How's that for name dropping?

Courier Terminal Systems name seven of the top ten Fortune 500 companies among their users. IBM speaks for itself. Add the name Plessey Data Systems and you have, in Australia, a unique combination. A combination providing the finest in equipment, versatility of application, and the sound service back-up of the Plessey organisation.

In Courier you are offered a versatile family of intelligent communications devices designed to give you more systems availability, additional features and better performance. Outstanding data communications at a cost that meets the most stringent budget requirements.

Courier. The name to call is Plessey Data Systems. Or post the coupon now.

Plessey Data Systems
122 Arthur Street
North Sydney NSW 2060 Australia
Telephone (02) 929 8299 Telex AA21339
We'd like to know more of Courier, IBM 3270 compatible, from Plessey. Please send us more information.

Name ____________________________
Title ______________________________
Company __________________________
Address ____________________________
P Code ____________________________
Contents:

92-100 Flowchart Optimisation in COPE a Multi-Choice Decision Table
   By B. DWYER and K. HUTCHINGS

101-106 A Course About Social Implications of Computers
   By G.K. GUPTA

107-111 Job Scheduling by a Front End Computer
   By B.J. AUSTIN

112-113 A Computer Program for Exploring the Homogeneousness of Randomly Sampled Categorical Data
   By J.P. HIGGINS

114-117 Batch Submission, Demand Terminal and "Hands-on" Computing in a Computer Science Tertiary Education Environment
   By R.A. JARVIS and P.N. CREASY

118-127 Artificial Intelligence Techniques in Computer Assisted Instruction
   By J.A. SELF

Short Communication:

128-129 Attribute Contributions to a Classification
   By G.N. LANCE and W.T. WILLIAMS

Special Features:

100, 106, 111, 129 Book Reviews

130 Forthcoming National and International Events

SUBSCRIPTIONS: All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc., P.O. Box 640, Crows Nest, N.S.W. 2065.

PRICE TO NON-MEMBERS: There are now 4 issues per annum. The price of individual copies of back issues still available is $1.50. Some are already out of print. Current issues are available at $3.50 per copy. All of these may be obtained from the National Secretariat, P.O. Box 640, Crows Nest, N.S.W. 2065. No trade discounts are given, and agents should recover their own handling charges. Special rates apply to members of other Computer Societies and applications should be made to the Society concerned.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining part-way through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $1.50 per copy. Members should write to the Honorary Secretary of their Branch about their requirements. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly of any change of address.

REPRINTS: 50 copies of reprints will be provided to authors. Additional reprints can be obtained, according to the scale of charges supplied by the publishers with proofs. Reprints of individual papers may be purchased for 50 cents each from the Printers (Publicity Press).

PAPERS: Papers should be submitted to the Editor; authors should consult the notes published in Volume 9, pp. 45-49 (or request a copy from the National Secretariat).

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: P.O. Box 446, Canberra City, A.C.T. 2601. N.S.W.: P.O. Box N250, Grosvenor St, Sydney, N.S.W. 2000. Qld: Box 1481, Rockhampton, Qld. 4701. S.A.: Box 2423, G.P.O., Adelaide, S.A. 5001. W.A.: Box F320, G.P.O., Perth, W.A. 6001. Vic.: P.O. Box 98, East Melbourne, Vic. 3002. Tas.: P.O. Box 216, Sandy Bay, Tas. 7005.

The Australian Computer Journal, Vol. 9, No. 3, September 1977
Flowchart Optimisation in COPE

a Multi-choice Decision Table

By B. Dwyer* and K. Hutchings*

The operation of a processor is described which will convert decision tables to COBOL programs. The notation for decision tables has been extended to allow the expression of arbitrary flowcharts. Multi-choice entry is presented as a convenient alternative to extended-entry notation.

The paper describes the conversion of a decision table to a feasible flowchart, the subsequent transformation of the flowchart to minimise storage required by the program, and selection of the sequence in which code is generated.

Keywords: decision tables, flowcharts.
CR categories: 3.50, 4.19, 4.22, 4.49, 5.24, 8.3

1. INTRODUCTION
COPE is a COBOL decision table processor with a number of novel and interesting features. The main features are:

i. It is procedure oriented. It is capable of expressing any arbitrary flowchart as a table, by listing its sub-graphs.

ii. The "extended-entry" notation has been rationalised and generalised, by the introduction of a "multi-choice" notation.

iii. The method which it uses to optimise its internally generated flowcharts is of general use, aside from its application in decision table processing.

iv. It tackles the ambiguity problem in a novel way, by determining the relative rank of conflicting rules. One merit of this approach is that the "else rule" is no longer a special case.

v. The syntax includes features for test program generation.

This paper will not describe COPE syntax or semantics in detail, as they are described elsewhere (Dwyer 1977). Rather, we wish to demonstrate some of the more interesting features of the processor, and especially to outline the algorithms it uses to convert decision tables to COBOL source programs.

The form of this demonstration will be to start with a flowchart, express the flowchart as a table, and then show how COPE would convert this table into a program with a flowchart at least as good as the one that started the demonstration. Along the way, we shall make some small excursions to examine difficulties not illustrated by the basic example.

We shall explain only sufficient syntactical features to clarify the example problem.

2. THE EXAMPLE FLOWCHART
The example we shall use is a flowchart which is familiar to many readers, the matching of a master and update file using the "high/low/equals" technique. The flowchart is shown in Figure 1.

The correctness of the flowchart relies on suitable

Figure 1: The original flowchart

*South Australian Institute of Technology, North Terrace, Adelaide, S.A. 5000. Manuscript received 27th July 1976 and in revised form 9th June 1977
The Australian Computer Journal, Vol. 9, No. 3, September 1977

Flowchart Optimisation in COPE

3. CONVERTING THE FLOWCHART TO A DECISION TABLE

Conversion of the example to a decision table begins with identifying a sufficient set of the flow paths in Figure 1. This requires the programmer to assign labels to certain nodes of the flowchart and list the details of the paths between labelled nodes. He must ensure that the set of paths includes every connection at least once.

The assignment of labels is to some extent arbitrary, and he could label every box on the flowchart. A minimum requirement is that he labels the entry-point(s) and at least one point in each loop (otherwise some paths will be non-terminating).

In the case of Figure 1, the two nodes labelled 1 and 2 are sufficient.

As a result of this labelling he may describe Figure 1 by means of the five partial graphs shown in Figure 3a. Each of the partial graphs may be considered as a processing "rule".

The graphs may be expressed by the programmer as the decision table of Table A, which is the form in which he would present it to the COPE processor.

The following notational points need explanation:

i. The first line of the table is a heading, the remaining lines constitute 9 "rows".

ii. The first and last rows of the table, the GROUP and NEXT GROUP rows, indicate the labels and label connectors.

iii. The rules, or flowpaths, run vertically downwards. An action box is indicated by an "X", and a decision box by its exit path ("Y" or "N").

iv. Where a row contains numerical values it is a multi-choice row. The value of a particular choice is obtained by substituting parameters for paired periods (. .). The number indicates the position of the parameter. For example, for the third entry in the fifth row, a three, the action selected is "PERFORM MATCHED-PAIR".

v. A hyphen indicates an empty, or unspecified entry.

The table so obtained is not the only possible one. A different choice of labels would have led to a substantially different table. Also, the same table could be expressed with minor differences in notation. For example, each of the multi-choice rows could have been expressed as a series of limited-entry rows.

There is no requirement for a COPE table to have all conditions preceding all actions; the sequence simply follows the sequence of boxes on the flow paths. Conventional decision tables, in which conditions must precede actions, are a subset of COPE tables. However, with appropriate labelling, any flowchart can be reduced to decision table format, as in the case of this example.

To interpret the GROUP and NEXT GROUP rows in conventional decision table terms, we may regard GROUP as a pseudo-condition which tests the value of the current label, and NEXT GROUP as a pseudo-action which sets it (i.e. a form of GO TO operation).

4. RANKING OF RULES

The programmer can always leave one exit point from a decision box unmarked, because decision boxes are logically exhaustive. For example, he may leave the "greater than" (>) branch of the first condition unmarked, since it is "greater than" by implication from the other two conditions. Likewise he may omit the "Y" label from the second condition. The resulting table is given in Table B.

In one way, Table B is preferable to Table A, because it does not indicate entries for conditions which need not be evaluated. But in order to interpret the fifth rule of Table B, it is necessary to understand that it does not hold irrespective of the relationship between UPDATE-KEY and MASTER-KEY, but only if Rule 2, 3 or 4 does not apply. Likewise the third entry in Rule 4 is to be interpreted in the light of Rule 3.

Consequently, selection of rules is made according to the following criteria:

i. At any one time, one and only one rule will be selected.

ii. All conditions specified in the rule must be satisfied.

iii. If more than one rule is satisfied, the highest ranking rule will be selected.

iv. One rule is higher ranking than a second rule if in the first row in which they differ, the first specifies a condition where the second does not, or the first rule has a lower valued numerical entry.

TABLE B: The example, showing implied conditions.

<table>
<thead>
<tr>
<th>123000 FILE-MERGE</th>
<th>NOTE DECISION TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>020#   1 2 2 2 2</td>
<td>GROUP</td>
</tr>
<tr>
<td>040#   - 1 2 2 3</td>
<td>IS UPDATE-KEY .. MASTER-KEY.</td>
</tr>
<tr>
<td>060#   - - N Y</td>
<td>&lt;=,</td>
</tr>
<tr>
<td>080#   X - - - -</td>
<td>OPEN INPUT MASTER, UPDATE.</td>
</tr>
<tr>
<td>100#   1 2 3 4 5</td>
<td>PERFORM ... INITIALIZE,</td>
</tr>
<tr>
<td>120#   1 2 3 4 5</td>
<td>MATCHED-PAIR,</td>
</tr>
<tr>
<td>140#   1 2 3 4 5</td>
<td>UNMATCHED-UPDATE,</td>
</tr>
<tr>
<td>180#   1 2 3 4 5</td>
<td>FINALIZE,</td>
</tr>
<tr>
<td>200#   X - X - X</td>
<td>READ MASTER, AT END MOVE</td>
</tr>
<tr>
<td>220#   X X X - -</td>
<td>READ UPDATE, AT END MOVE</td>
</tr>
<tr>
<td>240#   X X X - -</td>
<td>HIGH-VALUE TO UPDATE-KEY.</td>
</tr>
<tr>
<td>260#   X X X - -</td>
<td>X MATCHED-PAIR,</td>
</tr>
<tr>
<td>280#   X X X - -</td>
<td>UNMATCHED-UPDATE,</td>
</tr>
<tr>
<td>300#   X X X - -</td>
<td>NEXT GROUP.</td>
</tr>
<tr>
<td>320#   2 2 2 2</td>
<td></td>
</tr>
</tbody>
</table>

NOTE DECISION TABLE.
Flowchart Optimisation in COPE

This last stipulation ensures that conditions such as
1 2 3 – AGE > ... 65 60 18.
Are tested in the correct sequence, and not as follows:
IF AGE > 18 GO TO RULE-3.
IF AGE > 60 GO TO RULE-2.
IF AGE > 65 GO TO RULE-1.
GO TO RULE-4.
In which, no value of AGE could result in Rules 2 or 1 being selected.
Since, in general, there is no way in which the COPE processor can determine the correct sequence, the correct ordering of a multi-choice condition is the responsibility of the programmer.
Note that the ranking of rules is dependent on their entries, not on the order in which they are written. But in Table B the rules have been written in rank order.

5. ELSE RULES
The fifth rule of Table B has no condition entries, except for the GROUP row entry. It is therefore the lowest ranking rule in Group 2. It always applies, but is only selected if no other rule applies. It therefore has the status of an “else rule”. Likewise Rule 4 is an “else rule” of a kind, serving the sub-table in which UPDATE-KEY = MASTER-KEY.
In the example, there is no particular advantage in using implied conditions. But in many problems, implied conditions can lead to a more concise expression of the rules. In particular, where many flows join into a common path, the common path can be expressed as a single else rule.

6. LOGICAL COMPLETENESS
Both Table A and Table B are logically exhaustive because at any one time, at least one rule is bound to apply. But this is not transparently obvious from Table A. In the second row of the table, three choices are presented; <, =, or >. It is obvious from the meaning of the symbols that the alternatives are mutually exclusive. But, in general, it would not be possible for the COPE processor to determine, out of context, if a set of alternatives is logically exhaustive. Instead, the assumption is made that if a condition entry has no lower-ranking alternative it must be meant to exhaust the logical possibilities. It can therefore be treated as a comment.
In the case that the programmer has made an error, and the table is logically incomplete, this assumption would be false, so a warning diagnostic is issued by the processor.

7. RANKING OF ROWS
Many decision table processors manipulate the sequence of condition rows in order to obtain a more efficient program. A review of these techniques can be found in Pooch (1975). But this manipulation can only be done safely if the conditions are independent. That is to say, if the correct evaluation of one condition does not depend on the truth of another. King and Johnson (1975) describe a scheme for indicating dependencies which can be used to control the optimisation process.
The assumption made by the COPE processor is simpler but less flexible; the order of the rows cannot be manipulated. This throws extra responsibility on the programmer to ensure that the sequence he specifies is both logically correct and can lead to an efficient flowchart.

In defence of this approach, the authors have found that in typical uses of COPE, dependencies between conditions are common, and the gains which might be obtained from manipulating the sequence are marginal.
In many cases, the flowchart optimisation process we shall describe will remove some inefficiencies which result from a poor sequence of condition rows.

8. COMPONENTS OF THE PROCESSOR
The COPE processor consists of the following major functional modules:
i. The syntax analyser
ii. The flowchart generator
iii. The flowchart optimiser
iv. The code generator.
The purpose of the syntax analyser is to detect the occurrence of decision tables in the source program, check their format, and to build tables of their contents. The table entries are stored in a matrix, and the conditions and actions stored in a separate list. Only the matrix is used to generate and optimise the flowchart.
The flowchart generator transforms the table entry matrix into a feasible flowchart, expressed as a network structure.
The flowchart optimiser operates on this network by applying transformations which reduce the total number of nodes. The less nodes there are in the final flowchart, the less storage will be needed for the generated program.
Finally, the code generator produces a COBOL program from the flowchart, inserting appropriate procedure-names and GO TO instructions, and substituting parameters in multi-choice rows.
We will not discuss the syntax analyser further.

9. REPRESENTATION OF THE FLOWCHART
The entries in the decision table are stored in a matrix. Each cell of the matrix contains the following information.
i. The value of the entry, essentially as written in the table.
ii. A link to another cell called the “left successor”.
iii. A link to another cell called the “right successor”.
iv. An additional link called the “thread link”.
v. Various flags.
Eventually, each cell of the table which will become part of the final flowchart becomes linked into a network structure. If the entry represents a condition, the left successor link indicates the node which logically follows it on the flowchart if the condition is true, and the right successor, the node which follows if it is false. If the entry is an action or a label, the left successor link indicates the (only) logically following node. (See Figure 2.)
All those cells of which a given node is a left or right successor are called its predecessors.
The thread link is used to link, in various ways, all those nodes which currently are included in the network, into a simple sequence.

