Preliminary thoughts about a UNIversal TEAching Machine (UNITEAM)

by JOE K. CLEMA, R. L. DIDDAY and M. WESSLER

Colorado State University
Fort Collins, Colorado

INTRODUCTION

Computer Aided Instruction (C. A. I.) promises to make practical the goal of enabling each student to receive highly individualized instruction at some point in his educational career. C. A. I. has evolved over the last fifteen years from programmed instruction which itself has had a relatively short history as an educational procedure.

Programmed instruction has not changed dramatically since the time of Pressey,1 who in 1926 used it as a means of reinforcement in the educational process. Thirty years later Skinner2 saw the need for incorporating stimulus materials into a framework of instructional aids in order to provide education without a live teacher. However, the ultimate goal of a universal teaching machine which provides instruction to individuals in arbitrary fields has not yet been achieved.

We consider a universal teaching machine to be one which may be converted to teach any subject by simply changing the program data base and which can adapt the instruction to the needs of each individual student. It is desirable that the transformation from the natural language of the subject to the machine language of the data base be somehow simple.

TEACHING MACHINES AS QUESTION-ANSWERERS

For us the problems faced by a general teaching machine are similar to those faced by question-answering systems. A question-answering system is one which accepts information and uses it to answer questions. Question-answering systems are attempts to construct and manipulate data bases which are represented as associations among symbols which have meaning when humans see them. Newer systems are based on theorem provers,3 and when given some input question must choose efficiently from an infinity of possible deductions in order to arrive at an answer. At present the limiting factor in their usefulness is the difficulty in selecting an appropriate sequence of deductions in a reasonable length of time.

A question-answering system does not store as such all information which is available to the user. Instead, it maintains a compressed data base of compactly coded facts.3,4,5 The facts may reside in data structures3,4,6 such as lists or binary trees. There are many methods for deducing answers which have not been explicitly stored in memory. These include:

1. A set of prewritten programs, one for each class of question. These programs are a permanent part of the system and do not change from their initial state.
2. A translator which creates a subprogram each time a question is input to the system. This subprogram exists only as long as it is needed to answer the question.
3. A formal theorem prover using some subset of current known facts as axioms.

The problem with prewritten subprograms is when questions arise that require interaction among existing classes. Either the interaction must be anticipated or new subprograms must be written. In No. 3 the inference mechanism must be general enough to answer questions of a variety of types.

An important characteristic of all question-answering systems is the set of languages programmed into them. At the most sophisticated level are the dialogue languages3 which include:

1. A language for introducing facts into the system;
2. A query language to be employed by the user;
3. An answer language to be employed by the system.
In the process of answering questions the system may require additional information from the user. Thus it is necessary that a query language be employed by the user and an answer language be employed by the system.7

Question-answering systems also consist of one or more internal languages. These are machine languages which are used as intermediate steps in translation8 or for calculation and manipulation of data.

The problem of representing data may be derived into three parts:7,10

1. Determining the semantic content of the data;
2. Choosing a language to represent the semantic content;
3. Choosing an internal representation of the language chosen above.

For example, the sentence "GASP is FORTRAN based." may be expressed by the binary relation "is" as applied to the subject "GASP" and the predicate "FORTRAN based." In choosing a language to express semantic content we might use the notation of symbolic logic, and in choosing an internal representation we might express a series of binary operations as a binary tree structure.9,11

OUR APPROACH

A teaching program can be structured so that the burden of finding a path through the data, i.e., from question to answer, is placed on the student. To put it another way, we can generate proofs at random. Of course we will not know what theorem we have proven (what question we have answered) until we are through, but this seems to be the way that human teachers operate anyway.

Currently we plan to represent a topic as an ordered set of information units called concepts. A concept consists of an ordered triple (a production), a string of symbols called the question format, a list of concept labels called the error list, and an integer weight which serves as a measure of how well the student knows the concept.

In operation the system will be able to access a subset of concepts, namely those it has introduced so far. Associated with each student is a vector, the elements of which serve as the weights of each concept in the topic when he is using the system. When the student is asked a question, the system selects a concept with a probability skewed by his weight vector.12,13,14,15 This involves Monte Carlo methods and techniques for generating pseudorandom numbers. A certain number of deductions are then made, again chosen according to the weighted distribution. As each deduction is performed, the question format of each successive concept is taken into account so as to generate a reasonably well-stated question. At some point in the random deduction process a deduction is not made according to the production associated with the current concept. Instead, an alternate concept is chosen from the error list, and its production is carried out next. This allows the use of "what is wrong with . . . ?" type questions which we will demonstrate later.

When a student's error vector fits certain criteria, a new concept will then be introduced and used (with high probability) in the question generation process. It is our basic assumption that if a student can answer arbitrary questions which result from associations among a basic collection of concepts, he has learned the topic. It is in this respect that we call our programs teaching programs.

