THE GRADUAL ENCROACHMENT OF ARTIFICIAL INTELLIGENCE

Elaine Rich

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-05 February 1984

ABSTRACT

Many tasks that require ingelligence and that are now being done by people can be partially transferred to computers as individual parts of the problems become well-enough understood. This *gradual encroachment* of artificial intelligence (A.I.) means that A.I. can be of practical significance now, even though some problems remain unsolved. Examples of areas in which this is occurring will be discussed.

1. Introduction

The goal of artificial intelligence is to produce programs that rival human performance at a wide variety of so-called "intelligent" tasks. Although in some areas, that goal is close to being attained (see, for example, some of the current work in *expert systems*[1]), in others it remains a good deal farther away. But even in those more difficult areas, in which we do not yet know how to construct programs that perform well on their own, we can use what we do know to build programs that can provide significant assistance to people who are trying to perform a task. In other words, we do not have to wait until we have a complete understanding of an area before A.I. can contribute to it. Long before that, A.I. may contribute by taking some, albeit not all, of the load off the person responsible for the task. In fact, rather than a sudden take-over by a program of a task previously performed autonomously by a person, we may expect to see a gradual shifting of the burden from the person to the machine.

This approach to the computer solution of difficult problems can be described as the *gradual encroachment* of artificial intelligence. It is not a new idea. For example, a similar technique, called *incremental simulation* was used in the design of a speech understanding system[2] when some of the component processes were not initially well enough understood to be automated. Those processes were simulated by people, who communicated with the rest of the system in the same way in which the other system modules did. Gradually, the simulated modules were transformed into code as the techniques that they were using became apparent.

There are two important ways in which programs that have been encroached upon by artificial intelligence differ from their more conventional counterparts. The most important is their use of a large amount of structured knowledge about their domains. The second is their exploitation of *heuristic search* (search that is guided by knowledge) as a technique for solving problems for which a direct mechanism cannot be found[3]. The importance of these two things suggests one reason why the encroachment of A.I. often is gradual. As the amount of knowledge that a program possesses increases, and as the accuracy of its search-controlling heuristics (which are themselves a kind of knowledge) grows, the performance of the program itself usually improves. Since the build-up of the necessary knowledge is usually gradual, so too is the encroachment of the knowledge-based program into its problem arena.

In the remainder of this paper, several domains in which the encroachment of artificial intelligence is occurring will be discussed and the ways in which it is happening will be described.

2. Programming

The most fundamental task in which people interact with computers is programming, since without that, the other things would not exist. Eventually, we would like to have access to programs that can themselves do much of the programming that is now done by people.

2.1. Helping People Program

Automatic programming has as its goal the construction of exactly such programs, which can transform a set of specifications into a program meeting those specs. Although some progress has been made in this area, the field is still in its infancy[4] and the bulk of programs must still be written by people. Yet there are many ways in which computers can assist people in their programming efforts.

Compilers for high-level languages enable people to write programs using constructs that are closer to the natural structures of their problem domains than are the primitive operations that the target machine can execute. The compilers translate those higher-level constructs into the primitive ones. This can typically be done rather straight-forwardly, although as we move toward optimizing compilers that attempt to produce more and more efficient target code, we begin to see the increased use of the two main tools of artificial intelligence, search (e.g. to find patterns of use of variables) and knowledge (e.g. to store special-purpose, highly-tuned code segments).

Very high-level languages for specific domains go even farther in allowing people to specify abstract problem solutions and leave the details to the machine, which must exploit its database of domain-specific knowledge to complete the task. One widely used system that is an excellent example of this is MACSYMA, a program developed by the MATHLAB group at MIT, that performs symbolic mathematics. For example, the following are legal MACSYMA commands:

```
INTEGRATE(SIN(X)**3,X);
```

which computes the indefinite integral with respect to x.

```
DETERMINANT(M);
```

which computes the determinant of the matrix M using a method similar to Gaussian elimination.

MACSYMA is one of the earliest examples of the class of programs now known as *expert systems*, since these programs perform tasks that were previously done only by human experts. The key to the success of these programs is the knowledge that they have of their task domain. Although MACSYMA knows a lot about symbolic mathematics, it cannot do much on its own. Its importance lies in the fact that it can be used as an intelligent aide to a human mathematician in solving hard problems.

