Automated Tutoring in Interactive Environments: A Task Centered Approach

Ursula Wolz*
Kathleen R. McKeown+
Gail E. Kaiser++

Technical Report CUCS-392-88

Columbia University
Department of Computer Science
New York, NY 10027

Abstract

Tutoring in interactive computing environments is sometimes more properly understood as *consulting*. A tutor's implied curriculum may be adaptable to the user's knowledge and experience but still not meet the user's immediate needs — to get some task done. A consultant however, can dynamically adapt to address the task at hand. We present a user's goal-centered approach to tutoring in interactive environments, and describe how we automate certain tutoring strategies appropriate for consulting behavior. We have implemented our approach in GENIE, a question answering system for the Berkeley Unix Mail system. We focus on the pedagogical strategies employed by Genie to best meet the user's immediate needs.

keywords: Automated consulting, help systems, intelligent tutoring systems, interactive computing environments.

*Supported by ONR grant N00014-82-K-0256. +Supported by NSF grant IST-84-51438, by ONR grant N00014-82-K-0256, by a grant from Bell Communication Research, and by the Center for Advanced Technology. ++Supported by NSF grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens and Sun, by the Center of Advanced Technology and by the Center for Telecommunications Research.

1. Introduction

Interactive computing environments are designed to provide supportive resources for a range of users with different expertise and computational goals. Such environments may be as simple as mail systems and word processors, or encompass sophisticated data bases, design tools or programming languages. Yet all such environments contain an underlying set of functions or constructs with which users accomplish tasks. A problem arises in providing resources through which users can initially learn about the environment and then later extend their expertise. Standard curricula — even those provided on-line — require users to subordinate their immediate needs to the agenda embedded in the curriculum. We have developed a solution through the implementation of GENIE (GENerated Informative Explanations), an answer generating system that tutors specifically to the current needs of the user in the domain of Berkeley Unix™ Mail.

Automatic consulting provides a valuable test bed for theories about personalized tutoring because of the unique relationship between the consultant and the user. The knowledge conveyed in a consulting session is determined by the current task or goal of the user and the user's questions about that task. By contrast, in most tutoring, both on and off the computer, the teaching agenda is predetermined by an implied curriculum chosen by the tutor. This is not appropriate for a consultant since users have a multitude of different needs, backgrounds, and deficiencies in what they know. Furthermore, most tasks can be done in more than one way, and the core functions can be used in more than one task. A curriculum may cover all the tasks and functions, but will rarely capture the richness of the relationships between them. Although we focus on task centered environments, exploratory learning environments that encompass a curriculum can benefit from consulting. In task centered settings, consulting is based on tasks initiated by the user. In a learning centered setting, a tutoring agent initiates the task, but the student may still require consultation in order to accomplish it.

We take a user's goal-centered approach to consulting in which help given is a direct function of users' goals, their knowledge and the current context. Specific tutoring strategies determine the form and content of an answer to the user's questions. A user model represents what plans the consultant thinks the user has for achieving goals. An expert domain model

provides a representation for a consultant's knowledge of computational goals, plans to accomplish them, and functions of the environment that can instantiate plans.

We abandon both the classification of aspects of the environment according to level of expertise and the categorization of users as "novice", "intermediate" or ,"expert". Instead, information on a user's exposure to the environment influences the pedagogical agenda of the consultant. Decisions about how to answer to a user's question are based on an analysis of the discrepancies between the expert domain knowledge and the user model, as well as the current situation.

Consulting can be characterized as a three stage process of question understanding, problem analysis and answer generation. We concentrate on the latter two components: analysis, through an algorithm called the Plan Analyst; and generation, through an algorithm called the Explainer. GENIE attempts to answer a question by doing a two phase search of the knowledge base, first trying to construct a coherent relationship between the user's question and computational goal in an attempt to find the most appropriate information for a response; then trying to construct a coherent textual explanation that takes into account what the user already knows.

In this paper, our focus will be on the tutoring strategies GENIE employs in various settings. These include:

- **Introducing:** Presenting functions and goals that the user has not encountered before.
- **Reminding:** Briefly describing functions and goals that the user has been exposed to but may have forgotten.
- **Clarifying:** Explaining details and options about functions and goals to which the user has been exposed.
- **Elucidating:** Clearing up misunderstandings that have developed about functions and goals to which the user has been exposed.

We will also show how the content of the answer GENIE provides depends on the type of question asked and on an analysis of discrepancies between the user model and the expert domain knowledge.

The next section presents our perspective on tutoring and consulting in interactive

environments. Section 3 introduces the tutoring strategies we use. Section 4 describes the knowledge required to choose a tutoring strategy. Section 5 summarizes when different strategies are chosen. Section 6 presents GENIE, our experimental consultant system, and gives examples of how the strategies are used. Section 7 summarizes related work and section 8 presents conclusions and directions for future work.

2. Tutoring in Interactive Environments

Interactive computing environments are sometimes characterized as workbenches of tools with which a user can accomplish tasks. In this respect they are *procedural environments* in which learners develop skills rather than learn facts and associations between facts. In these environments, sophistication and complexity, and consequently power, are built upon simple interfaces. Good environments are often characterized as *customizable* — in which users can bend the tools to their own personal needs, and *extensible* — in which users can build new tools from those that already exist.

Expertise in a procedural environment is not simply a matter of knowing what functions allow you to perform simple tasks. One must know how functions interrelate or interfere with each other when attempting to achieve more complex goals. In other words, one must know *plans* for accomplishing goals where plans may be broken down into sub-goals or directly executed by functions. A particular plan is directly related to a particular goal, but the components of a plan, the steps or sub-goals, may occur in many different plans. Furthermore, in most computing environments there is often more than one plan to achieve a goal.

As in real workshops, learning skills and doing real tasks are intimately woven together. While initial learning may require extensive supervision, once the key concepts have been learned, users are expected to initiate their own goals, and solicit expertise from others only when necessary. Within such an environment, lending expertise is often thought of as ''consulting'' rather than tutoring.

It is important to articulate some of the differences between consulting behavior and tutoring. In more classic settings a tutor has an agenda based on an externally prescribed

curriculum and diagnosed student deficiencies. The tutor assumes some external force motivates the student to learn, or the tutor must find incentives to motivate the student. A consultant, on the other hand, has a limited agenda based on showing the user how to accomplish a task. Motivation for learning is a matter of being able to do the task independently of the consultant. Brown and VanLehn [5] introduced the notion of *felicity conditions* under which new learning is most likely to take place. Within a procedural setting, two situations are extremely motivating. The user attempted to do something and it didn't work. The user finds a task tedious and suspects there is a more efficient way to do it.

Since the user's current task initiates the tutoring dialogue, in the student's mind whatever learning takes place is secondary to actually accomplishing the task. A prescribed curriculum may not always be appropriate for a particular task. In a traditional curriculum it is often assumed that a set of skills must be learned in a particular order, and to a specified level of mastery. Within the context of consulting this is not necessary, since the user can always ask for help again later. Users' knowledge of the functions available within the environment may vary considerably depending on the tasks they do regularly and how much they attempt to customize the environment to their needs.

We do not claim that consulting behavior is appropriate to all kinds of tutoring; only that interactive environments provide a potentially rich metaphor for creating procedural learning environments. Even within these environments there is a place for more traditional tutoring. During the initial stages of using the environment, it seems imperative that the basic concepts and functionalities of the environment are introduced. Here a didactic approach would be more expedient. Later however, when users take off in different directions, consulting behavior is more appropriate, providing truly individualized instruction.

3. Tutoring Strategies

It is clear that automated consulting in interactive environments requires complex interactions between sophisticated processes and knowledge bases. We have chosen to carve a niche within this larger domain that has not been explored as fully as others — namely how best to choose what to say and how to say it. In order to prevent further clouding of the pertinent

issues we have chosen not to concern ourselves with *when* to say it. At the present time we view consulting as *passive* in which information is only presented after it is explicitly solicited through a user's question. Obviously a feedback loop through which the user can rephrase or expand on the question is necessary. We discuss this briefly in section 8.

