# ARTIFICIAL INTELLIGENCE AND AUTOMATIC PROGRAMMING IN CAI

Elliot B. Koffman
Computer Science Group
Electrical Engineering Dept
University of Connecticut
Storrs, Conn. 06268

Sumner E. Blount
Software Development Group
Digital Equipment Corp.
Maynard, Mass. 01754

## Abstract

This paper discusses generative computer-assisted instruction (CAI) and its relationship to Artificial Intelligence Research. Systems which *have a* limited capability for natural language communication are described. In addition, potential areas in which Artificial Intelligence could be applied are outlined. These include individualization of instruction, determining the degree of" accuracy of a student response, and problem-solving,

A CAI system which is capable of writing computer programs is described in detail. Techniques are given for generating meaningful programming problems. These problems are represented as a sequence of primitive tasks each of which can be coded in several ways. The manner in which the system designs its own solution program and monitors the student solution is also described.

## I. AI in CAT

There is currently significant interest in generative systems for computer-assisted instruction (CAI). A generative system has the capability to both generate and solve meaningful problems. It must also be *able to monitor a* student's solution, determine to what extent the student is correct, and provide pertinent remedial feedback.

Generative CAI systems free the course author from having to develop several alternative presentations of the same or similar material. In addition, the course author does not have *to* specify *correct* and incorrect answers and their consequences as the system can determine all of these on its own. These systems often have the capability to reply to student questions and allow him to divert the instruction to areas that interest him.

### Data-Base Systems

There have been a few CAI systems designed which are oriented around a structured network of facts. Wexler describes a system which combines generative CAI with frame-oriented CAI in that the course-author must specify certain question formats. The system generates parameters for these formats and searches the data base to determine the correct answer. It generates remedial feedback from its *network* if the student's solution is incorrect. This feedback can either be in the form of a correct statement based on the student's response or a trace of the steps required to retrieve the correct answer. On command from the student, it will generate statements from its data-base which conform to prespecified patterns.

This research has been supported by the National Institute of Education, Grant #0EG-0-72-0895. The University of Connecticut Computer Center Provided computational facilities.

Carbonell describes a system, SCHOLAR, in which the generation of several types of questions is done entirely by the system from the semantic network. Also, the system has the capability to interpret a variety of student questions and generate appropriate responses. As in the Wexler system, the semantic network must be constructed by the course-author.

Recent research with SCHOLAR (Carbonell[3]) has attempted to incorporate graphical information along with symbolic information in a semantic network. In addition, emphasis is being placed on improving the ability of the system to make inferences based upon the information contained in its semantic network and to take into account the fuzzyness or uncertainty associated with an inference. *(Carhonell* and Collins[4]).

Simmons[5] describes a system in which the semantic network is automatically produced from a page of text. The student can request paraphases of difficult passages and word definitions. In addition, the system can generate "fill-in" and "true-false" questions from its semantic network.

*This system* relies heavily on the use of an augmented transition network grammar (Woods[6]) both for the construction of the semantic network and for the generation of statements from this network. More recent research has focused on the use of a STRIPS like model (Fikes and Nilsson') to allow modifications to the semantic network which reflect a time sequence of events.

Research in a CAI system for meteorology (Brown, Burton and Zdybel[8]) has investigated the use of a semantic network to store static information and an augmented finite automata structure for storing information concerning the dynamics of a process. This latter structure shows, for all possible state transitions, the external and internal conditions which must be met before the transition can occur. Consequently, when a student poses a question, he excites the model and causes the generation of a tree of intermediate states until equilibrium is reached. Answering the student's question is accomplished by interpreting this tree of states in the appropriate manner.

As research continues, improvements due to Artificial Intelligence Research will enable future versions of data-base oriented systems to accept an unstructured or essay-type response from a student. These systems provide an impressive demonstration of the potential application of research in natural language communications and question-answering to CAI. However, there are still several problem areas in generative CAI which tie in very closely with Artificial Intelligence.

### Individualizing Instruction and Remedial Aid

The first problem is individualizing the

instruction received by each student or determining what is best for a student to study based on his previous interactions with the system. This is a problem in pattern recognition. The student must be classified as a particular type of learner. Based on this categorization and his past history, an appropriate concept should be selected for study.

This problem is being studied by Shea and Sleeaan . They propose a three level teaching system in which level one is strictly concerned with student interaction, record collection, and generation of problems to suit the criteria imposed by level 2. Level 2 is responsible for implementing a specific teaching strategy. Level 3 decides which strategy should be used and evaluates the performance of the lower levels to determine whether performance is satisfactory or whether a different strategy should be tried.

