A PROLOG-based Expert System for Tuning MVS/XA

Dr. Bernard Domanski

The College of Staten Island / CUNY ***

ABSTRACT

This paper will discuss some of the issues involved in building an Expert System that embodies tuning rules for IBM's MVS/XA operating system. To understand the components of an Expert System and their functions, PROLOG on an IBM PC (Turbo-PROLOG from Borland International) was chosen as the development environment. The paper will begin by defining the key concepts about Expert Systems, Knowledge Engineering, and Knowledge Acquisition. The reader will be given a brief overview of PROLOG, from which we can explain how an inference mechanism was developed. Finally, the paper will describe the Expert System that was developed, and additionally will provide a set of key issues that should be addressed in the future. It is our overall objective to provide new insight into the application of AI to CPE.

1. Introduction

In his acceptance speech for the A. A. Michelson award in 1985, Dr. K. Mani Chandy said that it takes some ten to fifteen years for research in the universities to become practical technology for industry. The field of *Artificial Intelligence* (AI) is a prime example; the last fifteen years have seen AI flourish in coming from the university sector to industry. AI languages such as LISP and PROLOG have been touted as the languages of "choice" for fifth generation computing. The Japanese have chosen PROLOG as the machine language of their logic processor for their fifth generation systems.[1] Expert System shells such as OPS5[2] and M.1[3] can help simplify and speed the development of these systems.

Considering Computer Performance Evaluation (*CPE*), some prior exploratory work has been done.[4,5] Vendors are now emerging with *Knowledge Based* systems for CPE (ISS, Boole & Babbage).

Although AI has had a long history, the application of AI to CPE is still immature. In CPE, many performance analysts have developed a set of "rules-of-thumb" for tuning large systems. Typically, this set takes five years to develop. As there is a tendency to promote quality personnel within a corporation, it is *critical* to extract the set of rules from the performance "expert" before (s)he is promoted to a new position. In essence, a "technology transfer" must occur.

This paper will discuss some of the issues involved in building an Expert System that embodies tuning rules for IBM's MVS/XA operating system. To understand the components of an Expert System and their functions, PROLOG on an IBM PC (Turbo-PROLOG from Borland International) was chosen as the development environment. The paper will begin by defining the key concepts about Expert Systems,

---

*** This work was done by the author while on a sabbatical leave from the College of Staten Island, a unit of the City University of New York (CUNY). Though this work was conceived and performed by the author *on his own*, colleagues at Bell Communications Research reviewed the work as it progressed. Their input is gratefully acknowledged here.

1. Feigenbaum, E. A., McCorduck, P., "The 5th Generation - Artificial Intelligence and Japan's Computer Challenge to the World", Addison-Wesley, 1984.
2. Forgy, C. L., OPS5 User Manual, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA., 1981.
3. M.1 is a product of Teknowledge, Inc., Palo Alto, CA.
4. Hellerstein, J., Van Woerkom, H., "YSCOPE: A Shell for Building Expert Systems for Solving Computer Performance Problems", Proceedings, Computer Measurement Group, Dallas, December, 1985.
5. Artis. H. P., "Using Expert Systems for Analyzing RMF Data",ibid.

Knowledge Engineering, and Knowledge Acquisition. The reader will be given a brief overview of PROLOG, from which we can explain how an inference mechanism was developed. Finally, the paper will describe the Expert System that was developed, and additionally will provide a set of key issues that should be addressed in the future. It is our overall objective to provide new insight into the application of AI to CPE.

2. What is an Expert System?

An *Expert System* is a knowledge based reasoning system that captures and replicates the problem solving ability of human experts. Expertise consists of knowledge about a particular area or *domain*, understanding of domain problems, and skill at solving some of these problems.[6]

Domain knowledge, rather than the complexity of formal reasoning methods, is the key to solving difficult problems. Knowledge can take many forms:

- Specific inferences that follow from specific observations,
- Necessary and sufficient conditions for achieving a goal,
- Probable causes of symptoms.

Knowledge can be represented in several ways, such as rules, frames, and logical predicates. In most current Expert Systems, knowledge is expressed in *rules*, and these *rule-based systems* will be discussed here. The skill of these systems increases at a rate proportional to the enlargement of their knowledge bases. They usually explain their conclusion by retracing their lines of reasoning and translating the logic of each rule used into natural language.

There are a variety of reasons for building systems that capture human expertise:

- Experts retire or are promoted, taking their knowledge with them.
- An expert's time is not particularly well spent answering users' questions.
- Expertise may be scarce.
- Expertise may be expensive to deliver.
- Experts are not always consistent.

Figure 1 shows a diagram of the most basic components of an expert system. The *knowledge base* is usually a collection of rules about the problem domain, supplied by the expert. Problem solving knowledge is usually used in consultations between the Expert System and the end user or with other programs. The *working memory* keeps track of what the system knows during each consultation. The *inference engine* looks in the working memory and the knowledge base to see what is true at any given time to resolve queries.

6. Hayes-Roth, "Rule Based Systems", Communications of the ACM, 28(9), 1985.

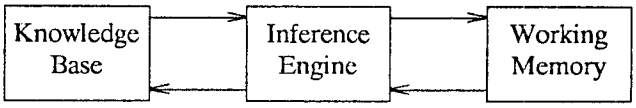


Figure 1

Consider a simple knowledge base with only 2 rules:

1. Rule 1: IF *a* THEN *b*.
2. Rule 2: IF *b* THEN *c*.

Each rule has a *premise* clause (IF), and a *conclusion* clause (THEN). If a user poses the question "IS *c* TRUE", the inference engine looks through **all** the rules in the knowledge base and tries to find one that would conclude about *c*. It finds rule 2: IF *b* is true, THEN *c* is true. Now, if the inference engine could discover whether *b* was true, then it would know whether *c* was true. So a query is made of working memory: "Is *b* true?" Again, the inference engine examines the rules in the knowledge base, this time looking for a rule that would come to a conclusion about *b*. It finds rule 1 - this leads to asking the question "Is *a* true?" If the user responded "yes", then the inference engine would:

- erase the query about *a*,
- place the fact that *a* is true in the working memory,
- use Rule 1,
- place the fact that *b* is true in the working memory,
- use Rule 2,
- place the fact that *c* is true in the working memory,
- resolve the original query, and
- tell the user "Yes, *c* is true."

This is a simple example of a **backward chaining** system. The inference engine works backwards through the conclusions of the rules, trying to determine whether the premises are true. In a **forward chaining** system, facts would be placed in the working memory and the inference engine would report any conclusions it could to the user. For instance, if "*a* is true" were entered, the system could report that both *b* and *c* were true. Forward chaining was used in our PC/PROLOG Expert System for MVS/XA. Backward chaining systems tend to be *goal driven* - they work backward from some specific goal (*c* above). Forward chaining systems tend to be *data driven* - the system is primed with certain facts and draws any conclusions it can.

Other information must reside in the knowledge base in addition to the rules. For example, consider the user interface. How does the Expert System know what question to ask the user when that becomes necessary? This information is stored in the part of the knowledge base that describes the *a*'s and the *b*'s - the **traits** characteristic of a given problem domain. For example, the trait USERS may carry with it the prompt "On average, how many users are connected to the system?"