Another way in which artificial intelligence has begun to encroach upon the programming process is illustrated by the Programmer's Apprentice[5]. The Programmer's Apprentice knows about general programming structures rather than about specific constructs geared to a particular problem domain. This knowledge about programming structures is contained in a library of *plans*, which can be combined to form complete programs. An example of the kind of thing that is described as a plan is the "trailing pointer list enumeration loop." This plan contains code that moves a pointer down a list, starting at the second element, and looks for a desired value, while at the same time, trailing a second pointer one element behind the first one. By using its plan library, the Programmer's Apprentice can help the programmer construct a new program. It can also help the programmer edit an existing program without causing undesirable side effects. It can do that because it knows how pieces of the program relate to each other.

Simple spelling correctors (such as the Do What I Mean (DWIM) facility of INTERLISP) use a small amount of knowledge about programming language syntax and current context to correct user errors. Verification systems, timers, and portability checkers provide programmers with additional sources of assistance. As the amount of knowledge contained in programs such as these grows, the amount of responsibility they can assume will also increase until all that is left for the person to do is to provide an accurate set of specifications (itself no simple task).

2.2. Helping People Learn to Program

Since we are still a long way from the destination point of automatic programming, there is yet another way in which artificial intelligence can begin encroaching on programming. It can be used to build systems that help people learn to program. Several efforts are currently underway to do this (for example, MENO[6].)

To build an effective programming tutor, it is necessary to be able to analyze a student's program and to detect three classes of things:

- Plans, such as the ones in the Programmer's Apprentice, that implement programming concepts.
- Bugs, which are places where the student's program differs from a correct program.
- Misconceptions, which are flaws in the student's understanding that are made obvious by the observed bugs.

Although such a tutor constitutes a form of computer-aided instruction (CAI), its structure needs to be radically different from that of the traditional frame-based CAI system, in which the order in which material will be presented is determined in advance of any particular tutoring session. Instead, the programming tutor must be structured as a knowledge-based reasoning program. In the rest of this section, the LISP tutor we have built will be described, since it illustrates how programming knowledge and search can be combined into a rudimentary programming tutor. Then the limitations of this approach will be described and some suggestions for how further encroachment into this area may arise will be presented.

The LISP tutor works by assigning the student a program to write. The tutor has been given an abstract outline for solving the problem that the program must solve. From that outline, it uses a set of programming plans to synthesize one or more programs that solve the problem. From then on, the tutor relies heavily on a matching process to guide its behavior. First the tutor compares the student's program to its own programs and selects the one of its programs that most closely matches the student's code. For many programming languages, the first step toward doing this is, of course, to parse the student's program. The MENO system mentioned above, which is a tutor for PASCAL, does this. But because of the syntax of LISP, the LISP tutor can skip this step and go immediatedly to the next one at which the use of specific programming plans is identified. For example, the tutor might have synthesized both an iterative and recursive solution to the problem and it must choose to continue with whichever approach the student used. This one program that is known to be a solution to the assigned problem gives the tutor somewhere from which to start in analyzing the student's program. The next thing that the tutor does is to try to transform its program so that it is as similar to the student's program as possible. It does this by comparing (matching) its program to the student's, and using the differences it finds to select, from a set of meaning-preserving program-transformation rules, those that should be applied to its program. Some examples of program transformation rules are shown in Figure 2-1.

```
(1)   (PLUS A B) → (PLUS B A)

(2)   (PLUS A 1) → (ADD1 A)

(3)   (NOT (AND A B)) → (OR (NOT A) (NOT B))

(4)   (< A B) → (NOT (OR (> A B) (= A B)))

(5)   (AND (A) (B)) → (AND (B) (A))

(6)   (COND (PRED ACTIONS1) (T ACTIONS2)) →
        (COND ((NOT PRED) ACTIONS2) (T ACTIONS1))
```

Figure 2-1: Some Program Transformation Rules

Of course, since the student's program may not be correct, an exact match between the student's and the tutor's programs cannot usually be obtained. But differences that cannot be eliminated pinpoint bugs in the student's program. So when the tutor has no more meaning-preserving program-transformation rules that appear to reduce the difference between its program and the student's, it begins to apply another set of rules, which introduce bugs. If, by doing so, a match between the tutor's program and the student's can be achieved, then the tutor will have identified the bugs in the student's code. The final step is then for the tutor to determine what underlying misconceptions led the student to make the errors he made. Appropriate advice can then be given to the student.

