. In writing a PROLOG program, one does not specify (directly) how a problem is solved but rather one uses data structures called predicates to describe the problem (e.g., facts and rules relating facts) and the goal. Goal resolution is accomplished via a logic-based inference procedure which is itself a part of PROLOG. Examples given in this section are based on Borland's TURBO PROLOG implementation.

An example of a PROLOG fact is father(tim,joe). This statement simply expresses the fact that tim is the father of joe. Predicates can be combined to form sentences which define more complex relations. For example, the sentence brother(X,Y) :- father(Z,X),father(Z,Y) could be used to define a brother relationship. In this case X, Y, and Z are treated as variables and the sentence expresses the fact that X and Y are brothers if X and Y have the same father Z. A PROLOG program is a collection of these types of constructs such as
father(tim,joe)
father(tim,matt)
father(ted,frank)
father(ted,bob)
brother(X,Y) :- father(Z,X),father(Z,Y).

The job of the PROLOG interpreter is to resolve goals based on these facts. Resolution of a goal can involve simply an indication of the truth of a statement. For example, if we expressed the goal Goal: brother(joe,matt), PROLOG would respond with True. If we tried Goal: brother(joe,frank), PROLOG would respond with False. Resolution of a goal can also involve finding all values of a variable which make a stated goal true. For example, given Goal: brother(X,Y), PROLOG responds with X=joe Y=matt, X=frank Y=bob indicating that there are two pairs of brothers. Note that PROLOG involves primarily symbolic rather than numeric computation.

The PROLOG language is especially useful for representing knowledge and their associated inference procedures and is thus commonly used to develop expert systems and other AI applications. We will see later, in our discussions of EG Expert and EG Network, how PROLOG can be used to represent knowledge in the form of rules and semantic networks. References on PROLOG and logic programming in general include Colmerauer (1985), Cohen (1985), Campbell (1984), Clocksin and Mellish (1984), Kluzniak and Szpakowicz (1985), and Kowalski (1979).
CHAPTER III. KNOWLEDGE-BASED COMPUTER SUPPORT

As mentioned earlier, a basic need for most users of statistical software is support (training and guidance) for the programming activities involved with such packages - this is especially true for inexperienced or occasional users. We will see in this chapter how knowledge-based systems are being developed and used to provide such support in other computer-related domains. In particular, we will see applications for operating systems (e.g., UNIX and VMS), components of operating systems (e.g., UNIX-Mail), and fourth-generation database systems. Existence and features of these systems provide motivation for the development of a knowledge-based support system aimed at statistical software.

Conventional Support

Conventional sources of computer software support (statistical software and otherwise) include documentation and printed manuals, computer-based tutorials, and classroom short courses. Downfalls of these conventional sources are well-recognized by several authors including Leigh et al. (1987), Bannon (1986), O'Malley (1986), and Lang et al. (1982). Briefly, short courses are rarely available at a user's convenience, computer-based tutorials
are expensive and seldom developed at deeper than an overview level, and manuals are often more general and abstract than might be best suited for a specific user and his/her problem. In addition, Lang et al. (1981) have found that "very few users take advantage of available courses [and tutorials], but tend to pick up the knowledge they need as they go along." On documentation, Bannon (1986) has written, "There is accumulating evidence that users do not read manuals, no matter how well-written."

In general, research has shown (Coombs and Alty, 1984 and O'Malley, 1986) that when users of computer software are in need of assistance, they prefer to consult other people (e.g., the "local expert") rather than to use manuals or other types of help available to them. As such, a new approach to provision of support for users of computer systems has been the development of knowledge-based support systems. These systems attempt to capture the characteristics and expertise of the human software consultant (i.e., the local expert) in the form of a computer program that can be made generally available to users.

**Question-Answering Systems**

An obvious advantage of a human expert for the provision of software support is that the expert can
communicate with users. For example, a human expert can (usually) quickly respond to a user question about what command to use for some situation or problem. This type of interaction generally produces a quicker answer than if the user would "go it alone" and try to find the information himself using manuals or help systems. The advantage is thus a reduction in the investment in time required to obtain necessary information, especially for users who are unfamiliar with the computer system and the documentation associated with the system. Question-answering systems thus represent an attempt to make system information more accessible to users by allowing users to express questions about the system in somewhat the same form as they would to a human consultant.

Two examples of knowledge-based support systems that can understand and respond to queries in natural language are the UNIX Consultant (Wilensky et al., 1984) and the QUIZ Advisor (Skuce et al., 1988 and Constant et al., 1987). The UNIX Consultant (UC) is a natural language help system which can understand and answer user questions about the UNIX operating system. For example, if a user types "How can I compare two files?", UC responds "To compare two files, type 'diff file1 file2'". Note that UC can only respond to questions, not engage in any form of general dialogue with the user. The QUIZ Advisor is able to answer
"how do I" questions about a fourth-generation software product called QUIZ (a database report writer). A typical question form is "How do I report an item only after a subtotal?". Initially, the QUIZ Advisor gives a generic answer in the form "Use <command> with <option>" (for the above example, "Use a FOOTING AT statement" is the generic answer). If requested, the Advisor can extend the answer and actually generate the QUIZ code necessary, although it seems that this capability is currently very limited and has not been a focus of the research project.

Figure 8. Internal operations in question-answering systems
In general, the internal operations of question-answering systems follow the outline in Figure 8. The input to the system is a user's query specified in natural language format. The first operation is to read the user's statement (query) and produce an internal representation of the statement's meaning. The representation is then passed on to a goal analyzer which (generally) uses a forward-chaining rule-based inference technique to determine the user's likely goal. Once a user's goal is recognized, the system plans out a solution to the user goal (i.e., finds the appropriate command) and once done, produces an internal representation of the solution. The last step is to translate the generated plan into a natural language response.

UC and the QUIZ advisor both follow this general design, but efforts have been focused on different components within the design. UC's strength lies in its ability to analyze the linguistic structure of user questions in order to recognize underlying user goals and intentions. In terms of trying to emulate the actions of a human consultant, the most important (and the most interesting) step in UC's operation is the goal-analysis step. Human consultants are able to translate a user's stated goal into a goal in terms of the software or computer system being used (and/or its documentation). For
example, the question "How do I cancel a print job" means, in terms of the UNIX system, "How do I remove a file from the line printer queue". Once the goal is determined in terms of the computer system operations, finding the appropriate command is usually straightforward. In fact, most of the time, UC simply matches a user goal (once determined) to a pre-stored planned solution associated with that goal.

The Quiz Advisor's operation is very similar but the focus seems more on the planning of the solution. In fact, the steps leading up to the planning stage are not nearly so distinct as they are in UC. In parsing the input query, a forward chaining set of rules is used to directly identify QUIZ constructs (commands, subcommands) relevant to answering the query. This is unlike UC which produces first a distinct representation of a goal and then (in most cases) matches that goal to some preplanned solution. The planning stage for the Quiz Advisor thus involves piecing together the constructs relevant to a solution in order to produce a meaningful reply. A unique feature of the Quiz advisor involves the development of its natural language grammar. Unlike UC, which uses a very general phrase analyzer to parse natural language input, the QUIZ Advisor's natural language grammar was constructed after an in-depth study of actual questions encountered by
consultants in the product's telesupport group. The study of questions was used not only in developing the grammar but also as a basis for defining the set of knowledge necessary to address typical user problems.

There are problems associated with question-answering systems. On the technical side, computer understanding of natural language input is a challenge due to the variety of natural language grammars that can be encountered along with the ambiguities sometimes present in natural language input. Thus, most systems require users to restrict their input to some particular grammatical form. Conveying these grammatical restrictions to the user is a problem - once a certain form of natural language input is accepted, users expect the system to be able to understand anything they enter. Another problem concerns the abilities of users to ask the right questions. Hartley and Smith (1986) have found that "inexperienced users find it difficult to identify their specific knowledge needs and ask clear questions." Thus the usability of question-answering systems is at issue for those users who are unable to formulate a question in such a way that they can obtain relevant information. Such might be the case for a new user whose goal is to learn about the system - the user would not necessarily know what to ask about or how to ask in a way that the system could respond.
Question-answering systems for statistical software could be useful in some cases. However, most application programs are far too involved to be concisely summarized in a short question. A typical regression application, for example, might involve reading data, fitting a model, plotting residuals, and saving predicted values in an output dataset. It would be quite inconvenient for a user to express such a lengthy request that completely describes his problem. Furthermore, a user might not be aware of all the things a particular package can do or even all the things he might want it to do. Thus, the user might not really be able to express a question that adequately describes the information he desires. Finally, there would be implementation problems involved with parsing user input in the form of lengthy, complex question structures.

A better option might be to allow the user to specify some major area of interest (like regression) and then let the system ask the user a series of questions in order to find out the specifics of the user's needs. This is the approach taken within our development of EG Expert (see Chapter IV). Another option would be to show the user a series of example programs meant to exemplify what options are available and how to implement them. This, to some degree, is the approach taken within our development of EG Network (see Chapter V).
Alternate Approaches

As an alternative to the question-answering systems that attempt to understand natural language input, Hartley and Smith (1986) have worked on an "intelligent" help system called EXPLAINER for UNIX-Mail which anticipates user questions and in particular can "generate menus of questions which seem to suit the working context and the user's knowledge, so that selections could be made." Their strategy is thus to generate a series of best-guess menus that "span the user's request". When a user hits the help key, the result is a menu of questions that seem appropriate given the user's previous actions and the resulting anticipated user goal(s). Obviously then, this system places heavy emphasis on the ability to correctly anticipate user goals and intentions based on recent command use and help requests in the current context. The primary activity involved in such a system is the matching of user actions with prestored plan grammars. A user model is also employed to keep track of what the user knows and to thus avoid offering him a question that he already knows the answer to.

We have embedded the philosophy of this approach into our EG Network system. In particular, after showing the user an example program, we offer him a list of other interesting and related examples that he could view. Our
selections, however, are not based on monitoring the user or updating some model of the user. Rather, our selections are based more on our heuristic knowledge about common applications and useful programming features. This selection procedure is described more fully in Chapter V.

Another approach that is somewhat related to question answering can be seen by looking at the TEACHVMS system (Billmers and Carifio, 1985). TEACHVMS is a rule-based expert system designed to help users learn the about the VMS operating system. The unique feature of TEACHVMS is that the assumed audience is a user who knows another operating system language (in this case TOPS20). The interface to TEACHVMS resembles the TOPS20 environment. Users enter TOPS20 commands which are familiar to them and the system responds with a command set that accomplishes (as closely as possible) the same result using VMS operating system commands. This system can thus take advantage of users' general knowledge about operating systems and their particular knowledge about specific commands within one operating system to help, instruct, or advise them about a new system. In this case, the "natural" language input most appropriate is the language of the operating system the user is already familiar with. Input is not in the form of a direct question but rather in terms of the TOPS20 command(s) used to perform a specific
operation with the implicit question being "how do I accomplish this same goal in VMS?" Again though, the general strategy followed is that found in Figure 8. Rule-based inference techniques are first used to translate the inputed command(s) into a specific goal and then to identify the command(s) necessary to accomplish that goal in an alternative operating system environment.