Our narrow view of tutoring is therefore concerned with understanding a user's question about how to do something, deciding what skills are necessary to do it, and choosing the appropriate way to present the skills to the user. We will focus here on the last of these. Others [26; 21] address understanding the question, and we describe how to determine what to skills to present elsewhere [28; 29].

We have identified four necessary tutoring strategies in computing environments through an analysis of various types of help. They specify the kind of information that is typically included. Two of the strategies we posit, *introducing* and *reminding*, are found as part of written reference material, whether on-line or off. They can be used to provide help about a function or goal the user has, either by introducing details about a function or plan that can accomplish the goal or by simply reminding the user of forgotten functions and plans. If a user already has some information about a function or plan, but it is incomplete or incorrect, then one of two other strategies, *clarifying* or *elucidating* is more appropriate. These strategies are corroborated primarily by informal observations of human consulting situations.

Our analysis of on and off-line help for Lisp, UNIX, Pascal, BASIC, Logo, and a number of word processing programs reveals that reference material tends to fall into three categories:

- **Reference manuals** that provide details and definitions of the environment. The material is either alphabetically ordered or grouped according to the function of the constructs.
- **Support manuals** that provide more explanation about how to use the functions. These are organized according to the function of the constructs.
- **Tutorials and textbooks** that introduce the concepts behind the functions. These tend to be much more explanatory and less definitional than reference or support material. Although they may be organized according to the function of the constructs, there is a greater emphasis on how constructs are combined.

Tutorials and to some degree support manuals are intended to *introduce* new material, while

the concise definitions in reference manuals can efficiently *remind* users of how functions work. All three kinds of material may help users clarify details or clear up misunderstandings of *functions*, but the user must possess strategies for locating the relevant information. Clarifying or elucidating *goals* only occurs in a limited way in tutorials and textbooks. The user's goal may not occur in a written text. Even if it does, texts tend to introduce the simplest techniques for accomplishing a task. Although the information necessary for learning a better way may exist, the user is responsible for finding that information and must often piece it together from various points in the book.

Evidence for two of our strategies, *introduce* and *remind* is also provided by Magers [16] and Borenstein [3] who have drawn a distinction between information that is *definitional* and *instructional*. Definitional information is more appropriate for reminding someone about something they have previously learned, while instructional information is more appropriate for introducing new information. These types differ not only in their format and level of detail, but also in their emphasis and the degree to which related information is included. We therefore choose to *remind* or *introduce* depending on the user's knowledge and goals. We further refine the distinction of Magers and Borenstein, however, by including the possibility of elucidating or clarifying.

We also extend our strategies by using them differently to satisfy distinct tutoring needs: the need for tutoring that is in *direct response* to the question and tutoring that is intended as *enrichment*. The former prevails in order to satisfy principles of informativeness: answer the question that was asked. But it is also possible to present new skills to the user opportunistically. For example if the user asks whether a particular plan will accomplish a particular goal, the consultant must respond informatively that it does or does not. However, if the consultant knows of a better way to accomplish the goal, the opportunity should be taken to mention it. Each strategy can be used both responsively and as enrichment.

In the following sections we describe each strategy in more detail and show examples of their use in reference material. Steps within a strategy that occur in response but not as enrichment are marked with '*'.

3.1. Introducing

Introducing is used to provide generic details about a function or plan assuming no previous background about it. In the reference materials, we found certain kinds of information consistently being presented in the support manuals, tutorials and textbooks. We have identified the kind of information used for introduction and provided an ordering on the information that is often, but not always, used in the texts. Informally, the process of introducing a goal consists of

1. Stating the goal. *
2. If the skill maps to a function, introducing the function, otherwise:
3. Summarizing the sub-goals for the plan for the goal.
4. For each sub-goal either introducing or reminding the sub-goal depending on whether the user model does or does not contain the sub-goal.
5. If it is a top level goal, reviewing the steps in the plan through an example.
6. Relating each step in the example to a sub-goal. *

Introducing a function consists of:

1. If not in the context of introducing a goal, stating the goal.
2. Presenting the syntax.
3. Describing the parameters.
4. Describing any preconditions that must exist for it to work.
5. Describing the effects (which is not the same as stating the goal.)
6. If not in the context of introducing a goal, giving an example. *

This strategy can be seen in the example shown in Figure 3-1 taken from a text book on Lisp [27]. In the first paragraph, the goal is stated (''define procedures such as COUNT-ATOMS'') and a summary of the steps (mapcar and apply) provided. The remaining paragraphs *introduce* the first sub-goal (following step 4). The second paragraph equates the sub-goal ''iterate'' with the function ''mapcar'' and the remainder of the section recursively uses the *introduce* strategy for the function (step 2). Following the second strategy for introducing functions, the third paragraph presents the syntax of the function (''supply the name of the procedure together with a list of things to be handed to the procedure one after the other'') and describes the parameters (''a list of things''). Preconditions are provided out of order in the last paragraph and the effects

are provided in the immediately preceding sentence (''MAPCAR is said to cause iteration of ODDP, because the MAPCAR causes ODDP to be used over and over again.''). Since this is not the top-level goal, we skip step 5, finishing introduction of the function and pop back to introduction of the sub-goal ''iterate'' continuing with step 5, reviewing through an example. There is only one step in the plan for this goal, MAPCAR, and an example is provided (again, out of order) through ODDP. Step 6 is not needed since there is only one step in the plan.

Dealing with Lists May Call for Iteration Using MAPCAR

A somewhat more elegant way to define procedures like COUNT-ATOMS is by means of the primitives MAPCAR and APPLY, two new procedures that are very useful and very special in the way they handle their arguments.

Iterate is a technical term meaning to repeat. It can be used to iterate when the same procedure is to be performed over and over again on a whole list of things. Suppose, for example, that it is to record which numbers in a list are odd. From the list (1 2 3), we expect to get (T NIL T).

To accomplish such transformations with MAPCAR, you supply the name of the procedure together with a list of things to be handed to the procedure one after the other. Consider the following, for example, where the primitive 'ODDP has the obvious definition:

```
(MAPCAR 'ODDP        :Procedure to work with.
        '(1 2 3))    :Arguments to be fed to the
                      procedure.
(T NIL T)
```

MAPCAR is said to cause iteration of ODDP, because the MAPCAR causes ODDP to be used over and over again.

There is no restriction to procedures of one parameter, but if the procedure does have more than one parameter, there must be a corresponding number of lists of things from which to extract arguments that can be fed to the procedure.

As shown in the following example, MAPCAR works like an assembly machine, taking one element from each list of arguments and assembling them for the procedure. .....

Figure 3-1: An example of Introducing

The strategy *introduce* as shown here provides a high level characterization of the type of information provided in the naturally occurring example. This use of strategies is quite similar to the use of schemas in work by McKeown [18], Paris [20] and McCoy [17].

3.2. Reminding

Reminding is used to present the bare minimum of information about a function or plan under the assumption that the reader has some knowledge about it from previous experience. Manuals most often use this strategy.

Reminding about a goal consists of

1. Stating the goal.
2. If the skill maps to a function, reminding about the function, otherwise:
3. Summarizing the sub-goals for the plan for the goal.
4. If this is the top level goal, reviewing the steps in the plan through an example.

Reminding about a function consists of:

1. If not in the context of introducing a goal, stating the goal.
2. Presenting the syntax.
3. Describing the parameters.
4. If not in the context of introducing a goal, giving an example.
5. If not in the context of introducing a goal, relating the function to other pertinent information.

An example of reminding is shown in Figure 3-2 taken from a Lisp reference manual [14]. The first sentence provides a general description of the action (step 1, strategy for functions). The syntax and parameters are then presented (steps 2 and 3) followed by an example (step 4). Additional information about side-effects is also provided (step 5).