Figure 2-2 shows an example of what the tutor can do. The student has been asked to write a program that accepts as its input a list of positive numbers and returns as its value the maximum element in the list. The student has written an iterative program to do this. The first program in the figure is one synthesized by the tutor, which also synthesized other programs, including a recursive one. But the tutor selected this one because it most closely matched the student's code. The program iterates through the list, keeping the biggest number so far in ?AC. The second program was written by the student. It contains two bugs. AC should have been initialized to zero rather than NIL. There also must be an explicit RETURN statement or the value NIL will be returned from the PROG. The tutor notices some semantically irrelevant differences between its program and the student's. It applies transformations to its program to reduce these differences. The third program is the tutor's program after those meaning-preserving transformations have been applied to convert COND into OR, merge two SETQs into one, and convert > into NOT < and NOT =. The differences between the transformed version of the tutor's program and the student's program are now due entirely to the bugs in the student's program. These differences can now be found by the bug detection rules and the bugs are easily identified.

Example: One synthesized Program for the MAX-EL Task:

```
(DEFUN ?FUNCTION (?LIST)
    (PROG (?AC ?LIST-LEFT)
          (SETQ ?AC 0)
          (SETQ ?LIST-LEFT ?LIST)
     ?LOOP (COND ((NULL ?LIST-LEFT) (RETURN ?AC))
                 (T NIL))
          (COND ((> (CAR ?LIST-LEFT) ?AC)
                      (SETQ ?AC (CAR ?LIST-LEFT)))
                (T NIL))
          (SETQ ?LIST-LEFT (CDR ?LIST-LEFT))
          (GO ?LOOP)))
```

Example Student Program:

```
(DEFUN MAX (LIST)
  (PROG (AC LIST-LEFT)
    (SETQ AC NIL LIST-LEFT LIST)     ;;;BUG: AC INIT TO NIL
   CONTINUE
    (OR LIST-LEFT AC)         ;;;BUG: SHOULD BE (RETURN AC)
    (OR (OR (< (CAR LIST-LEFT) AC) (= (CAR LIST-LEFT) AC))
        (SETQ AC (CAR LIST-LEFT)))
    (SETQ LIST-LEFT (CDR LIST-LEFT))
    (GO CONTINUE)))
```

Tutor's Program after Transforms:

```
(DEFUN ?FUNCTION (?LIST)
   (PROG (?AC ?LIST-LEFT)
         (SETQ ?AC 0 ?LIST-LEFT ?LIST)
    ?LOOP (OR ?LIST-LEFT (RETURN ?AC))
         (OR (OR (< (CAR ?LIST-LEFT) ?AC)
                 (= (CAR ?LIST-LEFT) ?AC))
             (SETQ ?AC (CAR ?LIST-LEFT)))
         (SETQ ?LIST-LEFT (CDR ?LIST-LEFT))
         (GO ?LOOP)))
```

Figure 2-2: An Example of the Programming Tutor

The LISP tutor illustrates the point that the power of an intelligent system comes from its use of a substantial knowledge base coupled with a judicious use of heuristic search. The tutor's knowledge of the problem to be solved is contained in its outline for the problem's solution. Its knowledge of programming is contained in four places: its plans for generating code from outlines, its meaning-preserving program-transformation rules, its bug-introduction rules, and its mappings between observable bugs and underlying misconceptions about programming. In addition, the tutor exploits knowledge about individual students and about typical types of students (in the form of *stereotypes*[7]) in order to constrain its search for appropriate transformation rules and to enable it to give advice that is tailored to the needs of each individual student. All of this knowledge enables the search-based matching process employed by the tutor to focus the attention of the tutor on exactly the part of the student's program where attention is needed -- the bugs.

The LISP tutor as just described works, but its effectiveness is limited to small, fairly simple programs. The reason for this is that without more control over the search process, the size of the space becomes unreasonably large if the program that is being analyzed is of even medium size. To correct this, and to enable the tutor to expand its scope and to encroach farther into the tutoring domain, several modifications are necessary. Some of these modifications just require that ideas that are already present in the tutor be expanded. For example, plans need to play a much bigger role and the catalogue of known plans needs to be expanded. But some dramatic changes to the structure of the tutor are also necessary. Two of these will be discussed briefly.

The LISP tutor has several types of rules that are used in different parts of its operation. To keep the system manageable, some degree of modularity is necessary. This can be achieved by dividing the system into a set of quasi-independent knowledge sources that communicate via a shared data structure called a *blackboard*[8]. Each of these knowledge sources can then contain its own working data structures independent of the others. When one knowledge source is operating, the fact that a large number of rules exist in other knowledge sources is of no concern to it. Its matcher can and will ignore them, thus limiting its space of possible matches to a manageable size. This organizational structure is important to the gradual encroachment of A.I. As the capabilities of a system are increased, it is not necessary to create a dinosaur. Instead we grow a bee hive.