Within both EG Expert and EG Network, we have developed processes whereby a user can view an example implemented in any of the packages supported. Using EG Network, for example, a user can simultaneously view on the screen example programs implemented in (at least) two different package languages. Our approach in EG Network, as we will see, is not to translate between languages as is done in TEACHVMS, but rather to associate already-complete example programs within the knowledge base.

DCL (Shrager and Finin, 1982), a system designed to help users learn the Vax/VMS operating system, is another example of an "intelligent" help system. This system, however, monitors user actions and provides unsolicited advice whenever appropriate. DCL is thus like the expert user who watches over your shoulder and breaks in whenever appropriate advise can be given (i.e., "Don't do that!" or "A better way of doing that is to ..."). In order to provide such unsolicited advise, DCL contains catalogs of
common user activities, inefficient plans novice users often employ to carry out these plans, and the corresponding more efficient methods. If DCL matches user activities to one of the inefficient plans, the user is interrupted and provided with information on the preferred method.

We have not attempted such a user-monitoring activity within either of the systems that we have developed. Although the idea is appealing, implementation seems impractical for anything much beyond a trivial example.
CHAPTER IV. EG EXPERT: A PROTOTYPE KNOWLEDGE-BASED EXPERT SYSTEM TO SUPPORT THE USE OF STATISTICAL SOFTWARE

In this chapter we discuss an initial experiment with a knowledge-based system developed to support the use of statistical software. EG Expert is a prototype knowledge-based expert system designed to answer general "how do I?" questions about statistical software. The system is capable of first directing a query process to extract relevant information from a user and then constructing an example program for the user to view and or use. Critical analysis of EG Expert prompted the development of EG Network, which is described in Chapter 5.

Background and Objectives

Graduate students in the statistical computing section of the Iowa State University Statistics Laboratory staff a university-wide consulting room to help support the use of statistical packages. The primary packages supported are SAS and SPSSX, although MINITAB and BMDP are also available. The main function of the consultants is to provide support for the programming aspect of package usage; that is, the consultants help users write and debug SAS code (for example). Although formal training is not a specific function of the consulting room, many of the
contacts can be viewed as short training sessions on the user’s topic of interest. The consultants do not provide a general statistical consulting service - clients generally know, or are assumed to know, what type of statistical analysis is appropriate for their problem, but they need help in using or learning to use a particular statistical package for their problem.

The consulting room is very popular as program development within these statistical packages can often prove time consuming and frustrating, especially for inexperienced or occasional users. Furthermore, manuals for the packages are not always readily available and when they are, tend to be overbearing and difficult to use. Our own observations support Lang et al. (1982) observations that users of computer software, when in need of assistance, prefer to consult other people (the programming consultants in our case) rather than to wade through manuals and search for the information themselves.

The consulting room provides a valuable service to users of statistical software in the university community but, because of time constraints, availability of consultants is limited to only four hours per day. Furthermore, the service is available only on a walk-in basis - users in need of assistance cannot call in their questions.
With these constraints in mind, we established a goal of developing an expert system which could emulate some of the activities of the programming consultants. The expert system would capture the expertise of the programming consultants in such a way that their expertise could be made generally available to users in the form of a computer program (operating on a DOS-based personal computer). Of course, this computer program could be accessed at a user's convenience in terms of both time and location. The particular consulting activity targeted for implementation was the answering of general "how do I" questions like "How do I read in this data with SAS?" Questions of this type are very common in the consulting room and can generally be answered by providing the client with an example program which they can modify for use with the particulars of their problem. EG Expert is a prototype implementation of an expert system designed to model such an activity.

System Overview

A user initiates a consulting session with EG Expert by choosing some major area of interest (e.g., reading data) from a menu of choices. Given the major area of interest, EG Expert proceeds to interrogate the user with questions about the particulars of the problem (Where is the data? What form is it in? and so on). Note that EG
Expert really only answers indirect and very general "how do I?" questions. Choosing "reading data" from the menu is much like asking a very general question like "How do I read in data?". Also, the system always offers the user a list of suitable responses to its questions, from which he can simply choose the most appropriate. The use of menus in general alleviates two problems. From a development standpoint, the high overhead of incorporating natural language understanding into EG Expert is avoided. From a usability standpoint, the choices eliminate the problem of a user not knowing how to ask or answer a particular question. Once EG Expert has all the necessary information from the user, it forms an annotated example program and presents it to the user. The user can then capture this example code in a text file and modify it as necessary for use. Operationally, this procedure seems to be a reasonable model of a human consultant's activities in a similar situation. That is, a user rarely confronts a consultant with a highly specific question - the initial question is usually very general in nature. The consultant then proceeds to ask the user more and more specific questions until he has enough information to provide the user with an answer.

The expertise EG Expert must possess in such a scenario involves two major areas. First, just like its
human counterpart, the expert system must know what information is relevant for a given major area of interest. Thus, if the user indicates that he wishes to "read some data", the system must, for example, know that information about the location of the data is relevant. In essence, the system needs to be able to ask all the right questions given a user's general goal and his responses to previous questions.

Secondly, the system must be able to process the responses and solve the problem. That is, given the user responses to appropriate questions, the system needs to be able to actually construct the example program. E.g. Expert accomplishes this as a two-part process. The first involves identifying what major characteristics to include in the example program. This set of general characteristics make up what is referred to as the generic solution. For example, if the system has found that the user is reading data from an external file, then it knows that one characteristic necessary in the example program is some indication to the statistical package about the external file's location and name. The second step involves mapping this generic solution into specific code elements (program statements or keywords) for a particular statistical package (the final example). For example, if the system finds that the user's data are located in an
external file, it must know that the INFILE statement is appropriate if a SAS example program is desired. Furthermore, the system needs to know where the INFILE statement is located with respect to other code elements also necessary (e.g., after the DATA statement and before the INPUT statement). Of course, since multiple packages are to be supported, EG Expert must be able to make this translation for any of the supported packages. In fact, it is because multiple packages are supported that EG Expert first generates a solution in generic code. Later, we will show that an advantage of this approach is that a developer can create the core knowledge base of EG Expert without regard to any particular statistical package.

A pictorial overview of EG Expert's operation is given below in Figure 9. PROLOG implementation of EG Expert and the representation of the knowledge (expertise) involved is covered in a later section. Note that selection of a major area of interest initiates the query process. This query process is further driven by the user's responses. The information obtained from the user during the query process is fed into the procedure for producing the generic solution. After the generic solution is produced the final example is produced for a particular package of interest.
Using EG Expert

This section shows EG Expert being used for a very simple example. The output from EG Expert is given in boldface while the user's responses are in regular typeface. A user initiates a consulting session with EG Expert by selecting the appropriate major topic of interest from a menu of choices. Assume that this user has chosen the topic "Reading Data", the session proceeds as follows:
What package [SAS, SPSSX, BMDP, MINITAB]? SAS

Data to be read from *inline* or from *external* file? external

Date type *system* or *raw*? raw

Data format *free* or *other*? free

An example SAS program:

data work; *identifies operation as data input
   *gives name to data set being created
   with the name work
   infile 'fname.txt'; *identifies source of data as external
   *identifies type of data as raw with
   keyword infile
   input y x; *identifies variable names as y and x
   *describes format of variables on
   input record as free format

** End of Example **

Had the user chosen MINITAB as the package of interest,
the example program generated would have been as follows:

read 'fname.txt' c1-c2 *identifies operation as data
   *data set name not relevant in
   MINITAB
   *identifies source of data as external with file specified in
   quotes
   *identifies type of data as raw with keyword read
   *identifies variable names as c1 and c2
   *describes format of variables on
   input record as free format by
   default
Note that the annotations follow the same pattern. This is because they are developed from the generic solution first and then specialized to the particular package during the translation step.

Implementation and Knowledge Representation

Production of the generic solution

Figure 10 outlines the internal organization of EG Expert's knowledge structures and inference procedures. Production of the generic solution is accomplished via an inference procedure whose goal is to identify what major characteristics to include in the example program. The user sees this inference procedure as a series of questions - a query process. The inference procedure that produces the generic solution is driven by three types of knowledge. These include knowledge about major topics supported by the system, general program characteristics available in example programs (not specific to any particular package), and relevant queries and user responses that help identify the specific set of characteristics to be included for a particular situation. We will now consider these elements of knowledge individually and give examples of their representations within EG Expert.
Figure 10. Overview of EG Expert's (internal) operation

- major topics
- characteristics available in example programs
- relevant queries and possible responses

-------------------

* inference procedure to produce generic solution

-------------------

* generated solutions

-------------------

* inference procedure to produce code for specific package

-------------------

* final example

-------------------

* coding elements for package SAS
* coding elements for package SPSSX
* etc. for all packages supported

(——— knowledge elements; ——— inference procedures; ****** generated solutions)
The query process is initiated by the user indicating some major topic of interest. To do this, a user simply chooses from a menu. The menu of choices is generated from an internal representation of the major topic areas currently known to the expert system. Thus, EG Expert has a rudimentary capability of "knowing what it knows" and of course only offers options which it can currently handle. Prolog representation of this knowledge is accomplished via simple predicates as:

```
  topic(readdata)
  topic(regression).
```

These two predicates are a simple representation of the fact that the system can currently help users with the major topics of data input and regression.

The generic solution to be produced can be viewed as an example program coded in a generic package language or pseudocode. The generic solution is made up of what are called structural elements together which serve to describe or specify the characteristics to be present in the final example. The query process seeks to identify what set structural elements should be included for a user's particular request within some major topic area. Thus, EG Expert has a representation of what structural elements are available. An example of this representation in Prolog is

```
  element(idi,"identifies operation as data input").
```

This
predicate shows that there is a structural element labeled as idi which represents the action of identifying the operation as data input. A generic example program can be represented as a set of these types of elements such as \{idi, dsn, ids, idt, file, ivar, format\}. This collection provides a general representation of an example program as shown in Figure 11.

idi : identifies the operation as data input
dsn : gives name to data set being created
ids : identifies source of data
idt : identifies type of data
file : identifies DOS file name with data
ivar : identifies variable names
format: describes format of variables on input record

Figure 11. Example of a general solution

Representation of the elements themselves is not enough. We also need a set of information that relates the presence of the structural elements to certain topics and certain responses to queries. Some examples of representation of these relationships are:

structure(idi):-topic(readdata).
structure(file):-source(external),type(raw).

The first predicate indicates that structure idi should be present in an example involving the general topic of reading data. The second predicate indicates that the structure file should be present in an example involving
input of raw data from an external file. The second predicate can also be interpreted as a rule of the form "IF the data is from an external source AND the data type is raw data THEN the structure file is required".