Where do concepts come from? Unlike the designers of question-answering systems, we make no effort to find an elegant or concise representation. We believe that concepts should be the embodiment of a global model of the topic encoded in "symbology" which is as close to the natural language as possible. We further envision that such a program could be developed in the following way.

Experts in a field such as mathematics, programming, etc., will be engaged to extract from their topic a set of concepts which describe it. By "describe" we mean that the concepts they provide can potentially lead to the derivation of all facts that they are trying to teach in their subject area. We can think of the topic as a vector space and the concepts as a set of basic vectors which describe it. We imagine that the process of extraction will be done on-line, i.e., at any point the designer can interrogate the concepts he has previously defined in order to see what, in fact, can be derived from them. Thus he can refine concepts which lead to false deductions.

It has been implicitly assumed that we can embody everything necessary about a topic within a reasonable number of concepts. Also, the usefulness of this technique depends on its efficient implementation so that a large number of students can interact with the program simultaneously and economically. The former requirement dictates that the productions associated with each concept be potentially powerful. The latter requirement dictates that they be easily manipulated. A similar problem is faced by those who write the software for computer languages. ALGOL is represented as a
“context-free” (Chomsky Type 2) set of productions, as opposed to Chomsky Type 1 which is “context sensitive,” which are easy to manipulate but which do not completely specify the language. Context sensitive aspects of the language such as the requirement that array sizes be explicitly declared have been shelved away in symbol manipulation routines, etc.

This solution is one we cannot take. Our goal is to expose all aspects of the subject within the framework of these concepts. We apparently will be forced to use a context sensitive scheme in every interesting case.

COMPLETED WORK

In order to test our approach, we have chosen to design a program, UNITEAM, which teaches the simple programming language BASIC. We have further simplified our immediate goal, choosing to divide the teaching into three phases. We hope that those phases can eventually be included in one general framework which will be applied to a broad range of topics. Phase 1 we term “motivational” in that we are trying to relate the new words “computer”, “memory”, “word”, “I/O”, etc., both to things with which the student is already familiar and to each other.

In Phase 2 the syntax of the language and its semantics with respect to the concepts of Phase 1 are introduced. In Phase 3 the semantics in terms of the world and more global techniques of programming are emphasized. The following reports our work in Phase 2, mentioning our approach to the other two phases only in passing.

Sample productions belonging to concepts which will be stressed in Phase 1 are things like:

```plaintext
COMPUTER HAS-AS-PART MEMORY
COMPUTER HAS-AS-PART CONTROLLER
COMPUTER HAS-FUNCTION PERFORM OPERATION
```

etc.

Sample questions might be something like:

```plaintext
WHAT PART OF COMPUTER HAS-FUNCTION PERFORM OPERATION?
```

etc.

There is much work to be done on this phase which is the most general phase. For example, we must understand what effect allowing various verbs (second element of the production) has on the system. In Phase 2 we deal almost exclusively with productions which have the same verb.

In Phase 2 we are currently using techniques similar to those used in compilers for languages expressed as context free grammars (Chomsky Type 2). Since there are also global (context sensitive) features, we make a slight addition. A typical production is

```plaintext
/line/ IS UNIQUE (/statement number/0/
statement/CR
```

which would appear in traditional Backus-Naur form\textsuperscript{17} as

```plaintext
/line/ ::= /statement number/ /statement/ /CR,
```

and says that a /line/ consists of a /statement number/, a /statement/, and a carriage return. It is a global feature not expressed in the second form above that no two lines may have the same statement number. In our system the expression “UNIQUE( )” will reference a subprogram which insures that any statement number that is generated through this production is unique. There will be an additional concept with the production

```plaintext
UNIQUE() HAS-FUNCTION EACH LINE
HAS A UNIQUE /statement number/
```

which summarizes the operation of the hidden subprogram for the student.

A second difference is that, unlike compilers, we will proceed from left to right which requires the rewriting of a few productions from their more commonly seen forms.