Another important modification to be made to the tutor's structure is to incorporate the idea of top-down problem decomposition. There are at least two ways this needs to be done. The first is to enable the tutor to analyze a large program into smaller subprograms that have restrictied interactions with each other. (Of course, this will only be possible if the original program possesses such a structure, which it should.) Then the detailed matching operations described above need only be applied to reasonable sized units at a time. The second thing that needs to be done is to separate the search for probable program structures from the proof of the existence of those structures. Heuristic rules that outline program transformations can suggest correspondences between the tutor's program and the student's. A program verification system can then be used to reason precisely and confirm (or deny) the suggested correspondence. For example, consider transformation (5) of Figure 2-1. That transformation is meaning-preserving provided that there are no side effects in either A or B. Since functions normally do not have side effects, it is a good transformation to try, but it is not always correct.

Current work on the design of programming tutors is attempting to build these capabilities into tutors that will then be able to go even further than the current ones do toward helping people learn to program.

3. Using Traditional Programs

Computer systems that perform straightforward tasks such as document formatting now exist in abundance. The bulk of the problem solving that must be done to achieve a user's goal (e.g. a properly formatted paper) is done by the user when (s)he selects the commands to be given to the system. Some systems offer the user rudimentary assistance in choosing those commands by providing on-line help facilities that access stored text via a predefined set of keywords. Ultimately one might envision a system that does not use commands at all but instead goes directly from an abstract statement of the user's goal to a final output. Such a system would have to be able to perform all of the problem solving now done by human users. Although we do not yet know how to build such a stand-alone system, we can begin to move in that direction by retaining the requirement that users write straightforward commands, but, at the same time, providing them some assistance in writing those commands. Notice the obvious analogy here to the programming domain discussed above.

The encroachment of artificial intelligence into this arena can proceed in steps. The first is to leave the user with the responsibility for deciding when (s)he needs help and to build a system that can respond intelligently to such requests for assistance. The next step is to build a system that itself takes on some of the responsibility for deciding when help is needed. Such a system would volunteer advice and assistance without being asked. Both of these steps will be discussed below, the first in more detail than the second since it is closer to being possible now.

3.1. Answering Questions

The idea of an automated help facility that answers users' questions is not new. (For a survey, see[9].) Simple keyword-based systems are widely available. To use one of these systems, the user must already have solved a great deal of his or her problem, since it is necessary to know the name of the program or command about which help is being solicited. So one can say, for example,

```
HELP FTP
```

The response to this command will be a paragraph of information about how to use the file transfer program FTP. But FTP is hardly an obvious name. A user who is really in need of help needs to be able to say

```
How can I copy a file from one machine to another on the
network?
```

The goal of current research in automated help facilities is to bring us closer to the point at which a user will be able to ask this latter question.

A complete interactive help system must contain all of the components shown in Figure 3-1 if it is to be able to handle dialogues such as the following:

User -	I need to delete 2 of my files.
System -	Use the DELETE command.

User - I tried that.

System - Have you checked the protection on the files?

To participate in this dialogue, the system had to: recognize the indirect question contained in the user's first statement; find an answer to that question; recognize the user's second statement that DELETE didn't work; figure out why it did not; and suggest the next action that the user should take in appropriate terms (it believes that this user understands how the protection mechanism works and just needs to be reminded). Although no current system possesses all of these pieces, work is being done on several of them.