3.3. Clarifying

Clarifying is used to compare two functions or two plans for a goal. This is done occasionally in tutorials and textbooks, but is essential for face to face consulting and questions, when the user often queries about the difference between two plans or specifically asks for a better way to achieve a goal.

To clarify a goal, we assume that between two alternative plans, one of them is clearly better. Which one is better can vary depending on context. Clarifying a goal consists of

MAPCAR **Function**

The mapcar function applies a function to each element of each list in a series of lists.

(**mapcar** *function list* **&rest** *more-lists*)

Examples

```
(mapcar #'1+ '(1 2 3))                          => (2 3 4)
(mapcar #'list '(smith) '(sue janie mary))      => ((SMITH SUE))
```

Comments

The iteration of mapcar applies to the elements of the shortest list in the series of lists; excess elements in longer lists are ignored. mapcar returns a list whose elements are the values returned by calls to function.

See also, Steele: 7.8.4. - p. 128.

Figure 3-2: An example of Reminding

1. Stating that a best way (B) exists for the goal.
2. Summarizing an alternative plan (P). *
3. Summarizing B.
4. Describing the relationship between B and P.
5. For steps in B that differ from P, introducing or reminding those sub-goals based on whether those goals exist in the user model, and describing the relationship between this step and the corresponding step in P.
6. Summarizing the steps of B through an example.

Examples from texts do not correspond exactly to our strategy as there is generally no context for identifying which of a set of alternatives is the better plan. In some cases, a context may be hypothesized and one of the alternatives is labeled as better for this case. An example in which context is not hypothesized is shown in Figure 3-3 taken from a Lisp support manual [25]. The goal is mapping over lists and is described in paragraphs 1 and 2. Several alternative plans (single step plans that map directly to functions) are then given (step 2 of the strategy). Although it is not explicitly identified as a best plan, because of the detail given, we can assume the author intends to identify mapcar as a best, or at least default, plan. A more detailed summary of mapcar follows in paragraph 3 (step 3), followed by an example (step 5). Following

this, each alternative function is compared with mapcar and differences noted (step 4).

7.8.4. Mapping

Mapping is a type of iteration in which a function is successively applied to pieces of one or more sequences. The result of the iteration is a sequence containing the respective results of the function applications. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

The function map may be use to map over any kind of sequence. The following functions operate only on lists.

mapcar *function list* &rest *more-lists* [function]
maplist *function list* &rest *more-lists* [function]
mapc *function list* &rest *more-lists* [function]
mapl *function list* &rest *more-lists* [function]
mapcan *function list* &rest *more-lists* [function]
mapcon *function list* &rest *more-lists* [function]

For each of these mapping functions, the first argument is a function and the rest must be lists. The function must take as many arguments as there are lists. mapcar operates on successive elements of the lists. First the function is applied to the car of each list, then to the cadr of each list, and so on. (Ideally all the lists are the same length; if not, the iteration terminates when the shortest list runs out, and excess elements in other lists are ignored.) The value returned by mapcar is a list of the results of the successive calls to the function. For example:

```
(mapcar  abs '(3 -4 2 -5 -6))  => (3 4 2 5 6)
(mapcar  cons '( a b c) '(1 2 3))  => ((a . 1) (b . 2) (c . 3))
```

maplist is like mapcar except that the function is applied to the list and successive cdr's of that list rather than to successive elements of the list. For example:

```
(maplist  '(lambda (x) (cons 'foo x))
          '(a b c d))
    => ((foo a b c d) (foo b c d) (foo c d) (foo d))
(maplist  '(lambda (x) (if (member (car x) (cdr x)) 0 1)))
          '(a b a c d b c))
    => (0 0 1 0 1 1 1)
   ;An entry is 1 if the corresponding element of the input
   ;  list was the last instance of that element in the input list.
```

mapl and mapc are like maplist and mapcar respectively, except that they do not accumulate the results of calling the function.

Compatibility note: In all LISP systems since LISP 1.5, mapl has been called map. In the chapter on sequences it is explained why this was a bad choice. Here the name map is used for the far more useful generic sequences mapper, in closer accordance to the computer science literature, especially the growing body of papers on functional programming.

Figure 3-3: An example of Clarifying

3.4. Elucidating

Elucidating is used to clear up misconceptions and will be used most often when dealing with an individual's problems. Because most texts do not specifically address an individual reader, elucidating is found even less often in texts than clarifying. However, some texts (particularly tutorials) sometimes will posit a possible misconception that can occur. Figure 3-4 identifies 5 kinds of invalidities that can be diagnosed based on work by Joshi, Webber, and Weischedel [10].

- A step in the plan has missing preconditions.
- A step in the plan is missing.
- A step in the plan is extraneous.
- A step in the plan that has missing preconditions is related to a step that is missing.
- A step that is missing is related to a step that is extraneous.

Figure 3-4: Types of invalidities found in plans

Elucidating a goal consists of

1. Stating that the plan does not work for the goal.
2. Summarizing the plan, identifying the problem.
3. If the problem is missing preconditions either state that no plan exists to satisfy them, or introduce or remind about a plan to satisfy them depending on whether the plan exists in the user model.
4. If the problem is a missing step, introduce or remind about it depending on whether it exists in the user model.
5. If there is an extraneous step, identify it, and describe why it is extraneous - what effects does it have that are redundant with some other step.
6. If missing preconditions are related to a missing step, identify the relationship between the two, and introduce or remind about the missing step depending on whether it exists in the user model.
7. If a missing step is related to an extraneous one, clarify the difference between them.

Two examples taken from a Logo tutorial are shown in Figure 3-5 [15]. In both cases the

first sentence identifies that there may be a problem with the plan "typing POTS." The first example illustrates a missing precondition — the workspace must contain procedures (step 2). The second sentence suggests a plan for viewing procedures ("... retype the ones you want", step 3.) The second example illustrates a missing step, namely to pause when the screen is full. The discussion of Ctrl-NumLock provides the missing skill (step 4.)

Printing Out Procedures - POTS Command

```
Possible Bugs:
1. If you don't see anything when you type POTS, you have no procedures in your
   workspace.  You can review the previous chapters and retype the ones you want.
2. If you have many procedures in your workspace, you may find that the titles go
   by too quickly on the screen and disappear as more titles come on.  If this
   happens, repeat the command POTS and press Ctrl-NumLock as soon as you see the
   title or titles you're looking for.  Ctrl-NumLock will pause whatever is on the
   screen.  Press any key to see the rest of the titles.
```

Figure 3-5: An example of Elucidating

---

4. Knowledge that Affects the Choice of Strategy

Strategies dictate the type of information to include in a response, but they will be used and combined differently depending on circumstances. Four kinds of knowledge influence how a consultant decides what to say and how to say it:

- **The Expert Model.** The consultant's expertise influences the choice of what should be presented.
- **The User Model.** The consultant's beliefs about what the user knows affect both what should be presented and the form in which to present it.
- **The Current Context.** The context in which the question was asked provides a basis for both content and form.
- **The Question Intent.** The intention of the user's question influences the focus of the response.

4.1. Representing Expertise

Expertise includes knowing which plan is most appropriate in a given situation, and therefore requires knowing the relationships between alternative plans for accomplishing a goal. For example, there are at least two ways in most mail systems to send a message to a set of

people. One can type each address in turn when prompted for the receiver of the message. One can also create an *alias* which is a named list of addresses that can be reused, and type the alias name at the prompt. The first method is most appropriate when the set occurs only in this instance, or is very small and easy to remember. The second method is more appropriate if over a period of time many messages will be sent to this set of people.