10. FLOWCHART GENERATION
The strategy in flowchart generation is to begin by considering the highest ranking rules. In this way, the meaning of each implied condition will be clear. Since specific condition entries will be dealt with first, the
Flowchart Optimisation in COPE

Figure 2: The table matrix before optimisation

implied condition is the residue, i.e. not the conditions specifically tested.

To simplify this process the processor transposes the rules, arranging them so that if one rule is to the right of another, it is also lower ranking. It can then begin with the left-most rule and work systematically to the right.

The next step is to determine the left and right successor of each node in the flowchart. Not every cell in the table will require to be represented by a node, and the development of the flowchart and determination of successors proceed simultaneously. When this process is complete, the flowchart is ready for optimisation.

This method will be seen to be analogous to the rule-mask technique of Kirk (1965), except that the rules are scanned at processing time and converted to a decision tree, rather than scanned at execution time.

A disadvantage of our method is that it is possible to contrive logically consistent decision tables in which the scanning of the rules by rank conflicts with the desired row by row sequence of condition evaluation. This is resolved by the COPE processor in favour of rule ranking, the order of row evaluation is adjusted, and a diagnostic message is issued. In such cases the decision tree generated can be inferior to that generated by Pollack's least-storage algorithm (Pollack 1965). Decision tables which cause this problem cannot be obtained directly from flowcharts, and seem to be rarely specified in practice. Since the authors wished to retain the advantages of resolving ambiguities by rule ranking and honouring the programmer's choice of condition evaluation, we consider our method satisfactory, especially as it is open to the programmer to resolve the conflict between rule and row ranking by correcting the table.

Furthermore, Pollack's algorithms are not consistent with a convention that represents dependencies between conditions by the sequence in which they are written. It is possible to devise examples where the sequence of condition evaluation, as adjusted by Pollack's algorithm, would not be safe. We acknowledge, however, that a synthesis of Pollack's and our approaches could be attempted, and could sometimes give better results.

10.1 Sorting for Rank

Each rule is considered as a string of entries. The rules are sorted according to the collating sequence; "Y", "N", "X", 0 through 99, hyphen. The resulting order satisfies the requirements for relative rank.

10.2 Determining the Left Successors

For the most part, the left successor of a condition, or action, is simply the next non-blank entry vertically below it in the same rule.

There are two exceptions. First, if the entry is an "N", the entry below will be the right rather than the left successor, because it follows logically if the condition is false, rather than true. However, it is simpler to treat the link temporarily as a left successor, and change the connection later.

The second exception is when the next entry in the rule is an entry in the NEXT GROUP row, because this is a connector, rather than an actual node. Temporarily however, the NEXT GROUP entry is designated as the successor. If a rule has no NEXT GROUP entry (i.e. a hyphen) the successor of the last entry in the rule is the exit point of the table. Its left successor link is set to a null value.

At this point, the partially structured flowchart is the same as the set of partial graphs shown in Figure 3a. Clearly

Figure 3a: The processing rules

The Australian Computer Journal, Vol. 9, No. 3, September 1977
left successor of the matching GROUP entry. This node is always the left successor of each connector to its corresponding label.

The appropriate place of connection is the node associated with the matching label. A left-right scan of the GROUP row locates the proper rule. The left successor link of the NEXT GROUP entry is redirected to the left successor of the matching GROUP entry.

There are a number of pathological cases which must be handled by the algorithm.

i. There is no matching GROUP entry. This is simply a coding error.

ii. The successor of the GROUP entry is the exit-point. This is handled by copying the null link.

iii. The successor of the GROUP entry is a NEXT GROUP entry. This would not resolve the linkage, since it represents a connection to another connector. The search must therefore iterate. A loop can be detected by observing that any valid number of iterations cannot exceed the number of rules.

After these operations, the partially developed flowchart of Table A would resemble Figure 3b.

10.4 Determining Right Successors

The objective now is to merge the subgraphs of Figure 3b so that each of the flows within a group begins from a common point (the entry-point). The decision boxes of the subgraphs must be merged into the decision tree of Figure 1. The flow of control when the condition test is satisfied has already been established as the left successor link. It is now necessary to determine the right successor link for each condition test, i.e., where flow must pass if the condition fails.

The appropriate branch target cell for a condition entry will always be in the left-most rule which satisfies the necessary conditions. (This follows from the left-right ordering of the rules by rank).

The conditions which a branch target rule must satisfy are that in the row itself it must contain a different entry, and that in all preceding rows it must contain equal entries. Or, if any such row is a condition row, either rule may contain a hyphen, which can be implied to be equal. (As a corollary, an "else rule" will always satisfy the conditions).

The appropriate cell is that found in the first row in which the rules have different entries. In any table derived from a flowchart, this will be the same row as the branching cell. If this is true, and the branch target cell contains a hyphen, or the row is a limited-entry (Y/N) condition, the target cell condition will not require evaluation, because the logical alternatives are exhausted. The right successor of the branch must then be equated to the left successor of the target cell. (Which is why the processor assigns left successor links to empty cells).

It may be that there is no rule which satisfies the conditions for containing the right successor cell. This happens in Table A when the processor attempts to find a right successor for the cell in Rule 5, Row 2 (UPDATE-KEY > MASTER-KEY). As mentioned earlier, this test is redundant. The processor equates the right successor to the left successor, and issues the warning diagnostic. This cell will be eliminated during subsequent optimisation.

Determination of the right successors proceeds in parallel with generation of the decision tree. Once it has been determined that a cell is the right successor of some node, it is the root of a sub-tree which must be subsequently elaborated.

Because the decision-tree is in general not a tree, but a network, care must be taken to elaborate each sub-tree only once. This is achieved by flagging each target cell when it is assigned. The elaboration of the sub-tree is deferred until all rules to its left have been examined for possible branches to it (i.e. Right successors are assigned rule by rule, from left to right).

Not every entry in the original decision table will necessarily be visited during the generation of the network, as the purpose of the process is to merge nodes of the partial sub-graphs. A node which occurs once in the original flowchart, but is part of more than one sub-graph, will be visited only once.

Figure 3b: Linkage of connectors

There is a simple correspondence between the decision table rules and the partial graphs.

The left successor links are set by a bottom-up scan, rule by rule. It is also useful at this time to set a left successor value into each empty cell of the matrix, pointing to the next non-empty entry in the rule.

10.3 Resolving GROUP Entries

The purpose of this step is to resolve the linkage from each connector to its corresponding label.

Each label, by convention, is potentially an entry-point to the flowchart. The entry-point is associated with the first node in the left-most (highest ranking) rule of its group. This node is always the left successor of the left-most GROUP entry with the particular value.

Each NEXT GROUP entry represents a connector. The appropriate place of connection is the node associated with the matching label. A left-right scan of the GROUP row locates the proper rule. The left successor link of the NEXT GROUP entry is redirected to the left successor of the matching GROUP entry.

Figure 1. The flow of control when the condition test is satisfied has already been established as the left successor link. It is now necessary to determine the right successor link for each condition test, i.e., where flow must pass if the condition fails.

The appropriate branch target cell for a condition entry will always be in the left-most rule which satisfies the necessary conditions. (This follows from the left-right ordering of the rules by rank).

The conditions which a branch target rule must satisfy are that in the row itself it must contain a different entry, and that in all preceding rows it must contain equal entries. Or, if any such row is a condition row, either rule may contain a hyphen, which can be implied to be equal. (As a corollary, an "else rule" will always satisfy the conditions).

The appropriate cell is that found in the first row in which the rules have different entries. In any table derived from a flowchart, this will be the same row as the branching cell. If this is true, and the branch target cell contains a hyphen, or the row is a limited-entry (Y/N) condition, the target cell condition will not require evaluation, because the logical alternatives are exhausted. The right successor of the branch must then be equated to the left successor of the target cell. (Which is why the processor assigns left successor links to empty cells).

It may be that there is no rule which satisfies the conditions for containing the right successor cell. This happens in Table A when the processor attempts to find a right successor for the cell in Rule 5, Row 2 (UPDATE-KEY > MASTER-KEY). As mentioned earlier, this test is redundant. The processor equates the right successor to the left successor, and issues the warning diagnostic. This cell will be eliminated during subsequent optimisation.

Determination of the right successors proceeds in parallel with generation of the decision tree. Once it has been determined that a cell is the right successor of some node, it is the root of a sub-tree which must be subsequently elaborated.

Because the decision-tree is in general not a tree, but a network, care must be taken to elaborate each sub-tree only once. This is achieved by flagging each target cell when it is assigned. The elaboration of the sub-tree is deferred until all rules to its left have been examined for possible branches to it (i.e. Right successors are assigned rule by rule, from left to right).

Not every entry in the original decision table will necessarily be visited during the generation of the network, as the purpose of the process is to merge nodes of the partial sub-graphs. A node which occurs once in the original flowchart, but is part of more than one sub-graph, will be visited only once.
Elaboration of the network starts from the successor of the left-most GROUP entry, and normally incorporates each rule into the tree. It is possible however that some conditions and actions cannot be reached from this principal entry point, but can only be reached from an entry-point of some other group to the right. Consequently, it is necessary to elaborate the network starting from each entry-point in turn. Normally, elaboration from the subsidiary entry-points will quickly terminate, because many of their sub-trees will prove to have been already visited.

When the whole network has been elaborated, the successor links of any "N" entries are interchanged, so that throughout the flowchart, the left successor is always the true branch of a decision box.

At this point the flowchart will resemble Figure 3c. This flowchart is a correct implementation of the rules, and is functionally equivalent to the flowchart of Figure 1.

11. FLOWCHART OPTIMISATION

Compared with flowchart generation, optimisation is a relatively simple process. It is based on three axioms.

i. If two actions are identical and have the same successor, one may be deleted and merged with the other. Any links referring to the deleted cell must be adjusted to link to the one remaining.

ii. If two conditions are identical and have both the same left successor and the same right successor, one may be deleted. Links must be adjusted, as in i. above.

iii. Any condition which has its left successor equal to its right successor is redundant and can be deleted. Any link referring to the cell must be redirected to its successor.

These axioms are applied iteratively to every cell in the network, until no further elimination is possible.

Two conditions or actions are judged to be identical if they are in the same row and contain the same entry. As a result, the search for nodes which satisfy the axioms is best made on a row by row basis. The choice of which of the two nodes to delete has no effect on the resulting flowchart.

An optimum order for scanning the nodes would start from the exit-point of the flowchart and work backwards through the network.

The authors did not favour this approach because it would need to deal with nodes which have multiple predecessors, and would also have to avoid being trapped by loops in the flowchart.

An order which seems more practical to us, and easier to implement, is based on the observation that the predecessors of a node will usually lie above it or to the left in the table. Consequently, elimination of nodes is usually efficient when it proceeds from right to left, bottom to top. If during elimination, the predecessor of an eliminated node does prove to be a node already processed, a further iteration is made. Since another iteration can be needed only if at least one node has been eliminated, this process must terminate.

To speed the optimisation process, all nodes remaining on the tree at the beginning of optimisation, are connected by the thread links into a single list, in right to left, bottom to top order.
The flowchart which would result from optimisation of Figure 3c is shown in Figure 3d. It is isomorphic with Figure 1. It is left as an exercise for the reader to see how the axioms have produced the necessary transformation.

11.1 A Note on the Optimiser

It seems to the authors that the axioms used by the optimiser, although local in character, are capable of producing surprisingly global results. For example, Table C(i) and Table C(ii) are equivalent, because Table C(ii) is a common sub-table in C(i) and the first condition of C(i) is thereby redundant. Table C(i) is therefore reducible to Table C(ii), and this will be achieved by patient transformation of its flowchart using the axioms. Similarly, a much cruder attempt could have been made in the flowchart generator to merge the partial subgraphs when producing an initial flowchart, and the rest left to the optimiser. The processor could have assigned a left and right successor to every cell in the table, without attempting to determine the decision tree. This was not done, because the alternative we have described is more efficient.

These same axioms could be made the basis of a general purpose program optimiser. This would contain three main functions:

i. Derivation of the flowchart from the program.
ii. Optimisation of the flowchart.
iii. Regeneration of the program.

Alternatively, the optimiser could form part of a compiler. A merit of the approach is that programs could be written with a greater emphasis on clarity, without sacrificing efficiency in the use of core storage.

The authors would like to note the close parallel between the flowchart optimisation process and the non-deterministic to deterministic reduction algorithm of finite automata theory. (Nelson 1968).

The performance of the optimiser is limited by the assumption that it may not alter the sequence of conditions or actions in the flowchart.

The authors defend this simplification on the grounds that dependencies between conditions are common, and that programmers are used to expressing these dependencies implicitly as the sequence of the conditions in the program text. On those occasions where conditions are independent and there is the scope for a choice of sequence, the programmer must exercise judgement. He can use his judgement to optimise execution time or storage, in the same way that he does in conventional programming. We felt it preferable to give the programmer control over the code in a familiar way, rather than learn a new technique for expressing dependencies. Indeed much of our thinking in the design of COPE has been aimed at simplifying the learning process and easing the transition to a different way of programming. Programmers like the assurance that the processor will not change the logical sequence of the rows.

There remain certain situations where the processor could still determine automatically that the interchange of two rows is safe, will not affect execution time, and would save storage. These cases have in experience been so rare that we have not considered their treatment worthwhile.

In contrast, it has been found extremely useful to monitor program execution under operational conditions by counting the executions of each rule. This requires the temporary addition of one multi-choice action to each table, and a small amount of code to display the final values of the counters.

The information so obtained highlights those areas where optimisation efforts will be well repaid. Our personal experience is that these are rarely the areas we expected, or that the improvements that suggest themselves are confined to exchanging condition rows within a table.

12. CODE GENERATION

The sequence in which code is generated is largely a matter of taste, although by convention, COPE always begins with the lowest numbered entry-point and finishes with the exit-point. This ensures that the table can be executed either as a PERFORMED subroutine or as an in-line routine.

The sequence we have found best is defined by a right-left pre-order traverse of the flowchart. (Knuth 1973). By "best" we mean two things:

i. The majority of condition tests are of the form IF <condition> GO TO <procedure-name> rather than IF <condition> NEXT SENTENCE ELSE GO TO <procedure-name>.

ii. The sequence of statements generated agrees reasonably well with what a programmer might have written.

This sequence may sometimes create one unnecessary GO TO, because it does not ensure that the last statement generated before the exit-point is necessarily one of its predecessors. This is illustrated by the code generated from Table A, shown in the Appendix. (If the paragraphs FILE-MERGE-1517 and FILE-MERGE-1518 were interchanged, a GO TO could be eliminated).