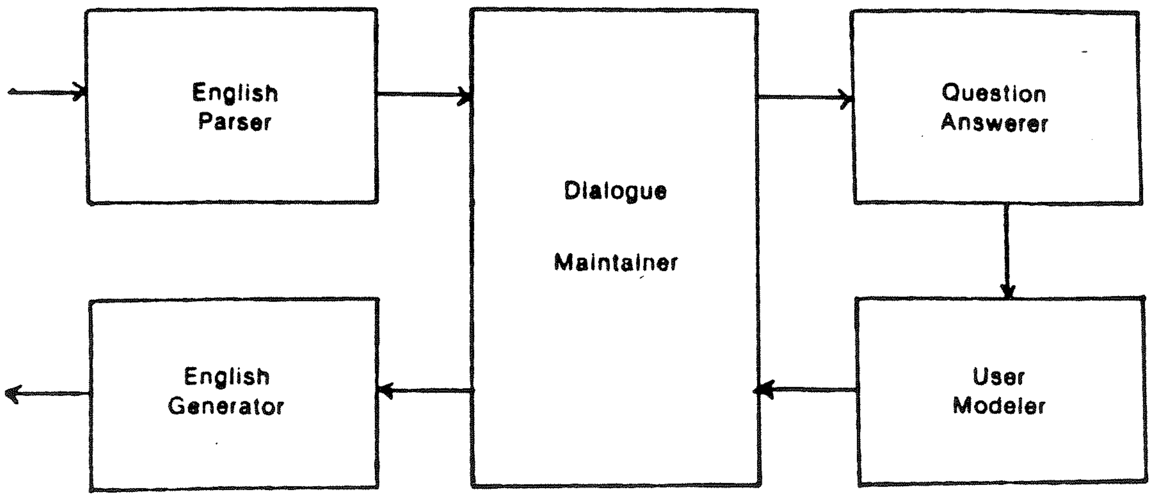

Figure 3-1: The Structure of an Interactive Help System

In the UNIX help system project at Berkeley[10] three of these components are being built: the English understander, the question answerer, and the English generator. Their system, UC, can answer questions such as, "How can I delete a file?" or "How can I get more disk space?" The natural language front end to the system was built using a general-purpose English analyzer, which was modified to handle the specific language forms that are used in talking about UNIX. The problem-solving component of UC is interesting because it exploits the powerful, general-purpose (as we have already seen) idea of stored plans that can be used to satisfy goals. UC's representation of a UNIX command includes a statement of the preconditions that must hold for the command to work and the state changes that are caused by the command. Simple requests, such as "How do I delete a file," can be answered directly by finding a command whose result matches the result the user desires. But to answer more complex questions, the interactions between goals and plans must be considered. For example, although one can get more disk space by deleting all one's files, most of the time that is not an acceptable solution because it conflicts with another goal of preserving one's previous work. To answer this type of question, UC must consider such interactions and find a way of satisfying the stated goal that does not, as a side effect, undo other, possibly unstated goals.

Another approach to the design of a system that answers questions about another program (the target program) is to give the target program itself to the question answerer and to provide the question answerer with the ability to reason about programs. This is the approach we are taking in the help system we are building, which answers questions about the document-formatting program, Scribe.

There are two significant advantages to be gained by using Scribe itself as the database for its help system. The first is that the effort that would otherwise be required to construct a database specifically for the help system is avoided. The second is that any separately constructed database will almost surely not describe every detail of the target system's behavior. This means that the help system will necessarily "bottom out" when the level of the user's questions exceeds that of the help system's knowledge. This is particularly important in Scribe, which is a complex system with many features that interact in unpredictable ways. The code for Scribe is not enough, by itself, to serve as the knowledge base for a help system. It must be augmented with some additional information, particularly a lexicon that maps between internal program objects (such as particular variables and procedures) and concepts in the problem domain (e.g. left margin, justification). In addition, a practical system needs to remember the answers it has computed so that simple, commonly-occurring questions can be answered quickly.

The Scribe help system can answer questions like, "How do I get my paper right justified?" or "How can I get double columns?" Although the complete system must possess all of the components shown in Figure 3-1, the current version of the system contains only the reasoning mechanism. Questions to the reasoning mechanism must be stated in a formal query language.

The help system answers users' questions by matching the questions against portions of the Scribe code and then chaining through the code to find answers at the appropriate level. The following examples illustrate this process. Although chaining can be described as a syntactic search procedure, it relies on semantic information to decide which paths to follow and which to prune. It is here that the help system must exploit knowledge about programming and about the individual users it is dealing with. The following examples show how the help system finds answers to common types of user questions:

1. "How do I get my paper right justified?" - Find the place in the code where justification is done. Find out what determines how it is done (e.g. a particular variable value). Then find out what user-controllable event(s) can cause the variable to be set appropriately. Notice that it is necessary to understand the scope of an action, since the user does not want to know how to get a single line right justifed but rather it is the whole paper that must be considered.

2. "What is the difference between the itemize and the enumerate commands?" - Find the code that is executed for each of the two commands. Compare them and report significant differences.

3. "What will happen if I put @bibliography in the middle of my file?" - Find the code that is executed when the command is processed. This code represents the highest level of description of what happens. Chain down through it (by looking at each procedure that is called and perhaps at each procedure that each of them calls, and so forth) until a description at the desired level of detail is found.