Other studies in the design of an intelligent monitor for CAI have centered around the use of a student model (summary of a student's past performance) and a concept tree, which indicates the pre-requisite structure of the course. As the syster. gains experience with a particular student, it updates his model and establishes to what extent he prefers to advance quickly to new material or build a solid foundation before going on. Based on its knowledge of the student and his past performance, it decides at which plateau of the concept tree the student should be working. All concepts on this plateau are then evaluated with respect to factors such as recency of use, change in state of knowledge during last interaction, current state of knowledge, tendency to increase or decrease his state of knowledge, and relevance to other course concepts. The highest scoring concept is selected, a problem suitable for his experience level is generated, and its solution is monitored.

In order to generate pertiment remedial feedback, a CAI system must determine the degree of correctness of a student response. If a simple number or phrase is expected, it is relatively easy to determine whether a student is correct. In mathematics, resolution-based theorem provers have been used to verify logical proof procedures[11]. Their initial application has been successful in assisting a student in constructing a proof by applying the rule of inference designated by him to earlier lines in the proof sequence. If the student's rule cannot be applied, he is given remedial feedback. Efforts to verify that a line typed in by a student is a valid inference from his previous work have been "moderately successful". A more ambitious goal now under study is to have the theorem prover suggest ways of completing a proof taking into account a student's incomplete or erroneous work.

Often it is not sufficient to tell a student he is wrong and indicate the correct solution method. An intelligent CAI system should be able to make hypotheses based on a student's error history as to where the real source of his difficulty lies. For example, a student may be having trouble with division because he does not know how to multiply and/or subtract. The CAI monitor must recognize this and redirect the student's efforts or he will never master the division algorithm.

This problem, of course, is even harder to resolve in natural language systems. None of these systems currently have an effective way of adapting the tutorial dialogue to exploit the information present in an incorrect student response.

Problem-Solving in CAI

A final probleiti area is pertinent to researchers attempting to use generative CAI in *a* problem-solving environment. This is the design of the problem-solver itself. Perhaps current research in problem-solving and planning techniques such as are evidenced in STRIPS and in languages such as PLANNER[12] and QA4[13] will find application in this area.

A generative CAI system has been designed around an introductory course in computer science at the University of Connecticut. Two-thirds of this course is concerned with an introduction to digital logic design; the remainder of the course teaches machine-language programming. An algorithmic approach has been used to teach the concepts in digital logic design through CAI. The specific approach taken has been described elsewhere[1] .

The software portion of the course Is concerned with teaching students how to write machine language programs for a small computer, very similar to the Digital Equipment Corporation PDP-8. The instruction set and organization are identical; the only difference is that student programs are restricted to pages 0 and 1 of memory. This consists of 377 (octal) words of core which is sufficient for beginning students to learn to program (and also to learn principles of memory conservatlyn).

The CAI system *Its* not used to replace the classes or textbook. It Is intended to provide practice and tutoring in problem solving similar to what one would obtain through homework problems. The advantage, of course, is that the student gets some guided direction and remedial feedback when he goes astray.

Since the system is generative and problems are constructed in a random fashion, it is necessary that it also be able to write its own solution programs. There has been significant effort expended in the development of programs with this capability. Early work by Simon[14] has been reported. Recent efforts by Manna and Waldinger[15] emphasise a theorem proving approach to writing programs. It was felt that this general approach would not allow programs which emphasize the requisite techniques of machine-language programming to be designed.

The remainder of the paper will focus on our heuristic approach to automatic programming. The system so designed will be referred to as MALT (MAchine Language Tutor).

## II. System Overview

Prior to describing the system itself, it would be worthwhile to outline some of its goals and constraints. It was important that this system function effectively as a tutor in machine language programming for approximately thirty students per semester, rather than just serve as a demonstration of what might be accomplished if additional computer resources were available. This meant that MALT had to be implemented on the time-sharing system currently available on the University's IBM 360/65 which is CPS (Conversational Programming System-IBM[16]), a dialect of PL-I. It also meant that response times to students should normally be on the order of 5 to 10 seconds or less. An additional restriction imposed by CPE itself was that no more than twelve pages (U8K bytes) of core memory be active at any user terminal.

A second goal of this system was that it be easily expandable. This means it should be possible

to add new problem types without having to increase the system's problem solving capability, This factor, and the requirement for a compact or modular structure in order to meet the twelve page core limit, resulted in the use of primitive tasks to serve as building blocks in the design of solution programs,

A third goal was that the system be capable of a high degree of individualization of instruction. The form of the problem posed for each student should be influenced by his previous experience with the system. The degree of monitoring should decrease as the student's level of competence increased; in fact, the system should assist the student by programming previously mastered sub-tasks *for* him.

## III. General Approach

A generative CAI system has the minimum components illustrated in Figure 1. The problem generator- is capable of providing an unlimited variety of meaningful problems. A vital information link exists between the problem generator and problem Solver. This link provides the system with the exact structure of the problem, and allows the problem solver to concentrate on the solution of the problem rather than its definition.