Since the selection of a set of elements is driven by a query process, representation of relevant questions about certain structural elements is required. Such representation is accomplished as

```prolog
query(ids):-write("\nData to be input *inline* or from *external* source ? "), readln(S),assert(source(S)),!.
```

This predicate simply identifies a query associated with the structural element ids (identify data source) that reads the user's input and asserts the appropriate response into working memory. For example, if the user indicates that the data are to be input from an external file, the fact source(external) would be added to the Prolog fact base.

The inference procedure and its associated query process, by referencing the above knowledge structures, seek to identify what structural elements should be included for a user's particular request. The user's choice of a major topic initializes this procedure by placing a set of structures on the "active" list (those to be included in example). Presence of structural elements on the active list lead to queries of the user and the
resulting information in turn leads to the activation of additional structural elements. The process continues and is thus dynamic, recursive, and can be viewed as a forward chaining inference procedure. The process ends when the system has no further questions to ask (i.e., the system needs no more information). An algorithmic representation of this process is

Activate structures based on major goal

Any queries associated with active structures?

YES - query user
activate additional structures or add additional facts based on response

NO - no additional information needed

Generic solution is complete.

A simple descriptive example of this process is given in Appendix A.

Production of the final example

In this section we discuss the translation of the generic solution into a program for a particular package. As mentioned earlier, this involves mapping the generic solution into specific code elements (program statements or keywords) for a particular statistical package. As such, the system must have a representation of specific code
elements for all packages. Some examples of this representation for the SAS package are:

\begin{verbatim}
sas(idi, "\ndata work; with the keyword data")
sas(ivar, "\ninput y x; " as y and x").
\end{verbatim}

The first predicate gives the SAS example code element and its annotation for the structural element idi. Note that the general annotation for this element is specialized by combining the general annotation string "identifies operation as data input" (see earlier example) with the above string " with keyword data input".

Comments and Conclusions

The prototype implementation of EG Expert was successful in showing that a software representation of the consulting activity involved in answering "how do I" questions was possible. However, there were some shortcomings and concerns that led us not extend the prototype into a full-scale system. These problems are discussed below. Recognition of these problems and their possible solutions steered us toward a new approach and the development of EG Network which is described in Chapter 5.

Problems in implementation

It is important to recognize that the inference procedures in EG Expert are independent of the particular
knowledge content. That is, it does not matter to the system what major topics are included, what structural elements are defined, or what particular languages are supported as long as the knowledge is encoded in the correct form. Of course, the more "knowledge" EG Expert has encoded, the more "expert" the system will be - the more situations it will be able to handle. This characteristic should allow for easy extension and modification of the knowledge within EG Expert. However, a major problem in implementation came into play when trying to extend this system to one of any reasonable magnitude. That is, to one that could handle enough situations such that it would be considered a useful system. Use of statistical packages involve a wide range of applications and it seems a formidable task to construct a system capable of handling even a significant number of interesting situations. Further, a knowledge base capable of handling such might be so large as to be intractable. Related to this was a more general situation of being able to fashion the knowledge base to suit the needs of any particular consulting site. We did not necessarily see our system as being one all-complete system which could be generally distributed, but rather a skeleton system that, for example, various departments could tailor to their specific needs and applications. For example, an MIS
Department might have a system focusing primarily on applications of data manipulation and reporting versus one involving several applications of ANOVA problems.

A major problem in extensibility and/or modifiability of the system involved simple recognition and definition (or redefinition) of a useful set of structural elements. Keeping in mind that collections of the structural elements together made up the general solution, elements had to be defined in such a way that they spanned the requirements of all particular packages supported by the system and to some degree the requirements of all particular examples the system could generate. Thus, changes in packages supported and or application areas supported necessarily involved changes in the set-up of the structural elements and their relations. Since the relations become somewhat complicated beyond any simple example, modification or addition of even one element might produce cause for a restructuring of the entire network or knowledge base. As a result, allowing for extensibility of the system in a general and flexible way would necessitate the production of a so-called knowledge acquisition module to manage and oversee any changes or additions to the knowledge base. The knowledge acquisition module would contain metaknowledge or knowledge about knowledge such that it could assist the producer in adding or changing knowledge within the system. Production
of such a KAM seemed a major task and was not attempted, rather a new approach was taken that removed the need for such a module. Similar problems occur when trying to add or modify the knowledge about particular coding for a certain package.

A general conclusion we made was that the granularity of the knowledge involved on the system was too small. In fact, as we will see in the next chapter, the granularity could go as far as complete examples versus elements which make up the examples.

Problems in Usability

More crucial than the problems described above in terms of implementation involved shortcomings we observed in the usability of the system from the user's standpoint. The traditional expert system situation of reading a series of questions, providing answers, and then finally getting an answer seemed a bit inflexible for the way we saw the system as actually being used. A typical user might be using the system not to obtain the answer to a specific problem but rather to extend their knowledge in a more general way. For example, a user interested in learning to read data might be interested in seeing several examples, one involving data input from a file, one involving data input from inline data, one from a system file existing on
an external drive, etc. EG Expert does not lend itself well to these types of interactions. Rather, the user must go through the same or an overlapping series of questions for each unique example they wanted to see. Further, if the user had a question about one very small detail of a problem, in order to construct a complete example, the system would still have to ask several questions for completeness, eventually coming to the one that is relevant. In other words, the user would be asked about things they already know because EG Expert needs a complete set of information to construct an example. We thought of modifying EG Expert in such a way that users could indicate or the system could deduce in some way the user’s knowledge prior to the query process being undertaken. In other words, the system could form a user model about the user and operate in a special way for unique classes of users. This again seemed like a major undertaking that could be avoided by simply rethinking the way in which such a system could be implemented.
In this chapter, we describe EG Network, a knowledge-based information system for statistical packages. An overview of the system is given along with a description of the Prolog implementation of the example network. Finally, methods for traversing the network are described.

Overview

EG Network represents a completely different approach to the supporting of statistical software when compared to EG Expert. Recall that EG Expert is a system capable of directing a query process to extract information from a user and then constructing a relevant example program based on the user's responses. It was found that the query process of EG Expert was much too inflexible for maximum user benefit and the procedure for constructing examples required knowledge constructs that were difficult to extend and or modify. EG Network is meant to be a more user-oriented system. It can be described as a knowledge-based information system that contains an integrated collection of example programs linked together in the form of a graph or network. The system assists users in accessing relevant
information and sets of information within the network in meaningful and flexible ways. Note that while EG Expert involved querying a user and constructing an example program, EG Network will contain already complete examples and the emphasis will be more on the provision of information to the user.

The core of EG Network is the network structure of nodes and links. The nodes contain the text representing example programs or program segments. The links represent associations between the nodes (i.e., between the example programs). A user will obtain information by moving throughout the network, viewing the contents of relevant nodes and making meaningful jumps to other interesting nodes. The knowledge in EG Network thus consists of two components. The first comes from the example programs. Each example is a representation of how to accomplish a certain task using a software package. This is the form of knowledge that the user is directly interested in gaining access to. The second comes from the links defined within the network. Each link represents knowledge about how two example programs are related. EG Network uses this knowledge to assist the user in deciding which nodes are relevant and which jumps are meaningful.
The Network of Examples

In order for EG Network to be able to assist users in traversal of the example network, it needs to know how the nodes (example programs) within the network are related. In this section, we see how a hierarchical structuring of the example programs gives rise to links within the network.

At the highest level of this hierarchy is the concept of a topic. Topics break examples into major areas such as reading data and regression analysis. Within a topic, example programs are further categorized by subject area. Within the topic of regression analysis, for example, some subject areas are specifying a model, creating output datasets, and requesting variable selection routines. Finally, within a particular subject area, examples are further classified according to the level of detail involved.

The highest level link between two nodes is directly related to the topic level and is thus called a topic link (tl). This link simply represents the fact that two examples are from different major topic areas as shown in Figure 12 (for SAS example programs). Example E1 is an example program within the major topic area of regression. Example E2 is an example program within the major topic area of reading data. If a user were currently located at
example E1, traversal via the tl link would take him to example E2. Of course, many different topic links might exist for any given node. To minimize the number of defined links, a topic-entry example is assigned for each major topic area. Any time a user moves within a particular topic area for the first time, he will be located at this topic-entry example. Topic links are then only explicitly defined between topic-entry examples.

![Figure 12. Topic link](image)

Within the same topic, we define another link between two nodes called an across-subject link (asl). This link represent the fact that two examples are about different subjects within the same major topic group. Figure 13 shows two examples involving the major topic of regression analysis. Example E3 is concerned with the subject of variable selection in regression programs while example E4 is concerned with the subject of hypothesis testing of the regression coefficients. Again, many possible across-subject links could be defined within a large network of
examples. To minimize the number of explicitly defined links, a subject-entry example is defined for each of the subject areas within a topic. The first time a user moves within a given subject area, he is located at this subject-entry example. Across-subject links are then only explicitly defined between subject-entry examples. Note that in can be the case that a topic-entry example might also serve as a subject-entry example for a particular subject.

```
E3
Proc Reg;
Model Y = X1 - X5
/ Method=Stepwise;
<-(asl)->
E4
Proc Reg;
Model Y = X1 - X5;
Test X1+X2=1;
```

Figure 13. Across-subject link

A related link is the within-subject link (wsl). This link connects two examples within the same subject area as shown in Figure 14. Both examples here are about the subject of variable selection in regression programs but example E3 involves stepwise selection while example E4 involves forward selection. Within-subject links are defined only for so-called primary examples within a subject as defined below.
Within a particular subject we can also define a detail-of link (dol). Figure 15 shows that this link simply represents the fact that one example is a more detailed version of another. In the context of our organization, example E6 is a detailed version of example E3 because it contains an extra level of detail within the code. Essentially, the two programs are examples of the same procedure - stepwise regression. Example E6 however is more detailed because it shows how to explicitly set the entry significance level. If a user were currently viewing example E3, a move to example E6 via the detail link might be of interest.