But when many rules are present in the knowledge base, complex consultations can occur. It can become difficult to predict what questions will be asked and in what order unless some precautions are taken. The builders of the famous MYCIN medical Expert System[7] found that doctors were somewhat bewildered when

the consultation questions jumped from one topic to another: "What is the patient's age?" "What was the result of test X?" "What is the patient's name?" and so on. It becomes necessary to use more control information. For instance, the system could be instructed "Ask about traits *a*, *b*, and *c* first, then try to find the value of trait *d* by forward chaining, then ask about traits *e* and *f*, ..." This consultation and control information is also kept in the knowledge base. As will be shown later, the PC/PROLOG Expert System for MVS/XA employed this concept in its structure of its knowledge base.

It is sometimes necessary to keep even more information in the knowledge base, such as special purpose algorithms, or methods needed to retrieve information from databases that reside outside the Expert System. Thus the task of **knowledge engineering** includes more than eliciting rules from the expert; all the traits (such as *a* and *b* above), and their characteristics and accompanying control information must be identified as well.

A natural question asked about Expert Systems is "what are the advantages of using a knowledge based system?" In general, knowledge based systems extend the speed, accuracy, consistency, availability, and affordability of computation from *clerical* to *intellectual* applications. Specifically,

- *Behavior* - Users can ask "Why?" questions to show why certain decisions were made. This is in direct contrast to a traditional data processing system, where the only way to verify the system's reasoning is to simulate the system's behavior.
- *Knowledge Independence* - A piece of knowledge (a rule) can be examined and easily changed somewhat independently from other pieces of knowledge in the system.
- *Separation* - Knowledge is kept separate from the reasoning process that uses it. Building an Expert System consists of changing *knowledge* rather than program code. The expert and the knowledge engineer can examine and change this knowledge since it is ideally represented in an English-like format. Because of this separation, rule based systems can be incrementally developed with steady performance improvements.

3. Knowledge Engineering / Knowledge Acquisition

What is Knowledge Engineering? This is the term given to the extraction, description, and computerization of expert knowledge. Knowledge consists of descriptions, relationships and procedures in a domain of interest. The major task in building an Expert System is the transfer of knowledge from its source into the Expert System; this is called **knowledge acquisition.**

Potential sources of knowledge include human experts, textbooks, data bases, examples, case studies, and personal experience. Within CPE, nearly all these sources can be used to elicit performance knowledge. The prototype system developed using PROLOG used knowledge that was extracted from technical papers, IBM manuals, and personal experience.

The process of building a comprehensive Expert System can be described as an iterative, cyclic interaction between the expert, the knowledge engineer, and the system itself:

1. The expert tells the knowledge engineer what rules to add or change.
2. The knowledge engineer makes changes to the knowledge base.
3. Several tests are run for consistency checking.
4. If any problems arise, they are discussed with the expert, and we go back to step 1.
5. The expert runs the modified system on new cases until new problems are discovered.
6. If no problems are encountered in a substantial number of cases, the expert stops checking the system.

---

7. Buchanan, B. G., Shortliffe, E. H., "Rule-Based Expert Systems: The MYCIN Experiments of the Heuristic Programming Project", Addison-Wesley, Menlo Park, CA., 1984.

Hayes-Roth outlines stages of Expert System knowledge acquisition that roughly correspond to systems analysis and software engineering. In this acquisition model, an Expert System grows by proceeding from simple to increasingly more difficult tasks and by incrementally improving the representation and organization of the knowledge. We outline this acquisition model below.

- *Identification* - Identify the problem area and define its scope while determining the resources needed and deciding on the goals and objectives of the system. Questions that must be answered include "What class of problems will the Expert System be expected to solve?", "What data is required?" and "What are important terms and their interrelationships?"
- *Structure* - Key concepts, relations, and information-flow characteristics needed to describe the problem-solving process are defined. Related questions might include "What is given and what is inferred?", "How are objects related?", and "What is the information flow?"
- *Formalization* - Concepts and relations are mapped into a formal structure suggested by the Expert System. Related questions include "What kinds of questions should be asked to obtain data?", or "What is the cost of data acquisition?"
- *Implementation* - A knowledge engineer combines and organizes the formalized knowledge to make it compatible with the information flow characteristics of the problem, resulting in a prototype that can be executed and tested.
- *Testing* - The performance of the prototype is evaluated and revised to conform to the standard defined by experts in the problem domain. Typically, this is an iterative process: an expert evaluates the program's performance, while the knowledge engineer revises the knowledge base.

It is beyond the scope of this paper to examine knowledge acquisition techniques in detail. Note that most of these techniques require that knowledge be elicited from information sources and first placed into an **information base.** From there, it is analyzed and organized into knowledge bases, and then used with an Expert System. The knowledge must be tested for *necessity and sufficiency*, and the knowledge bases must be refined accordingly. Techniques in knowledge acquisition often have roots in psychology: the Delphi technique[8] , and Personal Construct Theory.[9] The interested reader is encouraged to examine these and other sources of knowledge acquisition information.

Our PC/PROLOG based Expert System for tuning MVS/XA faced the same problem of knowledge engineering / knowledge acquisition. Though we chose as our initial knowledge source written material (e.g. papers, IBM manuals), we first addressed questions of identification. We felt that the most common types of tuning problems for MVS/XA are I/O related; thus, the class of problems for our first prototype was limited to I/O and paging/swapping. We next needed to identify the data that would be required, and this was inferred from our information sources. For example, we decided that it would be important to know whether or not dedicated page volumes were being used. Overall, the objective of our system was to gather information about the presence/absence of performance "symptoms", analyze these to form a "diagnosis", and then suggest a "treatment."

In defining the structure, we grouped certain traits together that are related to specific problems. For instance, values for pend time, connect time, disconnect time and IOS queue time are related to poor I/O response time - we could derive a value for I/O response time from these traits. The procedures for finding a problem could then be informally described by grouping the key traits together, inferring any additional information, and finally describing (informally) the relationships (rules) between the traits and the problems to be solved.

---

8. Jagannathan, V., Elmaghraby, A. S., "MEDKAT: Multiple Expert Delphi-Based Knowledge Acquisition Tool", Computer Science Department Technical Report, University of Louisville, 1985.
9. Boose, J. H., "Personal Construct Theory and the Transfer of Human Expertise", Proceedings, National Conference on Artificial Intelligence, Austin, Texas, 1984.

The formalization of structure into a more formal representation is usually made easy when an Expert System shell is used. The format of rules is specified by the shell, so the transformation does not have to be difficult. In our case, no such shell was used; our problem was compounded because we were developing our own shell. In a later section of the paper, we will describe the steps that were taken to build a shell and to determine a "language" for expressing CPE rules.

Implementation implies that the system be built. This process will be described later. But an interesting question about the focus of rules comes up. We might build a rule like "If page delay time is greater than 100 msec, then you should examine the page/swap dataset configuration." Here, the presence of the symptom (page delay time > 100 msec) implies a potential diagnosis, which would have a corresponding treatment (e.g. consider dedicated page volumes). We will call this a *forward leading predicate*. But now consider the following: if a treatment was "use dedicated page volumes", then couldn't the Expert System ask "are you using dedicated page volumes?" If the answer were yes, then suggesting this as a possible treatment is redundant and confusing. This is an example of a *backward looking predicate* - the presence of a potential treatment would imply that the Expert System should look for *other* treatments, or, reply that the system does not know a treatment for the problem.