Procedural knowledge can be characterized as a *web* of interrelated goals for doing tasks, plans for accomplishing those goals, steps within plans that are either goals themselves (sub-goals), or functions that describe the actions available in the environment. Choosing the functions that will execute the plan for a goal is a matter of navigating the web, making decisions about what plans, sub-goals and ultimately functions to use.

4.2. The Expert Model and The User Model

The web is the basis of both our Expert Model and User Model. Presumably the former is considerably richer than the latter. The Expert Model is traversed in order to locate relationships between goals, plans and functions. Encoded within it is the knowledge to present choices between plans. For our purposes we view the expert model as a static declarative structure that is searched in order to locate information. Extending, updating and modifying the Expert Model is discussed in section 8.

The User Model is also a web of relationships between goals, plans and functions. It is used both to decide what to present and how to present it. In the example above, if a consultant knows that a user has a very sparse knowledge of sending mail, it may decide to describe how to type the list of addresses rather than create an alias, since the latter will require introducing yet another unknown function. On the other hand, if the consultant thinks the user has a number of inefficient plans[1] for this goal, it might introduce aliasing. The User Model also affects the choice of strategy in answering the user's question. The form of an answer will depend on whether the consultant thinks the user does or does not already know something, or whether it thinks the user has a misconception. Constructing and maintaining a User Model is also

[1]In some contexts our first plan is less efficient than the second. Another less effecient plan is to laboriously retype the message to each addressee.

discussed in section 8.

From this perspective, expertise and complexity of functions can be characterized by the richness of the web, rather than as simple spectra as described by Chin [6]. Although spectra give the illusion of quantifiable criteria, the methods used to develop and validate them are often superficial at best. In our model, functions that would be classified as ''hard'' or ''advanced'' can be better characterized as requiring an understanding of complex interactions in order to navigate the web. Those that are ''simple'' or ''basic'' have more straightforward paths. Similarly, classifications such as ''novice'', ''intermediate'' and ''expert'' are hard to quantify. It our approach they are unnecessary because users are judged by what they know about the current task, rather than how much they know about the entire environment. It is perfectly plausible for a user to have an extensive web of knowledge about a portion of the environment, and almost no expertise about others. Like a spider web, density and strength of paths occur where they are most useful in accomplishing specific tasks.

4.3. Current context

Although the User Model provides a means for choosing the most appropriate plan for a goal, the current context is also relevant. For example, if the user wants to reply to a message sent to a group of users, rather than initiate correspondence to the group, his or her current collection of old mail may contain a message that was addressed to all members of the group. Many mail systems contain a function for replying only to the individual who sent the message and a similar function for automatically ''carbon copying'' the reply to all who received the original message. The choice of how to reply to the group may be mitigated upon whether the collection of old mail contains such a message.

The context also plays an important part in the content of the answer. Rather than using ''canned'' examples, the explanation of what to do can be based on actual objects in the environment. In an electronic mail environment the objects include messages, users and named collections of both.

4.4. Question Intent

The last kind of knowledge that influences tutoring in a procedural environment is the intent of the user's question. This is often termed the *discourse goal*, since it is the goal of constructing some expression to elicit some information. It is distinguished from the computational goal of trying to accomplish some task in the environment. In order to decrease confusion between the two, we refer to the computational goal about which help is sought as the "goal", and the discourse goal as the question intent.

Within a procedural environment it is possible to reason about, and consequently ask questions about, the relationship between goals and plans, goals and functions, plans and functions, goals and other goals, plans and other plans, functions and other functions. The range of question intentions reduces to those in figure 4-1. The utterance identifies a goal or plan (or function) or both, and implies an assumption about their validity. Its form also implies an expected answer. Therefore, in order to reply informatively, a consultant system must provide the information users expect, namely a goal, plan, function or relationship. It must also confirm or deny their assumptions about the validity of the goal, plan or function mentioned in the question.

5. Choosing a Tutoring Strategy

The knowledge described in the previous section provides four basic parameters for choosing a tutoring strategy. These are:

- **A question intent - QI**, that provides an expected discourse focus of a goal, plan and/or function.
- **A computational goal - G**, which is either identified by the user in the question, or is inferred by the consultant from the plan or function specified in the question.
- **A stated plan - S**, which only exists if the user identifies it in the question. If it does exist it may not be the same as a plan in the User Model — the user may have just learned it from someone else.
- **A User Model plan - U**, for the computational goal, which is the plan that the consultant thinks the user has used in the past for accomplishing the goal.
- **A best plan - B**, for the computational goal which is inferred by the consultant from the Expert Model given the current context and the goals, plans and functions in the User Model. It may or may not be the same as S or U.

Question Intent	Question Identifies	Question Assumes	Question Expects in Reply	Intuitively
What plan/function is required to satisfy the goal?	Goal	Goal is possible	Plan or Function	How do I do it?
What goal is satisfied by this function or plan?	Function or Plan	Function or Plan satisfies some Goal	A goal	What does it do?
Does this plan satisfy this goal?	Function or Plan and Goal	Plan or Function satisfies Goal	Confirmation of assumption or explanation	Does it do it?
This plan doesn't work, for this goal, what's wrong?	Function or Plan and Goal	Plan or Function does not satisfy Goal	Confirmation of assumption or explanation	What's wrong with it?
Is there a better way for this goal?	Function or Plan and Goal	Plan or Function is not best way to satisfy goal	Confirmation of assumption or explanation	What's a better way?
How are these Goals, Plans or Functions similar or different?	Pair of Functions, Plans or Goals	Plans, Goals or Functions exist	Relationship between the pair	What are the similarities or differences between them?

Figure 4-1: Information imbedded in the question intention

Each subsection below will describe how these parameters affect the choice of strategy and whether the strategy is used responsively or as enrichment. When the distinction is not important we will refer to both goals and functions as *skills*.

5.1. Introducing

A skill is introduced responsively when the consultant thinks the user has no previous knowledge of it, doesn't know the best plan for the goal, or has a faulty plan for the goal. An exception is when the question intent specifically asks for a better way, in which case the clarify strategy is used. More formally:

```
Introduce responsively when:
```

```
(AND (QI <> get a better plan B than S )
     (OR (U does not exist)
         (AND (U exists)
              (OR (U is invalid)
                  (U <> B) ]
```

For example in the domain of Berkeley Unix Mail, if the user knows nothing about replying to a message and asks "How do I answer a message" (i.e. specifies a goal), the function `Reply` is introduced. Conversely if the user asks "what does `Reply` do?", (i.e., specifies a function) the goal *reply to a message* is introduced.

Introducing as enrichment occurs when the plan stated by the user does work and none of the plans the consultant thinks the user knows is the best plan. As shown later in section 5.4 when the stated plan does not work, the consultant must respond to the question by elucidating why it doesn't. Only then may the consultant provide enrichment and introduce the best plan. More formally:

```
Introduce as enrichment when:

        (AND (U exists)
             (U <> B)
             (S exists)
             (S is not valid))
```

For example if the user asks "To send mail to a group of users, do I type `alias` at the `TO:` prompt?", and the consultant believes this is the only plan the user knows (it is also the plan in U), then the consultant will first elucidate why the plan does not work, and will then introduce as enrichment the best plan, which may not require the alias command at all.

5.2. Reminding

A skill is reminded responsively when the user does not state a plan and the consultant thinks the user already knows the best plan. Reminding is also used when the stated plan is a plan the consultant thinks the user already knows. More formally:

```
Remind responsively when:

        (AND (QI <> get a better plan B than S)
             (OR (AND (S does not exist)
                      (U exists)
                      (U = B))
                 (AND (S exists)
```

```
(U exists)
(U = S))))
```

For example, if the consultant has seen the user reply to messages in the past and the user asks, "How do I answer a message?", (no S is stated), then the user just needs to be reminded about the command; a long introduction is not necessary. Similarly, if S is stated: "Does `reply` let me answer a message?", then a terse affirmative answer is appropriate.

We have not found a reason to use reminding as enrichment. In fact it seems rather pedantic.