Examples like E6, which is a more detailed version of E3, is referred to as a secondary example within a subject. Nodes like E3, which in this case is not a more detailed version of some other example, is referred to as a primary example. Within-subject links, explained above, are only defined between primary examples. All subject-entry
examples are primary examples. Additional primary examples can serve as entry points to other sub-subjects within a particular subject area.

\[
\begin{align*}
E3 & \quad \text{Proc Reg;} \\
& \quad \text{Model } Y = X1 - X15 \\
& \quad / \text{ Method=Stepwise;} \\
& \quad \text{(dol)} \\
E6 & \quad \text{Proc Reg;} \\
& \quad \text{Model } Y = X1 - X15 \\
& \quad / \text{ Method = Stepwise} \\
& \quad S\text{lentry} = .25;
\end{align*}
\]

Figure 15. Detail-of link

Two secondary example programs that are both details of the same node are also related. This link is called a same-detail-of link (sdo). Figure 16 shows this association. These two programs are both more detailed examples of the stepwise regression procedure in example E3. Example E6 shows how to set the entry significance level while example E7 shows how to set the stay significance level. If a user were currently viewing example E6, a move to E7 via the same-detail-of link might be of interest.

\[
\begin{align*}
E7 & \quad \text{Proc Reg;} \\
& \quad \text{Model } Y = X1 - X15 \\
& \quad / \text{ Method=Stepwise;} \\
& \quad S\text{lstay} = .20; \\
& \quad \text{sdo} \quad \longrightarrow \\
E6 & \quad \text{Proc Reg;} \\
& \quad \text{Model } Y = X1 - X15 \\
& \quad / \text{ Method = Stepwise} \\
& \quad S\text{lentry} = .25;
\end{align*}
\]

Figure 16. Same-detail-of link
The set of examples above (E1-E7) and the described links might be pictured as a hierarchical network structure as shown in Figure 17. In this simple example, there are only two major topics: Regression and Data Input. The topic-entry examples (**) are E1 and E2. Within the topic of Regression, there are three subjects: Basic Regression, Variable Selection, and Parameter Testing. The subject-entry examples (*) are E1, E3 and E4 respectively. Note that E1 serves as both a topic-entry example and a subject-entry example. Within the area of Variable Selection, another primary example is found in E5. More detailed examples of E3 are found in E6 and E7 (secondary examples).

Within this small example set of nodes, it is now easy to identify all (8) possible links of the types defined earlier. First there is a topic link defined between the topic-entry examples, E1 and E2. Within the topic of Regression, there are across-subject links defined between all subject-entry examples. In this case, all possible links are defined between examples E1, E3, and E4. Within the subject of Variable Selection, there is a within-subject link between the primary examples E3 and E5. For example E3, there are two detail-of links defined for the secondary examples E6 and E7. Finally, examples E6 and E7 are linked via a same-detail-of link.
Note that for this set of seven examples, we have defined a total of eight possible links. The links all represent meaningful associations between examples. If, on the other hand, we had attempted to define links between all possible pairs of nodes, we would have had to define 21 such links (n(n-1)/2 in general). For a large set of examples (which we would need to have a useful system), defining all possible links and their meanings within the knowledge base would be a difficult and many of the links would not be very meaningful. By limiting our links to be of a certain form, we are taking advantage of our heuristic knowledge about how example programs can be usefully related within our network. In terms of choosing alternative links out of a given node, we are then reducing the solution space from one of all other nodes to one of a smaller set of nodes all of which satisfy one of a few well-known relations.

Consider example E1. Figure 17 show us that there are three links leading from E1: the topic link to E2, the across-subject link to E3, and the across-subject link to E4. Now, if a user has viewed the contents of node E1 and now wishes to see another example, his choices are E2, E3, or E4. These represent moves that the developer of the knowledge base has deemed to be meaningful for a user. Without these defined links, the user would be faced with a
### Example descriptors

E1: Multiple regression  
E2: Simple data input  
E3: Stepwise selection  
E4: Testing linear combination of parameters  
E5: Forward selection  
E6: Entry significance level  
E7: Stay significance level

### Links recognized

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Example Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>topic</strong></td>
<td>tl(E1,E2)</td>
</tr>
<tr>
<td><strong>across-subject</strong></td>
<td>asl(E1,E3), asl(E1,E4), asl(E3,E4)</td>
</tr>
<tr>
<td><strong>within-subject</strong></td>
<td>wsl(E3,E5)</td>
</tr>
<tr>
<td><strong>detail-of</strong></td>
<td>dol(E6,E3), dol(E7,E3)</td>
</tr>
<tr>
<td><strong>same-detail-of</strong></td>
<td>sdo(E6,E7)</td>
</tr>
</tbody>
</table>

**Figure 17. Network of examples**
choice between all other nodes in the network (E2-E7 in this example). Especially important is that the defined links can give the user an indication of what type of information he will be led to given his choice.

The defined links can also serve to limit the allowable paths through the network. Given the links defined in Figure 17, we see that the only way for a user at E1 to view E7 is to first view E3. This makes sense because the example in E7 is a more detailed version of the example in E3. The links can thus represent prerequisites information for certain examples. In this case, it makes no sense to view node E7 if node E3 has not been viewed first.

Another benefit of having a given set of predefined links is that algorithms can be developed, in terms of these links, that serve to make suggestions to the user about which link is best given his current location. EG Network can then "compute" its recommendations in terms of link activation. In other words, if a user is currently viewing an example program and wishes to be advised on what example to see next, EG Network can provide this advice in terms of a link or set of links. For example, the system might recommend that the user "view a detail of the current example" (execute a detail link) or "view another example within the same subject" (execute a within-subject link).
A trivial example of a recommendation algorithm is to always give preference to a certain link type (e.g., an across-subject link). A more complicated example might involve ranking the links based on the amount of future information available along paths given the choice.

Before describing the PROLOG implementation of the network structure, we discuss a final type of link called the "language link" (ll) which associates two example programs that accomplish the same thing but are implemented in different package languages as shown in Figure 18.

```
Proc Reg;
Model Y = X1 - X5;
```

\(\rightarrow (ll)\) \(\rightarrow\)

```
Regress c6 on 5
c1-c5.
```

Figure 18. Language links

These two programs are both examples of regression programs, one being implemented in SAS the other in Minitab. The existence of language links is kept simple by the concept of parallel networks. Essentially this means that the network shown in Figure 17 is replicated for each package language supported and each node is associated with its corresponding node in another language network via the language link. Regardless of what languages are supported, a language network for generic package called 'description'
is maintained. The examples in the generic language network are not really program examples, but rather English descriptions of the task to be carried out as shown in Figure 19. The core network of examples and links can be developed with respect to this generic language and can thus represent examples of using statistical packages in general versus using one particular package language. Given this core network of examples and links, additional language links can be easily incorporated into the network.

![Diagram](image)

**Figure 19.** Language link SAS to description

**Prolog Representation of the Example Network**

The set of nodes within the network can be viewed as a collection of frames structured like that shown in Figure 20. Within Prolog, this representation is accomplished using a set of three predicates of the form

- `eg("E3","Variable Selection","none")`
- `lang("E3","SAS","Proc Reg;\n Model ...")`
- `entry_subject("E1","E3").`
Each \texttt{eg()} predicate has three entries. The first entry corresponds to the node id slot and represents an internal identification number for the node. The second entry is a subject area descriptor. So, for the above example, node E3 is within the subject area "Variable Selection". The third entry corresponds to the Detail-Of slot and indicates the id number of the node for which E3 is a detail of. In this case, E3 is not the detail of any node so the entry is "none". E3 can thus be identified as a primary example.

<table>
<thead>
<tr>
<th>Node id #</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>Regression</td>
</tr>
<tr>
<td>Topic_Entry</td>
<td>No</td>
</tr>
<tr>
<td>Subject</td>
<td>Variable Selection</td>
</tr>
<tr>
<td>Subject_Entry</td>
<td>Yes</td>
</tr>
<tr>
<td>Detail_Of</td>
<td>None</td>
</tr>
<tr>
<td>Package</td>
<td>SAS</td>
</tr>
<tr>
<td>Text</td>
<td>'&quot;Proc Reg; Model Y = X1 - X5; / Method=Stepwise; &quot;'</td>
</tr>
</tbody>
</table>

Figure 20. Frame representation of an example program

The \texttt{lang()} predicate also has three entries. The first entry is again the id number and serves as a hard link to the \texttt{eg()} predicate. The second entry tells us to which particular language network the node belongs. In this case, the example is for the SAS package. The third entry is simply the text making up the actual program
example (\n is a Prolog control code for a line feed). Note that by separating the eg() and lang() predicates, we can provide a more efficient representation of examples across packages. Using this design, to add parallel examples for the generic Description package and the Minitab package, we simply need to add new lang() predicates like

\[
\begin{align*}
\text{lang}("2", "Description", "Forward Multiple regression") \\
\text{lang}("2", "MTB", "Stepwise c6 on variables c1-c5").
\end{align*}
\]

Note that the information on subject classification and detail status need not be repeated across packages. Furthermore, operations involving movements around the network can be defined without regard to any particular package.

The entry_subject() predicate simultaneously identifies the topic-entry example and a particular subject-entry example. The first entry is the id of the topic-entry example, in this case node E1. The second entry is the id of a subject-entry example within the topic associated with node E1, in this case E3. An entry_subject() predicate like this is used to identify each subject-entry example. For example, if E4 is also a subject-entry example, then the predicate entry_subject("E1", "E2") would also be present. If a node does not have an entry_subject() predicate associated with
it, then it is not a subject-entry example - "NO" entries in the topic-entry and subject-entry slots are not explicitly represented. Separation of this particular piece of information was done primarily for convenience in later operations.

An entire collection of these frame structures make up a representation of the network of nodes. The set of predicates for the topic of Regression for the network shown in figure 17 are as follows (for languages SAS and Description):

entry_subject("E1","E3")
entry_subject("E1","E4")

eg("E1","Basic Examples","none")
eg("E3","Variable Selection","none")
eg("E5","Variable Selection","none")
eg("E6","Variable Selection","E3")
eg("E7","Variable Selection","E3")
eg("E4","Parameter Testing","none")

lang("E1","Description","\n Multiple regression example.\n Dependent variable is Y, independent variables are X1 X2 X3 X4 and X5.")
lang("E1","SAS","\n Proc Reg; \n Model Y = X1 X2 X3 X4 X5; ")
lang("E3","Description","\n Stepwise selection. ")
lang("E3","SAS","\n Proc Reg; \n Model Y = X1 - X15 \n Method=Stepwise; ")
lang("E5","Description","\n Forward selection.")
lang("E5","SAS","\n Proc Reg; \n Model Y = X1 - X15 \n Method=Forward; ")
lang("E6","Description","\n To set entry significance level for \n stepwise method to .25.")
lang("E6","SAS","\n Proc Reg; \n Model Y = X1 - X15 \n Method = Stepwise \n Slentry = .25; \n ndefault is .15 */")
lang("E7","Description","\n To set stay significance level for \n stepwise method to .10.")
Within the EG Network design, this entire set of predicates would be held externally in a file readable by the Prolog package. A similar collection of nodes for other topics would likewise be held in other external files, each topic in its own file. The topics and their corresponding files are recorded using predicates like

```
topic("Data Input","data_eg.dba")
topic("Regression","reg_eg.dba")
topic("Analysis of Variance","anova.dba")
```

To keep track of the current topic, the predicate `ifile()` is used. For example, if the current topic is Regression, then the predicate `ifile("Regression")` is present in working memory. When a topic is changed, the `ifile()` predicate is replaced as necessary.

The nodes in the example network, represented by the `eg()`, `lang()`, and `entry_example()` predicates described above, reside in external data files. Links of the type described earlier can now be defined in terms of the predicate structures used to represent the nodes. These definitions can be set up within the primary database (knowledge base) as they are not dependent on the
particular information in the external files, only the form of the information known to be present.