The structure of each problem is illustrated *by* the *AND-OR goal tree in* Figure 2. *A complex* programming problem is represented as three sub-problems dealing with input of information, processing of core-resident information, and program output respectively. There are several alternatives for each of the sub-problems. For example, (I , I , ..., I } is the set of input sub-problems where I,is the null sub-problem. Each sub-problem, in turn, is decomposed into a sequence of logical tasks. Some of these tasks are primitive tasks which the system can solve directly in one or more ways (1 and I-,-). Other logical tasks consist themselves of a sequence of primitive tasks (i ,).
ID

There are thirty-five primitive tasks programmed in MALT. They would be used repeatedly in the design of a complete program. Sample tasks might be: the initialization of pointers to data, initialization of counters to keep track of the number of loop iterations; reading or printing an ASC-II character; and transferring data into and out of memory. A large variety of problems can be constructed using only this set of primitive tasks.

Figure 3 is a block diagram for the MALT system. The previous student performance determines what type of problem will *be* generated. This problem is presented to the student in natural language and also passed on to the system as an ordered triple of sub-problems. Next, a list of logical tasks for each of the sub-problems is presented to the student. The system representation of this "flow-chart" is a sequence of calls to problem-solver routines. These problem-solver routines solve the programming problem and interact with the student to monitor his solution.

As the student undertakes each task in the programming process, a corresponding problem-solver routine is entered by the CAI system, which guides the student through the construction of that part of his program. During this phase, the student is constantly being given feedback as to the correctness of his program. If his program introduces logical errors, the system will point these out and offer helpful suggestions for their correction. If the system feels that the student might benefit from observing his program in operation, it also has the capability to simulate statement by statement program execution.

The system is constantly evaluating the student's performance and updating his permanent file. This is necessary because his achievement determines not only the difficulty of the problems given him, but also the amount of interaction which he receives during the design of his program.

It should be stressed at this point that the system can only solve the primitive tasks. Complex problems *can be constructed using* these tasks *as* building blocks. The system's knowledge of the primitive tasks and their relationships enables it to generate the solution for the complete programming problem.

As is indicated by the above discussion, the logic sequence which the system will use to code the problem is pretty well determined by its implied structure. The only freedom that exists is in the selection of the code to implement each primitive task.

This is perfectly satisfactory for a system whose sole function is to produce the machine-language code which accomplishes a particular programming problem. However, this approach should be justified in a CAI environment.

There are normally many ways to flowchart and code a programming problem. The system could accept the student's solution in its entirety, derive its own solution program, execute both programs, and compare their results. Such an approach, although it would give the student complete freedom and verify his program, would be worthless as a teacher. Essentially, it would say to the student who has made a mistake: "Your answer is wrong, but I have no idea why. Here is my solution; perhaps you can figure out what is wrong yourself." A student who can do this does not really need *a* CAI tutor.

The approach we have taken is to say: "You are just learning how to code in machine language. Your problem can be flowcharted in this way. If you will follow my flowchart, I can teach you something about machine-language coding," We feel (and so do our student guinea pigs) that the beginning student gains more from a little guidance at this stage and learns quite a bit about flowcharting through observation.

Perhaps, current research in program synthesis and verification will soon remove this restriction. It would then be preferable to allow the student to design his own program to the extent he is able. The system could verify that the portion of the program constructed so far does, in fact, accomplish what the student intended. At this point, the system would take over and synthesize the remainder of the solution. If the student's portion of the solution were only partially correct, the point at which his program went astray could, perhaps, be located and the correct solution continued from there.

## IV. Problem Generator

As has been mentioned, each problem may be thought of as consisting of three (or less) distinct phases. When students first start off using the system, they will normally be presented with a problem that consists of just a single phase. As they gain competence and experience, more difficult problems will be generated for them.

Problem generation consists of selecting a pa"Uh through a tree of depth three as shown in Figure 4. Branches A1 through A7 consist of the seven strings (including the null string) describing the input sub-problems. B , B , B are the three meaningful processing steps (out of a total of thirteen) which can be performed on the input format described by A,. $^{C}$jiji $^{c}$^i2 $^{are}$ $^{tlie}$ possible forms of the output sub-problem (out of a total of 10) for a problem beginning with A and B .

The probability of a particular branch being selected is a function of each student's competence. These probabilities change as the student progresses through the course material. There are more than *HO* different paths through this tree which represent meaningful problems. Before presenting a problem format (or path) which has been .selected, the system checks to see that it is "different" from one previously worked by this student.

In addition, once the format has been selected, various parameters must be randomly generated. These parameters might represent specific memory registers or character's to be searched for. The complete problem is described internally a?, a vector consisting of the three selected brandies from the problem tree followed by all required parameters. Sample problems are shown in Table 1.

TABLE 1

Sample Problems

Note: All Randomly Generated Parameters Are Underlined.

1. Add the contents of register 150 to the cuntenU; of register 167.

2. Print out the message "Hello".

3. Read in a series of ASCJ-l1 character's ending with a * and store them starting, in location 11?Q.

4. Read in 31 A'SC-ll characters and store them starting at iocation 300. Search register 3Q0 through 330 for the largest number.

b. Read in a series of 3-digit numbers and store them starting at location 25Q. The input will end when the first character of a number[1] is a "X".