With this brief introduction let us sketch the operation of the system as it would proceed in generating a question. For this example we will generate a statement as it would appear in BASIC\textsuperscript{18} except for one error. We will not write all the productions which are necessary, but suppose that concept 23 has been chosen. Concept 23 looks like this:

```plaintext
C23-< /assignment/> IS LET / variable /
= / arithmetic expression /
.QUESTION FORMAT = ....
.ERROR LIST = \& (null)
.WEIGHT = ....
```

We will have three push-down stacks:\textsuperscript{19,20} One stores the concepts in order of application, one stores that part of the final string we have completed, and one stores the parts we are still working on. We call these the C stack, T stack, and W stack respectively.
So far the stacks look like this:

<table>
<thead>
<tr>
<th>C Stack</th>
<th>W Stack</th>
<th>T Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LET</td>
<td>LET</td>
</tr>
<tr>
<td>C23</td>
<td>/variable/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/arithmetic expression/</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next we choose a concept which has a production that tells us something about the top-most (non-terminal) element of the W stack. Suppose we choose

\[ C_{36} \rightarrow /\text{variable/ IS / identifier/} \]

\[
C_{23} \
\vdots \
/\text{variable/} \
/\text{arithmetic expression/} \
\vdots
\]

Then the stacks would look like:

<table>
<thead>
<tr>
<th>C Stack</th>
<th>W Stack</th>
<th>T Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/identifier/</td>
<td>LET</td>
</tr>
<tr>
<td>C35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C23</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/arithmetic expression/</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At some point in the processing the student will encounter a concept which has a relatively high weight, meaning that the student has had trouble with it before. Instead of copying the right side of that concept's production onto the W stack, the program copies the right side of a production belonging to one of the concepts in the current concept's error list. This fact is noted on the C stack.

Suppose that later in the processing the top of the W stack contained/arithmetic primary/, and instead of an appropriate match we choose from the appropriate error list

\[ C_{24} \rightarrow /\text{logical expression/ IS /arithmetic primary/} \]

\[
C_{23} \
C_{35} \
C_{31} \
C_{41} \
C_{35} \
C_{37} \
C_{35} \
C_{23} \
C_{28} \
C_{25} \
C_{37} \
C_{35} \
C_{23} \
\]

The system prints

\[ \text{WHAT IS WRONG WITH THIS /assignment/?} \]

\[ \text{LET B40 = A20 < 7.} \]
Note that /assignment/ is the left side of the production of the bottom concept in the C Stack (C23).*

If the student’s answer looks something like

/logical expression/ SHOULD HAVE BEEN /arithmetic expression/,

i.e., if it names expressions which pinpoint where the error was made, the program accepts it. If the student’s answer includes expressions found higher in the C Stack than the ERROR flag, for example, /relation/ SHOULDN’T BE THERE, the program responds:

YES, BUT CAN YOU BE MORE SPECIFIC?

Otherwise the program limits the productions which caused the error and any others in the C Stack that the student might want to see.

We have not yet settled on the best way to locate an error nor on the best way to update the weights of those concepts used in generating a question.

Phase 3 is dedicated to teaching the student how to write small, meaningful programs. It will begin with concepts which describe how the various BASIC statements alter the variables referenced within them. The next batch of concepts will introduce bits of programs such as “short routine”, “sum an array”, etc. Finally, the highest concepts will be oriented toward a canonical representation of a few simple programs. Thus, if all goes well, at the end of his relationship with the system the student will be asked questions about what specific, complete programs do.

CONCLUSION

We have outlined our initial approach to the problem of creating a universal teaching program. The fundamental ideas are similar to the theorem proving approach, i.e., the technique used by question-answering systems, but runs “backwards,” generating questions which test the student’s ability to manipulate the basic concepts of the topic he is to learn. We have given an example which, while extremely simplified from the general case, demonstrates a sophisticated capability when compared to present day teaching programs. It remains to be seen if the techniques we devise can be generalized sufficiently to accept concepts derived from diverse topics.

The parameters used to evaluate such a system are access time, CP (Central Processor) time, I/O (Input/Output) time for Control Data Corporation 6000 machines, mass storage, and core requirements. We believe that all four can be cut drastically by utilizing the random access capability of the Control Data Corporation 6400 on which our program is being implemented. That is to say that the system will be maintained on disc. Since the selection of concepts is inherently random, this approach seems intuitively correct. The elimination of chaining, i.e., searching for data sequentially, will minimize access time, and CP time will be reduced by holding data manipulation to an absolute minimum. Since the system will reside on mass storage units, core capability will be greatly enhanced.

The principal advantage of UNITEAM over other systems (PLANIT, PLATO, ETC.) is that UNITEAM is the most adaptive program yet devised. Certainly the ultimate C.A.I. will include verbal communication and pictures, as does PLATO. PLANIT can recognize responses in a number of ways which UNITEAM at its present stage of development cannot.

PLANIT can derive data from several sources in order to provide a basis for making instructional decisions. This examination of various sources of data and records is incorporated into UNITEAM to formulate the weights (UTILES) of the value of various topics to be presented to the student.

1. Student performance on a question;
2. Cumulative student performance;
3. Student entry characteristics;
4. Student preference.

Student performance includes accuracy of response, time of response, etc. Student entry characteristics include the ability and educational background. It is felt that UNITEAM should make use of the prior knowledge of an individual, including such things as grades, IQ, interests, teacher ratings, etc.