Given the eg() representation, we can immediately define a within-subject link. In particular, consider the Prolog statement

\[
\text{within-subject}(X,Y) :- \text{eg}(X,S1,\_),
\text{eg}(Y,S2,\_),
\text{S1=S2}.
\]

The statement can be read in rule form as "IF node X is in subject area S1, AND node Y is in subject area S2, AND S1 is the same as S2, THEN Nodes X and Y have within-subject link."

It would seem that we could define an across-subject link in a similar fashion, replacing the S1=S2 with S1<>S2. However, recall that across-subject links are to be defined only for subject-entry nodes. The appropriate definition then uses the entry-subject() representation as

\[
\text{across-subject}(X,Y) :- \text{entry-subject}(T,X),
\text{entry-subject}(T,Y).
\]

The statement can be read in rule form as "IF node X is a subject-entry example (under topic-entry example T), AND node Y is a subject-entry example (under topic-entry example T), THEN nodes X and Y have an across-subject link."

This eg() representation also leads directly to the definition of the detail-of link using the statement
detail-of(X,Y):-eg(X,_,Y). The interpretation of this statement is obvious given the structure if the eg() node.

In a similar fashion, the same-detail-of link can be recognized via a Prolog statement like

same-detail-of(X,Y):-eg(X,_,K),eg(Y,_,K).

This statement can be read as "IF node X is a detail of node K, AND node Y is a detail of node K, THEN nodes X and Y have a same-detail-of link."

Finally, topic links and language links are not explicitly represented in forms like those given above. This will become apparent in the next section where we describe usage (traversal) of the example network.

**Traversing the Network of Examples**

A user obtains information from the system by moving throughout the network and viewing nodes (examples). EG Network's role is to assist the user in making the moves and, thus, in deciding what information to view. This assistance can come in terms of individual moves or sets of moves (paths). For example, a user could request that EG Network guide him through a particular topic, showing him all relevant information in a meaningful order (this mode is called user browsing). Alternatively, the user could take the initiative and proceed through the network using the provided mapping tools (this mode is called mapped
traversal). These two types of support have been fully implemented and are described in this section. Other types of traversal support have been experimented with and are also described.

User browsing

When a user enters EG Network in browse mode, he sees a list of topics from which he can choose. Selection of a major topic area places the user at the topic-entry example designated for that topic. Assume that the user selects Regression analysis from the menu. The browse screen is set up as shown in Figure 21.

The top of the browse screen has a status line identifying the current topic and subject area. Three windows labeled as F9, F10, and Actions are also present. F9 and F10 are windows to the textual content of the current node. These windows can be opened to any of the package networks supported. For example, in Figure 21 window F10 is opened to the SAS network and thus the window contains the SAS version of the current example. Likewise, F9 is opened to the Description network (the generic package code) and thus contains a descriptive version of the current example. At any one time then, a user simultaneously views corresponding nodes from two of
Figure 21. Browse screen with SAS example and Description

the parallel networks. Pressing the F9 or F10 function keys toggle the contents of their respective windows. For example, if the F10 function key is pressed for the above example, the window closes to SAS and opens to the next package network supported (like MINITAB). This process is akin to moving to a new example via a language link. Continually pressing the F10 key can thus show you the current example implemented across various packages with the description of the example staying open in the <F9> window. Appropriate settings of the <F9> and <F10> windows
can also provide the user with the screen shown in Figure 22. In this case, we are simultaneously viewing the same example program implemented in MINITAB and SAS. If a move is made to a different node (see below), the windows remain linked to their respective packages and a new example, implemented in both packages as above, is now shown. In this regard, in moving through the network, we actually are moving through the various package networks in a parallel fashion. This process provides a very good way for learning a new package given knowledge about another. For example, a user who knows MINITAB could use the above setup and browse through various familiar MINITAB examples and see at the same time the corresponding SAS examples. Of course, if no other package language is known, the Description package provides the familiar examples. We now describe how a user can move through various examples using the browse tools supported by EG Network.

The <Actions> window shown in the above figures identifies for the user what browsing actions are available. When the user initially enters browse mode, EG Network analyzes the set of examples and plans a somewhat flexible path for the user to take. The path is essentially an ordered list of nodes to visit and the user proceeds through this ordering example by example using the
Figure 22. Browse screen with MINITAB and SAS windows

PgDn and PgUp keys. In addition, the user can take larger steps (at the subject level) using the <F7> and <F8> keys.

The path is developed using only knowledge about how examples can be linked within the network. We now exemplify this process using the set of examples shown in Figure 17. To construct the path, EG Network first accumulates a list of all subject-entry examples within the topic chosen. The resulting list is \{[E1], [E3], [E4]\}. Note that the examples within the list are all connected via across-subject links. By construction, the first
subject-entry example found will correspond to the topic-entry example (E1). The ordering of the other subject-entry nodes is currently only dependent on the physical location of the eg() constructs within the knowledge base; that is, the examples are listed in the order they are found (which corresponds to the order in which they were entered into the knowledge base by the developer). The procedure could be modified to allow for some form of index ranking but this has not yet been done. The next step is to expand this list around each of the subject-entry nodes. This expansion involves adding to the list all of the primary examples within each subject area or equivalently, adding all within-subject links to the subject-entry examples. For the examples of Figure 17, the only other primary example is E5 and the resulting list is thus \{[E1], [E3, E5], [E4]\}. Note that within the {} grouping, the first examples of each of the [] groupings are linked via across-subject links. Within a [] grouping, the examples are linked via within-subject links. The final step is to expand this list around each member, adding all detail-of links from secondary examples. E6 and E7 both have detail-of links to E3, so the resulting list is \{[E1], [E3, (E6, E7), E5], [E4]\}. All examples within the () groupings have same-detail-of links between their members. Prolog implementation of this procedure is given in Appendix C.
As mentioned earlier, the user proceeds through this ordering example by example using the PgDn and PgUp keys and can take larger steps (at the subject level) using the  
<F7> and <F8> keys. For example, if, within the middle of a subject, the user finds that he is no longer interested in seeing examples within that subject, he can use the F8 function key to immediately move to the next subject-entry example in the list. Likewise, the F7 key will take the user back to entry-subject nodes previously viewed. Figure 23 shows the screen that might be present after the user has hit the F8 function key from the situation in Figure 21 (these examples do not correspond to E1-E7 above). Notice that the status line has identified the new subject descriptor and the window contents have been updated accordingly.

A user continues to utilize the action keys as necessary to traverse the network within the topic chosen. In essence, by viewing sets of examples in this way the user is receiving an example-based tutorial on the particular topic chosen. Of course, the tutorial can be specialized by the user if he chooses to skip less interesting subjects. Finally, if a new topic is desired at any time within the browse (before reaching end), the F6 function key can be used.
Figure 23. EG Network after executing next section move

Note that we have only provided the user with the opportunity to move example by example or subject by subject. It might seem that we should also provide opportunities for movement sub-subject to sub-subject and so on. However, recall that this mode is meant to be guided not user controlled. That is, in this mode the user is requesting that EG Network guide him through a particular topic and show him all relevant information in a
meaningful order. Some variations on this mode have been experimented with and are described later. The next section deals with the opposite situation - a mode where the user is in complete control of all movements within the network. This mode is called mapped traversal.

**Mapped traversal**

In user browsing mode, EG Network has pre-selected a path through a topic of interest; that is, EG Network has developed a suggested movement at each node. Mapped traversal puts the movement decision at each node in the hands of the user. EG Network's role in this case is to let the user know what other information is available and provide him with the mechanisms for moving to that information. In essence, mapped traversal mode involves implementing the link movements described earlier and allowing the user to choose which movement to make. Figure 24 shows the screen when a user enters mapped traversal mode. Note that the only difference is in the actions window - the two example windows operate as in user browsing mode. The actions window lists five possible actions the user can initiate. These are described below.

**Other Subjects** When a user selects this option, the first thing EG Network does is to examine the network, identifying all possible across-subject links and
accumulating their subject descriptors into a list. The list is presented to the user who can then choose which particular subject he wishes to move to. Figure 25 shows the screen after a user has selected this option. Note on the subject list that an asterisk (*) is placed next to all subjects which have been viewed (in this case only the current subject). Once a user selects one of the new subject areas, EG Network executes the across-subject move and places the user at the entry-subject example corresponding to his selection. As was the case in browsing mode, the ordering of the subject list is based on the physical location of the eg() constructs within the knowledge base - subjects are listed in the order they are found. Again, the accumulating procedure could be modified to allow for some form of index ranking but this has not yet been done. This ranking could be entered at the time that the knowledge base is developed or EG Network could calculate a weighting number based on, for example, the number of examples within that particular subject (assuming that subjects with a lot of examples are the more important subjects).

Recall that across-subject links are explicitly defined only for subject-entry examples. Thus if the current example being viewed is not a subject-entry example, no across-subject links are found. In this
Multiple regression example.
Dependent variable is Y, independent variables are X1 X2 X3 X4 and X5.

Figure 24. EG Network screen in mapped-traversal mode

situation, EG Network takes advantage of a concept called inheritance. Briefly, if 'Other Subjects' is chosen for any non-subject-entry example, EG Network identifies the subject associated with the current example and 'inherits' the across-subject links from the relevant subject-entry example. Referring again to the examples in Figure 17, if 'Other Subjects' is requested for example E7, EG Network will automatically associate with E7, the across-subjects links attached to E3 (the relevant subject-entry example).
Figure 25. Selection of Other Subjects

Other Examples

The Other Examples action operates much like the Other Subjects action except that the relevant link is the within-subject link. Again, within-subject links are defined only for primary examples within a subject. Thus if 'Other Examples' is requested while viewing a secondary example, EG Network associates with that example, the within-subject links of the corresponding primary example. For instance, if 'Other Examples' is requested for the secondary example E7, the relevant
within-subject links are those associated with the primary example E3 since E7 has a detail-of to E3.

**Details**
The Details action accumulates a list of all nodes which have a detail-of link to the current node being viewed. Again, the user can select from a menu which particular detail node he wishes to move to. No inheritance procedures need be employed for this operation.

**Other Details**
The Other Details action accumulates a list of nodes with a same-detail-of link to the current node. This allows a user to directly move to a different detail without first moving back to the detailed node and reselecting the Details action. An example of the results of using this action is shown in Figure 26. Originally, the user was viewing a simple example on stepwise regression. He then selected the Detail Action and moved to the new example showing the specification of entry significance levels. Finally, he has now selected the Other Details action to view other examples which are also details of the original simple stepwise regression example. In this case, an asterisk (*) is placed next to all detail nodes which have already been viewed.