6. Read in 24 (octal) four digit numbers and store them starting at iocation 242. Search registers 242 thru 265 for the 1st number which begins with the ocTal digits "7£". (Example 7QXX)

7. Multiply the contents of register 211 by the contents of register 310. Finally print out the 4-digit contents of the Accumulator.

8. Search registers 160 thru 20'5 for the octal number 7215. For registers 160 thru 205^ print out the register number, 4_ spaces, and the octal contents of that register.

9. Assume that a table has been set up starting at location 120 consisting of a 2-character symbol followed by a number-, there are W_ of these entries.
Search the table for the symbol "AN" and retrieve the corresponding number. If it is not in the table, then halt the program. Finally, print out the 4-digit contents of the Accumulator.

The next step is to generate the corresponding list of logical sub-tasks. These are generated separately for each of the three major phases of the problem. An example is shown in Table 2. As mentioned above, the student solution would be monitored after the generation of all the sub-tasks associated with each phase of the problem.

TABLE 2

EXAMPLE Or PROGRAM SUB-TASK GENERATION

YOUR PROBLEM IS To WRITE A PROGRAM WHICH WILL;

a) Read in 20 (octal) ASCII characters and store them in registers 240 thru ?f>0.

b) Form the absolute value of the contents of register 240 in the Accumulator.

cl Finally, print out the 4-digit contents of the Accumulator.

Here- are the subt.asks for a.

1) Initialize a ptr to register 240.
2) Initialize a ctr with the value of -20 (octal).
3) Read a character,
4) Store it away using the ptr-.
5) Update the ptr.
0) Update the ctr and if it's not ;;,cro, jump back to start of loop,

here are the sub-tasks for b.

J) Bring the number in register 240 l." the Accumulator.
2) Check the sign of the ACC and if it's negative, then form it's 2's complement.

Here are the sub-tasks for c.

1) Store the contents of the Aecumul.-it.ur temporarily.
*2)* Set up a subroutine which will print a character,
3) Get the number to be printed and rotate it so the 1st (or leftmost) octal digit" is on the right.
4) Mask out the lefr q bits, add 260, and call the print subroutine,
5) Do the same for the 2nd digit,
b) Do the same for the 3rd digit.
7) F.et the number, mask it, add 260, and print it.

The system representation of these sub-tasks consists of a string of two digit numbers. This sequence indicates which of the thirty-five primi'ive tasks of TtiHchine language programming muct be performed, and in what order, to program each sub-problem correctly. Interpretation of this string by the control routine results in a sequence of calls to the individual problem-soxver routines. The manner in which the problem-t.olvers monitor the machine code being entered by the student will be described next.

## V. Problem Solver and Monitor

Each of the logical sub-tasks is broken down further into a sequence of primitive tasks (35 total) which the system can solve. The student builds his solution by coding one task at a time as Implied by Figure 2.

The system provides the student with varying degrees of interaction depending on his past performance.

A beginning student will be led by the hand and each program statement will be requested and checked individually. An intermediate student receives fewer hints and will normally enter a group of statements at a time; an advanced student benefits from the system's ability to code its own programs and will be provided with the code for the simpler sub-tasks. This is important as it frees him to concentrate on more difficult areas and prevents the boredom which would result from constant repetition of trivial tasks.

Table 3 provides an example of the problem solver/monitor interaction with a student. The square-bracketed comments have been added for clarity. Each underscore indicates the start of a student response. M377 stands for memory register 377. The system is operating at a moderate degree of interaction.

TABLE 3

EXAMPLE OF STUDENT INTERACTION

```
Your problem is to write a program which will:
Read in 10 (octal), 1-digit numbers and store their
value starting in register 205.

Is this problem OK?
yes

Here are the sub-tasks for the 1st line:
1)  Initialize a pointer to register 205,
2)  Initialize a counter with the value of -10 (octal).
3)  Read a digit and mask out the left 9 bits.
4)  Store it away using the pointer.
5)  Update the pointer
S)  Update the counter and if its not zero, jump
    back to start of loop.

SUB-TASK 1:
Use M377 to store the pointer value, and M376 as the
pointer.
M377:
105
Wrong!  "0205" is the correct ptr value.

M001:  [user program starts in register 1]
7200 [clear the ACC]
M002:
1377 [get the value of the pointer]
M003:
3376 [deposit into pointer]

SUB-TASK 2:
Use H375 to store the counter value and M374 as the
counter.
M375:
0012
"0012" can't be used as the counter value.  It should
be either "0010" or "7770".  Try again!
M375:
7770
Now set the counter value into the counter.
M004:
7200 [clear the ACC]
You don't need a "CLA" here but it's ok.
M005:
1375
M006:
3374 [set up the ctr]

SUB-TASK 3:
You'll need 3 instruction to read a character.
M007:
6031 [check the keyboard flag]
M010:
5007 [Jump to M007]
M011:
6036 [Read a keyboard Char.]

Let's store the mask in M373
 What number should we use as a mask?
M373:
£077
Wrong! "0007" is the correct mask because it should
contain a "1" corresponding to every bit of the
 Accumulator we would like to Keep.

 Now, "AND" the ACC with the mask.
 M012:
 £373 ["AND" the ACC]

 SUB-TASK 4:
 M013:  3776 [system generates code for easy sub-
 task]

 SUB-TASK 5:
 M014:
 2377 [increment the pointer]
 Wrong!  The ptr is in M376 so "ISZ PTR" should be:
 2376.

 SUB-TASK 6:
 M015:
 2374 [increment the counter!]
 M016:
 5005 [Jump to MOOS]
 Wrong! the body of the loop begins in M007 so it
 should be:  "5007".

 Now we should halt the program
 M017:  7402
```