Testing our prototype consists simply of letting an "expert" use the system, and incorporating any additional knowledge the expert gives into the system. For this, our prototype was given a "why" facility - the ability for the user to ask how a particular diagnosis was reached. In addition, the prototype was given an editing capability over the contents of the knowledge base. Using commonly used PC editing keys, the expert could **add, delete or modify** not only the rules in the system, but the control mechanism, and the contents of the help, diagnosis and treatment messages that the system uses. In short, the expert is given full control of all of the knowledge that the system has during the entire testing process. After testing, these features were left in the Expert System.

4. PROLOG

Over the past two decades, the price of hardware has dropped dramatically, while the costs associated with software development have now become the dominant portion of a total system budget. This rapid rise has influenced the development of new programming tools that simplify system development. PROLOG is the result of years of effort in this area. PROLOG was developed at the University of Marseilles, France by Alain Colmerauer in the early 1970's. It was designed to be a convenient tool for *PRO*gramming in *LOG*ic (hence the name). It can be more powerful and efficient than many traditional programming languages (e.g. PASCAL, FORTRAN, etc.)

A PROLOG program uses a description of a problems' facts and rules, and then finds all possible solutions to the problem. In PROLOG, a programmer describes *what* must be computed, rather than how the computation should be carried out. This declarative (rather than procedural) approach eliminates well known errors that are common in other programming languages (e.g. one too few iterations in a loop). So aside from some initial declarations, a PROLOG program consists of a set of *facts* (e.g. john likes mary, tom likes sam), and a set of *rules* (e.g. jean likes X if tom likes X). PROLOG *deduces* that jean likes sam. Goals like "find every person who likes sam" can be asked of a PROLOG program. PROLOG uses a built-in backtracking mechanism that, once one solution has been found, causes PROLOG to reevaluate any assumptions that were made to see if some new variable values will provide additional solutions.

To better understand the syntax and operation of PROLOG, consider the following sample program:

```
clauses
   watches(bill,bob).
   watches(john,jane).
   watches(fred,felicia).
   watches(mitch,bill).
   watches(brenda,greg).
   watches(bob,bob).
   watches(fred,greg).
   watches(bill,phillipe).
```

The clause "watches(bill,bob)" is the fact "bill watches bob." Thus, if we pose as a goal "watches(mitch,bill)", PROLOG would reply "true". "watches(bill,jane)" would result in a reply of "false". Variables begin with upper case characters; thus a goal of "watches(john,Who)" (analogous to asking Who does john watch?) causes a reply of "jane". But the goal of "watches(fred, Who)" results in **both** "greg" and "felicia". The "_" character is called the blank variable; a goal of "watches(fred,_)" is analogous to asking "does fred watch anyone?" This would result in the reply true.

If we add the *rule* "happy(eric) if watches(brenda,eric)" (i.e. eric is happy if brenda watches eric), the goal "happy(eric)" ("is eric happy") would be satisfied. The rule "nervous(Who) if watches(bill,Who)" would be satisfied with "bob" and "phillipe"; this is analogous to asking "Who is nervous", and answering using the rule "someone is nervous if bill is watching them." Thus, PROLOG looks through the already established facts and concludes that since bill watches both bob and phillipe, that both bob and phillipe are nervous.

Though there are many interesting features of the language (the reader is encouraged to examine the cut operator, as well as backtracking), the most commonly used *data structure* is known as a **list.** Analogous to arrays, lists are collections of objects (elements) of the same type, separated by commas and placed inside square brackets. For example, **[1, 2, 3]** is a list of the integers 1, 2, and 3. PROLOG can manipulate a list by dividing it into two parts: a ***head*** and a ***tail***. The head is the first object on the left in the list; the tail is the rest of the list. Syntactically, a PROLOG list is written with a vertical bar separating the head from the tail **[head | tail].** Lists are treated just like other objects in PROLOG, and PROLOG has built-in facilities (predicates) for manipulating the elements of a list.

5. The Development of the Expert System

The reader should, as this author did, feel comfortable with the concept of an Expert System, as well as some PROLOG basics. The discussion now turns to building the system itself. Consider that this author is (by definition) a programmer of *procedural* languages (FORTRAN, PL/I, C, etc.) Thus, PROLOG is somewhat unnatural, and causes some conceptual problems that have to be overcome. For example, consider the following rule: Bad I/O response time exists if pend time is greater than or equal to 10% of the sum of disconnect plus connect time, and IOS queue time is greater than the sum of pend time, disconnect time and connect time. In PROLOG,

```
bad_resp(IOSQ, PEND, DISC, CONN) if
   PEND >= (DISC + CONN) * 0.10   and
   IOSQ > (PEND + DISC + CONN).
```

This simple rule implies the our PROLOG program should:

- Prompt the user for the value of each variable (IOSQ, PEND, etc.),
- Provide help with each prompt should the user not understand what is being asked for,
- Validate each resulting reply - here, with a simple numeric test,
- Pass the replies as parameters to this rule that does the evaluation (bad_resp(IOSQ, PEND, DISC, CONN)),

- If the rule is satisfied, provide the user with the diagnosis (poor I/O response time because of these symptoms),
- Look up a treatment for this diagnosis.

Our first version of the system has the rule (knowledge) coded in PROLOG; thus it is part of the reasoning process. Given all the rules to be coded, plus the help, diagnosis and treatment messages, the resulting program was rather long, and strongly resembled a traditional procedural program.

Recall that an objective of an Expert System is to *separate* the knowledge from the reasoning process. Our first version mixed the knowledge and reasoning process together. Thus, we examined the rules a little closer, since they were the driving force behind the design of the program. When invoked, rules result in true or false. Thus, if the prompts were structured as true or false questions, the inference mechanism could be greatly simplified. For example, consider the following:

```
demand_paging_problem(P1, P2) if
  ask(i1, P1) and
  ask(i2, P2).

ask(Index, Prompt) if
  prompt(Index, Prompt) and
  write("Is ",Prompt,"? ") and  /* an ordinary write   */
  readln(Reply) and             /* read the user reply */
  yes(Reply).                   /* validate the reply  */

prompt(i1,"Demand Paging High").
prompt(i2,"storage isolation used for IMS or CICS").
```

The inference mechanism "ask" is now divorced from the knowledge of what prompts should be used. The prompts themselves are associated with specific *traits* (i1, i2). For illustration, we have included the prompt clauses within the program, but the collection of prompt clauses is ordinarily kept within a *separate, editable file.*

Note though that the control information - ask about trait i1 then trait i2 relative to demand paging - is still part of the PROLOG program. Thus, we need to carry this "separation" of knowledge from reasoning one step further. We create in a separate, editable file, a **knowledge base** of the following form:

- prompt(trait, prompt messages).
- condition(symptom, [list of traits]).
- diagnosis(symptom, [list of traits], treatment).
- treatmsg(treatment, treatment messages).
- diagmsg (treatment, diagnosis messages).
- helpmsg (trait, help messages).

where:

- *prompt* - associates with each trait the corresponding prompt message to ask the user. "Is" is printed before each prompt message implying a true/false question.
- *condition* - associates a list of traits with a particular symptom. For example, condition(ratehi, [i1, i2, i3]) would imply that traits i1, i2 and i3 are all associated with the symptom ratehi. This knowledge is used to *control* the flow of prompting.
- *diagnosis* - associates the true/false values of traits with whether a particular symptom exists. If the symptom does exist, *treatment* is the index to invoking the correct diagnosis and treatment

messages. For example, consider

— diagnosis(ratehi, [i1, i3], fixrate).