**Other Topics**
This action simply allows the user to choose a new general topic and thus is an implementation of the topic link.
Other Methods

Shortcomings associated with each of the traversal methods described above have led to additional experiments. The first is most directly involved with user browsing. For completeness sake, a network of examples might be very large with some showing rather obscure features of a package language that are only employed in rare instances and others showing very detailed examples that might only be of interest to a select few users. Since the path that
EG Network creates through a particular topic is complete, involving all examples present in the knowledge base, a user traversing the network via this path may be forced to view examples that are not of interest to him. This is not a severe problem since all the user need do is press the PgDn key to go on. However, this brings up the point of how EG Network might customize its created path based on characteristics of a particular user. One simple solution that has been experimented with is to label each example in the knowledge base as being either a 'common' or an 'uncommon' application. EG Network still creates a complete path but if the user so desires, the examples marked uncommon can be masked out of the list and thus be made unavailable to the user. At any time, however, the user can switch modes and either have uncommon examples included or excluded. This idea could be expanded upon to allow for further subsetting of examples. For instance, examples could be categorized as being appropriate for new, common, or experienced users.

The above ideas are somewhat appealing, but have not been pursued for the following reason. Under close examination, it seems that the developer of the knowledge base could avoid this problem by simply creating subject groupings that correspond to different categories of users. For instance, rather than create only a subject area called
"Data Input", one could create subjects called "Basic Data Input", "Intermediate Data Input", and "Advanced Data Input". This would eliminate the problem of a user seeing inappropriate examples for their level of expertise.

Another traversal method experimented with is meant to be a midpoint between the user browsing method, which allows little user control, and the mapped traversal method, which demands complete user control. This method is called reactive traversal and involves incorporating user feedback into EG Network’s suggestive process. This method is much like the user browsing method except that the path is determined dynamically as the user traverses the network and views examples. The process begins with EG Network accumulating a list of examples (or a path) just as was done in user browsing mode (in fact, the same list is created). At each step along the path, however, EG Network shows the user an example and then asks for user feedback on the example shown. This feedback is kept simple by simply asking the user whether or not the example shown was of interest. An affirmative response results in EG Network showing the user the next example on the list. In fact, if the user gives an affirmative response at every node, the path followed will be exactly that which would be followed under the user browsing method if the user simply paged through the list. A negative response, on the other hand,
forces EG Network to reevaluate its next proposed move. The new decision is based on where in the network the user is currently located. Some examples of rules used in the process for making such a new selection are given in Appendix D. Once an alternative example is selected, the user is shown that example and is now positioned at that new choice on the list. So, the next example shown to the user will be the next one on the list after the new selection. The process continues as above form that point on.

Comments

EG Network can help users traverse the example network by either developing a path through the network for the user to follow or by providing him with enough information at each step so that he can make the decision. In the former case, the resulting activity is system-initiated in that the user has little input into what links are traversed when. In the latter case, the resulting activity is user-initiated in that the user decides what link to traverse at each stage. Both modes are useful. If a user knows nothing or very little about a package, he might wish for the system to make all the decisions on what information he sees. If, on the other hand, a user knows the package fairly well, he might just be looking up how
some small detail or some item that has been forgotten. In this case, the user would like to be in complete control so that only relevant information is viewed. Of course, EG Network still provides assistance in this situation by letting the user know what information is available where.
CHAPTER VI. SUMMARY

Knowledge-based programming techniques have typically been used to develop statistical expert systems that help users correctly apply a statistical tool. In this research, we have investigated an alternative application. In particular, we have shown that knowledge-based programming techniques can be used to develop support systems for users of statistical computer software. Current statistical software packages (SAS for example) are extremely powerful but program development can sometimes prove time consuming and frustrating, especially for inexperienced users. Furthermore, manuals for software are often cumbersome and rarely contain the useful rules of thumb or shortcuts employed by expert users. With respect to these ideas, we consider knowledge-based systems that can help people use and learn to use statistical software.

The first system developed, EG Expert, is a prototype knowledge-based expert system designed to answer general "how do I?" questions about statistical software. Using knowledge about typical applications in statistical software, EG Expert first queries a user to extract information about his problem. Based on the information received, the system then builds a generic description of the example program to be generated. The generic example
is not specific to any particular package and can be thought of as a pseudocode representation of the example. EG Expert can then translate this generic example to any particular package language using knowledge specifically relating the generic elements with package commands.

The second system, EG Network is less like a traditional expert system and more like an intelligent information system. EG Network contains an integrated collection of example programs linked together in the form of a graph or network. A user obtains information from the system by moving throughout the network and viewing nodes (examples). EG Network's role is to assist the user in making the moves and in deciding what information to view. The emphasis of EG Network is more on the provision of information to the user. The result is a system more flexible from a user's standpoint and easier to produce and maintain from a developer's standpoint.

Nevertheless, further research needs exist for EG Network. The most pressing is the need to develop a substantial database of examples and to submit the system to extensive testing. We were able to evaluate the system during the development process but feedback from potential users is critical. Such feedback will help us to further refine and develop the traversal methods implemented. In addition, full scale development of a large database of
examples will allow us to better study the development process and further refine the tools for such.

In conclusion, this work can be viewed from two perspectives. First and foremost, it is an investigation into the use of knowledge-based and related programming techniques for statistical application. Most work in the literature focus on the analysis aspect of statistical applications of AI. Our work, on the other hand, focuses on systems that can serve as assistants or tools and make a user more productive. Secondly, this work can be thought of as an investigation into computer-based support for use of computer software. Some work exists in the literature regarding this topic but ours is the first to focus specifically on statistical software. Furthermore, our methods, in EG Network especially, differ substantially from methods employed by others in this area.
BIBLIOGRAPHY


Winston, P. H. 1984. AI. Addison-Wesley, Reading.
ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. William J. Kennedy, for his guidance and support throughout this project. I would also like to acknowledge the rest of my committee for their time and interests. Thanks also go to everyone in the Statistics Lab for their friendship and support.

I would also like to thank my family for all of their love and support over my years in school. Special thanks to my wife Nancy. Her constant love, understanding, and encouragement made completion of this degree possible.
APPENDIX A. EG EXPERT INFERENCE PROCESS
User indicates that they would like an example of reading data.

System begins to look for structural elements associated with major goal of readdata; finds

- s1: identify operation as data input
- s2: give name to data set created
- s3: identify source of data
- s4: identify type of data

S3 instigates the query "Data to be input inline or from external source", assume answer is external, add fact that source of data is external.

S4 instigates the query "Data in form of system or raw values", assume answer is external, add fact that form of data is external.

No more queries found.

Additional structural elements identified based on facts (external, raw):

- s5: identify DOS file name containing data
- s6: identify format of variables in input record
- s7: identify variable names

Final generic program is s1-s7

identify operation as data input
give name to data set
identify source of data
identify type of data
identify DOS file name containing data
identify format of variables in input record
identify variable names

This generic form is now ready for conversion to any package.
APPENDIX B. PROGRAM LISTING FOR EG NETWORK
code = 3048

domains
  list = symbol *
  llist = integer *

database
  mdesc(symbol, string)
  term(symbol)
  q_topic(symbol)
  link(symbol, symbol)
  level(symbol, symbol)
  gmove(symbol, symbol, string)
  cogal(symbol)
  nat_query(symbol, string)
  solist(list)
  olist(list)
  asap(symbol)
  asap(symbol)
  ad(symbol)
  active(symbol)
  already(symbol)
  base(symbol)
  bel(integer, symbol)
  cnt(integer)
  final(symbol)
  hid
  lang(symbol, symbol, string)
  ifile(symbol)
 .lang_list(list)
  nat_list(list)
  left_win(symbol)
  next(symbol)
  nodetalls
  over(symbol)
  prev(symbol)
  raf(symbol)
  right_win(symbol)
  seen(symbol)
  eq2(symbol, symbol, symbol) /* node, nat, detail_of */
  nat(symbol)
  commands(symbol, symbol, symbol)

include "nwutil.pro"
include "menu2.pro"
predicates
  ch_resx(string,string,string)
detidnsts(list,list)
detldnsts2(list,list)
detail(symbol)
othac_detail(symbol)
othac_example(symbol)
othac_subject(symbol)
make_ideal
check_term(symbol)
assert_terms(list)
target_matches(symbol,symbol,symbol)
gat_querystring(symbol,string)
accum(list)
process(list)
cover(list)
list(list)
cover_list(list)
search
overview
browse
eg2details(symbol)
all(list)
allnet(list)
check(char)
child(symbol,symbol)
chk_sect(symbol,symbol)
chk_passct(symbol,symbol)
chk_last(symbol,symbol)
chk_prav(symbol,symbol)
chkgoal(symbol)
chkasen(symbol)
clearact
clearaq
clearold
clrdate
clrfile
collect(symbol)
nocollect(symbol)
do
doneit
empty(list)
get_code(string)
get_code1(string)
go(symbol)
head(list,symbol)
head3
in(symbol,list)
mchk(symbol,symbol)
mark_list(list,symbol)
mark
next_section(symbol,list,symbol)
next_none(symbol,list,symbol)
next_prev(symbol,list,symbol)
next_up(symbol,list,symbol)
prv_section(symbol,list,symbol)
qnext_section(symbol,list,symbol)
process(key)
repeat
special
start
split(list,symbol,list)
sublist(list,list)
tail(list,symbol)
topic(symbol,symbol)
wait(key)

write_all(list)
process_query
count_matches(symbol,list,integer)
str_to_list(string,list)
checkin(symbol,list,integer)
matches(list,list,integer)

classes

/* Set up for windows */

mgoal:-
  retractall(clist(_)), retractall(aclist(_)),
  retractall(link(_,_)),
  makewindow(1,14,15,"",1,0,10,80), /* left window */
  makewindow(2,12,0," Messages ",6,0,1,40), /* dialogue window */
  makewindow(4,12,15," Actions ",0,40,10,30), /* actions window */
  makewindow(3,13,15,"",10,0,15,80), /* right window */
  makewindow(10,7,0,"",0,0,8,40),
  shiftwindow(1),clearwindow,
  shiftwindow(3),clearwindow,
  shiftwindow(4),clearwindow,
  shiftwindow(2),clearwindow,
  shiftwindow(10),clearwindow,
  makewindow(9,26,0,"",0,0,25,80),
  shiftwindow(9),
  clearwindow,
  makewindow(8,26,15," Welcome to EXAMPLES ! ",4,10,10,60),
  start, do.