Several conventions have been established to facilitate the generation of program segments and monitoring of student programs. All user programs begin in location 001 and all program constants are placed at the top of memory beginning with location 377 and preceeding downwards. The middle areas of memory, locations 120 through 350, are reserved for lists and tables to be used by the student's program.

The existence of a program loop is assumed by the system whenever a pointer or counter is initialized. The physical start of the loop is the first memory register after the initialization process. By monitoring the beginning of a loop in this manner, the system can easily determine if the student has correctly designed his end-of-loop decision sequence. The most common programming mistake of this kind occurs when the student attempts to jump back to the initialization sequence instead of the main body of the loop.

Another program parameter which must be kept track of is the accumulator status. The simulated computer has neither a non-destructive deposit nor a destructive load instruction. Hence, the accumulator must be cleared prior to loading it with a given number and must be reloaded after a number has been deposited in memory if the number is still needed. The status of the accumulator is going to determine which of several alternatives is to be pursued by certain of the problem-solver routines in the design and checking of a program segment.

A complicating factor in determining accumulator status is the existence of logical branching or program jumps. The accumulator status may differ

depending on whether a sub-task was entered sequentially or through a program jump.

Forward jumps to yet unprogrammed sub-tasks also present a problem, as the memory location in which the new sub-task starts is not yet known. Consequently, HALT keeps track of the first memory location of each sub-task. If a jump is made to a previously programmed sub-task, the required instruction is provided immediately. In the case of forward jumps, a note is made of the memory location in which the jump instruction belongs and the sub-task to be reached. When this sub-task is finally programmed, HALT completes all prior jump instructions which reference it.

There are two important techniques used by the MALT system to judge the correctness of a student's program. The most common method is to analyze in detail each segment of the program as it is typed in, to determine if it performs the required functions. This is done on an instruction-by-insfruction basis so that there is immediate feedback to the student.

Immediate verification implies that the system must have a detailed knowledge of the status of the user's program at all times. As the student formulates each response, the system also generates what it considers to be an appropriate answer. If the two do not match, the system must determine if other responses are possible. If so, the student's answer is compared with all such reasonable possibilities. When the system finally decides that the response supplied by the student is in error', it informs him as to the reason for this determination and supplies the best program alternative.

If the student's response matches any of those which the system generated, then it is accepted by the system as a valid alternative to its own solution. Since this was not the expected result, however, the system must adjust its representation of the users program status to reflect the new conditions.

In the rare event that there are too many acceptable ways to program a particular sub-task, the program segment supplied by the student is simulated to determine its correctness. To verify the user's program through simulation, all conditions of the machine which might possibly affect final program results are determined. For example, if the program is intended to perform a particular operation depending upon the status of the overflow link register, then only two initial states are necessary; the program is tested with a zero Link and again later with a non-zero Link,

Once the various initial states have been determined, the program segment can be simulated under each condition. The system decides, following each simulation, if normal program termination occurred. Conditions which might cause abnormal termination are such things as infinite loops, undefined instructions, or program branches which are directed outside the user's program segment. Any such conditions are corrected immediately by the student, the current set of initial conditions is re-established, and simulation is attempted again.

If any particular terminal condition indicates that the user's program did not perform its function correctly, MALT attempts remedial action. Since it is aware of the exact results which should have been obtained, it can provide a concise description of the error. It cannot, however, isolate the location of the error in the user's program. This determination must be left up to the student. However, the problem has been greatly simplified due to the system's diagnostics and the user's ability to observe his program in execution. If the student is unable to correct his program segment, MALT will generate a correct program segment for him.

## VI. Conclusions

The system has been implemented in the CPE (16) language on. the IBM 360/65 at the University of Connecticut Computer Center. Students can use this system whenever' they desire. There is also a batch-mode simulator of this computer which they use for class projects of a more ambitious nature.

Student reaction to MALT has been very favorable. They feel this system helps to bridge the gap between what they have learned in class, of from the textbook, and what they need to know to program independently in batch-mode.