— prompt(i1, "Demand Paging High").

— prompt(i3, "Storage Isolation Used for IMS or CICS").

This implies that the condition ratehi is present if traits i1 and i3 are both true - demand paging is high and storage isolation is being used for either IMS or CICS. A minus sign "-" in front of a trait, e.g. -i2, would imply that trait i2 is tested for falsehood. On closer examination, *diagnosis* represents how tuning rules are coded. If it is possible for a symptom to be caused several different ways, then each possibility can be coded; the inference mechanism automatically checks for all possible solutions.

- *treatmsg, diagmsg* - associate a treatment with a set of messages - treatmsg deals with treatments, diagmsg deals with diagnoses.
- *helpmsg* - associates a trait with a set of help messages that try to give more information about the particular trait.

Thus, each of the above represents knowledge, and is kept in a file that can be edited by the user. The inferencing mechanism is thus reduced to processing these predicates; it is short, and contains no knowledge. Conceptually, it can be used in any expert application as long as the syntax outlined above is followed.

Several features were added to the general inferencing mechanism: after a consultation, the user can invoke a built-in editor, which resembles the PC word processing program WORDSTAR.[10] Rules can be added, deleted or modified. Care should be taken so as not to violate the syntax outlined above - no syntax checking is done. In addition, a "Why?" feature was added. After each diagnosis is given, the user is given the opportunity to ask the Expert System why that diagnosis is being given; in other words, how that conclusion was reached. The inferencing mechanism *backtracks* from the treatment using the diagnosis and prompt predicates to replay the rules that were invoked using English.

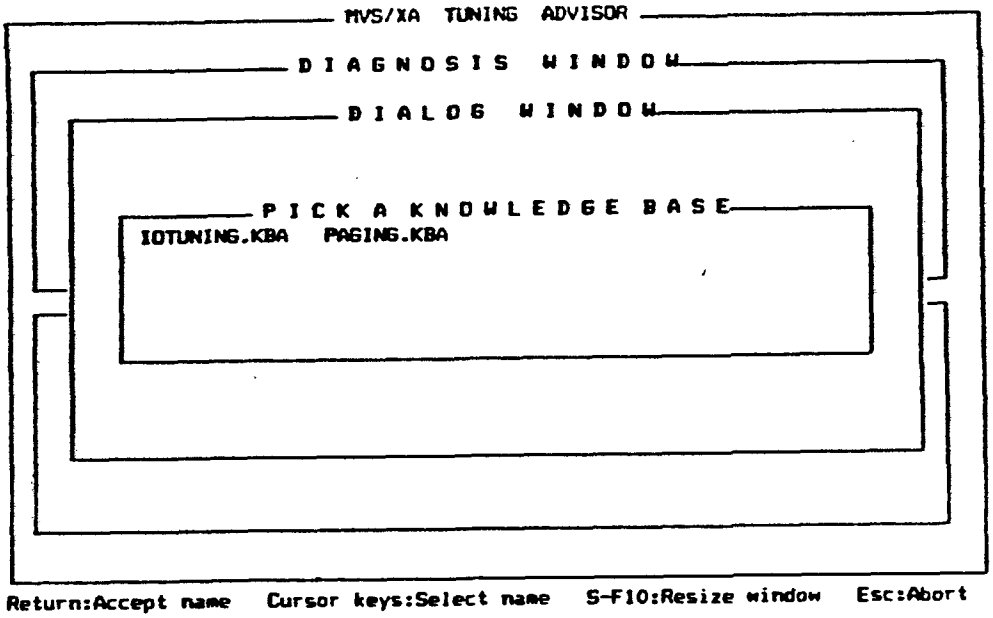
6. A Consultation Session

It is difficult for the reader to get a firm understanding of the Expert System described unless (s)he has the actual floppy disk with the program available. To eliminate this difficulty, this section will display parts of a sample consultation session with the Expert System. The discussion will show the screen displays a user would see, and will provide an overview of the overall flow of the system. Finally, we will give a short discussion of how a knowledge base can be edited.

Required Files - First, the program (henceforth called *ADVISOR*) must be present along with at least one knowledge base. Knowledge bases are ordinary text files that can be created by any ordinary PC editor that does not use embedded control characters. *ADVISOR* knows that files are knowledge bases if they end with the suffix KBA (for *K*nowledge *BA*se). In our prototype, we created two knowledge bases, IOTUNING and PAGING.

The following is what is displayed when *ADVISOR* is invoked:

10. WORDSTAR is a registered trademark and product of the MicroPro International Corp.

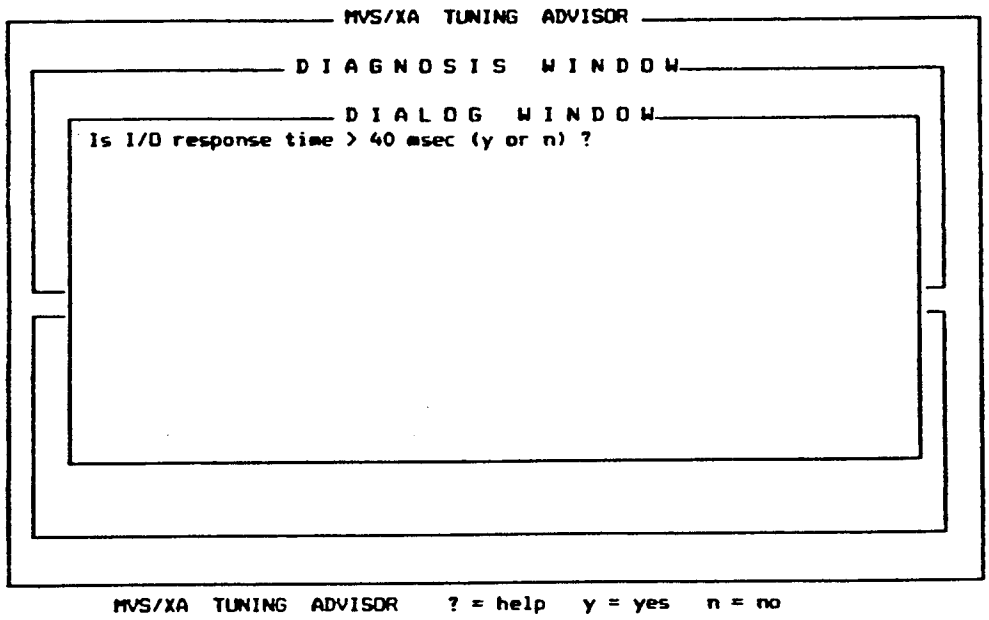


The user uses the PC arrow keys to select which knowledge base they want to use; in essence, this is actually equivalent to selecting a *knowledge domain* to be exploited.