/* Basic Loop and setup */
do:-repeat,wait(X),process(X),fail.
    wait(X):-readkey(X).
    wait(X):-wait(X).
    repeat.
    repeat:-repeat.

chkgoal("browse"):-assert(cgoal(browse)),
    process(ckey(6)),shiftwindow(8),removewindow,
    all(L),qsort(L,SL),assert(lang_list(SL)),
    allnat(N),assert(nat_list(N)),
    assert(right_win("SAS")),
    assert(left_win("Description")),
    shiftwindow(4),
    browse,
.
chkgoal("search"):-assert(cgoal(search)),
    process(ckey(6)),shiftwindow(8),removewindow,
    all(L),qsort(L,SL),assert(lang_list(SL)),
    allnat(N),assert(nat_list(N)),
    assert(right_win("SAS")),
    assert(left_win("Description")),
    shiftwindow(4),
    search,
.
chkgoal("overview"):-assert(cgoal(overview)),
    process(ckey(6)),shiftwindow(8),removewindow,
    all(L),qsort(L,SL),assert(lang_list(SL)),
    allnat(N),assert(nat_list(N)),
    assert(right_win("SAS")),
    assert(left_win("Description")),
    shiftwindow(4),
    overview,
.
chkgoal("special"):-process(ckey(6)),shiftwindow(8),removewindow,
    all(L),qsort(L,SL),assert(lang_list(SL)),
    allnat(N),assert(nat_list(N)),
    assert(right_win("SAS")),
    assert(left_win("Description")),
    shiftwindow(4),
    special,
.
chkgoal(X):-write(X," not yet available !"),readin(,),clearwindow,shiftwindow(4),!.

start:-shiftwindow(8),write("Goal ? "),readin(G),chkgoal(G),!.
next_section(A,L,B) :- eg2(A,N1), next_element2(A,L,B), eg2(B,N2,_), N1>N2.
next_section(A,L,C) :- next_element2(A,L,B), next_section(B,L,C).

next_none(A,L,B) :- eg2(A,N1), next_element3(A,L,B), eg2(B,"nona").
next_none(A,L,C) :- next_element3(A,L,B), next_none(B,L,C).

next_det(D,[H|_],B) :- ag2(H,_,D).
next_det(D,[L|_],B) :- next_det(D,[L|_],B).

next_up(A,[H|_],B) :- eg2(B,"nona"), child(A,B).
next_up(D,[L|_],B) :- next_up(D,[L|_],B).

split([H|T],H,T).
split([_|T],T,T).

prev_section(A,L,C) :- reverse([L|L1]), next_section(A,L1,B), eg2(B,N1), eg2(C,N1).

next_section(A,L,B) :- eg2(A,N1), next_element2(A,L,B), eg2(B,N2,_), N1>N2,
            net_query(N1,Q),
            mkwindow(0,26,15,",",4,12,6,60), shiftwindow(8), clairwindow,
            write(q), readchar(R), removewindow, check(R).
next_section(A,L,C) :- next_element2(A,L,B), next_section(B,L,C).

chk_last(_,"null") :- shiftwindow(10), field_str(7,0,35," Last example 1")
            shiftwindow(4),!.
/* chk_last(A,N) :- eg2(A,N1), eg2(N,N2), N1<>N2, shiftwindow(10), field_str(7,0,35," Last example in
            this section 1")
            shiftwindow(4),! */
chk_last(A,N) :- retract(active(A)), assert(active(N)), go(N),!.

chk_prev(_,"null") :- shiftwindow(10), field_str(7,0,35," First example 1")
            shiftwindow(4),!.
/* chk_prev(A,N) :- eg2(A,N1), eg2(N,N2), N1<>N2, shiftwindow(10), field_str(7,0,35," First example in
            this section 1")
            shiftwindow(4),! */
chk_prev(A,N) :- retract(active(A)), assert(active(N)), go(N),!.

chk Sect(A,N) :- eg2(A,N1), eg2(N,N2), N1>N2, shiftwindow(10), field_str(7,0,35," Last Section for this
            topic 1")
            shiftwindow(4),!.
chk Sect(A,N) :- retract(active(A)), assert(active(N)), go(N),!.

chk_psect(A,N) :- eg2(A,N1), eg2(N,N2), N1<N2, shiftwindow(10), field_str(7,0,35," First Section for
            this topic 1")
            shiftwindow(4),!.
chk_psect(A,N) :- retract(active(A)), assert(active(N)), go(N),!.

/* Process Keystroke Commands */
process(fkey(1)):-process(pgdm),!.

process(fkey(3)):-active(A),olist(L),head(L,A),!.
process(fkey(2)):-active(A),olist(L),head(L,NN),link(H,A),
    next_section(A,L,B),
    retract(active(A)),assert(active(B)),go(B),!.
process(fkey(2)):-active(A),olist(L),head(L,NN),not(link(H,A)),
    eg2(A,B,\"none\"),next_non(A,L,NN),eg2(N,\"none\"),
    retract(active(A)),assert(active(N)),go(N),!.
process(fkey(2)):-active(A),olist(L),head(L,NN),not(link(H,A)),
    eg2(A,B,\"none\"),next_non(A,L,NN),eg2(N,\"none\"),goGli,
    next_section(A,L,B),
    retract(active(A)),assert(active(B)),go(B),!.
process(fkey(2)):-active(A),olist(L),split(L,A,B),eq2(A,B,\"none\"),next_split(D,L1,N),
    retract(active(A)),assert(active(N)),go(N),!.
process(fkey(2)):-active(A),olist(L),split(L,A,B),eq2(A,B,\"none\"),olist(L),
    next_element2(A,L,B),eq2(B,\"none\"),
    retract(active(A)),assert(active(B)),go(B),!.
process(fkey(2)):-active(A),write(\"No suggestion (next)\"),olist(L),
    next_element2(A,L,B),retract(active(A)),assert(active(B)),
    go(B),!.

process(fkey(4)):-active(N),
    make_window(8,25,15,\"Level\",4,12,6,60),shift_window(0),clear_window,
    readin(\"Level\"),
    assert(\"Level(N,Level)\"),remove_window,!.

/* Look around */
/* process(fkey(7)):-active(A),findall(X,move(A,___,X),L),
    manu(16,41,120,120,\"Info Available\",1,C),
    write(C),!.

/* */

/* Change Sections */
process(fkey(8)):-goal(browse),active(A),olist(L),next_section(A,L,N),chk_section(A,N),!.
process(fkey(7)):-goal(browse),active(A),olist(L),p_we_section(A,L,N),chk_psection(A,N),!.
process(fkey(9)):-goal(special),active(A),olist(L),next_section(A,L,N),
    eg2(N,N,\_),net_query(N,N),retract(olist(\_)),
    mark_list(L,N),findall(X,astap(X,L)),retractall(astap(\_)),
    assert(olist(L)),
    chk_section(A,N),!.
process(fkey(6)):-clearold,clrfile,findall(T,topic(T,_),Tlist),
menu(16,41,120,120,Tlist,"Choose a Topic",1,C),
pick(C,Tlist,K),topic(K,File),existfile(File),
assert([file(File)],consult(File),
shiftwindow(10),clearwindow,str_len(K,NI),
field_str(2,1,7,"Topic: "),
field_str(2,8,NI,K),field_attr(2,8,NI,15),
/*goal(G),
str_len(G,NG),field_str(1,1,7,"Mode: "),
field_str(1,8,NG,G),field_attr(1,8,NG,15),*/
  .
process(fkey(6)):-write("File not available"),nl,1.
/* Toggle left window */
process(fkey(9)):-left_win(R),lang_list([H|T]),next_lang(R,[H|T],N,H),
retractall(left_win(_)),assert(left_win(N)),
active(A),go(A),!.
/* Toggle right window */
process(fkey(10)):-right_win(R),lang_list([H|T]),next_lang(R,[H|T],N,H),
retractall(right_win(_)),assert(right_win(N)),
active(A),go(A),!.
process(pgdn):-active(A),solist(L),next_element2(A,L,N),chk_last(A,N),!.
process(pgup):-active(A),solist(L),prev_element(A,L,N),chk_prev(A,N),!.
process(home):-retractall(active(_)),solist(L),head(L,N),
assert(active(N)),go(N),!.
process(end):-retractall(active(_)),solist(L),tail(L,N),
assert(active(N)),go(N),!.
/* Quit (Esc) */
process(esc):-doexit.

process(char(‘q’)):-process_query,!.

process(char(‘w’)):-makewindow(14,26,15," Query ",1,1,5,65),shiftwindow(14),
clearwindow,write("Input Descriptor: "),nl,readin(Q),
active(K),assert([misc(Q,K)]),clearwindow,removewindow,!

process(char(‘d’)):-findall(D,detail(D),L),dasthrs(L,Hlist),
menu(16,69,120,120,Hlist,"Other ... ",1,C),
pick(C,L,K),
retractall(active(_)),assert(active(K)),go(K),!.
112

process(char('o')): findall(D, other_detail(D), L), detstrs(L, Hlist),
    menu(16,69,120,120, Hlist, "Other ...", 1, C),
    pick(C,L,X),
    retractall(active(_)), assert(active(K), go(X), 1).

process(char('e')): findall(D, other_example(D), L), detstrs(L, Hlist),
    menu(16,69,120,120, Hlist, "Other ...", 1, C),
    pick(C,L,X),
    retractall(active(_)), assert(active(K), go(X), 1).

process(char('s')): findall(D, other_subject(D), L), detstrs2(L, Hlist),
    menu(16,69,120,120, Hlist, "Other ...", 1, C),
    pick(C,L,X),
    retractall(active(_)), assert(active(K), go(X), 1).

process(char('t')): findall(T, topic(T), L),
    menu(16,69,120,120, L, "Other ...", 1, C),
    pick(C,L,X),
    retractall(active(_)), assert(active(K), go(X), 1).

/* Edit existing example - language */

process(char('1')): left_win(X), active(K), lang(N,X,Code),
    makewindow(9,26,15,X,0,15,40), gotowindow(9),
    editmsg(Code, Ncode, "","Edit example description", 0,"", _,
    retract(lang(N,X,Code)), assert(lang(N,X,Ncode)), removewindow, go(N), 1.

process(char('2')): right_win(X), active(K), lang(N,X,Code),
    makewindow(9,26,15,X,0,40,15,40), gotowindow(9),
    editmsg(Code, Ncode, "","Edit example description", 0,"", _,
    retract(lang(N,X,Code)), assert(lang(N,X,Ncode)), removewindow, go(N), 1.

process(char('s')): right_win(X), active(K), lang(N,X,Code),
    makewindow(9,26,15,X,10,0,15,80), gotowindow(9),
    display(Code), removewindow, 1.

process(X): shiftwindow(2), clearwindow, write("Key not recognized: ", X), nl,
    header3, active(L), go(L), 1.