This past semester, students spent two weeks using HALT and were then given a week to get a rather sizeable problem coded and running in batch-mode. All but one student managed to accomplish this.

A questionnaire was distributed to the class. The results of this questionnaire are tabulated in Table 4. It appears that the students feel that this experience was beneficial and good preparation for learning to program independently. On the whole, students were not bothered by the fact that MALT requires them to adhere to a particular "flowchart". As Indicated by question seven, improvements must be made to the algorithm which determines that a generated problem is sufficiently different from previous problems presented to that student.

TABLL' 4

Student Evaluation

for questions 1-9 the numbers of students giving the following responses are tubulated.

| | Strongly Disagree | disagree | Uncertain | Agree | strongly Agree |
|---|---|---|---|---|---|
| 1. The system was useful in introducing me to machine language programming. | 2 | 1 | 18 | 12 | |
| 2. It was relatively easy to learn to Use the batch version of the assembler since I had been introduced to programming concepts through MALT. | 0 | 5 | 4 | 15 | 7 |
| 3. since the sub-tasks were always laid out for me, 1 felt very constrained using MALT. | 0 | 19 | 9 | 5 | 0 |
| 4. Because the sub-tasks were laid out, I only learned the mechanics of programming and didn't really understand what was going on. | 1 | 16 | 8 | 5 | 2 |
| 5. The approach taken in printing out the sub-tasks was good as it taught me how to organize a machine-language program. | 0 | 2 | 7 | 20 | 4 |

6. The problem became more difficult as my level increased,
   1 3 7 19 3

7. There was a good variety in the problems I received in MALT.
   1 12 6 13

S. In general, I enjoyed the interaction with MALT.
   0 3 6 21 3

9. In general, I preferred the use of CAI in this course to conventional homework.
   0 2 4 11 16

Overall . we feel that MALT is an effective demonstration of what can be accomplished in CAI with the limited use of AI techniques. It should be stressed that MALT's design has been influenced by AI research, but certainly much more could be done in the way of incorporating AI Research in problem solving and program synthesis. The desire to produce a working system with reasonable response time on an existing time-sharing system precluded this possibility. Hopefully, MALT will challenge others with an interest in CAI and AI to pursue this goal further.

## REFERENCES


1. Wexler, J. D., "Information Networks in Generative Computer-Assisted Instruction," IEEE Trans. on Man-Machine Systems, Vol. MMG-11, No. 4, December 1970, pp. 181-190.

2. Carbonell, J. R., "AI in CAT: An Artificial Intelligence Approach to Computer-Assisted Instruction," IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, Dec. 1970, pp. 190-202.

3. Carbonell, J. R., "Artificial Intelligence and Large Interactive Man Computer Systems," Proc. of the 1971 Joint National Conference on Major Systems.

4. Carbonell, J. R., and Collins, A. M., "Natural Semantics in Artificial Intelligence, " Bolt Beranek and Newman Working Paper, March, 1973.

5. Simmons, R. F., "Natural Language For Instructional Communication," In Artificial Intelligence and Heuristic Programming, Edinburgh Univ. Press 1971, pp. 191-198.

6. Woods, W., "Transition Network Grammars for Natural Language Analysis," Comm, Assoc, Comput. Mach. Vol. 13, Oct. 1970, pp. 591-606.

7. Fikes, R. E., and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, II, 1971, pp. 189-208.

8. Brown, J. S., Burton, R. R., and Zdybel,"A Model-Driven Question-Answering System for Mixed-Initiative Computer-Assisted Instruction", IEEE Trans, on Systems, Man, and Cybernetics, SMC-3, No. 3, pp. 248-257.

9. Shea, T. 0., and Sleeman, D. H., 1972, "A Design for Adaptive Self-Improving Teaching System," Working paper, Department of Computational Science, University of Leeds, England.

10. Koffman, E. B., "A Generative CAI Tutor for Computer Science Concepts," Proceedings of the AFIPS 1972 Spring Joint Computer Conference.

11. Goldberg, A, and Suppes, P., "A Computer-Assisted Instruction Program for Exercises on rinding Axioms," Tech. Report #186, Stanford University, Institute for Mathematical Studies In the Social Sciences, June, 1972.

12. Hewitt, C., "Planner; ft Language for Proving Theorems in Robots," Proceedings of the 1369 International Joint Conference on Artificial Intelligence, Ed, D. E. Walker and L. M. Norton, 1969, pp. 295-302.

13. Rulifsan, J. F, Derksen, J. A. and Waldinger, R. J., "QA4: A Procedural Calculus for Inductive Reasoning," SRI Technical Note #73, 1972.