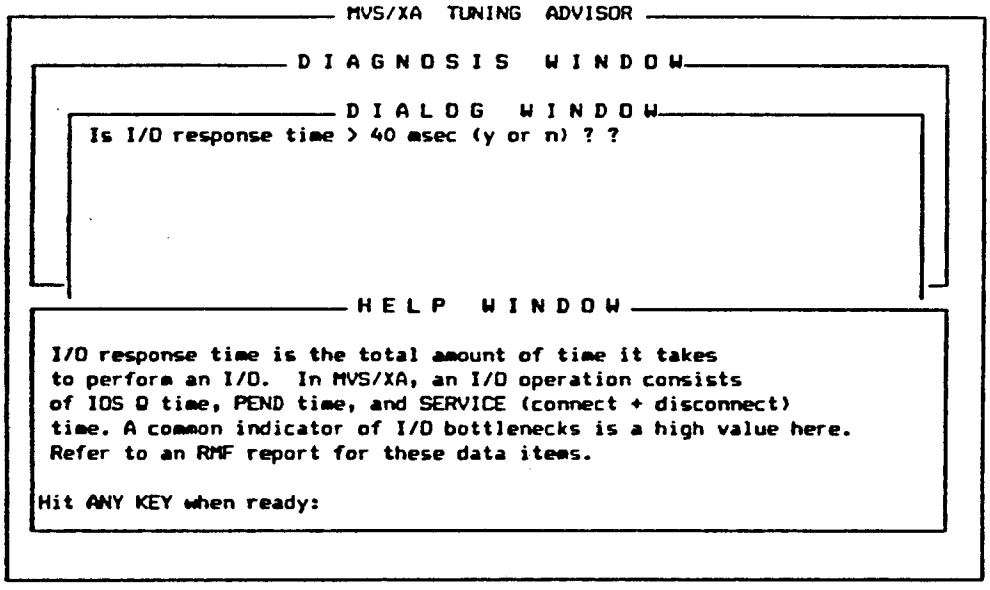
At this point, we point out the different *windows* that *ADVISOR* uses.

- *DIALOG* - This window is the primary means of communication between *ADVISOR* and the user. Performance questions are posed to the user, and each reply should be followed by hitting the RETURN key.
- *HELP* - Should a user need assistance in understanding a question, the HELP window will display that information. HELP is invoked by replying with a ? to the question posed in the DIALOG window. In addition, the HELP window is used later to explain why a particular diagnosis was arrived at.
- *DIAGNOSIS* - After a set of related performance questions is asked, any performance problem found will be noted in the DIAGNOSIS window. Here, the user is given the opportunity of asking why a particular DIAGNOSIS is being made.
- *TREATMENT* - Often, a particular diagnosis will have one or more possible remedies, or TREATMENTS. Each possible TREATMENT is displayed in this window successively, each one being preceeded by giving the user the opportunity to ask WHY.
- *EDIT* - Once a consultation session has ended, the user is given the opportunity to edit a knowledge base. Here, a *full-screen* editor is invoked that allows the rules in the knowledge base to be modified, deleted or extended.

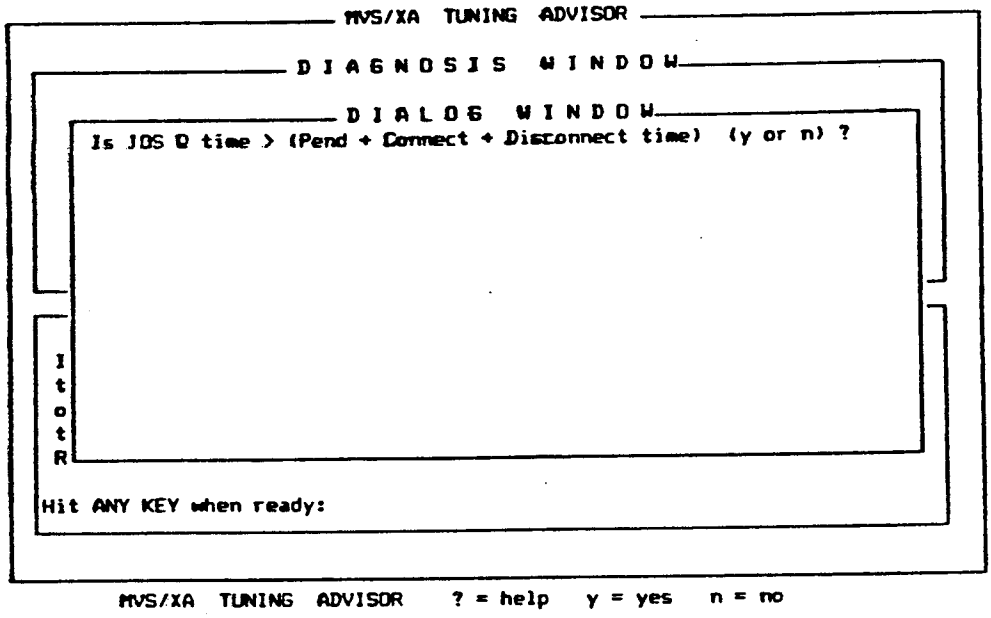
We now examine several screens to become familiar with the "look and feel" of the system. As each question is asked in the DIALOG window -



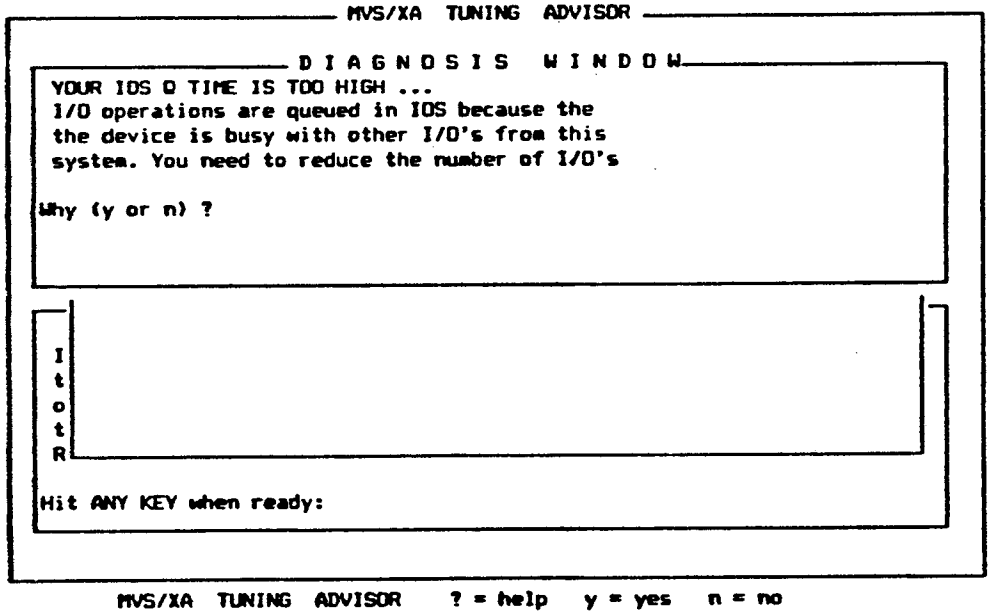
the user can ask for help by typing ? and hitting the return key -