5.3. Clarifying

A skill is clarified responsively when the intent of the question is specifically for a better plan for the goal. A plan must be stated, and clarification occurs if it is not the best plan. More formally:

```
Clarify responsively when:

        (AND (QI = get a better plan B than S)
             (S <> B))
```

For example the user asks "Normally, to send mail to a group of users, I just type all the addresses at the `TO:` prompt, is there a better way?" The consultant may decide that it is time to introduce *aliasing*, and will clarify the difference between using a new plan that includes the `alias` function and the user's stated plan[2].

Clarifying as enrichment occurs whenever the question intent is not specifically asking for a better way, but the plan stated is not the best plan. More formally:

```
Clarify as enrichment when:

        (AND (QI <> get a better plan B than S)
             (S exits)
             (OR (U <> B)
                 (S <> B)))
```

For example, the user asks "Can I type more than one address at the `TO:` prompt" and the

---

[2]Note that asking "What's the best way?" is not the same as asking "What's a better way?" The former is actually a form of "How do I do it?"

context suggests that the best way would be to create an alias. First since the plan is the focus of the question, the consultant must responsively introduce or remind about the plan depending on what it thinks the user knows. Only then would it say, "by the way..." and clarify aliasing as enrichment.

5.4. Elucidating

A skill is elucidated responsively when the user states a plan for a goal that the consultant does not think is valid. Five invalidities were presented in figure 3-4, section 3. More formally:

```
Elucidate responsively when:
        (AND (S exists) (S is not valid))
```

For example if the user asks "To send mail to a group of users, do I type `alias` at the `TO:` prompt?", the consultant notices that a precondition to using the `alias` command is that one be at the `Mail>` prompt. The consultant provides a solution: return to the `Mail>` prompt, create the alias, then begin to compose the message.

Elucidating for enrichment occurs when the consultant thinks the user has a plan for the goal that is not valid — (U is not valid). Since the plan in the User Model is never the focus of discourse (unless it is equal to the stated plan, in which case the stated plan is still the focus of the discourse) it can never be elucidated in response. However under some circumstances it may seem opportune to address what the consultant thinks the user knows. This must proceed in a delicate manner since it is based on knowledge the consultant believes rather than knows.

6. The Use of Tutoring Strategies in GENIE

In order to answer a user's question, a consultant must *understand* the question, *analyze* the problem phrased in the question to find the best solution, and *generate* an answer that is informative. GENIE consists of three processes, and Understander, a Plan Analyst and an Explainer, that operate on three knowledge bases, a World Model, and Expert Model and a User Model. Since the focus of this paper is on tutoring strategies, this section will show how the form and content of an answer is developed by the Explainer. The other processes, as will the

knowledge bases will be described briefly.

The Explainer uses the rules in section 5 to choose a strategy and the schema from section 3 to flush out an answer. We have focused on issues of deep structure generation: that is, what to say rather than the surface level wording through which to say it. Consequently, this discussion will describe the directives produced by the Explainer, but will not show in detail how actual English text is produced. Issues pertaining to surface generation are addressed in section 8.

Four example questions will show how seven different answers occur depending on the user's knowledge and the context in which the question was asked. Through the first question we will show how the form of an answer varies depending on the tutoring strategy chosen, while the skill chosen remains the same. The second shows how the answer varies with the skill that is described, while the tutoring strategy remains the same. Finally, through two more questions we show how the intent of the question itself influences both the form and content of the answer.

6.1. Understanding and Analysis in GENIE

The Understander must be able to parse a user's question and identify the components listed in figure 4-1. It must identify the goal, plan or function to which the user is referring. It must determine what the user assumes; for example, that the identified plan does or does not satisfy the identified goal. It must also determine what the user expects as an answer: either a goal, plan or function, or a relationship between them. From this it must identify the intent of the question. Since research by Wilensky [26] and Pollack [21] has been devoted to question understanding, building a robust understander in GENIE is currently a low priority. At the present time, we produce the stated goal, plan or function, and the user assumptions, expectations and question intent by hand, and provide these as input to GENIE.

Given a plan or function and a goal, the Plan Analyst uses the Expert Model to find a "match" between them. A match is defined as confirmation that a plan or function satisfies a goal. Given a function and goal the Plan Analyst confirms or denies that they match. Given a plan and a goal, it tries to find a match. If it is unsuccessful it uses the cases in figure 3-4 to identify the mismatch. Given a plan, the Plan Analyst attempts to locate the goal it satisfies. If

no actual match emerges from a list of candidates, then the goal with the least complex mismatch is chosen, but is marked as a mismatch along with its causes.

Given a goal, the Plan Analyst searches the Expert Model for the "best" plan for the goal using two sets of heuristics. First it tries to choose steps in plans using "world knowledge" such as efficiency and temporality based on the current context or World Model. For example, users normally prefer plans that accomplish goals now rather than later, or that require fewer rather than more steps. As will be seen in section 6.4 in a mail environment the choice may be affected by whether a message, address or alias already exists. The second set of heuristics compares the candidate plans to knowledge in the User Model. The plan whose substeps occur most frequently in the user model is considered the *best plan.*

6.2. Knowledge Bases in GENIE

The Expert Model is a network of the computational goals that can be satisfied in the computing environment. Figure 6-1 shows the structure of this frame-based knowledge representation. Computational goals contain links to alternative plans for satisfying the goal. A plan can be linked to a sub-goal or an ordered sequence of sub-goals that describe how it can be executed, or to a function that executes it directly. Encoded within a computational goal are links that describe the relationship between plans.

Functions describe the operators of the environment. Their representation includes information about the correct syntax of the function, any preconditions and effects, and the actions associated with parameters. Preconditions define a state that must be true before a function can be correctly executed. They may also contain a link to a goal that could satisfy it. Effects encode the actions of functions when applied to the World Model. Currently the World Model is represented as a simple add/delete list that describes possible states in the mail environment. Therefore effects are encoded as directives to add or delete a state from the World Model.

The User Model has the same representation as the Expert Model. It contains a history of what the user has done in past sessions in terms of what goals have been accomplished and what

plans and functions were used to accomplish them. Its goals may contain plans that were attempted, but didn't work, or plans that do not exist in the Expert Model.

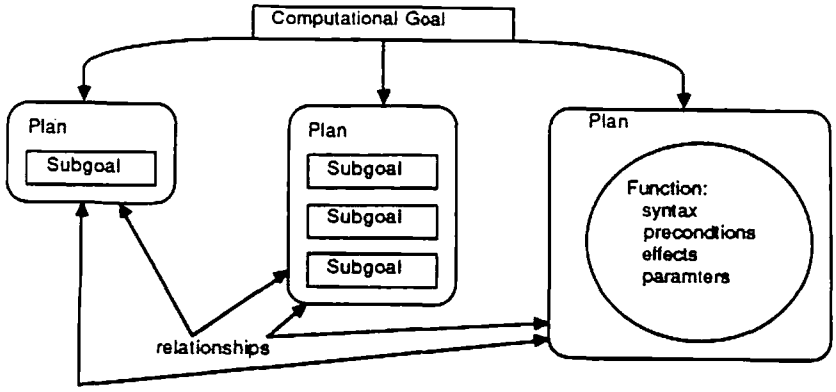

Figure 6-1: GENIE's frames for knowledge representation

6.3. The User Model Affects the Choice of Strategy

We first present an example where the best plan (B) chosen by the Plan Analyst is described by the Explainer through three different tutoring strategies depending on what the user knows. Consider the question "How can I answer a message?". The question identifies a goal (*reply to a message*) and has a question intent (QI) to receive a plan in response. Assume that the World Model contains a message that the user is currently reading that was sent only to the user, not to some group that included the user. Figure 6-2 is a graphic representation of the Expert Model required to answer this question. The World Model dictates that the Plan Analyst choose as the best plan B:

```
Goal: reply.to.message (satisfied by)
   P2: (through steps)
     Goal: reply.now (satisfied by)
        P1: (through steps)
          Goal: reply.only.to.sender (satisfied by)
             P2: (through steps)
               Goal: start.single.reply (satisfied by) ...
               Goal: compose.message (satisfied by) ...
```

Three different tutoring strategies are invoked depending on what is in the User Model. In all three cases we assume that the user has a plan for *compose.message*.