If a node of the flowchart has more than one predecessor, it can be reached in-line from only one of them, and must be reached from the others by means of GO TO's. We have found that the most attractive code, on average, is obtained by making the last connection generated the in-line connection. This will be the last connection examined in the right-left pre-order traverse.

The method actually used is to make a left-right post-order traverse, stacking the nodes using the thread links. The last connection to be generated will now be the first one traversed, and each node is flagged on its first visit. When a link is found pointing to a node which has already been visited, the link will need to be generated as a GO TO. The node it points to is flagged to show that it will require a procedure-name and its sub-tree is not traversed a second time.

When the thread links are unstacked, the nodes will become available in reverse sequence, which therefore corresponds to a right-left pre-order traverse.

The traverse begins from the successor of the lowest-numbered entry-point. If there are additional entry-points, the stacks of their sub-trees are appended, not stacked, onto the existing stack, to ensure that the lowest numbered entry-point will be generated first.

Any node which is the successor of an entry-point is specially flagged. Entry-points are not treated as flowchart nodes, but as attachments to nodes. They are not included
in the code generation stack. When a node is being generated as part of the COBOL program and has the flag set indicating that it is the successor of an entry-point, the cells in the GROUP row will be examined to determine which of them have the node as a successor, and entry-point procedure-names will be generated as needed.

The stack greatly simplifies code generation. When a successor link is equal to the thread link, no GO TO is necessary to link to the successor, otherwise a GO TO is necessary. At least one of the successors of a condition test (usually the left) will always require a GO TO link.

The COPE processor does not attempt to build more
complex COBOL than conditional GO TO or action statements. This ensures that we do not add so much to the complexity of the programmer’s conditions or actions that they become too complex for the COBOL compiler. The resulting program text may not look very elegant (it bristles with GO TO’s); but it is efficient.

13. CONCLUDING REMARKS

The authors have presented an alternative, more general, view of decision tables, regarding them as descriptions of flowcharts, by enumeration of a sufficient set of sub-graphs. Conventional decision tables are a subset of these tables.

We have shown how a processor can derive a flowchart from a table, by connecting and merging sub-graphs. The resulting flowchart may in some cases result in a program which is more economical of storage than one transcribed directly from the original flowchart.

We have also suggested that optimisation of flowcharts, similar to that described, could be used in applications other than decision table processing.

We have largely shirked the issue of optimising the sequence of condition evaluation, believing that the programmer would rather show condition dependencies implicitly in the order of the rows, than to state them separately in the manner of King and Johnson (1975).

In our opinion, the combination of a procedure oriented approach to decision tables, new syntactical features, and an efficient optimiser, has yielded a powerful and practical tool for COBOL program development.

References


Book Review


This book is a collection of invited papers presented at the Symposium on Analytic Computational Complexity organised by the Computer Science Department of Carnegie-Mellon University in April 1975. Abstracts of other contributed papers are also included. There are important and interesting papers by leading researchers in this area, and I have repeated here the table of contents for the readers information:

Introduction: J.F. Traub.

Some Remarks on Proof Techniques in Analytic Complexity: S. Winograd.


Maximal Order of Multipoint Iterations Using n Evaluations: H. Woźniakowski.


The Use of Integrals in the Solution of Nonlinear Equations in N Dimensions: B. Kacewicz.

Complexity and Differential Equations: M.H. Schultz.

Multiple-Precision Zero-Finding Methods and the Complexity of Elementary Function Evaluation: R.P. Brent.


(Continued on page 106)
A Course About Social Implications of Computing

By G.K. Gupta*

This paper presents a brief description of a course about social implications of computing for B.Sc. honours year Computer Science students. The description includes a list of topics discussed and a partially annotated bibliography of some of the references used in the course.

Key Words: Social issues, computer science education, computers and society.
CR Categories: 1.52, 2.0

1. INTRODUCTION

The last 30 years have seen enormous advances made in computer technology and an exponential growth in the use of computers in the Western countries. This continuing growth in the use of computers is reshaping our technological society. The social implications of the new technology are of crucial interest to the public at large. Evidence of this interest is readily apparent in the time and space devoted to the issue of data banks and privacy in the Australian mass media in the last few months.

Also in the last few years, many members of the computing profession have become increasingly concerned about the impact of computer technology on society. This concern, for example, is reflected by the recent setting up of the IFIP Technical Committee TC9 — "Relationship between Computers and Society" and the formation of a new technical department of "Social Impacts of Computing" in the journal Communications of the ACM. In the field of computer science education this concern is reflected by an ever increasing number of institutions, mostly in North America, introducing courses dealing with the social impacts of computing at various levels. However a look at the computing courses available at the various Australian universities seems to suggest that many computer science departments in Australia do not feel strongly about the inclusion of such a course as part of the computer science education. Based on a survey of the limited information available about the various curricula, the only universities discussing social issues in their computer science curricula are Melbourne, Monash, Sydney and Tasmania. Tasmania and Sydney both introduce social issues in their first year courses which were started in 1975 and 1977 respectively. At Melbourne, social issues is introduced in an optional third year unit on computer applications.

This author feels that it is important that the computer science students of today (who presumably will be the leaders in their field at the start of the 21st century) be exposed to the discussion which is going on in the computing community and society in general about the present and potential social implications of computing. Therefore a course (i.e. a set of lectures/seminars) dealing with the social implications should be included in all tertiary computer science degree programs. This paper briefly describes one such course which has been offered at Monash University since 1975.

2. DETAILS OF THE COURSE

Several courses dealing with the social impacts of computing have been described in the literature. For example, Horowitz, Morgan and Shaw (1972) describe a course suitable for undergraduate and graduate computer science majors while Horowitz and Horowitz (1973) describe a course in which students with computing background as well as those without were encouraged to join. The course described in this paper is designed for B.Sc. Honours year computer science students. In the author's opinion, the Honours year is perhaps the best place to discuss the social issues because the students are better able to appreciate the capabilities and implications of computers only after 2 to 3 years computing experience.

Therefore the objectives of our course are very similar to those of the course described by Horowitz, Morgan and Shaw (1972). These objectives are to educate those students who have chosen computing as their career on the present and future impact of computer technology, to discuss some of the difficult moral questions concerning the responsibilities of the computing professionals, and to get an appreciation of the views of society in general about the use and misuse of computers.

Though the objectives of our course are similar to that of Horowitz, Morgan and Shaw (1972), the organization and presentation of the course described here is quite different. Part of the reason for this difference is that the time available for the course is minimal. Another reason for the difference is that in the last few years the literature available about social issues has increased substantially. Availability of these additional books and papers has made it possible to organize the course around them. A partially annotated bibliography of references which were thought to be important is therefore included.

Given the objectives of the course, it is essential that the course involve a minimum of "lectures" from the lecturer. All students must be given a chance to take part in the discussion. To encourage student involvement the course is organised as follows.

---

*Dept. of Computer Science, Monash University, Clayton, Victoria, 3168, Australia. Manuscript received 30th November 1976 and in revised form 15th April 1977

The Australian Computer Journal, Vol. 9, No. 3, September 1977
In the first lecture, the students are presented with a list of topics expected to be covered in the course. Each student is required to choose some aspect of a given topic, get approval from the lecturer (so that all important topics are covered) and then present a seminar of about half an hour. Each student also writes an essay of about 1500 words discussing the same topic.

Once a topic is chosen, each student is provided with a list of references which he needs to consult. The students are not expected to search for relevant articles in the literature, though some students do, because the time available for the course is very limited and the relative weight of the course is only about 3% of the honours year.

The students are advised that the essay and seminar should attempt to present clearly the issue(s) being discussed and then present a brief survey of some of the literature available on the topic. In addition the students are encouraged to present their own views during the discussion following the seminar. An attempt is made to restrict the discussion to systems which are technically feasible at the present time, therefore avoiding the possibility of Science Fiction fantasizing.

The organisation of the course in 1976 was somewhat different to that in 1975 because of very different class sizes and also due to the changes which were made to the structure of the honours year courses. In the academic year 1975, the honours year course work offerings in computer science at Monash University consisted of 24 units of (usually) 9 (50 minute) lectures each, out of which each student was required to choose 18 units. The social issues course described here was presented as one of these units and was taken by almost all honours year students. In addition several third year students, who were required to do two honours year units, elected to take this unit. The total number of students taking the unit was 18. Due to this larger than average honours class-size, the lecturer presented only the first seminar himself and the rest of the talks were presented by the students. In fact an extra 2-hour session had to be arranged so that all students got the opportunity of presenting seminars.

In the academic year of 1976, the structure of the honours year course was changed and some 9 units including the unit dealing with social issues were made compulsory. Also this meant that the third year students were not allowed to take the units compulsory in the honours year. Due to the absence of the third year students and also because the 1976 honours year class was smaller, the total number of students taking the social implications unit was 7. Due to the smaller class-size, the seminars presented by the students were not able to discuss all the important issues and therefore several half an hour seminars had to be presented by the lecturer himself.

The seminar presented by each student and the essay which he submitted were the basis for assessment. No examinations were conducted in this unit.

3. STUDENT REACTION

No attempt was made to systematically assess the effect of this course on the students. However the feedback received by the author suggests that most students reacted favourably to the course. This is also reflected by the comments made by the 1976 honours year students in a document presented to the department which reviewed the honours year courses. The comment about the social implications unit was "a most stimulating and useful part of the course which could possibly be extended".

4. CONCLUDING REMARKS

On the basis of the experience gained in the last two years, the following comments can be made:

(a) The course should preferably be presented in the first half of the academic year. Towards the end of the academic year the students become too busy in their projects and preparation for examinations and therefore the second half of the year seems not to be suitable for discussion of social issues.

(b) Presence of part-time students and staff members in the class usually results in a livelier discussion.

(c) To cover all the topics presented in the list adequately, the time allotted to the unit should be around 15 lectures.

5. TOPICS

The following list of topics is presented to the students and an attempt is made to discuss most of the topics listed here. Students are also advised to refer to "A Problem-List of Issues Concerning Computers and Public Policy", Comm. ACM, Vol. 17, No. 9, 1974, pp 495-503. This is the most comprehensive list of issues known to the author.

(a) General:


(b) Information services for home use:

The concept of Community Information Utility (CIU). Technical feasibility and capabilities, potential uses and dangers involved in setting up the CIU. Speculation about social effects of the CIU. Control and funding of the CIU.

(c) Computers and Employment:


(d) Computing Profession and the Computing Industry:

Computing profession, professionalism, role of computing societies, code of professional conduct, certification, unionism. The computing industry, dominance of the world market by IBM, continuing litigation in USA and its implications. Computing industry in Australia, Europe and Japan. Government support for the computer industry in Australia. Patents/copyrights of Software.

(e) Computers in Education and Computing Education:

Computer assisted (aided) instruction (CAI) – discussion of some promising systems, e.g. PLATO and TICIT, Role of teacher in CAI. Aims and present state of computing education in high school. Computer literacy. Computer Science education in colleges and Universities. Content of university courses in computing – the Australian scene.
6. LITERATURE
A vast amount of useful literature is now available in the field of social implications of computing. Several books are also available which may be used as textbooks and these are listed in section 6.1 below. The literature is classified according to the topics listed in section 5. Only the important references are listed here. A partially annotated bibliography of about 150 recent references has been compiled and a copy may be obtained by writing to the author.

6.1 General (Including possible text books)

This report of the ACM Committee on Computers and Public Policy is an excellent summary of the issues concerning the social implications of computers. Since this is the most comprehensive listing of the issues, all students were required to read this report.


A very good candidate for text book which discusses most of the issues listed in Section 5. The book also provides technical details about files, simulation etc. for students not exposed to this background material.


One of the earlier books discussing social implications of computing. It is an introductory book which can be used as preliminary reading material for the course.


This book is based on a one-semester course on Computers and Society offered at the University of British Columbia. The book deals with historical perspectives, impact of computers on employment, education and health care. Also discusses issues related to computer utility, privacy, information and the political process and others. A list of about 400 references is presented.


“... The plan for the conference and the selection of papers was motivated by the belief that all levels of society should take stock of the growing use of computers in all walks of life and decide what it wants to do with computers for the benefit of everyone...” from the preface. These proceedings of an IFIP conference held in Vienna in April, 1974 also include a summary of the conference by the editors. The conference recommended to IFIP to encourage the development of an international “computer bill of rights” and it was suggested that this bill should protect the individual from any misuse of computer based systems. Details of this recommendation are included in the editors’ summary.


This report of the ACM Committee on Computers and Public Policy is an excellent summary of the issues concerning the social implications of computers. Since this is the most comprehensive listing of the issues, all students were required to read this report.


This little paperback was written by a New York attorney who has been a legal counsel to AFIPS and ADAPSO. The book is unique in that it not only discusses the social issues but also suggests solutions. It was recommended that all students read this book.

6.2 Information Services for Home Use


The author criticizes the approach of Sackman and Boehm regarding the desirability of establishing a Community information utility (CIU). He argues against the construction of a CIU in a prototype community and advocates establishing a moratorium on the construction of a prototype CIU until the year 2000 (also see discussion on this paper in Comm.
A Course About Social Implications of Computing

6.3 Computers and Employment

1. ARCHER, D.B. (1974), "Computers and Employment" in Mumford and Sackman (1974), pp 199-208. The article discusses the short term and long term consequences of computer technology for the worker. The authors suggest that in the long term the work force will benefit by the use of computers but in the short term, dislocations is the main problem which requires government intervention.

2. FORD, G.W. (1974), "Computers and the Quality of Working Life", Proc. of the 6th Australian Computer Conference, pp 1012-1025. The author discusses issues related to quality of working life of people in the computing industry. Among some of the issues discussed are industrial relations in the computing industry, discontentment and disillusionment among employees of computer organizations due to corporate dishonesty particularly in marketing, the need for technological assessment, the need for new education programs for system analysts, systems engineers and marketing people.


4. WHISLER, T.L. (1970), "The Impact of Computers on Organizations", Praeger Publishers, New York. The book is based on a study of several life insurance companies. It examines questions like: What led to the introduction of the computer? What patterns of computer use and application emerge? The book discusses the impact of computers on organizational structures, on decision-making, on authority and control, on job content. Some very useful information in the form of summary of responses by various companies is presented.

6.4 Computing Profession and the Computing Industry

1. ACM (1973), "Proposed ACM Code of Professional Conduct", Comm. of ACM, Vol. 16, No. 4, pp 262-268. This includes an introduction by the then President of ACM, a report of the Professional Standards and Practices Committee, the proposed ACM code of Professional conduct and proposed ACM policy regarding procedure in professional conduct cases. It is followed by a statement opposing the proposed code of professional conduct by P.J. Denning, G. Glaser, C. Hammer and G. Salton on pp 268-269.