14. Simon, H. A., "Experiments with a Heuristic Compiler," 1963, JACM, Vol. 10, No. 4, October 1963.

15. Manna, Z., Waldringer, R. J., "Toward Automatic Program Synthesis," CACM, Vol. 14, No. 3, March, 1971.

16. IBM Corporation, Conversational Programming System, (CPS) Terminal User's Manual, IBM Report GH20-0758-0 1970.

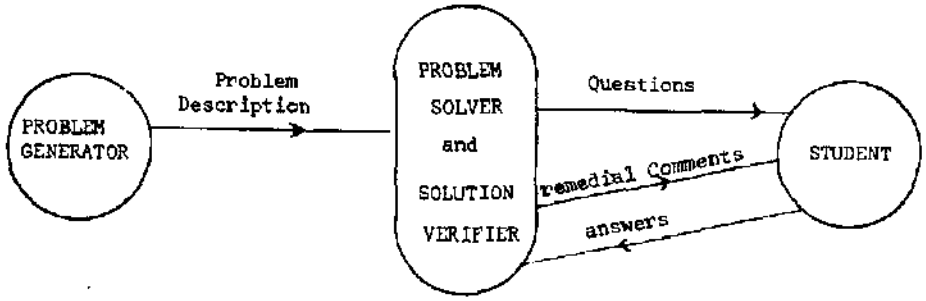


Figure 1

Minimal Generative CAI System

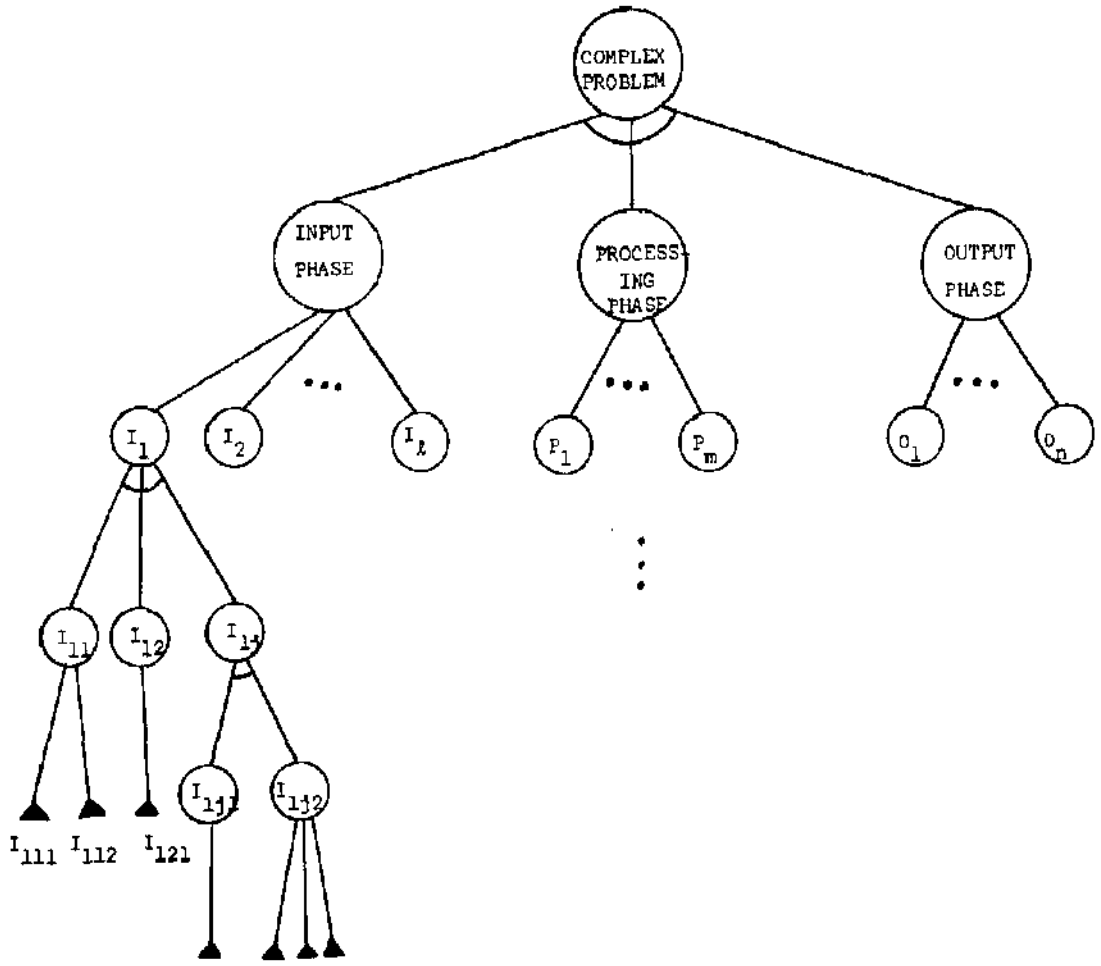


Figure 2

Problem Structure

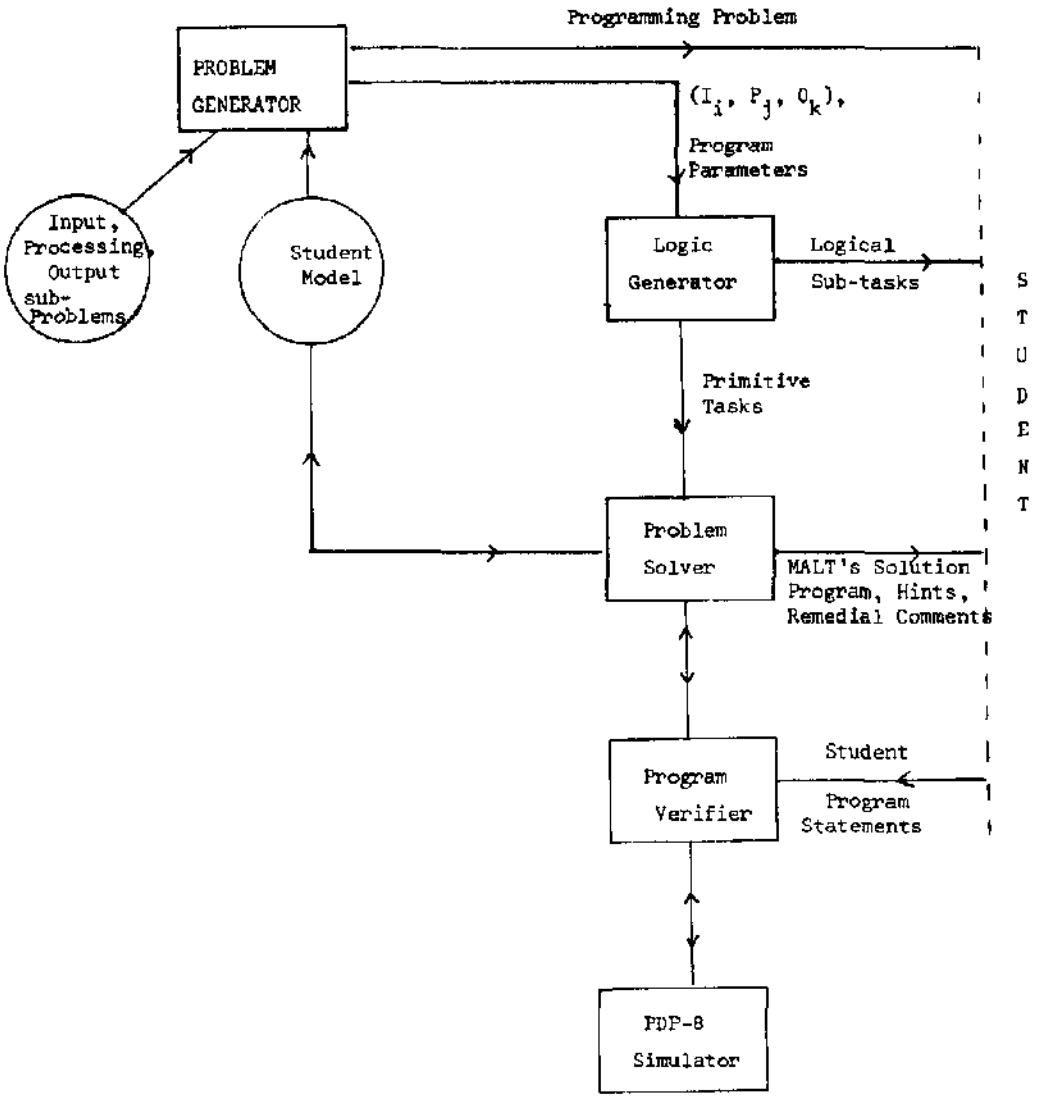


Figure 3

System Block Diagram

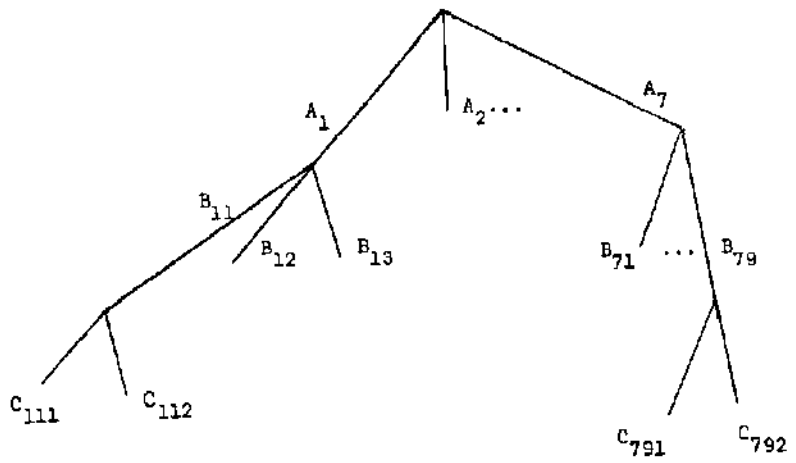


Figure 4

Tree of Problems