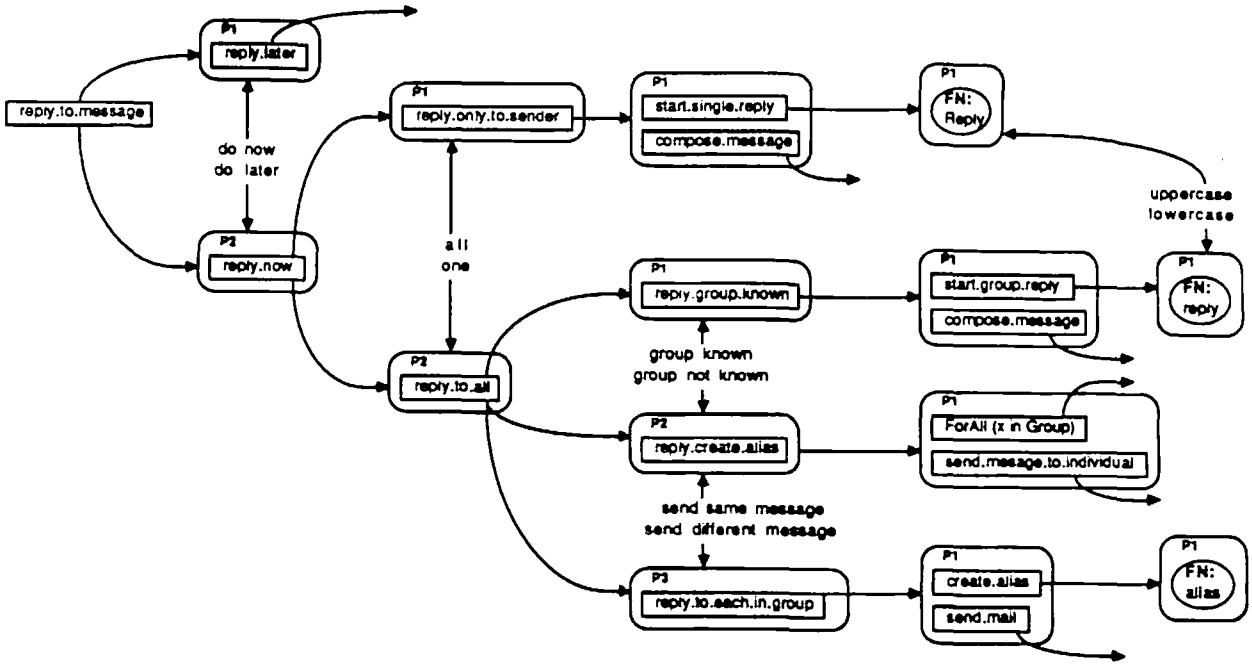

Figure 6-2: GENIE's expert model knowledge of replying to a message

In the first case the User Model contains no plan for the goal, that is GENIE has no knowledge that the user has a plan like B to reply to a message. The Explainer chooses to *Introduce Responsively* because no plan U exists in the User Model for the goal. Figure 6-3 shows botha process trace of the directives that are assembled for producing text and the resulting text.

In the second case the User Model contains a plan U that is identical to B, that is, GENIE believes the user has replied to messages correctly in the past. The Explainer chooses to *Remind Responsively* because U exists and is equal to B. Figure 6-4 presents a portion of the process trace.

In the third case, the User Model contains a plan that does not exist in the Expert Model, but that is valid:

```
Introducing:  reply.to.message
  STATEGOAL:  reply.to.message
    SUMMARIZE:  reply.to.message.P2
      Introducing:  reply.now
        STATEGOAL:  reply.now
        SUMMARIZE:  reply.now.P1
          Introducing:  reply.only.to.sender
            STATEGOAL:  reply.only.to.sender
            SUMMARIZE:  reply.only.to.sender.P1
              Introducing:  start.single.reply
                STATEGOAL:  start.single.reply
                  IntroducingFunction:  Reply
                    PRESENT-SYNTAX: [Reply]
                    DESCRIBE-PRECONDITIONS:
                      (mode = reading.messages)
                    DESCRIBE-EFFECTS:
                      Delete (mode = reading.messages)
                      Add (addressee = current.sender)
                      Add (interrupt.mode = compose.message)
                  Reminding:  compose.message
                    ......
        EXAMPLE: (FN: Reply, Compose.message)

In order to reply to a message it is assumed you want to reply right
away.  In order to reply right away it is assumed you want to reply
only to the sender.  To do this, you must indicate you wish to reply
and compose a message.  You can indicate you wish to reply by using
the command 'Reply'.

The syntax is: Reply

To use this command you must be in read mode. The
command removes you from read mode, makes the addressee of your
message the current sender and temporarily puts you in write mode.

To compose a message just type your message and end with <esc>.
For example,
        Type the command: Reply
        Then just type your message and end with <esc>.
```

Figure 6-3: *Reply to a message* when User Model contains no plan for the Goal

```
Reminding: reply.to.message
  STATEGOAL:  reply.to.message
  SUMMARIZE: P2
  EXAMPLE: (FN: Reply, Compose.message)
```

Figure 6-4: *Reply to a message* when User Model contains a plan for the goal

```
Goal: reply.now (satisfied by)
   P: (through steps)
    Goal: save.message (satisfied by) ....
    Goal: leave.read.mode (satisfied by) ...
    Goal: send.message (satisfied by) ...
```

The Explainer chooses to *Clarify Responsively* because the user has a valid plan for the goal, but that plan isn't best. Figure 6-5 presents a portion of the process trace.

```
Reminding: reply.to.message
  STATEGOAL:  reply.to.message
  SUMMARIZE: U
  EXAMPLE: (FN: Reply, Compose.message)
Clarifying: reply.to.message
  SUMMARIZE: B
  COMPARE_STEPS(send.message,reply.only.to.sender)
    SHOW-RELATIONSHIP: (initiate addressee) (sender is addressee)
    Introduce:  Reply
```

Figure 6-5: *Reply to a message* when user model does not contain best plan

6.4. Content Varies Within A Strategy

We now show how the Plan Analyst's choice of a plan for a goal varies the content, but not the form of the answer. Consider the question "To reply to a group of users I reply to each individually — is there a better way?". The question identifies a goal — *reply to all* and a plan (S) — *Forall (x in group) send.message.to.individual.* It has a question intent (QI) to get the best plan. Assume that the User Model contains the goals *compose.message* and *send.mail*, but does not contain *create.alias* or *start.group.reply*.

For the first case, assume the World Model contains a message that was addressed to a group. The Plan Analyst determines that *reply.group.known* is the best plan since the group is known and it is less work than the stated plan S. Since QI = get a better plan B than S, and S <> B, the Explainer chooses to *Clarify Responsively.* Figure 6-6 presents a portion of the process trace.