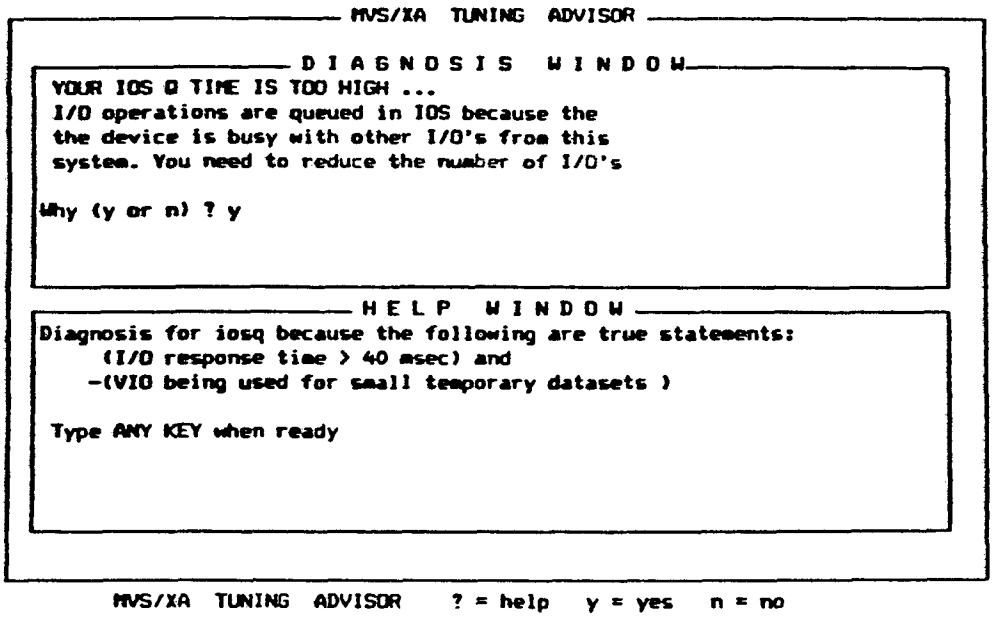
When finished with help, typing any key will return the user to the DIALOG window, where (s)he is reminded to reply with a *y* (yes) or an *n* (no). From there, the next question is asked -



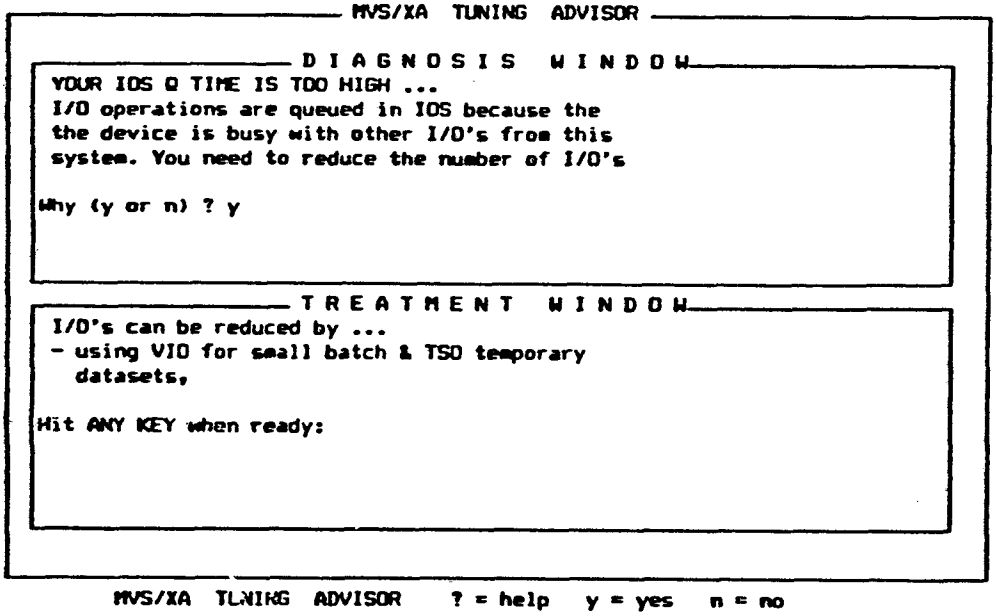
Once all of the questions or *traits* corresponding to a logical sequence have been asked, descriptions of any performance problems detected appear in the DIAGNOSIS window -



Here, the user is given the opportunity to ask what caused (WHY) this diagnosis to be made by *ADVISOR*. If the user replies *n* to the WHY prompt, the TREATMENT window appears (more later). If the user replies *y* to the WHY prompt, the HELP window reappears containing an English-like replay of the prompts used to arrive at the current diagnosis -

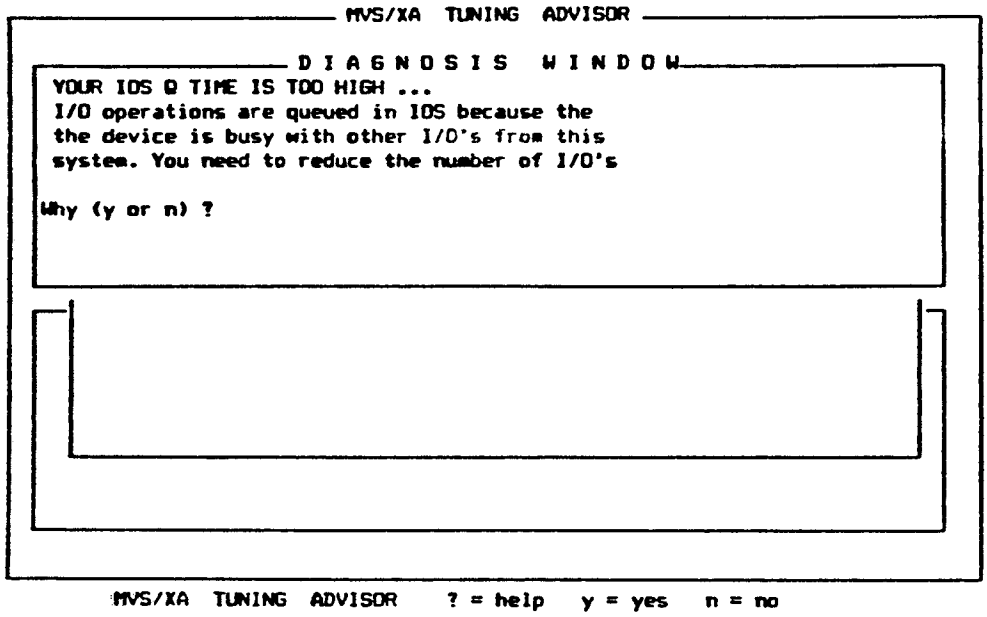


Notice, if a particular prompt is preceeded with a hyphen (-), this implies the *negation* of the particular prompt. Next, the user types any key which causes the HELP window to be replaced by the TREATMENT window. Here, a possible suggestion is made that the user can adopt to ultimately help tune the actual MVS system.

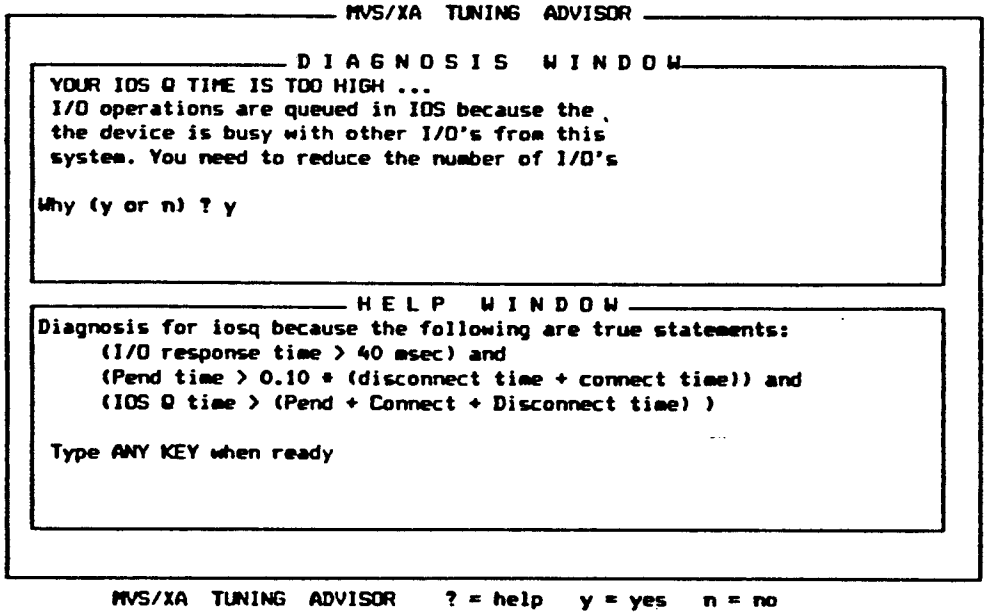


As before, the user types any key which causes *ADVISOR* to continue.