6. FINERMAN, A. (1975), "Professionalism in the Computing Field", Comm. ACM, Vol. 18, No. 1, pp 4-14. The author criticizes the four types of institutions — academic, industry, government and the professional society that educate, employ, regulate and mould the computer practitioner and suggests that these institutions can make improvements to help computer practitioners achieve professional status.


8. MOONMAN, E. (1971), Ed., "British Computers and Industrial Innovation – the Implications of the Parliamentary Select Committee", George Allen and Urwin Ltd., London. In November 1969, the British House of Commons established a Select Committee to consider Science and Technology, and this appointed a sub-committee to examine "the prospects for the U.K. Computer Industry in the 1970s including the possibilities of international collaboration and the functions of government in this field, both as policy maker and user". The main effort of the sub-committee was on the purchasing policy of the government before it was dissolved without producing a report to the House due to the 1970 General election. This book presents a review and evaluation of the massive evidence presented to the sub-committee. The book includes a review of the government support for the hardware industry, a review of the government purchase policy towards software, a review of the subject of telecommunications as it concerns the Post Office, a
review of evidence on computer training and education and an article on structure and competitiveness of the European industry.


The author presents a brief history of the Australian Computer Society and looks at some of its principal aims. The author then criticizes the apathy and indifference of the members of ACS, discusses several other problems facing the ACS and makes recommendations.

### 6.5 Computers in Education and Computing Education


This report of the ACM Curriculum Committee on Computer Science contains the recommendations on academic programs in computer science. These are the first detailed recommendations for curricula in Computer Science.


 Discusses the development of PLATO system and describes PLATO IV and its capabilities.


These proceedings contain several papers of interest regarding computer literacy, computing in high school and computer science education.


The author criticizes the lack of theoretical content in the undergraduate computer science programs in Australian universities, speculates on the reasons for this and makes recommendations.


The author presents the view that techniques and objectives of a computer scientist are different than that of a technologist or a mathematician. The special quality of computer science is illustrated by means of examples from Theory of Programming Languages.

### 6.6 Computers and Privacy


It is a report to the Committee on Scientific and Technical information (COSATTI) of the Federal council of Science and Technology, U.S. Government. Reports cover the following issues

1. the right of entry and access to information systems
2. the freedom of information
3. the right of privacy
4. anti-trust issues
5. property rights
6. copyright issues in the United States
7. the international copyright situation.


This report was prepared by the Ombudsman Committee on Privacy, Los Angeles Chapter of the ACM. The committee formed two sub-committees; one to study the individual's right to privacy and the other to study the security of data processing systems and data. The report includes a data centre security check list, a code of fair information practices and the U.S. Privacy Act of 1974.


The convenor of the Social Implications Committee of the ACS discusses controls and safeguards needed to ensure protection of individuals' privacy. Also briefly discusses the legislative position in Australia and social responsibility of technological experts.


Discusses the developments overseas and in Australia in the field of privacy. Devotes considerable attention to the technical safeguards that need to be implemented to ensure security of data.


A collection of readings used in a graduate course in Berkeley. Most articles are about the technical problems of security and a good knowledge of programming and operating systems is required.


The OECD seminar was held in June, 1974. Five sessions were held which considered

1. Regulatory instruments for Data Protection.
2. The Personal Identifier Issue and Privacy.
3. Right of Citizen Access to their file.
5. Accepting the Costs of Data Security.

Many well known experts in the field have contributed to these proceedings. Also the proceedings include The Swedish Data Act, The U.S. Privacy Act of 1974 and also a good summary of the proceedings.


The author is a member of the U.K. Data Protection Committee. He discusses developments in the field of data banks and privacy and summarises the results of various studies by seven principles which are regarded as the common objectives.


An important comprehensive report which discusses record keeping for various purposes and then makes recommendations to the Department of Health, Education and Welfare and the U.S. Government. (For a summary of the committee's conclusions and
6.7 Computers and Man


The author discusses research work in the fields of Cognitive Simulation (CS) and Artificial Intelligence (AI) and asserts that workers in these fields assume that man functions like a general-purpose symbol manipulating device. The author presents the view that this assumption is the basis of the optimism (by workers in the fields of CS and AI) that proper programming should be able to elicit human-like behaviour from digital machines. The author then discusses the assumption in detail and concludes that it is not justified. The book concludes with a discussion of the future of AI.


The author presents a view that the direct effects of the computer technology which are usually highlighted in the press are not as important as the much more subtle side effects. He asserts that just as microscopes changed the view of disease, the computer will change the way man looks at himself.


... the book contains the arguments, which are in essence, first, that there is a difference between man and machine, and second, that there are certain tasks which computers ought not be made to do, independent of whether computers can be made to do them ... (from author’s preface).

(Reviews of this book by B. Kuipers and J. McCarthy have been published in ACM-SIGART Newsletter, June, 1976).


7. ACKNOWLEDGEMENTS

The author expresses his appreciation to the referees for their comments.

8. REFERENCES


Book Review

Knuth (Knuth, 1973) and Borodin and Munro (Borodin and Munro, 1976) are three interesting books which may be consulted to see the blend of all three mathematical disciplines. This is as it should be, and the science cannot but benefit from the synthesis of previously diverse strands of thought.

This book should be in every university and college library, and probably also on the bookshelf of computer scientists and mathematicians with more than a cursory interest in numerical methods.

Norman Y. Foo
Basser Dept of Computer Science, University of Sydney

REFERENCES


Job Scheduling by a Front End Computer

By B. J. Austin*

The paper describes a scheduler operating in a front end computer (a Control Data 3600) which selects jobs to be sent to the main processor (a CYBER76). The scheduler distinguishes between purely computational jobs and those which will require disc packs or magnetic tapes (and therefore must wait for operator attention). Computational jobs are selected by an algorithm which aims to produce a mixture of many short and fewer long jobs. Jobs requiring demountable media are selected in an empirically determined "mix" which is intended to utilise all types of tape and disc drive fully. The scheduler is cognisant of the state of job execution within the CYBER76 via a single status bit. The operators control the throughput of jobs requiring tapes or disc packs, and they can adjust parameters to cater for cases of one or more drives being temporarily unavailable. In spite of the limited nature of these control mechanisms, the scheduler can cope effectively with the many states of the system normally encountered in a day — heavily loaded prime shift, lightly loaded evening shift and the transition between these states.

Key words: Job scheduling, front end processor and phrases

CR Categories: 3.80, 4.35

Introduction
The Division of Computing Research (DCR) operates a system consisting of a Control Data CYBER70/Model 76 connected to a Control Data 3600. The CYBER76 acts as the main computing power, while the 3600 provides interactive service and batch entry to the CYBER76, both at the central site and remotely through CSIRONET, the C.S.I.R.O. network. The 3600 runs under the DAD operating system which is described in Austin, Holden and Hudson (1967). The connection between the 3600 and the CYBER76 consists of code in a CYBER76 Peripheral Processing Unit (PPU) and in the 3600, together with appropriate hardware couplers, and has been described in Austin, Ewens and Hudson (1976).

The operating system of the CYBER76 (SCOPE) has elaborate scheduling of jobs once they have entered execution, both in regard to the swapping out of quantum-expired jobs and the allocation of assignable resources, but there are virtually no mechanisms for controlling the selection of jobs from the set of candidates for execution. A user may nominate a priority for his job. In SCOPE, highest priority will be used to select a job for initiation, but the scheduler to be described subverts this by restricting the number of jobs awaiting initiation, so that priority becomes meaningless and SCOPE can only select those jobs chosen by the front end scheduler.

It is reported that the Control Data Corporation plans a scheduler in a version of SCOPE soon to be released, and advance specifications of the scheduler would indicate that considerable generality and sophistication will be provided. The proposed improvements to SCOPE are described in a General Integral Design by Lawson (1976). The CERN installation has implemented a scheduler described by Martin (1973), and some other CYBER76 sites have adopted this work. The work described in the current paper represents only a few man weeks of effort, and while it will never equal the performance of a scheduler internal to the main processor, it has enabled the DCR system to perform quite adequately and efficiently for about two years, from its initial implementation in September 1974.

An important parameter set by the operators is the width of multiprogramming — i.e. the maximum number of jobs that may be executed simultaneously. The DCR system is intended to provide rapid turn round to short jobs, especially those submitted by console users, while servicing a background of other jobs, many of which require operator intervention to mount magnetic tapes or disc packs. A job awaiting operator intervention still occupies a multiprogramming slot. The choice of a number of slots becomes a compromise between good response for interactive users combined with many other partially-completed jobs at risk, or erratic response combined with low risk and possible low CPU utilisation.

Prior to the work described in this paper, it was not uncommon for multiprogramming widths in excess of 100 to be used, and the loss of system resources at a system failure was often considerable. Furthermore, the large number of partially completed jobs often caused a failure by exceeding the available mass storage. It is now normal practice to operate the system with a multiprogramming width of 16-24, in which approximately half the jobs are able to proceed, and the rest are awaiting operator action.

Job Classes and Subclasses
The scheduler recognises three classes of jobs. These are:

1. Short computational jobs (up to 8 seconds) submitted by interactive console users. "Computational" implies that no magnetic tapes or disc packs need be mounted by the operators. Disc packs, here and throughout the paper, do not include the system disc packs, nor a number of packs assigned to user groups and left permanently mounted.
2. Computational jobs not submitted interactively, up to a maximum time limit. This maximum is assembled as

"Copyright 1977, Australian Computer Society Inc.

*Division of Computing Research, Commonwealth Scientific and Industrial Research Organisation, Canberra, A.C.T. Manuscript received 30th November 1976 and in revised form 15th June 1977"
Job Scheduling

infinity, but can be set by operator command to lower values, including zero.

(3) Other jobs. This class caters in particular for jobs requiring demountable media. All jobs in this class are initiated under operator control.

Classes 2 and 3 are further subdivided according to job attributes, and a notational scheme is required to represent this division. The number of subclasses and their definitions are established in a general fashion from these tables.

Class 2 is divided into five subclasses 2.0, 2.1, 2.2, 2.3 and 2.4, and where a reference to a general subclass is required, the form 2.1 will be used. Subclass 2.0 contains all jobs with time limit less than or equal to 4096 seconds (infinity), while the limits for 2.1, 2.2, 2.3 and 2.4 are 256, 64, 32 and 8 respectively. Note that a given subclass contains all those with lower time limits.

Class 3 is divided into three subclasses, depending on the demountable media required. Subclass 3-Y caters for jobs requiring disc packs, while 3-N and 3-M are for jobs requiring nine and seven track tapes respectively. Where a general subclass must be referred to, a form such as 3.D will be used. Note that class 3 may contain jobs not requiring demountable media at all, or jobs requiring more than one type of demountable media. A job may be a member of none, one, two or three subclasses. A purely computational job would be considered as class 3 if the time limit mentioned in (2) above was exceeded. In effect, this implies that operator action is needed for the job to be started.

Job Queues

CYBER76 jobs are queued within the 3600 in a common structure used in all file output operations. (The transmission of job files to the CYBER76 is just another case of file output as far as the 3600 operating system is concerned.) Items are placed at the tail of a queue and are then identified by a queue index (QUIN). Items will normally be taken from the head of a queue, but any queue index can be accessed and retrieved.

The CYBER76 job queues are named CY1 and CY2. CY1 is intended to be reserved for an interactively submitted job (class 1), but as a matter of fairness it has been decided also to place re-transmitted files at the end of CY1, regardless of their original queue. Thus a CYBER76 failure during job transmission causes the job to be re-entered into CY1. All other jobs are placed at the tail of CY2, without regard to the SCOPE priority parameter, and without distinction between classes 2 and 3.

When the CYBER76 and the 3600 are in communication, the 3600 will present job files one at a time, for input and subsequent execution. CY1 is always serviced first, and must be empty before CY2 is examined. Furthermore, jobs are taken from CY2 only if free multiprogramming slots exist in the CYBER76. This information is transmitted to the 3600 every few seconds. It constitutes one of the two control mechanisms of the scheduler.

Obviously, one bit of information is a very poor basis for job selection, and further work on the project may result in moving the scheduling process into CYBER76, where many more control bits would be available.*

When all of the above conditions are met, a scan of CY2 will commence. The scan will result in at most one job being selected for transmission, and pointers into the CY2 queue will be updated. These pointers are the means of dividing a serial queue into sections reflecting job classes and subclasses, and the management of the queue pointers is the main theme of this paper.

Items within the CY2 queue are addressed by their queue index values (QUIN). Three primitive operations are available, viz:—

1. Read item at position QUIN, or greater.
2. Delete item at position QUIN.
3. Write item at position QUIN.

Reading is normally done with no QUIN specified, (QUIN = 0), in which case the highest queue index currently in existence is found, and the queue item is returned with an index one greater. This places the item at the tail of the queue.

CYBER76 job queues are named CY1 and CY2. CY1 is intended to be reserved for an interactively submitted job (class 1), but as a matter of fairness it has been decided also to place re-transmitted files at the end of CY1, regardless of their original queue. Thus a CYBER76 failure during job transmission causes the job to be re-entered into CY1. All other jobs are placed at the tail of CY2, without regard to the SCOPE priority parameter, and without distinction between classes 2 and 3.

When the CYBER76 and the 3600 are in communication, the 3600 will present job files one at a time, for input and subsequent execution. CY1 is always serviced first, and must be empty before CY2 is examined. Furthermore, jobs are taken from CY2 only if free multiprogramming slots exist in the CYBER76. This information is transmitted to the 3600 every few seconds. It constitutes one of the two control mechanisms of the scheduler.

Obviously, one bit of information is a very poor basis for job selection, and further work on the project may result in moving the scheduling process into CYBER76, where many more control bits would be available.*

When all of the above conditions are met, a scan of CY2 will commence. The scan will result in at most one job being selected for transmission, and pointers into the CY2 queue will be updated. These pointers are the means of dividing a serial queue into sections reflecting job classes and subclasses, and the management of the queue pointers is the main theme of this paper. Each subclass of classes 2 and 3 has an associated queue pointer.

Items within the CY2 queue are addressed by their queue index values (QUIN). Three primitive operations are available, viz:—

1. Read item at position QUIN, or greater.
2. Delete item at position QUIN.
3. Write item at position QUIN.

Writing is normally done with no QUIN specified, (QUIN = 0), in which case the highest queue index currently in existence is found, and the queue item is written with an index one greater. This places the item at the tail of the queue.

Reading is normally done with a QUIN specified. This will return the lowest item whose queue index is greater than or equal to QUIN. The queue index of the item found is also returned. The head of the queue can be accessed by QUIN = 0, and operations within the body of the queue are facilitated by the read primitive. The queue pointers which define the job subclasses are intended for use with the read primitive, and their definitions may seem somewhat unnatural unless this is appreciated.

Scanning of CY2 is performed in one of two modes, depending on an operator command. Normally jobs within class 2 are sought, and the algorithm steps through the subclasses in such a way as to deliver a mixture of long and short jobs. Note that a given subclass contains all subclasses with lower time limits. The second mode is entered upon receiving a command from the operator, and the scheduler will attempt to input a batch of class 3 jobs, in a mixture appropriate to the processing speed and availability of suitable disc drives or tape transports.