In the second case, assume the World Model contains a message sent only to the user. The Plan Analyst determines that the best plan is *reply.create.alias* since no group message exists

```
Clarifying:  reply.to.group
  STATE-BETTER-WAY-EXISTS:  reply.to.group
  SUMMARIZE: U
  SUMMARIZE: B
  SHOW-RELATIONSHIP: (group known) (group not known)
  COMPARE-STEPS:  (start.group.reply, ForAll(x in group))
    Introducing:  start.group.reply ...
  COMPARE-STEPS:  (compose.message, send.message.to.indivdual)
    Reminding:  compose.message ...
  EXAMPLE: B
```

Figure 6-6: Clarifying that the best plan is *reply group known*

and it is less work than the user's plan. Since QI = get a better plan B than S, and S <> B, the Explainer still chooses to *Clarify Responsively*. Figure 6-7 presents a portion of the process trace.

```
Clarifying:  reply.to.group
  STATE-BETTER-WAY-EXISTS:  reply.to.group
  SUMMARIZE: U
  SUMMARIZE: B
  SHOW-RELATIONSHIP: (send same message) (send different message)
  COMPARE-STEPS:  (reply.create.alias, ForAll (x in group))
    Introducing:  start.group.reply ...
  COMPARE-STEPS:  (send.mail, send.message.to.indivdual)
    Reminding:  send.mail ...
  EXAMPLE: B
```

Figure 6-7: Clarifying that the best plan is *reply create alias*

6.5. Question Intent Influences Form and Content

The two questions in the previous sections generated five different answers even though they were asking essentially the same thing — "How to reply". We believe this illustrates the fundamental difference between generated answers and canned text. The former provides flexibility in answers the latter cannot. We illustrate this point further with two more questions that also ask about *reply.to.message*. Both questions require *Elucidating Misconceptions*, but the form and content differs because the stated plans fail in different ways.

In the first case, consider the question "I'm trying to reply to a group of users using `reply`

but I only seem to be able to reply to the sender of the message. Why?'' The stated goal is *reply.to.all* and the plan implied in the question is *(*`FN: reply`*, compose.message)*. Assume that the message in the World Model was only sent to the user, and the user knows about the `alias` function. The fault lies in the fact that there are no other group members in the message header, causing a precondition of the function `reply` to be violated. The Plan Analyst finds the fault and suggests using the *reply.create.alias* plan. Given that S is invalid, the Explainer chooses to *Elucidate Responsively* and expands on a missing precondition. Figure 6-8 presents a portion of the process trace.

```
Elucidating:  reply.to.all
  STATE-BROKEN-PLAN:  reply.group.known.P1
   ProblemIn:  MissingPrecondition (exists group)
               IN start.group.reply
   Fix:  reply.create.alias.P1
     Reminding: reply.create.alias
```

Figure 6-8: Elucidating when a precondition is missing

In the second case consider the question ''I'm using `reply` to reply to the sender of a message, but seem to send mail to everyone else to whom the message was addressed. Why?'' The stated goal in this case is *reply.only.to.sender* and the implied plan is *(*`FN: reply`*, compose.message)* with the sub-goal *start.single.reply* being satisfied by the function `reply`. Assume that the message in the World Model was sent to a group of users. Given the knowledge in figure 6-2 the Plan Analyst notices that `FN: reply` seems to be an extra step while `FN: Reply` is a missing step[3]. From the figure these steps are related both at the level of the function — by the case of the first letter and two levels up in the distinction between *reply.to.all* and *reply.only.to.sender*. Since S is again invalid, the Explainer chooses to Elucidate Responsively expanding on an extra step related to a missing one. Figure 6-9 provides a portion of the process trace.

[3]For the Unix uninitiated character case matters, so `Reply` and `reply` are indeed two different functions. We admit this is grossly user unfriendly and should not occur in the first place, but the example is too good to resist.

```
Elucidating: reply.only.to.sender
  STATE-PLAN-BROKEN: reply.only.to.sender.P1
   ProblemIn: ExtraMatchesMissingStep
              Extra: start.group.reply
          Missing: start.single.reply
   DescribeMatch: Extra: FN: Reply (uppercase)
              Missing: FN: reply (lowercase)

              Extra: reply.to.all (all)
              Missing: reply.only.to.sender (one)
              Introducing: reply.only.to.sender
```

Figure 6-9: Elucidating when a missing step matches an extra step

7. Related Work

The represention of our Expert and User Models is based on work by Goldstein [9], Genesereth [8] and Clancey [7]. Goldstein introduced the notion of a network, a *genetic graph* of knowledge that represents stages of development of a student's understanding of a domain. Rather than overwhelm the beginner with an optimal method, the genetic graph provides a basis for choosing what to say. Its structure however, is based on degrees of student development which, as we argued earlier, is very hard to quantify and evaluate. Since our web is task connected as well as knowledge connected we should have less difficulty approaching a problem from a novel perspective. Genesereth's *dependency graph* of goals and plans suggests that this is the case. Genesereth focused primarily on diagnostics. He was therefore not concerned with encoding semantic information about the relationships between plans, which we see as crucial to explanation. Clancey first articulated the need for the separation of domain expertise from tutoring knowledge. Although he implicitly captures goal/plan knowledge in his rules, unlike GENIE his tutors can not reason abstractly about relations between goals and plans. Finally, none of these researchers seriously addressed the problem of generating a coherent response.

Constructing and maintaining a User Model has proven to be a difficult task in building Intelligent Tutoring Systems. One problem is reliably determining what the user knows. A second is choosing how to represent the user's knowledge. A third is updating the User Model dynamically. Although we will not try to justify it here, we believe that the web paradigm of

goals, plans and functions offers insight into solutions to the first two problems. The third, like updating the Expert Model, falls within the domain of knowledge acquisition and learning.

Research by Brownand Burton [4] and by Sleeman and Smith [24] illustrate the difficulty in determining what the student knows in unguided settings. The difficulty lies in diagnosing what the student doesn't know based on unexpected behavior. Since we do not expect any behavior in the first place, we avoid this problem by waiting for the user to notice that something is wrong. Our user model is a reference point for choosing the form and content of the consulting dialogue, but is rarely the focus. In only one instance, elucidating as enrichment, does GENIE expect to find a ''buggy plan'' in the User Model. This case is a byproduct of our representation rather than its focus and therefore does not play a significant role in how GENIE works.

We address the second problem by restricting our representation to knowledge of *what the user knows how to do* rather than the larger domain of *what the user knows*. For example we do not attempt to draw analogies to knowledge the user might have of things outside the environment. The second problem is also addressed by the nature of the domain. Since the user is engaged in using an environment, it should be possible to build a system to monitor his or her actions. For example, our Marvel software development environment [11] monitors all commands entered by the programmer. Inferences could therefore be drawn on whether particular goals have been attempted and successfully accomplished and through what plans or functions. Selker [23] and Quilici [22] among others have demonstrated how such plan recognition is possible and useful for developing a User Model.

Other research on User Modeling and natural language generation falls into two categories: either stereotypes are used to represent large categories of users and answers are geared to a category, or explicit beliefs and goals of the user are represented and reasoned about to determine an answer. Chin's work [6] on the UC system [26] best exemplifies the stereotype approach and is most relevant to our system. Chin's system provides help within the Unix environment based on a dual set of stereotypes. Users are classified as novice, intermediate, or expert and functions are classified as easy, intermediate[4], and hard. The system uses rules

[4]Chin actually uses two levels of intermediate.

stating how much detail to provide about the different classes of functions depending on which class the user falls into. Our approach is in direct opposition to Chin's: it is based on the assumption that knowledge cannot be quantified in discrete chunks and users are unlikely to learn and progress from one neat division to another. Rather, they are likely to learn functions based on tasks they have performed and this will vary substantially depending on the tasks.

A second body of work by Allen and Perrault [1], Pollack [21] and Appelt [2], uses detailed information about user beliefs and plans in combination with a formal reasoning system to determine what to include in an answer. Because they have relied on such detailed formal models in combination with a theorem prover, they have tended to operate in limited, well constrained domains, producing shorter responses than those at which we aim. Pollack is an exception, although she has focused more on the representation required to produce helpful responses for plan invalidities and less on the generation of the responses themselves. Like Pollack, we also represent details about beliefs, plans and goals. However, we combine the representation of beliefs with the use of tutoring strategies rather than a theorem prover, to allow us to produce more complex responses as part of a less well constrained domain.