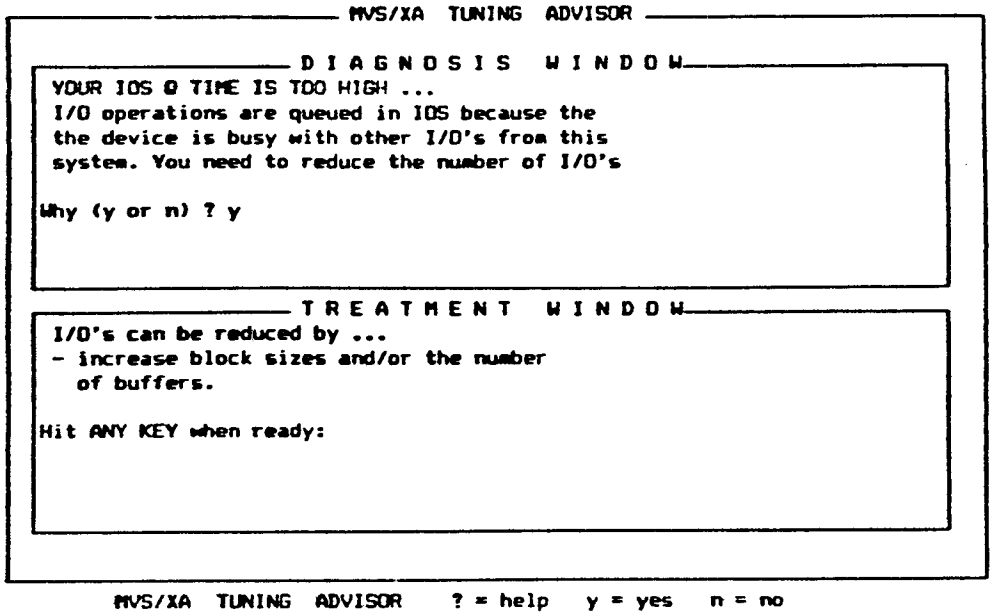
If a particular diagnosis reappears following the disappearance of the TREATMENT window,



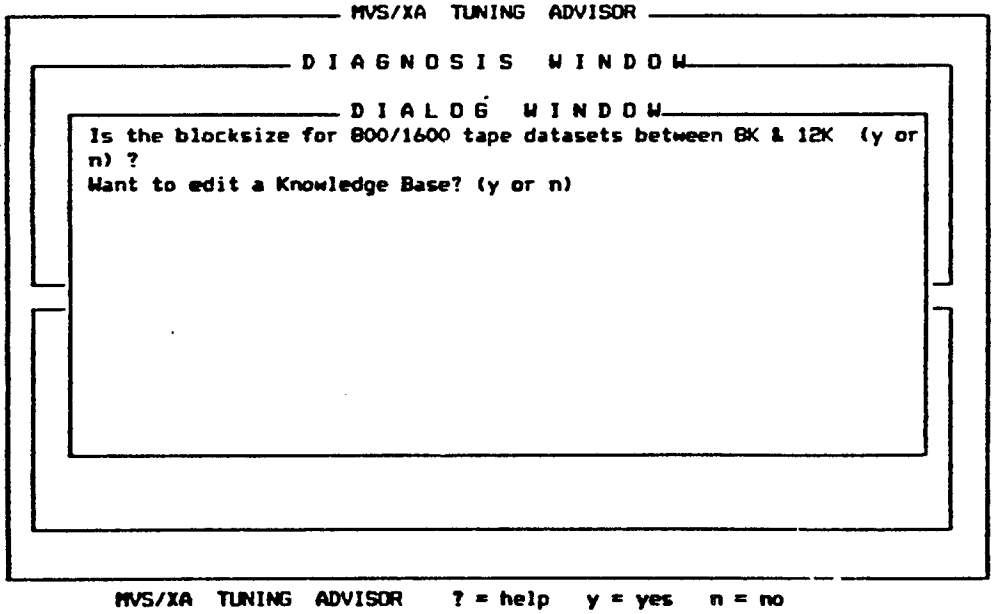
it implies that the current diagnosis has been reached by invoking additional rules. Again, the user can ask WHY, and we can see that a different reply is given in the HELP window.



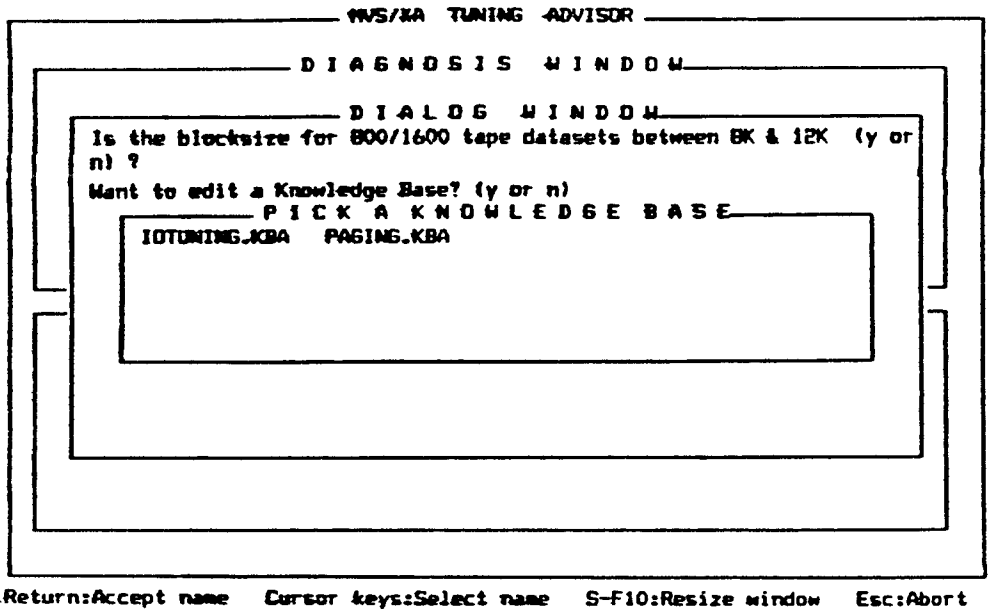
And, as before, the HELP window is replaced by the TREATMENT window, this time giving a *different* remedy the user might apply to the actual MVS system.