In either scanning mode, the jobs queued in CY2 are examined as possible candidates for execution. It is necessary to access the job file, as well as the CY2 queue, to obtain the job attributes. The parameters which determine selection or rejection are all contained on the first statement of the job file (the so-called Job Identification Statement). These parameters are:—

1. Time limit
2. Disc pack mount limit
3. Disc drive limit
4. Nine track tape mount limit
5. Nine track drive limit
6. Seven track tape mount limit
7. Seven track drive limit

Two of these parameters, numbers (4) and (6), have been added by DCR, while the rest are in the standard SCOPE operating system (see SCOPE 2.1.3 Reference Manual (1975)).

All parameters are optional and have defined default values. The time limit defaults to 8 seconds, while all other parameters default to zero. The pairs of parameters (2), (3); (4), (5); (6), (7) are required to distinguish between the maximum number of operator mounts required by a job and the maximum number of drives required at any one

* Further work by B.J. McHugh has divided the multiprogramming slots (JCBs) into classes and reserves a minimum number of slots for short jobs. The 'available slots' status reported to the 3600 now refers only to the remainder.
Job Scheduling

The normal scan procedure is controlled by an integer COUNTER (initial value zero) and an array of queue index values, MAXQUIN. The search for jobs within subclass 2.1 is dependent on the pointer MAXQUIN(I), which is defined as the queue index of the last job rejected when a job of subclass 2.1 was being sought. The next such scan will commence from MAXQUIN(I) + 1. Since each subclass contains all subclasses corresponding to lower time limits, the pointers MAXQUIN(I), I = 0, 1, 2, 3, 4 are made to form a non-decreasing sequence by adjusting MAXQUIN(I), I = J + 1,..., 4 whenever MAXQUIN(J) is updated as part of the scheduling process.

The scheduler selects subclasses as follows:—

1. COUNTER := COUNTER + 1
2. M := MSB (XOR (COUNTER, COUNTER + 1))
   where MSB = Function yielding the most significant bit position and XOR = Bit by bit Exclusive OR
3. I := MAX (0, 4 - M)

A search for a job of subclass 2.1 is then commenced at position MAXQUIN(I) + 1. It should be remembered that subclass 1 includes all higher subclasses, a different approach from that of Austin, Hanlon and Russell (1974). The algorithm is a modification of the Towers of Hanoi procedure described in Rouse Ball and Coxeter (1892), which may be more familiar to readers if the carry propagation of a half adder is considered. In this case classes are generated in approximately binary weighted fashion, viz. 4, 3, 4, 2, 4, 3, 4, 2, 4, 3, 4, 1, etc. This has the effect of producing a mix of many jobs and progressively smaller numbers of larger jobs. The bias against large jobs can be adjusted easily by the numbers and boundary time limits (LIMIT(I)) of the subclasses.

The scan can result in three possibilities — the queue is empty beyond the addressed position, a job with acceptable properties is found, or the next job has an excessive time or requires tapes or disc packs. The queue exhausted condition is handled by setting the subclass I - 1 and trying again. When subclass 2.0 (the maximum) fails to produce a job, the queue must be empty of purely computational jobs, and scanning ceases. When a job is found but fails the tests applied to it, the pointer MAXQUIN(I) is set to its queue index. This ensures that the next scan for subclass 2.1 commences beyond the rejected job. No pointer adjustment occurs when an acceptable job is found since after transmission to the CYBER76 its queue entry will be deleted. The resulting "hole" in the queue structure is skipped over by the read primitive.

Scheduling Algorithm — Demountable Media

The scheduler enters a different mode when the operator initiates a class 3 scan. The algorithm is controlled by an integer COUNTER and three arrays COEFF, ACTION and QUIN. The operator command contains a two digit number, which is stored in COUNTER. As a job is selected by each successive scan, COUNTER is decremented by one, and the normal Towers of Hanoi algorithm is restored when COUNTER reaches zero.

The arrays each have three entries, corresponding to disc pack, nine and seven track tapes. The QUIN array has a definition similar to that of MAXQUIN. To be precise, QUIN(D) gives the queue index of the last job rejected when a job of subclass 2.D was being sought. At any time, one or more of the subclasses may be disabled by the scheduling algorithm. Membership of such a subclass will cause job rejection. If a job at queue index Q is rejected, the QUIN pointers of non-disabled subclasses are set to the maximum of their current value and Q. The QUIN array has the effect of dynamically subdividing the queue by job attribute, and it is used to improve scanning efficiency.

The COEFF array contains empirically chosen values which give an estimate of the relative ability of each device type to service mount requests. The factors which are relevant include:—

1. The numbers of each device type currently available
2. The speed of devices
3. The ease of loading packs or tapes.
4. The degree to which the operating system assists or hinders the operators.
5. The typical characteristics of device usage.

In the DCR system there are four disc drives, six nine track drives and two seven track drives available and the assembled values of the corresponding COEFF elements are 4, 12 and 2 respectively. The values can be varied by the operators to cater for abnormal states of the system, such as a single drive unavailable (reduce COEFF value proportionately) or a controller down (set COEFF value to 0).

The operators are instructed to initiate a batch of about 10 jobs, so that at most about half of the jobs being multiprogrammed can be held up by need for operator action. This operator intervention is the second control mechanism of the scheduling algorithm. Furthermore, the operators are instructed not to initiate a fresh batch of class 3 jobs until the tape or disc mounting operations of the previous batch have been completed. This is, of course, quite difficult to determine, in cases where jobs compute for a considerable period before requesting a tape or pack mount. In addition, the designers of SCOPE have gone to some trouble to avoid the deadly embrace problem in device assignment, and this has unfortunately resulted in keeping knowledge of tape requirements concealed from the operators for as long as possible. Nevertheless, the attempt to complete a batch before initiating the next has some value in reducing the loss of system resources if the CYBER76 should fail. There is also, of course, a possible wastage of device utilisation inherent in this philosophy, in that a batch of jobs could require much tape activity with no disc pack mounts. The scheduling algorithm addresses
this point by attempting to select a mixture of tape and pack jobs but the possibility of idle devices still remains.

The COEFF array is intended to control the makeup of a batch of disc pack and tape jobs in such a manner that all devices are equally loaded and so that the instructions to complete one batch before initiating the next do not result in inefficient usage of devices. Obviously, such a simple overall average cannot cater for dynamic variations in job characteristics, but it does produce a reasonable mix. It is important to note that the COEFF array has been based on the potential of the devices to handle requests, and is independent of the current queue population.

The ACTION array is set when the operator invokes a scan for class 3 jobs. For each device type D=Y,N,M the value is computed by

\[ \text{ACTION}(D) = (\text{COUNT} \times \text{COEFF}(D) + 9)/10 \quad \ldots \quad (1) \]

The integers 9 and 10 give an approximate weight of 0.4, 1.2 and 0.2 to the three device types and insure that ACTION(D) is non zero if COEFF(D) is non zero. The ACTION array is intended to control the number of mount requests which will be displayed to the operators in respect of each device type — i.e. the job mix. When a job is accepted, the ACTION array is decremented by the number of mount requests for each device type on the job identification statement. When an ACTION element becomes zero or negative, jobs which require the corresponding device type will be passed over. Thus a subclass is enabled only when its ACTION is greater than zero.

The algorithm proceeds as follows. Firstly, the minimum QUIN(D), D=Y,N,M for which the corresponding ACTION(D) is greater than zero is found. Call this Q. The CY2 queue is examined from position Q + 1. Three cases then have to be treated — the queue is empty beyond the addressed position, an acceptable job is found or a job is found but requires a device type for which the ACTION value is not greater than zero.

When a job is accepted, the number of mounts of each required device type are subtracted from the ACTION array. A job is acceptable even if this subtraction makes an ACTION element go negative. This is an arbitrary decision taken for coding convenience. When a job is rejected, the QUIN pointers are updated for each device type for which ACTION is greater than zero. Note that it is possible for a scan for a class 3 job to encounter a job which is really class 2. Such a job would be accepted, because it does not fail any of the tests regarding device type requirements.

The queue empty case is given a relatively complex treatment. If no ACTION element has been exhausted (less than or equal to zero) then the queue is truly empty, and not merely empty beyond the addressed position. In this case, all the queue index array values are zeroed (both MAXQUIN and QUIN). Otherwise, it is assumed that the queue might possibly contain some jobs requiring a device type for which the ACTION element has been exhausted. The ACTION array is recomputed using equation (1) and a fresh scan commenced. This caters for the transition to an empty system, and will, of course, produce job batches which differ from the "optimum" mix defined by the COEFF array. It also implies that some cognisance is taken by the scheduler of the queue population in extreme situations. It was felt to be better to accept jobs even outside of the optimum mix if the queue size is very small or if the distribution of device requirements in the queue differs markedly from the optimum. If it is true that most jobs involve 1-2 actions the recomputation algorithm will not be invoked except when the queue is almost depleted.

Operator Control/User Interrogation

The operators have a number of parameters of the scheduling algorithm under their control, and they may adjust the multiprogramming width in the CYBER76. Within the 3600, the operators may alter the following

1. The size of a "batch" of class 3 jobs
2. The relative proportions of disc pack, nine track and seven track tape mounts.
3. The maximum time limit for class 2 jobs. Jobs with a greater time limit are then considered as class 3.

The proportions of different mount types would be adjusted to cater for an unavailable device, and a COEFF element may be set to zero to block jobs requiring that device type. The maximum time limit parameter is a vestigial remnant of an early version of the scheduler. The normal (assembled) value is 4096 — effectively infinity. The only use of this parameter in normal operation is prior to a system shutdown, when it will be set to zero to block non-interactive jobs. Nevertheless, other values are catered for in the code. In particular, the Towers of Hanoi scheme for selecting a subclass 2.1 of computational jobs is followed by a test to ensure that LIMIT(1) does not exceed the operator-set time limit.

The operators can interrogate and change the parameters of the scheduler from a 3600 console. They are expected to make adjustments to the COEFF array in response to equipment becoming unavailable or available. In addition, it is possible to ascertain the number of items queued in CY2, and the date/time of submission of the earliest job.

A recent addition allows operators or users to effectively interrogate the delay experienced by each of the scheduling subclasses — i.e. the QUIN and MAXQUIN pointer arrays. The information presented is the queue index and date/time of submission of the next job which will be considered as a possible candidate in each subclass. This, of course, does not imply that the specified job will be accepted, but does give an accurate picture of how the processing of each subclass is progressing. A user can make a reasonable estimate of the delay likely to be experienced by a job with any given characteristics. For a class 3 job, it is necessary to take the greatest delay of all the subclasses which include the job.

Discussion

Scheduling by a front end computer depends on adequate feedback of the internal state of the main processor. In this paper, the only feedback mechanisms are the returned multiprogramming width and the operator initiation of tape or disc pack jobs. While these mechanisms are clearly crude, the scheduler performs well in a variety of system loading conditions. The queue scans will dynamically establish three subsections of the CY2 queue bonded by the 3.Y, 3.N and 3.M pointers. The relative positions of these pointers will indicate which job attributes currently cause most delay before job acceptance. This ordering may vary throughout the day. Similarly, the MAXQUIN pointers will advance quickly for the smallest time limit and progressively more slowly for each larger subclass. The two scan algorithms can produce a graceful transition from an overloaded state to an unloaded state. When the CYBER76 is idle, the scheduler degenerates to a

The Australian Computer Journal, Vol. 9, No. 3, September 1977
Job Scheduling

"switched-off" condition.

The 3600 queue structure is designed to have rapid access, and an open ended ability to accommodate very large queues. Unfortunately, "very large" as it was understood before the advent of the CYBER76 is rather smaller than the size which the CY2 queue sometimes attains. If the CYBER76 is unavailable for half a day the CY2 queue will exceed one thousand times, and queue accesses will be costly.

In addition, the method of scanning jobs involves a queue access followed by an access of the job file. This could be avoided by encoding the job attributes within the queue item itself, but in the DAD system queued job files are not locked, so that the user could change a job file after submission. Steps to guard against this trick could be taken, but the effort has not been deemed worthwhile.

It is inherent in the scheduling procedure that several examinations of a given job may occur before it is submitted for execution. In general, this overhead is acceptable. However, a very large CY2 queue can cause the 3600 to give very erratic response to console users, for example, because of the time-consuming scans (which occur in interrupt disabled mode). These effects are particularly noticeable at startup time, when the MAXQUIN and QUIN arrays have not been established. These arrays are provided primarily for efficiency reasons, and until values reflecting the queue structure have been found, many abortive queue item/job file examinations may occur. Once a steady state has been achieved, the scanning overhead drops.

Similarly, if the operator should enable a subclass 3.D, which has previously been disabled (i.e. COEFF(D) = 0), the queue pointers have to be re-established, with momentary higher scanning overheads.

The algorithms presented in this paper are essentially crude. In particular, the operator initiation of batches of jobs requiring human invention may seem inept. Nevertheless, the concept that the computer should function at a rate convenient to the operators, rather than that the operators should be forced to respond at the pace of the machine (a frightening thought with something as powerful as the CYBER76), is an effective way of utilising both human and mechanical resources.

Acknowledgements

Acknowledgement should be made of the contribution made by officers of DCR in discussions on scheduling, particularly Dr. G.N. Lance. A number of facets of the scheduler were programmed by other members of the Central Operating Systems Group, for example B.V. Munter who provided a routine for decoding a job's scheduling requirements.

References

"SCOPE 2:1.5 REFERENCE MANUAL", (1975): p3-1, Control Data Corporation, Publications and Graphics Division, Arden Hills, CDC Pub. No. 60342600, Revision J.
A Computer Program for Exploring the Homogeneousness of Randomly Sampled Categorical Data

By J. P. Higgins*

This paper describes an ANSI FORTRAN program for exploring the homogeneity structure of randomly sampled sets of categorical data. The samples of each set should be from populations whose characteristics are described by a nominated set of binary categories. Each set is considered separately as a contingency table. An asymptotically chi-square statistic is used for hypothesis testing and to provide a dissimilarity measure. The program uses free-format input facilitated by means of key words.

Key Words: Dissimilarity, Homogeneousness, CR Categories: 3.12, 3.3, 5.5

1. INTRODUCTION

A commonly encountered problem in biological research is the statistical analysis of a data set about which very little is known regarding its distribution or internal structure. This paper considers the case where the data are a set of samples in the form of counts relative to a nominated set of categories. The data are therefore in the form of a contingency table. Typical examples of this type of data are:

(i) samples from termite populations described according to taxonomic characteristics and,

(ii) samples from blow-fly populations classified according to chromosome rearrangements.