/* Header Routines */

header3: goto(overview), active(_), shiftwindow(4), clearwindow, nl,
    write(" Next Example : <PgDn>\n"),
    write(" Previous Example : <PgUp>\n"),
    write(" Previous Section : <F7>\n"),
    write(" Next Section : <F8>\n"),
    header3: goto(overview), active(_), shiftwindow(4), clearwindow, nl,
    write(" Interested : <FI>\n"),
write(" Not interested : <P2\n").

header3.

go(Id)=eg2(Id,_,_),chkasem(Id),header3,
shiftwindow(10),field_str(4,0,38," Section:",
field_str(4,38,7),str倌(N,N1),
field_str(4,10,N1,N),field_str(4,10,N,15),
field_str(7,0,38," "),field_str(6,0,5,I),
shiftwindow(1),clearwindow,left_win(2),get_code(Cod),
concat(" <FB> : ",I,I),
framewindow(15,N1,1,"210\0191\0192\0193\0194\0195\0196\0197"),
window_str(Cod),
shiftwindow(3),clearwindow,right_win(N),get_code(NCode),
concat(" <FD> : ",N,N1),
framewindow(15,N1,1,"201\0187\0188\0189\0190\0191\0192\0193"),
framewindow(15,N1,1,"210\0191\0192\0193\0194\0195\0196\0197"),
window_str(NCode),shiftwindow(4).
/*,nmt(N),
concat(" Section : ",N,N1),
framewindow(15,N1,1,"201\0187\0188\0189\0190\0191\0192\0193").

get_code(Cod)=right_win(X),active(N),lang(N,X,Cod),1.
get_code("An No Example Available").
get_code(Cod)=left_win(N),active(N),lang(N,X,Cod),1.
get_code("An No Example Available").

topic("Test2","test2.dba").
topic("Test2","rttest.db").
topic("Data input and manipulation","new3.db").
topic("Regression analysis","new2.db").
topic("Analysis of Variance","X").
topic("Descriptive Statistics","X").

/* Clear database for new topic */
clrfil=retractall(ifile(_)).
cleardet=retractall(net(_)).
clearold=clearreg,clearact,cldet,retractall(ref(_)),retractall(lang_list(3)),
retractall(net_list(_)),retractall(seen(_)),retractall(base(_)).

clearreg=retractall(eg2(_,_,_,_)),retractall(lang(_.,_,_)).
clearact=retractall(active(_)),retractall(right_win(_)),retractall(left_win(_)),
retractall(seen(_)),retractall(base(_)).
dosxit-ifile(X),clrdet,clearact,clrfil,shiftwindow(2),
retractall(net_list(_)),retractall(lang_list(_)),
retractall(clist(_)),retractall(sclist(_)),
retractall(goat(_)),
write("Save changes ? "),readchar(Chk),check(Chk),
save(X),removewindow,clearold,
shiftwindow(1), removewindow,
shiftwindow(3), removewindow,
exit.
doexit:-exit.
check('o'):-1,fail.
check(_).

collect(H) :- lang(_,_,_), not(already(H)), assert(already(H)).
all(L) :- findall(H, collect(H,L), retractall(already(_))).

nocollect(H) :- e2(_,"none"), not(already(H)), assert(already(H)).
allnot(H) :- findall(H, nocollect(H,L), retractall(already(_))).

head([],H).
tail([],_).
tail([_ | _], T) :- tail([_ | _], T, P).
in(X,[X|_]) :- in(X,Y).
in(X,[_ | *]) :- in(X,Y).

sublist([|],_).
sublist([H | T], L) :- in(H, L), sublist(T, L).

chkseen(Id) :- seen(Id), 1.
chkseen(Id) :- assert(seen(Id)), 1.
empty([]).
/* Search commands */

/* Number of Items in list1 that are also in list2 */
matches([], 0).
matches([H | T1], L2, N) :- checkin(H, L2, T), matches(T1, L2, N), N =< T.

/* Evaluates to 1 if X is in List, else 0 */
checkin(X, L, 1) :- in(X, L).
checkin(X, L, 0) :- not(in(X, L)).
/* checkmatch(Node, Number=eq(Node, Text, _, _, _), */
str_to_list("", []).-1.
str_to_list(" ", []).-1.
str_to_list("n", []).-1.
str_to_list(String, [H | T]) :- fronttoken(String, H, Rest),
str_to_list(Rest, T).
count_matches(N,Key,Count) :-
    eq2(N, _, _), get_query_string(N, T), str_to_list(T, Tlist),
    matches(Key, Tlist, Count).

get_query_string(N, T) :- q_topic(Q), lang(N, Q, T).

process_query :- write("Key in query "), readIn(Q), str_to_list(Q, Qlist),
    nl, write_all(Qlist).

write_all(Q) :- count_matches(N, Q, Count), write("Node ", N,
    " has ", Count,
    " matches\n"), readIn(_), fail.
write_all(_).

eq2details(E) :- eq2(D, _, E), write(" Detail: ", E), nl.

child(C, A) :- eq2(C, _, A).
child(C, A) :- eq2(B, _, A), child(C, B).

accum(P) :- retractall(step(_)), nst_list(L), cover(L), findall(X, step(X), P),
    retractall(step(_)).

cover([ ]). 
cover([E/T]) :- findall(X, eq2(X, E, "none"), L), covernet(L), cover(T).
covernet([ ]). 
covernet([E/T]) :- findall(X, eq2(X, _, E), L), assert(step(E)), covernet(L), covernet(T).

browse :- accum(L), assert(olists(L)), assert(slist(L)), oproc(L).

oprocces((E, _)) :- retractall(active(_)), assert(active(E)), go(E), !.

special :- assert(cgoal(special)), accum(L), assert(olists(L)), mark_list(L, "common"),
    findall(X, step(X), L1), retractall(active(_)),
    assert(slist(L1)), oproc(L1).

overview :- accum(L), assert(olists(L)), assert(slist(L)), oproc(L).

/*mark_list(L, "common");
   findall(X, step(X), L1), retractall(active(_)),
   assert(slist(L1)), oproc(L1).*/

mark_list([ ],_).
mark_list([E|T], M) :- mchk(E, M), mark_list(T, M).

mchk(E, M) :- eq2(E, _, _), level(E, M), assert(estep(E)), !.
mchk(_, _).
/*ind*ll(N,n#t_qu»ty(W,"null"),Vli#t),oov#:(Nlimt),findmll(X,"*p(X) ,P),
m«m#rt(oli«t(P) ,I).
*/
list([]).
list([H|T]):-ag2(H,N,D),write(H, " = ", N, " = ", D),nl,list(T).

search:-accum(L),assert(dlist(L)),assert(solist(L)),oprocess(L),
makewindow(14,26,15," Search Template ",5,5,15,65),shiftwindow(14),
clearwindow,nl,
write(" Descriptive terms : 
"),nl,
write(" Target package : 
"),nl,
write(" Target terms : 
"),nl,
write(" Reference package : 
"),nl,
write(" Reference terms : 
"),
cursor(1,21),readin(D),
cursor(3,21),readin(T),
cursor(4,21),readin(TT),
cursor(6,21),readin(R),
cursor(7,21),readin(RT),
str_to_list(D,Dlist),nl,assert(q_topic("Description")),
write("Descriptive matches"),nl,write_all(Dlist),retractall(q_topic(_)),
target_matches("Target",T,T),target_matches("Reference",R,RT),
readin(_,),removewindow().

target_matches(P,T,_):-not(lang(_,T,_)),write(P," not available ").
target_matches(P,T,T):-lang(_,T,_),
str_to_list(T,Tlist),assert(q_topic(T)),nl,
write(P," matches ",T),nl,
write_all(Tlist),retractall(q_topic(_)).

make_index:-lang(_,"Description",T),str_to_list(T,Tlist),assert_terms(Tlist),fail.
make_index.

assert_terms([]).
assert_terms([H|T]):-check_term(H),assert_terms(T).

check_term(H):-not(term(H)),assert(term(H)).
check_term(_).

detail(D):-active(A),eq2(D,A).
octher_detail(D):-active(A),eq2(A,X),eq2(D,X),K="none".
octher_example(E):-active(A),eq2(A,G,_,_),eq2(E,G,"none").
other_subject(S) :- solist(L), head(L, H), ag2(H, _,_), link(H, S).

dathdni([]), []).
dathdni2([H|T], [H1|T1]) :- ag2(H, _,_), mdesc(H, H2), ch_seen(H, H2, H1), dathdni(T, T1).

dathdni2([], []).
dathdni2([H|T], [H1|T1]) :- ag2(H, _,_), ag2(H, H2, _), ch_seen(H, H2, H1), dathdni2(T, T1).

ch_seen(H, H2, H3) :- seen(H), concat("", H2, H3), 1.
ch_seen(\_, H, H1) :- concat(" ", H, H1).
APPENDIX C. USER BROWSING PATH ALGORITHM
The main predicate used to create the path is `accum()`. A call to `accum(P)` returns a list of node id numbers in the list variable `P`. This list represents the path. See Appendix A for complete code of EG Network.

Accumulation Code:

```prolog
accum(P):-retractall(step(_)),
    allnet(L),
    cover(L),
    findall(X,step(X),P),
    retractall(step(_)).

allnet(L):-findall(H,ncollect(H),L),
    retractall(already(_)).

ncollect(H):-eg2(_,H,"none"),
    not(already(H)),
    assert(already(H)).

cover([]).

cover([H|T]):-findall(X,eg2(X,H,"none"),L),
    covernet(L),
    cover(T).

covernet([]).

covernet([H|T]):-findall(X,eg2(X,_,H),L),
    assert(step(H)),
    covernet(L),
    covernet(T).
```

Given the following `eg()` constructs in the database:

- `eg2("E1","Basic Examples","none")`
- `eg2("E3","Variable Selection Routines ","none")`
- `eg2("E4","Hypothesis Testing ","none")`
- `eg2("E5","Variable Selection Routines ","none")`
- `eg2("E6","Variable Selection Routines ","E3")`
- `eg2("E7","Variable Selection Routines ","E3")`

A call to `accum(P)` returns `P=[E1,E3,E6,E7,E5,E4]`. 
APPENDIX D. REACTIVE BROWSING DECISION EXAMPLES
In reactive browsing mode, a user is shown an example and then asked whether or not the example is of interest. A negative response results in the following decision process for selecting the next example to be shown. The decision is based solely on the current node being viewed and its location within the network. The following are examples of implemented rules. Let C represent the current example.

1) IF C is the topic-entry example
   THEN change topic.

   If a user is not interested in the topic-entry node, it is an indication that the user is not interested in the topic.

2) IF C is a subject-entry example
   AND C is not a topic-entry example
   THEN go to the next subject-entry example.

   If a user has viewed beyond the topic-entry example, than the topic is seemingly interesting but this particular subject is not.

3) IF C is a primary example
   AND C is not a subject-entry example
   AND there is another primary example within current subject
   THEN go to next primary example within current subject.

   If a user has viewed within a particular subject, than the subject is seemingly interesting but this particular sub-subject is not.

4) IF C is a primary example
   AND C is not a subject-entry example
   AND there is no other primary example within current subject
   THEN go to next subject-entry example.

   If a user has viewed within a particular subject, than the subject is seemingly interesting but this particular sub-subject is not. However, there is nothing else in the subject to see, so move on to the next subject.

5) IF C is a secondary example
   THEN go to next primary example.

   If a user has viewed and is not interested in this detail, then he will seemingly not be interested in others, move on.