Feedback is a major factor in determining which material and at what depth that material should be presented next, thus enabling the program to adapt to the needs of the individual. All previous feedback including the most recent information in one sense is reevaluated at each step since UNITEAM is based on stochastic methods. Although UNITEAM bears some Markov chain similarities, it is more complex, and thus final evaluation will rest with performance of students rather than mathematical analysis. Thus, total examination of feedback determines to a large degree which direction one next proceeds.

* The concept numbers correspond to a list we have used but which is not included in this paper.
At the present time feedback consists of:

1. Student records;
2. Evaluation of performance;
3. Number of prompts, hints, cues needed;
4. Elaboration on a student’s response as a means of reinforcing a concept.

Unlike other C. A. I., UNITEAM employs Monte Carlo methods in addition to a weighting scheme associated with topics, concepts, etc. This system provides a completely non-linear capability and innumerable paths of deductions as UNITEAM develops necessary logic at each stage or step in the question-answering process. UNITEAM is capable of providing an almost unlimited number of pathways through the instruction because its responses are dependent upon each individual student who is making his own decisions, and it is very unlikely that any two students would choose the same path through UNITEAM.

APPLICATION OF UNITEAM

The system design of UNITEAM envisions the following to be provided by the instructor:

1. Text or subject material (data) to be punched on cards;
2. List of key words which students are to know;
3. Table of weights (utiles) for various topics and concepts;
4. Table of yes-no and true-false questions.

Instructors are not required to know anything about programming or how UNITEAM works. They must know how to present material, and if they wish, UNITEAM will handle the weights of all topics and concepts equally by default. The table of key words provides ready access to the determination of important concepts.

At the current stage of UNITEAM’s development, response time is not a problem. This, of course, is because at its present stage of development, the question-answering is one of essentially true-false or yes-no. Development is centering on multiple choice responses, and definitions at this time, and the response time problem is appreciated as the question-answering becomes of a more complex nature. Disc storage rapidly becomes a problem, as we envision the capability of UNITEAM to essentially control access to all knowledge about a particular course being taught that an instructor might like to include. For the controlling system (UNITEAM) disc storage is not a problem, but the course material (data) rapidly can become unwieldy. With the development of such hardware as Precision Instrument Company’s trillion-bit laser mass memory, one might hope that this problem will soon be eliminated or at least greatly reduced.

REFERENCES

1 S L PRESSEY
   A simple apparatus which gives tests and scores—and teaches
   In School and Soc Vol 23 1926 pp 773-776
2 B F SKINNER
   Teaching machine
   In Science Vol 128 1958 pp 969-977
3 C C GREEN
   The application of theorem proving to question-answering systems
   Information Service 1970
4 J R SLAGLE
   Experiments with a deductive, question-answering program
   In Comm ACM Volume 8 1965 pp 792-798
5 J D BECKER
   An information processing model of intermediate level cognition
   In Stanford Artificial Intelligence Project Memo AI-119
6 D E KNUTH
   The art of computer programming—Volume I—Fundamental algorithms
   Chapter 2 Addison-Wesley Publishing Company Reading Massachusetts 1968
7 C C GREEN
   The application of theorem proving to question-answering systems
   Information Service 1970
8 F R A HOPGOOD
   Compiling techniques
9 D E KNUTH
   The art of computer programming—Volume I—Fundamental algorithms
   Chapter 2 Addison-Wesley Publishing Company Reading Massachusetts 1968
10 J R SLAGLE
    Experiments with a deductive, question-answering program
    In Comm ACM Volume 8 1965 pp 792-798
11 D MAURER
   Introduction to programming
   Stanford University Press 1967
12 F R A HOPGOOD
   Compiling techniques
13 T H NAYLOR J L BALINTFY D S BURDICK K CHU
   Computer simulation techniques
   John Wiley and Sons Inc New York 1966 Chapters 3 and 4
14 J M HAMMERSLEY D C HANDSCOMB
   Monte Carlo methods
   John Wiley and Sons Inc New York 1964
15 PLANIT author's manual
Systems Development Corporation Sections I and II 1971

16 A D CALVIN
Programmed instruction
Indiana University Press Bloomington Indiana 1969

17 P NAUR et al
Revised report on the algorithmic language ALGOL 60
In CACM Volume 6 page 1 1963

18 J G KEMENY T E KURTZ
Basic programming
John Wiley and Sons Inc New York 1960

19 F R A HOPGOOD
Compiling techniques
pp 4-10 52-54

20 D E KNUTH
The art of computer programming—Volume I—Fundamental algorithms
Addison-Wesley Publishing Company Reading Massachusetts 1968 pp 234-238

21 PLANIT author's manual
Systems Development Corporation 1971 Sections I and II