In order to examine the relationships within such data sets, a test for homogeneity is often carried out, and the asymptotic chi-square statistic is used to test for significance (e.g. Kullback, Kupperman and Ku 1962). In the circumstances where the null hypothesis "the populations are homogeneous" is rejected, the experimentalist generally requires further information on the interrelationships within the set. In particular, answers to such questions as "is there a homogeneous subset" or "which samples are homogeneous with this sample" are often needed.

This paper describes an ANSI FORTRAN program, called HOMGEN, which pragmatically attempts to provide answers to such questions as mentioned above.

2. DEFINITIONS

(i) Test-statistic used in HOMGEN

This is derived from the likelihood-ratio method and is given as:

\[ X = 2 \sum_{k=1}^{r} \sum_{i=1}^{c} x_{ki} \log_e \left( \frac{x_{ki}/n}{x_k \cdot x_i/n^2} \right) \]

where

- \( r \) is the number of samples (rows),
- \( c \) is the number of categories (columns),
- \( x_{ki} \) are the cell counts in the contingency table,
- \( x_k \) are the marginal totals for the rows,
- \( x_i \) are the marginal totals for the columns,
- \( n \) is the grand total,
- \( X \) has \((r-1)(c-1)\) degrees of freedom.

(ii) Dissimilarity of two samples

The dissimilarity \( X(i,j) \) of two samples, \( i \) and \( j \), is obtained as follows. The two samples are considered as a \( 2 \times c \) contingency table. \( X(i,j) \) is defined as the value of the test-statistic for this table.

(iii) Dissimilarity measure between a sample and a subset of samples

This is based on definition (ii). The sample in question is considered, in turn, with each sample of the subset and the dissimilarity of the pair is calculated. The dissimilarity measure between the sample and the subset is then defined, by the relationship below, as the sum of the dissimilarity measures for the above pairs of samples.

\[ S(i,T) = \sum_{j \in T} X(i,j) \]

where

- \( i \) is a sample of the original set,
- \( T \) is a subset of the original set.

3. DESCRIPTION OF THE ALGORITHM

The data are considered as a contingency table and the test-statistic is calculated from definition (i). Under \( H_0 \) "the populations are homogeneous" and with large samples, the distribution of the test-statistic is known to be asymptotically chi-square with \((r-1)(c-1)\) degrees of freedom (see Kullback 1968). If the probability of the test statistic is below the significance level chosen by the user then the program prints out the relevant results and stops. However, should the initial test of the data be significant the program may take one of two paths determined by the user. The options available are: produce a maximal, homogeneous subset which best represents the original set of samples or; find a maximal, homogeneous subset of samples most similar to a nominated subset of samples.

In each case the required subset is produced by means of a rejection criterion based on the dissimilarity measure of definition (iii). To find the maximal, representative, homogeneous subset the dissimilarity measure between
Homogeneity of samples

Each sample and the subset consisting of the remainder of the set is found. The most dissimilar sample is then rejected. That is, the sample corresponding to \( \max \{ S(i,T) \} \) is rejected (where \( T \) is the subset composed of the remaining samples). This reduced set is tested for homogeneity and if the test is significant the procedure is repeated until, either a homogeneous subset is produced or, there are only two samples remaining.

The procedure to find the maximal, homogeneous subset, similar to a nominated subset, resembles that mentioned above. However, there are two differences:

(i) the samples of the nominated subset are excluded from the rejection procedure;
(ii) considering only the complement of the nominated subset, the dissimilarity of each sample from the nominated subset is calculated, rather than the dissimilarity from the entire set.

The most dissimilar of these samples is then rejected. That is, the sample corresponding to \( \max \{ S(i,T) \} \) is rejected (where \( T \) is the nominated subset). The reduced set is tested for homogeneity and if the test is significant the procedure is repeated until any one of the following situations occurs: a homogeneous subset is produced, only two samples remain or, only the nominated subset remains.

Although the null hypotheses for the tests are “the populations are homogeneous” they are not the same hypothesis since each refers to a different set of samples. Consequently after a null hypothesis is rejected and a reduced set of samples is formed, we then set up a new null hypothesis.

As mentioned previously, the test-statistic for the original set of samples is known to approximate a chi-square statistic provided that the samples are large and independent. However, if a sample is omitted from the set on the basis of the data then the remaining samples are no longer independent as required. We then consider the test-statistic to have a pseudo-chi-square distribution for the purpose of significance tests. In order to compensate for introduced bias it was decided to increase the critical region whenever a sample is rejected. This adjustment is carried out by a method, based on intuition, that seems to be reasonable.

In order to calculate the new or adjusted significance level the current level is divided among the samples in proportion to their sizes. If a sample is rejected then, the significance level is increased by that portion of the old level allotted to the sample that has been rejected, \( N_k \) is the total number of counts in the set before the \( k \)th rejection.

4. PROGRAM INFORMATION

Input to the program is in free-format. This is effected by means of key words; the use of which are summarised below:

(i) EXCLUDE
   This enables the user to have one or more of the samples omitted.
(ii) HEADING
   Places a heading, chosen by the user, on each page of output.
(iii) INCLUDE
   Allows the user to test for similarity between samples.
(iv) LEVEL
   Determines the significance level of the test.
(v) MESSAGE
   Places a label, chosen by the user, on the set of data on the output file.
(vi) OUTPUT DATA
   Controls the printing of the data.
(vii) PARTITION
   Allows the user to request the representative, homogeneous subset.
(viii) SAMPLE
   Provides for the input of the data.
(ix) TABLE
   Allows the user to request the printing of the contingency table of the original set of data.
   All key words have default values which are pre-set for each set of data and inserted by the program if the key word is missing. The only exception is SAMPLE, which must be present in the first set at least.

There is considerable error checking done on the data. As well as validity, the data are checked for consistency and it is reasonably hoped that no detectable error can go undetected.

The program is capable of multiple runs and errors in one set will not affect the results or diagnostics of another set, provided that such sets do not share the same sample data.

Some idea of the running time of the program can be gained from two jobs, run on a CDC CYBER76. The first was on a set of data consisting of 10 samples with 6 categories. The set was tested at three different significance levels, in succession, in the one run. The job took 0.089 s to execute. The second job consisted of 149 samples with 3 categories. The set was tested twice in succession with different INCLUDE directives. This job took 2.122 s to execute.

A user’s guide, program listing and source code of HOMGEN are available from the author.

References

A Case Study Comparison:

Batch Submission, Demand Terminal and “Hands-on” Computing in a Computer Science Tertiary Education Environment

By R. A. Jarvis* and P. N. Creasy*

Whereas the conventional batch mode of processing student jobs for large classes at an introductory level may be technically efficient in terms of processor and peripheral equipment utilization, it is not difficult to argue that demand access provides a more effective Computer Science teaching environment, especially if a remote entry point can be provided inexpensively in the vicinity of teaching centres. A case can also be made for “hands-on” usage of a small in-house computing system in providing a comprehensive experience of a limited but complete computing system, particularly for Computer Science courses built around special peripheral instrumentation. These three different types of computing modes are compared through case study observations relating to actual teaching experience at A.N.U. by Computer Science academic staff. The logistic, economic and academic factors involved in determining which computing mode is best suited to which course is carefully examined. The rationale behind the development of Computer Science facilities in relation to academic motivations is given, together with details of the relevant courses.


CR Categories: 1.52, 4.20, 4.30

1. INTRODUCTION

At the Australian National University Computer Science courses are offered at second, third and fourth year (honours) and are incorporated into the degree structures of the faculties of Arts, Economics and Science; there is no Engineering school on campus. The Computer Centre is responsible for campus-wide provision of computing facilities in relation to two large installations, an IBM 360/50 and a Univac 110/42 (recently upgraded from a Univac 1108) and plays a shared managerial role in relation to several other systems distributed about the University. Computer Science has an in-house Data General NOVA disc operating installation (see Fig. 1) with 32K words of memory, a 256K word disc, a 400 cpm card reader, a 356/1110 lpm line printer, a 300 cps paper tape reader, a 60 cps paper tape punch, a small magnetic cassette tape unit and a CRT keyboard display.

The split academic structure of A.N.U., with pure research associated with the Research Schools and the more familiar research/teaching role associated with the Faculties makes the question of student access to computing facilities a delicate one, but one which, fortunately, has in most part been solved through a considerable degree of goodwill and co-operation between the Computer Centre and the Computer Science department.

Although it is admitted that various student groups will have special Computer Science course needs, care has been taken in design of curricula to emphasise central themes in Computer Science, staff resources being limited.
unconstrained optimization) and Image Processing and Automata Theory (including pattern recognition, image processing, graphics, automata theory). Fourth year honours consists of the four formal units, Artificial Intelligence, Compilers: Theory and Practice, Advanced Computer Architecture and Information Systems, together with an advanced computing project with assessment weighting of one third of the year's work. This outline is given to provide a background against which the rationale for the various modes of computing access used for our courses can be established.

At the outset it must be recognised that the use of computer facilities to provide a rich learning environment for Computer Science students cannot be directly compared with industrial, commercial or applied research use in terms of efficiency either in relation to processor/peripheral utilization or in terms of human resource expenditure/convenience. Nevertheless the costs (human resources as well as machine related) involved must be studied and balanced realistically against academic goals attainment.

Essentially, this paper addresses itself to this question of balance as it reveals itself in our recent teaching experience at A.N.U.

Sections 2, 3 and 4 present brief outlines of the attributes and usage of Batch Submission, Demand Terminal Access and In-house/"Hands-On" modes of computing, respectively, in relation to specific courses. Section 5 is devoted to comparative comments which lead to the conclusions presented in Section 6.

2. BATCH SUBMISSION

Batch submission has several points in its favour for running large numbers of jobs such as those from our introductory unit (class size 150 to 200) — although these points are not necessarily based on the benefits to the student or the course: (1) the jobs can be scheduled at an organisationally convenient time, (2) they do not need a modern time-sharing system, (3) advantage can be taken of card punches which are being less frequently needed by other users, (4) congestion on terminals is avoided. Most of these points have been taken into account in deciding on a batch submission mode for the introductory unit.

The jobs (currently ALGOL W), prepared by the students and submitted within the department, are run on the IBM 360/50 which is approximately 2 km away. Before the installation of the Unvic 1108 (May 1972) overloading problems with the IBM 360/50 made it unfeasible to expect better than an average of 2 to 3 day turn-around, a period generally tolerated by non-student computer users throughout the campus. More recently, with the acquisition of the Unvic 1108 and consequent slackening workload on the IBM 360/50, four submissions per day have been implemented. This does not necessarily mean that each student submits jobs proportionally more often (Hall and Kidman, 1975); however, the decrease in turn-around time, together with an increase in tutorial and program consultation assistance, has considerably enhanced the rate at which students grasp programming and problem solving fundamentals. It should be added here that our experience has shown ALGOL W on the IBM 360/50 to be an excellent student compiler, with ample meaningful diagnostic statements invaluable to students who are just learning the language. No "student-job" type compiler (e.g. load & go) for a suitable language is currently available on the UNIVAC 1108.

A simple job control system with prepunched cards screens the students from the complexities of a large scale operating system with its powerful but frustrating alternatives which would confuse the novice. As a further attempt to overcome some students' apprehension at approaching a computing machine for the first time, an interactive time-sharing BASIC system is made available for the first few weeks of the course. This is provided on an 8K NOVA computer within the Computer Science submission room so that adequate assistance is always available. With this BASIC experience some valuable programming and problem solving fundamentals will have been quickly learned and, at least for some students, the limitations of the language will have become evident through experimentation, opening a well motivated path into ALGOL.

Some shortcomings of our batch submission practice are related to shortage of card punching facilities (or, alternatively, long delays if professional key punching is used), the problem of lost card decks and listings and sometimes of cheating. Although many complimentary things can be said of the demand terminal access mode discussed in the next section, it is our feeling that large classes of introductory students can fulfill most of the academic goals of such a course using batch facilities with good turn-around times and supplemented liberally with tutorial and program consultation assistance. On the positive side (though not a strongly compelling argument) is the idea that the discipline of carefully checking listings as an effective debugging procedure for small programs sometimes develops with a not-too-easy access to the machine; the practice of mindless debugging by machine through repetitive submission interleaved with minor "hit and miss" program variations certainly does not flourish in such an environment (see Hall and Kidman, 1975). Our present inability to provide adequate interactive facilities for our large introductory class (resource allocation problem) does not, for most part, worry us seriously. It is felt, however, that the situation will be much improved when the remote job entry (card reader/line printer) station close to the centre of our operations is functioning (Figure 2), as job handling and turn-around will be vastly improved (assuming the availability of a suitable student compiler on the Univa 1100/42). Despite the discussion

Figure 2: Demand Terminal, Remote Job Entry and C.A.I. Facilities.
above we would not want to give the impression that we consider batch submission as either suitable or adequate for more advanced classes.

3. DEMAND TERMINAL ACCESS

The "student job" batch compiler has certain limitations (e.g. severe restrictions on file usage, inadequate language choice). The more advanced user, however, is required to learn what can be very complex job control language before being able to make use of all the system facilities. Given this commitment, the choice between batch and demand terminal access is dependent upon job requirements: batch mode of operation is usually sufficient for production processing whilst demand terminal access provides a superior basis for the experimental emphasis of program development (Wilkes, 1968).

Because most Computer Science students' computing work is of the latter variety there is a clear advantage in them being given access to demand terminals, particularly when they can be expected (in advanced classes) to make the above learning commitment; this will make available to them a large selection of utility packages including a variety of compilers, source editors and on-line debugging facilities. These practical advantages are strengthened by the requirement that students for whom Computer Science represents a major study area develop a thorough understanding of the underlying principles of the processes involved in program development whilst demonstrating proficiency in facility usage. The immediacy associated with this type of interaction changes the students' view about the whole process of solving problems by computer. This change of view touches upon source preparation, modular testing procedures, dynamic debugging ideas and the man/machine inter-relationship as a whole. The whole computing process becomes mobile and dynamic. It has been observed (Schatzoff, Tsao and Wig, 1967) that using interactive facilities as nothing but fast turn-around batch has little to recommend it. The student should be made aware of his responsibility to acquire the skills to exploit all the facilities made available to him via demand access. We would regard the relatively free access to demand terminals for advanced Computer Science students (third year and beyond) as practically indispensable and would not be happy with anything less than 10 hours per week terminal availability for each of these students.

A computer link via modems was established between Computer Science and the Computer Centre's Univac 1108 early in 1973. The attachment was made directly to the communications module controller on the host machine though it is planned eventually to link via a PDP11/50 front end machine. At the remote end the 2400 baud line (full duplex) was initially administered by a 4K NOVA concentrator serving several ARS33 teletypes. More recently a number of CRT terminals including some with graphic capabilities were added and an 8K version implemented to handle the expanded system. The most recent development (see Fig. 2) will allow connection initially of some 16 terminals and provide both RJE facilities and the capability of stand alone processing within a real time disc operating system environment. These developments serve a computer user community of staff and students in the immediate geographic vicinity of the installation. Direct, University-owned lines will be used up to 9600 baud. These facilities should support demand access requirements of a user community of approximately 100 without congestion and with a fairly high expectation of casual (unbooked) terminal availability. (Our own third year, fourth year honours and postgraduate students would number in the vicinity of 45, but as the same demand terminal access facilities are made available to the academic staff and authorised students in departments other than Computer Science, the total user community in our vicinity would perhaps be near 100.)