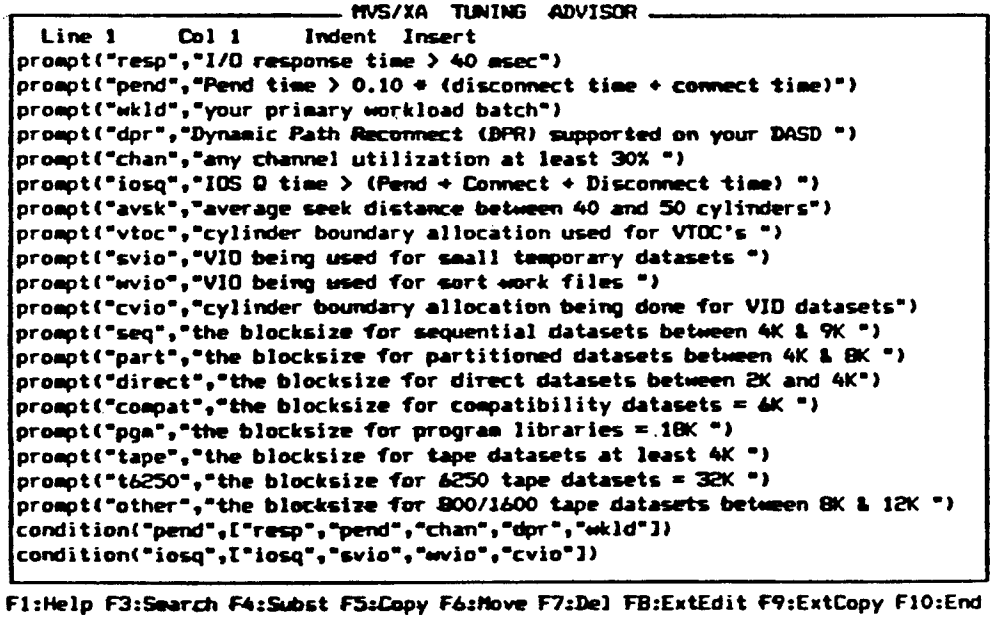
Once the consultation session has ended, the user is given the opportunity to edit a knowledge base - (a no reply ends the session).



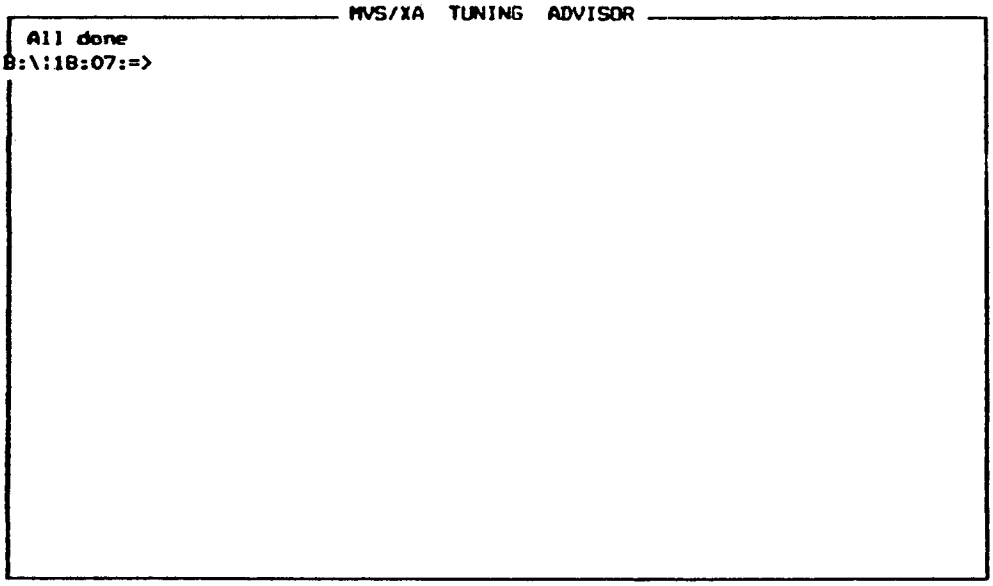
As at the start of program execution, the user selects a knowledge base with the C arrow keys followed by enter. Here, a full-screen editor similar to WORDSTAR is invoked that operates on the knowledge base.



The arrow keys, PgUp, and PgDn are used to move around the window, the Ins and Del keys are used to set/unset insert mode and for character deletion (respectively). Help on all the editing functions is available by hitting the F1 key. The editor has search (F3), substitute (F4), copy (F5), move (F6), and block delete (F7) capabilities. F8 is an *auxiliary* edit function. When invoked, the user can edit yet another file while remaining within the edit session for the knowledge base. This is similar to the split-screen facility under TSO/ISPF on the mainframe. F9 is an external copy function. Another file (e.g. a knowledge base previously created) can be copied in part or as a whole to any section of the knowledge base being edited. F10 is used to end the editing session; here, the user is asked if (s)he wants to save the file just edited.



Finally, the session ends.



7. Issues for Future Development

As this PC/PROLOG based system is still in its infancy, several issues regarding where the system should next go need to be expressed. The following issues should apply to any Expert System for CPE.

1. *Location Help* - Rather than just explaining the meaning of traits in help messages, add information about where to find measurement(s) associated with the trait; e.g. on page 2, bottom of an RMF report. This is an example of using an additional class of knowledge that, until now, only the expert has.

2. *Automatic Data Reduction* - Ideally, a "front-end" mechanism that could read and analyze measurement sources like RMF automatically would greatly simplify the user's interaction with the system. Rather than putting the burden of responsibility on the user to get the answers to the prompts, a program(s) could be designed to extract the corresponding answers, and download these to the PC. A potential source for much of this data could be the MICS data base (Morino Associates), which is a repository for many different types of measurements. The MICS data dictionary already assigns names to many measurements, and these names could be related to the traits we referred to previously.

3. *Links to other PC packages* - If data were downloaded from the mainframe to the PC, it could be stored in a spreadsheet or a database. Products like 1-2-3 from Lotus Development or dBASE-III from Ashton-Tate could then be used to do the analysis on the PC rather than the mainframe. These could be invoked from PROLOG directly, as the interface is already defined.

4. *Certainty Values* - As Artis pointed out (5), some rules are not as clear cut as one would like. If a certainty value, say between 0 and 100, was attached to each rules' conclusion, a *certainty threshold* could be specified for the premises. Below this threshold, rules are considered to be false. This would allow *fuzzy reasoning* to be a part of the system.

5. *External Program Execution* - As in many mainframe offerings, it is sometimes convenient to have an **exit** facility. This would allow the PROLOG system to escape to execute some user-specified program, and then return. This could greatly increase the potential applicability of the system.

8. Conclusions

The objective of this paper was to provide new insight into CPE by applying AI techniques. An Expert System is a complicated object with many advantages; foremost among these is that expert knowledge is

elicited and stored in an executable form.

The system described is only a beginning at applying current technology. It was **never intended** to replace the human expert. On the contrary, the original intent was to use the system as a training mechanism, where a new CPE practitioner could pose performance scenarios to the Expert System, and see how these would be analyzed, and what recommendations would be made.

It is our belief that similar applications will flourish in the next decade. Yet, though the size of knowledge bases may increase, we believe that the human inference mechanism is still better than any that will be built during that time. We conclude with the following from Aristotle: "It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of the subject admits, and not to seek exactness when only an approximation of the truth is possible."

9. Acknowledgements

The author would like to take this opportunity to thank those referees and Editorial Review board members of the Computer Measurement Group for their constructive comments and suggestions. Though the work was conceived and performed solely by the author on his own time, we would like to thank those staff members of Bell Communications Research for reviewing the work as it progressed. Their input is gratefully appreciated here.