8. Summary and Conclusions

The great hope of educational computing, that computer technology can deliver unprecedented individualized instruction, remains to be fulfilled. This paper describes a step in that direction, through the development of GENIE, an automated consultant for interactive environments. We have demonstrated how the form and content of an answer are chosen based on the current needs and knowledge of the user. Our knowledge representations and the algorithms and heuristics used to traverse them allow us to abandon spectra of user expertise and functional difficulty. We therefore circumvent the need for a broad sequential curriculum and have much greater flexibility in when, what and how to present skills to users.

Our demonstration system GENIE is by no means as sensitive as a skilled human tutor. At the present time, it cannot handle certain aspects of context, one cannot ask questions in any natural way, the knowledge bases must be updated by hand, and the answer that is generated could stand stylistic improvement. These are all primarily implementation issues, and in the

paragraphs that follow we address each of them in turn.

One major goal of this work was to answer questions within the current context. We provide the context in symbolic form as input to GENIE. Our assumption is that it can be derived by a subsystem that observes users' actions. In order to answer "what if" questions where the user presents a hypothetical context, a different subsystem would be required that allows users to construct a context in some friendly way such as natural language. Furthermore in human discourse a question is rarely answered fully through a single utterance. Often a user's initial question spawns other questions that may be combinations of question types and require a mixture of answering strategies. Here too, a different subsystem, that monitors a discourse history, would contribute to the construction of the current context.

At the present time we make the simplifying assumption that the user either knows or does not know about things in the domain. By definition, if something exists in the User Model, even if it is wrong, then it is known. If it does not exist, then it is not known. In practice this assumption is inadequate first because knowledge is not binary, and secondly because tutoring can occur across a broader continuum. Rather than fall for the seduction of simple categories of levels of mastery, we would like to evaluate and record knowledge in a less quantified manner. This in turn will affect how GENIE's heuristics and strategies must be changed.

We have thus far avoided consideration of a "front end" for asking questions. Our justification has been that our interests lie in generation rather than understanding of language. We are now at a stage where we have a clearer picture of what information we must get from the user and are in a better position to design an understanding component for demonstration purposes. We have begun work on a simple Augmented Transition Network [30] parser that will be able to extract the information described in figure 4-1.

Clearly, all possible goals and plans for those goals cannot be known when a computing environment is set loose on users. Furthermore, if the environment is extensible, even the relationships between functions cannot be fixed. We view the problem of extending, updating and modifying both the Expert and User Models as one of knowledge acquisition and learning and therefore not within the immediate scope of this research. Work on machine learning by

Lebowitz [13] has influenced the design of our knowledge representations, and we are confident that both can be updated automatically in the future.

Finally, the current version of GENIE relies on textual templates to produce actual text. We have discovered that the prose generated is stylistically stilted, referentially awkward, often redundant and therefore inadequate. We are currently exploring the use of a functional unification grammar [12; 19] that should allow us to produce more graceful text.

To summarize, GENIE is an automated consultant that attempts to meet the contextual needs of the user by basing its answers on the user's current task, and those tasks the user has successfully accomplished in the past. Our current plans are to develop a front end for demonstration and testing purposes, to make use of a functional unification grammar to produce better text, and to expand both the analysis and explanation facilities to handle a wider variety of consulting situations. Through these projects we hope to shed more light on the complex tutoring that occurs during consulting in interactive environments.

References

1. Allen, J. F. and Perrault, C. R. "Analyzing Intention in Utterances". *Artificial Intelligence 15*, 1 (1980), pages 143-178.

2. Appelt, D. E.. *Planning Natural Language Utterances*. Cambridge University Press, Cambridge, England, 1985.

3. Borenstein, N.S. *The design and evaluation of on-line help systems*. Ph.D. Th., Carnegie Mellon University, April 1985.

4. Brown, J.S. and R.R. Burton. "Diagnostic Models for Procedural Bugs in Mathematics". *Cognitive Science 2* (June 1978), 155-192.

5. Brown, J.S. and K. VanLehn. "Repair theory: A Generative theory of bugs in procedural skills". *Cognitive Science 4* (1980), 379-415.

6. Chin, D. N. *Intelligent Agents as a Basis for Natural Language Interfaces*. Ph.D. Th., University of California, Berkeley, 1988.

7. Clancey, W.J. Tutoring rules for guiding a case method dialogue. In *Intelligent Tutoring Systems*, Academic Press, London, 1982, pp. 201-225.

8. Genesereth, M.R. The role of plans. In *Intelligent Tutoring Systems*, Academic Press, London, 1982, pp. 137-155.

9. Goldstein, I.R. A genetic graph model for tutoring. In *Intelligent Tutoring Systems*, Academic Press, London, 1982, pp. .

10. Joshi A., B. Webber and R. Weischedel. Living up to expectations: Computing expert responses. Proceedings of AAAI-84, American Association of Artificial Intelligence, 1984, pp. 169 - 175.

11. Kaiser G. E., P. H. Feiler and S. S. Popovich. "Intelligent Assistance for Software Development and Maintenance". *IEEE Software* (May 1988), 40-49.

12. Kay, M. Functional Grammar. Proceedings of the 5th meeting of the Berkeley Linguistics Society, 1979.

13. Lebowitz, M. An experiment in intelligent information systems: RESEARCHER. In R. Davies, Ed., *Intelligent Library and Information Systems*, Ellis Horwood, London, 1986.

14. *LISP Language Reference (M-Z)*. Hewlett-Packard Company, Fort Collings, CO, 1986.

15. *Logo: Programming with Turtle Graphics*. International Business Machines Corporation, Boca Raton, Florida 33432, 1983.

16. Magers, C. S. An experiemntal evaluation of On-line HELP for non-programmers. CHI'83 Proceedings, 1983, pp. 277-281.

17. McCoy, K. F. The ROMPER System: Responding to Object-Related Misconceptions Using Perspective. Proceedings of the 24th Annual Meeting of the ACL, Association of Computational Linguistics, New York City, New York, June, 1986.

18. McKeown, K.R.. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Studies in Natural Language Processing. Cambridge University Press, Cambridge, England, 1985.

19. McKeown, K. R. and C. L. Paris. Functional Unification Grammar Revisited. Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Stanford, Ca., July, 1987.

20. Paris, C. L. *The Use of Explicit User Models in Text Generation: Tailoring to a User's Level of Expertise.* Ph.D. Th., Columbia University, 1987.

21. Pollack, M. *Inferring domain plans in question-answering.* Ph.D. Th., Moore School, University of Pennsylvania, May 1986.

22. Quilici, A.E., G. Dyer and M. Flowers. Understanding and advice giving in AQUA. UCLA Artificial Intelligence Laboratory, Los Angeles, CA, 1985.

23. Selker. T. Cognitive Adaptive Computer Help - A Research Overview. T.J. Watson Research Center, IBM, T.J. Watson Research Center, Yorktown Heights, N.Y., 1988.

24. Sleeman, D.H. and M.J. Smith. "Modelling student's problem solving". *AI Journal* (1981), 171-187.

25. Steele, G. L.. *Common Lisp, the language.* Digital Press, Bedford, MA, 1984.

26. Wilensky, R., Y. Arens, and D. Chin. "Talking to Unix in english:An overview of UC". *Communications of the ACM 27*, 6 (June 1984), 574-593.

27. Winston, P.A. and B.K. Horn. *Lisp, second edition.* Addison Wesley, Reading, MA, 1984.

28. Wolz, U. Analyzing user plans to produce informative responses by a programmer's consultant. CUCS-218-85, Department of Computer Science, Columbia University, New York, NY, 1985.

29. Wolz, U. and G.E. Kaiser. A Discourse-Based Consultant for Interactive Environments. Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications, 1988, pp. 28 - 33.

30. Woods, W. An Experimental Parsing System for Transition Network Grammars. In *Natural Language Processing*, Rustin, R., Ed., Algorithmics Press, New York, 1973.