The new terminal room itself is designed to provide a relatively pleasant (low noise, air conditioned and carpeted) environment which, in addition to providing a concentration of terminals allowing a certain amount of group self help in learning access protocols, can be used for computer terminal staff-directed tutorials which will greatly facilitate the acquisition of terminal usage and advanced programming skills. Furthermore, a computer aided instruction laboratory is being set up to support a range of courses in the Faculty. A tutor can demonstrate to as many as 25 students using a large screen graphics terminal linked via a scan converter to distributed composite video TV monitors. The video distribution system will not need to be used for less than about 8 students and more detailed information can then be displayed in the large screen terminal. These two kinds of tutorial mode will, to a large extent, complement one another, the first providing a practice workshop and the latter a demonstration utility with graphics as well as alphanumerics.

4. THE IN-HOUSE MACHINE AND "HANDS-ON" USAGE

A number of different approaches to teaching machine architecture and assembly language have been discussed (Sale and Bromley, 1974). In the machine architecture and assembly language course components we felt that it was important to teach about a real machine but preferably one that students could use "hands-on" at a later stage. To meet this requirement we decided to use our NOVA computer which we considered adequate though not ideal for the purpose. The assembly programming which immediately follows the machine architecture section of the course has a practical requirement which involves writing a number of assembly programs to be executed on the NOVA.

A batch system with immediate execution of the assembler object code was initially used with the operator taking a memory dump if the job crashed the system. Though relatively realistic, this mode was abandoned because of the time involved in the memory dump and system reload. In the current system the object code is interpreted, trapping and reporting infinite loops and errors which would normally crash the system. Only the memory reference instructions (approximately half) require interpretation with the remainder being executed directly.

"Hands-on" computing provides a unique hardware/software familiarity of considerable value in a teaching environment, particularly in Computer Science subject areas concerned with specialised peripheral instrumentation. That "hands-on" experience can be made available only on a small in-house machine is assumed as it would be impracticable and unjustifiably expensive to use a large central installation for this purpose (for a relatively large number of students each requiring separate solo sessions). Many of the attributes of "hands-on" computing complement but are not shared by demand terminal access. Some of these relate to access to console switches leading...
to more effective assembly language debugging, exposure to realistic resource allocation exercises (memory, peripherals etc.) and to both software and hardware quirks, the accommodation of which can provide worthwhile practical experience. The responsibility for powering on, bootstrapping and error reporting etc., together with direct contact with processor and peripheral hardware and a complete software system create a special type of computing environment which all Computer Science students should have the opportunity to experience. System development and interrupt structure design exercises take on special significance in this "total" though small computing facility. Obviously, also, some students (fourth year) may become involved with hardware projects such as the recent provision of a hardware stack (Edwards, 1975). Hardware access to the processor and device interface is indispensable for such projects.

In addition to the equipment list in the introduction a number of specialised interfaces and peripherals (Jarvis, 1974) have been added; these include a hybrid interface which links with a flying spot photographic transparency scanner, a flying spot/moving table/video microscope scanner, a desktop analog computer, storage CRT display screen, a graphics tablet, a joystick, and more recently, an industrial robot experimental set-up for hand/eye research. Most of these specialised additions have been developed for academic staff research but, where practicable, they are absorbed into the teaching of the workshop oriented course in third year which includes sections on pattern recognition and image processing.

It is worth while noting, but without undue emphasis, that, whereas the minicomputer "hands-on" environment has hitherto been considered as substitute (with clearly apparent shortcomings in terms of complexity and sophistication) for a large installation, recent technological developments and cost reductions have made the minicomputer a worthy subject of Computer Science study in its own right.

5. DISCUSSION

We have consistently emphasised the academic requirements of computing access rather than machine cost. In our planning we have had, in most part, to work within the constraints of existing installations and software available.

Although system overhead costs are generally larger for demand access than for batch submission on a job for job basis, the actual effective cost comparison must take into account the human resource element which presumably will be smaller for demand terminal access provided the user is proficient in its use, particularly during program development. By "proficient" is meant well informed and practised in the business of fulfilling a given task efficiently in terms of both machine and human resources utilization. In relation to system overhead costs, our batch system operation on the IBM 360 using a fast student compiler would most certainly be less expensive than the demand system which makes use of the full UNIVAC 1108 facilities.

Given the existence of card-reader and line-printer on a large system for batch and say a drum for demand, the main additional costs will be preparation card punches for batch and interactive consoles for demand. Remote consoles are relatively inexpensive, the prices for the Computer Science system for an 8K computer with 10 CRT consoles with interfaces averages out to approximately $2500 per console. Over 5 years this amounts to about $600 per year (including maintenance). Against this is the cost of card punch rental of about $800 per year. Added to these must be the cost of lines, paper etc. for the demand system and transport, cards etc. for batch. Thus the costs for the external equipment is about the same for batch and demand (allowing for fewer card punches than consoles).

The in-house system, whilst not the general purpose system in the same sense as the central facilities, does have a fairly high utilization. The capital cost of the equipment is approximately $35,000 with a maintenance cost of about $5,000 per year. The cost cannot be directly compared with the equivalent computing cost on the central facility because (as previously mentioned) it provides a unique mode of computing.

Follow the computer access modes made available to a Computer Science student who enrolls in all our courses including a fourth year (honours): the student will first meet the machine through interactive BASIC on terminals, then via batch submission of ALGOL W jobs to the IBM 360. Next he will have assembler jobs run for him on a NOVA in-house machine he already knows something about from lectures. At the beginning of third year he will be introduced to the demand terminal connected to the UNIVAC 1108 and will be expected to become familiar with a number of high level languages including ALGOL, SIMULA 67 and LISP. Later he will be required to execute and debug assembly code designed to explore interrupt structures on a stand-alone NOVA, his object code having been produced from his source by assembling on the larger (disc operating system) NOVA installation. At the same time he will be given a chance to gain access to the in-house disc operating NOVA system for general project work. Finally, in fourth year (honours) he will be able to utilize any or all of the various facilities he has come across over the previous two years.

6. CONCLUSION

In a tertiary education environment a case has been made for exposing the student to all these three modes of computing to which this paper is directed. Whilst demand terminal access and "hands-on" is indispensable for the advanced student, we feel that the batch processing mode does not disadvantage students in our introductory course and in fact can be seen having the advantage of simplicity. We are left with the feeling that the availability of a suitable multi-user demand access facility (for say ALGOL W) may well have changed our stance on this last point.

REFERENCES


Artificial Intelligence Techniques in Computer Assisted Instruction

By J.A. Self*

Artificial intelligence research should have much to offer to computer-assisted instruction. This article surveys some recent attempts to apply artificial intelligence techniques to develop more sophisticated computer-assisted instruction systems. The topics covered include representing knowledge, problem-solving, psychological modelling, learning and communicating. While many of the applications are only in demonstration systems and often do not take full advantage of the results of artificial intelligence research, there is clear evidence of progress towards more "intelligent" computer-based teaching systems.

KEYWORDS AND PHRASES:
CAI, computers in education, problem-solving, psychological modelling, generative CAI, student models.

CR CATEGORIES:
1.50, 3.32, 3.60.

Introduction
After fifteen years or more, computer-assisted instruction (CAI) is beginning to convince the sceptics that it has a role to play. In particular, the PLATO system (Bitzer, 1976) cannot fail to impress. On the other hand, a moment's thought shows that such systems are often impressive in the absence of, rather than because of, a deep understanding of how to write teaching programs. Teaching usually involves abilities such as communicating in natural language, knowing something about the student's problem-solving and learning processes, and structuring knowledge so that it may be used appropriately, all of which are problem areas in artificial intelligence (AI), in which most computer scientists, except perhaps AI researchers, seem to believe that little progress has been made. In fact, of course, the truth is not so extreme: some CAI programs can be and obviously are written by skirting round the difficult problems and AI has been making steady progress towards solving problems of relevance to CAI. So much so that in recent years there has been an increasing volume of work attempting to apply AI to help design more flexible, more understanding and less machine-dominant teaching systems. It is the aim of this survey to summarise this work.

CAI involves the writing of programs intended to enable a student to learn. Initially, these programs tended to adopt the logic of "programmed instruction", which had independently become popular in educational psychology. The simple 'output-input-compare-branch' flow of control of such programs made them easy, if tedious, to write and a breed of programming languages, called author languages, were developed for this purpose. While author languages have become more powerful in, for example, their display and answer-matching facilities, it remains true that the most important clause in their specifications is that the language must be usable by someone with little programming experience. This naturally tends to limit the sophistication of the teaching programs written. There are often additional reasons for the mundane character of many present CAI programs. For example, even the highly-developed PLATO system, complete with the latest technological innovations, cannot disguise the fact that if response times on a multi-terminal system are to be reasonable then the processing time per terminal must be small (Wilcox, Davis and Tindall, 1976). Unfortunately, teaching is difficult and it seems optimistic to expect good teaching to result from the use of the simplest programming languages and programs written to operate in a 'speak before you think' mode. There is therefore a trend towards considering how CAI programs ought to be written, with less emphasis on the practicalities of CAI.

Since almost all of AI has some potential relevance to CAI it is necessary to delimit carefully the work to be described. We will consider only attempts to develop a teaching system (or components of a teaching system), even if the system is primarily a demonstration one. AI work which includes only general comments about its relevance to CAI will not be discussed. Since the aim here is to survey existing systems, we make little attempt to identify and describe those concerns of present AI which are likely to influence future CAI systems. It is assumed that the reader has some familiarity with AI itself. In a rapidly developing field it is important to be aware of both the accomplishments and the limitations of the various approaches: in pointing out limitations, there is of course no intention of imputing incompetence to the system architects. The problems are difficult and, while the achievements so far are considerable, efforts towards solutions are, we suspect, in a primitive state of evolution.

In the following sections we will consider in turn the relationship between CAI and some of the major issues with AI programs, namely representing knowledge, problem-solving, psychological modelling, learning and communicating.

Representing knowledge
Since CAI is largely concerned with imparting knowledge, the way that knowledge is represented in a CAI system is of central importance. In AI the 'representation

*Department of Computer Science, University of Melbourne, Parkville, Victoria, 3052. Manuscript received 4th March 1977 and in revised form 12th July 1977.

"Copyright 1977, Australian Computer Society Inc.
General permission to republish, but not for profit, all or part of this material is granted, provided that ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society."
of knowledge problem, i.e. the problem of representing relevant knowledge, in a relatively general manner, so that a program may efficiently find, use or modify desired items of knowledge, has become increasingly fundamental and the subject of much controversy.

In most present teaching programs the knowledge is embedded in the program, enabling it to answer relevant questions or to provide a statement of its knowledge. Often the program has no way of making the knowledge explicit other than by using statements pre-specified by the author of the program. Such techniques of representation severely restrict the program's ability to understand the topic and hence to convey that understanding to the student. The student may soon attempt to cross the boundary of questions that the program has been explicitly programmed to be able to answer. More general methods of representing and manipulating knowledge are needed so that the program may itself create answering procedures and explanations.

The first steps in this direction can be regarded, oversimplifying greatly, as attempts to separate content from control. Such programs have three components: a 'knowledge base', specifying, in some structured way, the facts to be taught; an 'interpreter', which retrieves facts from the knowledge base; and the 'teaching strategy', which determines how the interpreter is to be used to generate questions or answers.

The most popular approach has made use of semantic networks (from Quillian, 1969). Carbonell (1970) is usually considered to have originated the method, in his SCHOLAR system (described more recently by Collins, Warnock, Aiello and Miller, 1975), which was intended to provide mixed-initiative CAI where the student interacts with a program acting as an intelligent tutor. Here, the networks are completely static structures, specified by the teacher in advance of a lesson, which is a marked difference from their use in language understanding systems, where one of their main justifications has been that it is comparatively easy to incorporate new information dynamically.

Carbonell's premiss that "all questions involve information retrieval in one form or another" is a debatable one (Norman, 1973). However, Carbonell and Collins (1973) have proceeded to make suggestions about how problems of imprecision, incompleteness, uncertainty and natural inference might be handled using a semantic network. Carbonell stressed the importance of the similarity of the network structure and the presumed information structures of the teacher and student. He proposed to model student errors by introducing small perturbations to the network but, in a nice understatement, conceded that he had "not yet developed to their fullest extent the modelling diagnostic capabilities". In fact, student errors were neither modelled nor acted upon.

However, Carbonell's work remains a milestone in CAI. His was the first serious investigation of tutorial dialogues and how they could be produced by an intelligent conversational computer tutor. He introduced a new kind of CAI, based upon using the computer as an analytical tool.

At about the same time, Wexler (1970) implemented a system remarkably similar to Carbonell's. Wexler also used a static semantic network and, like Carbonell, took his main examples from geography teaching. The two main differences from Carbonell's work are the processes for answering student questions (described below) and the use of a conventional author language for specifying teaching strategies. Wexler's system requires the teacher to specify questions, either in full or parameterized. In the latter case, the system completes the question on the basis of (student-independent) weights attached to objects in the network. While the use of author language techniques may make Wexler's system seem less of an advance, it has to be said that in SCHOLAR this issue was ignored since only a very general 'agenda' was specified, resulting in a somewhat disconnected conversation.

Modifications to the semantic network approach derive from inadequacies of the networks themselves as a general form of knowledge representation. Brown, Burton and Zdybel (1973) used a semantic network and simple inferring technique to represent the inherently factual information about meteorology but the more functional aspects required the use of augmented finite-state automata. About twenty automata were defined, specifying, for each transition, the prerequisite conditions, any computations that need to be performed, and some text enabling a description of the transition to be given. Then, to answer questions about causal chains of events, the global automaton was executed and the state changes described.

The method of representing knowledge as expressions in a formal logical system has been largely in CAI, primarily because the proof procedures are clearly too inefficient to enable a real-time conversation to be sustained and also because resolution-type methods do not give a descriptive model of students' problem-solving processes. Coles (1969) did, in fact, propose the development of a CAI system based upon theorem-proving in predicate logic but the proposal appears not to have been accepted.