The power of the help system derives from its use of the two main tools of artificial intelligence: knowledge and heuristic search. The major source of domain-specific knowledge in the system is the Scribe code itself. The way that this code is organized and represented is thus critical to the performance of the help system. The needs of the help system suggest both a particular programming language design and a programming discipline to be followed by people who code the systems for which help facilities are desired. Fortunately, the demands of the help system compliment, rather than conflict with, many of the other demands placed upon programs (such as that they be readable, maintainable, and verifiable), so it is reasonable to expect them to be met. The help system also appeals to a database of knowledge about the individual users with whom it is communicating. In addition to its specific knowledge about Scribe and its users, the help system must exploit more general programming knowledge in order to derive answers to questions by examining code. This more general knowledge forms the basis of its heuristic search through the code to look for appropriate answers. Interestingly, this more general knowledge looks a lot like the programming knowledge contained in the plans of the Programmer's Apprentice and the plans of the LISP tutor. As our catalogue of such programming knowledge grows, the effectiveness of these programs that use that knowledge grows and further encroachment happens.

3.2. Volunteering Advice

Direct help facilities, such as the ones described above, are useful when users know that they need to know something. But one of the things that shared knowledge enables one to do is to increase the bandwidth of a communication channel. To exploit knowledge to increse the bandwidth between a user and a program requires that the program take a more active role in recognizing when it needs to apply its knowledge. It must then apply the knowledge appropriately to provide indirect assistance to the user. Some of the circumstances in which this is needed include: when a complete statement of a required command is long but the need for the command can be inferred from context, when an explicit error is made in a command, when a correct command is used in a way that will not produce the result that the user desires, and when there is a better way to do what the user is doing. Sometimes the user should be given explicit advice. At other times, the system should just proceed and do what it knows the user intended.

Simple indirect assistance is already provided on many systems. For example, on some systems you can type

```
        edit pap<esc>
```

instead of

```
        edit paper
```

provided that you have only one file whose name begins with "pap". Simple error correcting capabilities are provided by such systems as DWIM in INTERLISP. If a command is in error, DWIM will perform simple changes (such as transposing a pair of letters) to see whether it can derive a legal command. But such systems work completely syntactically. They exploit no semantic knowledge of the sort a person who was looking over your shoulder would have. Artificial intelligence can encroach into this arena, elucidation process consists of three steps:

- Plan, which means to analyze the data and produce candidate substructures that can be used to constrain the later search process.
- Generate possible structures that are consistent with the results of the planning step.
- Test those structures, using additional chemical knowledge to determine whether the proposed structures are consistent with the observed data.

DENDRAL has been used extensively in chemical research. It has contributed to the analysis of several previously unknown organic molecules. DENDRAL's power comes from two sources:

- Its ability to search systematically through a large space of structures and to find *all* of the ones that match a given problem.
- Its ability to exploit knowledge about chemical structures to constrain its search to only those molecules that are likely matches for a given problem.

Thus we see the two common threads of artificial intelligence appearing yet again.

A more recent part of the DENDRAL project is meta-DENDRAL, which is a program that infers structure elucidataion rules of the sort that are used in DENDRAL. Thus meta-DENDRAL constitutes one further encroachment step over that taken by DENDRAL, which relied on people to supply its rules.

4.2. An Interactive Assistant

DENDRAL helps chemists analyze existing chemical compounds. Another important task that chemists must do is to discover ways to synthesize desired compounds. Artificial intelligence is also encroaching on the performance of this task.

To synthesize a desired molecule, it is usually necessary to perform a sequence of reactions that lead from available starting molecules to the desired structure. Planning such a sequence requires both knowledge (of the individual reactions that are known to occur) and search to construct a plausible sequence that will lead to the desired result. Artificial intelligence can encroach on this process in the following stages:

1. Provide efficient access to the large database of known reactions that must be considered. At this stage, the human chemist must select appropriate reactions and decide how to combine them.
2. Infer new structures to be added to the database. This can happen in two ways. First, the experience from stage 1 can be used to improve the organization of the reaction database so that stage 1 can be performed more efficiently and so that further planning can be automated. Second, new reactions can be inferred from patterns of existing ones.
3. Select a sequence of reactions that is likely to accomplish a desired synthesis.

There exist several system designed to perform step 1 (for example, Wipke's system[12]). But as the size of the reaction database grows, the speed of these systems decreases substantially because of the large