Procedural representations of knowledge, however, have been part of CAI since the first generative program as any program which generates problems must have some procedure for generating answers. Unfortunately, these question-answering procedures have generally been used only to answer questions when there are in fact, at least three further rules for procedural representations. First, and most easily, they may provide intermediate results (or a trace), as in for example, Brown et al's meteorology system. Secondly, it may be possible to generate explanations of the procedure itself rather than simply to present a snapshot (or snapshots) of its execution. Thus, Siklossy (1970) proposed "to explain to the student the problem-solving behaviour of the performance program". The system of Stansfield (1974), described below, is also relevant to this question. The third role is to compare the system's procedure for deriving an answer with that hypothesised for the student and to use any differences detected to determine the next instructional action. This will lead into deep theoretical water but some tentative, more pragmatic, steps will be described below.

Problem-solving

It is now generally agreed that a CAI system "has to have the problem-solving ability it tries to teach" (Laubsch, 1975) because it should be able to answer questions and use its knowledge of the problem-solving processes to generate explanations or to understand student errors. It was to provide this second capability that Wexler (1970) avoided the use of a standard interpretation of semantic networks: he required the teacher to specify the derivational processes.
and made these specifications available at run-time for the purposes of remedial and expository instruction.

In general, a student faced with a problem will, like an AI program, not be able to obtain an immediate solution and must initiate some search for one. It is natural therefore to look to AI problem-solving programs to see if they can enable a CAI system to develop some understanding of student's problem-solving processes. Some of the issues are clarified by a program, inspired by an AI symbolic integration program (Slagle, 1963), to assist students in learning introductory integration techniques (Kimball, 1973). This program assumes that the student is in one of a set of problem-solving states, that he knows a set of problem-transforming techniques, which he will try in succession, and that there is a probability $p_{ij}$ that the student will choose technique $j$ if he is in state $i$. Kimball gives a sample dialogue which amounts to guiding the student through the problem-reduction graph shown in Fig. 1. In this dialogue, the suggestion to use integration by parts with $u = \log x$ was made by the program after a request for help from the student, but all other steps were initiated by the student.

Looking more deeply, however, reveals that the system's problem-solving approach depends entirely upon a set of archived problems and solutions established by a teacher. A problem is selected from this archive on the basis of the student's presumed knowledge of integration techniques. Alternatively the student may suggest a problem. In either case, the system does not make direct use of a complete problem solution, retrieved from the archive or generated by the program, since this may well not correspond to a correct student solution. Instead, the program orders the possible integration techniques, for each expression to be integrated, by counting their use in the archived solutions for similar problems. This enables the system to monitor the student's choice of technique to apply but the lack of an overall view of the problem-solving process does cause some difficulties. For example, since deviations from recommended techniques are allowed, the program must interrupt a student whose solution length exceeds a "problem length threshold" estimated by the program. If the student requests help, the system generates a hint on the basis of its ranking of the integration techniques. Of course, the recommended technique may not solve the problem, which leads to the dubious contention that "such negative experiences will broaden the student's judgement by causing him to search for less likely solution schemes". In short, the fact that programs exist which solve symbolic integration problems is largely irrelevant to the design of Kimball's system.

A similar example, derived this time from another AI standard, the DENDRAL system (Buchanan, Sutherland and Feigenbaum, 1969), has been developed by Sleeman (1975). A student is presented with the molecular formula of some compound and its nuclear magnetic resonance spectra and has to determine the molecular structure. He does this by specifying each group in turn and the associated spectral peak. The system assumes that the molecules are linear and saturated and, with the further restriction that the structure has to be developed group by group from one end, this means that at each stage there is a single correct next step. While this enables the system to immediately point out any deviations from the correct solution path, it does imply that the method is not directly applicable to problem-solving situations where there is a choice of solution strategy.

Any performance program (not just AI ones) can be converted into a teaching program; for example the core of the SOPHIE system of Brown, Burton and Bell (1975) for teaching electronic troubleshooting is a general purpose circuit simulation package. However, as we will see, the simulator in SOPHIE is used not merely as a problem-solver but as a detailed symbolic representation of the knowledge of its tutorial domain, which concerns one particular power supply. The SOPHIE system is undoubtedly the most advanced operational computer tutor, as may be inferred from the sample conversation of Fig. 2.

The simulation model is used in the following ways:
(a) most obviously, to make measurements.
(b) to answer questions, not only about the given circuit but about a hypothetically modified one, e.g. "If C2 is shorted is the output voltage zero?", requiring the simulator to be invoked with a modified circuit model.
(c) to evaluate student hypotheses, e.g. "Is it possible that resistor R9 is open?". The simulation model is modified to satisfy the hypothesis and all the student's measurements are repeated within this model. If the actual measurements made agree with these hypothetical values the hypothesis is consistent.
(d) to generate hypotheses in response to a student request for help. The system first produces a list of possible faults consistent with the output voltage and then prunes this list by eliminating all faults which when introduced into the simulation model contradict measurements that the student has made. Thus, extensive use is made of the simulator but the simulator alone is not enough. A sophisticated monitor is required to ensure that the simulator is used appropriately. More recently, the SOPHIE system has been incorporated...
Electronic troubleshooting involves, according to Brown et al, a kind of abductive reasoning (reasoning from examples to theory) rather than deductive reasoning. It is perhaps not surprising therefore that the system’s inferential power lies in a set of special-purpose routines incorporating knowledge of circuit components rather than in some general theorem-proving process. Perhaps more surprising, given the emphasis in AI (at least until recently) on using axiomatic logics to represent knowledge, is that resolution-based theorem-provers have not even found use in systems to teach logic.

Goldberg (1974) and Smith and Blaine (1976) have implemented similar systems to teach mathematical logic. The primary mode of use in both cases involves the student constructing a proof of a theorem and the system checking each step of the proof. Fig 3 shows a modified dialogue from Goldberg (1974). In practice, the rules used by the student are specified in some conventional shorthand. Whereas Goldberg’s system deals only with first-order logic, Smith and Blaine consider many-sorted theories.

In both systems, a general purpose proof checker is provided, based not on resolution or any similar procedure but on natural deduction. Perhaps this is because of a feeling that a teacher should be able not only to solve problems but also to solve them in ways similar to the student’s, since then the teacher may be better able to
DERIVE \( (VX) \ ( (VY) \ ((Y+X = Y) \rightarrow X = 0) ) \)

working premise

\[ (1) \quad (VY) \ y+x = y \]

universal substitution in 1

\[ Y :: 0 \]

\[ (2) \quad 0+X = 0 \]

commute addition \( A+B = B+A \)

\[ A :: 0 \]

\[ B :: x \]

\[ (3) \quad 0+X = X+0 \]

replace equivalent expression from 3 in 2

\[ (4) \quad X+0 = 0 \]

zero axiom applied to 4

\[ (5) \quad X = 0 \]

from 1 and 5 have conditional proof

\[ (6) \quad (VY) \ ( (Y+X = Y) \rightarrow X = 0) \]

universal generalisation of 6 OVER \( x \)

\[ (7) \quad (VX) \ ( (VY) \ ((Y+X = Y) \rightarrow X = 0) ) \]

Figure 3. An example of a proof construction, adapted from Goldberg, 1974. (Computer output in capitals, student input underlined)

appreciate the student’s difficulties.

However, these two systems do not in fact solve problems. Just as Kimball’s program did not integrate expressions so in this case proofs of theorems are not derived. Error checking is therefore limited to determining whether the application of a rule is appropriate. Similarly, the systems can generate advice to the student only about individual rules but cannot derive advice about proof strategies.

The problem of specifying the steps of a logical proof is similar in many ways to that of specifying the statements of a program. Koffman and Blount (1975) developed a machine language tutor (MALT) in which complete problems were generated from a set of 26 primitive problems for which solution schemes were preprogrammed. The problem is presented to the student in natural language, together with a list of logical sub-tasks. The student specifies (and the system generates) each statement of a solution. The given logical sub-tasks impose a strategy upon the student (cf. Sleeman’s system for determining molecular structure) and generally enable each individual statement to be independently verified. If the student’s answer is not obviously correct or incorrect, the correctness of the program segment is determined by simulated execution under various suitable test conditions determined by the system. This execution has to be monitored by statement to detect run-time errors and there are obvious difficulties in detecting infinite loops. If the simulation indicates that the program does not function correctly, the system may display the conditions under which the program fails, the effect of each statement, or a complete correct program — but the system cannot isolate the cause(s) of the failure of the student’s program.

Koffman and Blount consider that MALT is “an effective demonstration of what can be accomplished in CAI with the limited use of AI techniques” and it is natural to wonder just which such techniques have been so demonstrated. The problem-solving component is indeed limited since MALT pre-determines the student’s solution strategy. This may be appropriate for teaching beginning students machine language programming but it is not an approach which can be applied to the teaching of higher-level programming. Such a system would have to deal much more explicitly with the purposes or goals of program segments and with comparisons of alternative problem-solving strategies. Also, the ‘automatic programming’ component makes no use of any standard automatic programming techniques. There is a very good reason for this of course — such techniques are nowhere near well enough advanced to be usable in a working system with a reasonable response time. However, MALT’s heuristic approach will be applicable to only very restricted problem domains and simple programming language constructs. Similarly, the attempts to determine whether a student’s program is correct (and if not to find the bugs) are unsophisticated compared to theoretical or AI approaches (e.g. Goldstein, 1975).

A system to teach introductory programming in BASIC (BIP: Barr, Beard and Atkinson, 1975; Barr and Beard, 1976), unlike the MALT system described above, does not generate problems and solutions dynamically; instead, it has a file of some 100 problems, together with model solutions, sets of suitable test values and hints to guide the student (cf. Kimball’s system). The correctness of the student’s solution is determined by executing the model solution and the student’s solution for the test values specified and looking for discrepancies. Again, then, no serious attempt is made to analyse the student’s program, which leaves the system unable to say much more than that the program “looks OK” or “doesn’t seem to solve the problem”. All the advice that the system is able to give has been explicitly specified by the teacher beforehand.

Some idea of the difficulties that would be involved in automatically checking solutions, generating hints, etc. is given by a system developed by Ruth (1976). This system attempts to understand a student program, i.e. to determine what the program is supposed to accomplish and to locate and describe any errors in the program. The system’s understanding is derived from

(a) The instructor’s description of the task. This description is intended to give a delineation of possible strategies for solving the problem. The language in which these descriptions are written is a delicate compromise between the need to be general enough to cover the broadest possible range of programs and the need to be specific enough to ensure that there is no ambiguity concerning intended effects. Of course, the system must include an analyzer able to code, rearrange and recognize the algorithm steps of a description.

(b) A built-in body of knowledge about common programming constructs and common errors. For example, the system has ‘experts’ for determining the equivalence of iteration or conditional statements.

The Australian Computer Journal, Vol. 9, No. 3, September 1977
The system proceeds by attempting to recognize the student's program as an instance (possibly erroneous) of some intended algorithm. These intended algorithms are not derived from the task description (as ideally they would be) but are specified by the instructor. The system assumes, then, that the student will adopt one of a small set of pre-specified strategies, rather than impose a strategy upon him as MALT does.

The analysis is not done on any strictly formal basis but makes use of a variety of techniques. The method of 'symbolic execution', familiar in program-proving, is used, where generalized assertions about the values of variables are calculated through the program. The equivalence of predicates is determined using a 'mathematical programming' approach, which seems a rather strange one to adopt in preference to some kind of theorem prover. The equivalence of expressions is found by algebraic simplification. Ruth argues that this combination of methods enables a more natural description of the program-proving problem to be given than does the conventional one involving theorem-proving in predicate logic. The latter method tends to provide only a verdict and no explanation of programming errors.

This system, while it does illustrate how an intelligent program analysis could improve attempts to use computers to aid teaching programming, also reveals the difficulties that lie ahead. The specification of task descriptions is far from a trivial operation — indeed it appears to be rather more difficult than specifying programs themselves. The conversion of the student’s program into an equivalent one in the simple language in which analyses are actually performed would not in general be a straightforward operation and one can imagine that errors detected in the simple language may not be easily correlated (by the student) with errors in his own language. Moreover, the programming task of embedding heuristic knowledge about programming methods is a daunting one.

This apparent obsession with teaching programming is not mere self-interest but reflects the fact that the methods developed can be applied to other problem-solving situations. For example, consider the NLS-SCHOLAR system (Grignetti, Hausmann and Gould, 1975), derived from Carbonell's system, for teaching how to use the NLS editor which involves the student giving commands in the NLS language to produce required changes in a file's contents, a process analogous to specifying statements in a language like BASIC. The correctness of the student's response is found by direct evaluation — "all the system has to do is access the correct sequence of NLS commands for the task, perform them on a fresh copy of the student's file, and then compare the results". As indicated above, such a process provides information only about the manifestations not the causes of student errors.

Similarly, with the SOPHIE system, an electronic circuit defines a relation between input and output voltage in much the same way that a program defines one between input and output data. Troubleshooting circuits is akin to debugging programs. The same issues arise: is a student's circuit considered correct if it gives the same input-output pairings as a known working circuit?; does each component of his circuit have to be the same as one in the working circuit?; how can the system generate an appropriate set of test settings from the infinity of possible settings?; how can the system ensure that erroneous circuits do not fall catastrophically?

All of the applications of problem-solving techniques discussed above have been concerned with solving problems which the student has also to solve. There is a further application which has been largely unrecognized and therefore unexplored. This application derives from the fact that teaching is itself a problem-solving activity, i.e. at any time the teacher's problem is to transform the student from his present state to some goal state using one or more of a set of possible teaching actions. The teacher has some (usually intuitive) notion of the likely effect that each action will have on the state and attempts to estimate the student's state both from this learning model and the student's responses to teaching actions. The teacher also has some idea of the 'distance' from any state to the goal state, which is used to help determine appropriate operations to apply. The advantage of recognizing that teaching programs should be regarded as examples of problem-solving programs if they are intended to simulate intelligent human teachers is that it becomes clear that such programs should be written in languages which enable them to make full use of those problem-solving techniques that have been developed. This trend has begun: as we will see shortly, there has been increasing use of AI programming languages to write teaching programs.

Psychological modelling

If a CAI system is to be provided with problem-solving capabilities so that it may be able to say something about problem-solving processes (and not merely make use of them) then, in order that what is said may be comprehensible to a student, it is necessary that the problem-solving processes be comparable to those of the student. In other words, we need models which are descriptive rather than normative.

An example will clarify the distinction. Consider the system to teach integration techniques (Kimball, 1973) described above. A symbolic integration algorithm exists which for many types of algebraic expressions does not have to generate a problem-reduction graph and is guaranteed to give a correct solution (Risch, 1969). The method is derived from some theoretical work of Liouville in the nineteenth century. But although the algorithm may be the most efficient and reliable known, it may not be ideally suited for use in a CAI system. Such an algorithm may be used to answer questions of the form "What is the integral of ..." but may not be appropriate for "How do I integrate ..." questions (assuming that it is not Risch's algorithm which the student is expected to learn). If the student is expected to use the heuristic transformation method, then the teaching system ought to be able to adopt a similar method, as in the Slagle (1963) program, since only if a CAI system has some understanding of a student's problem-solving processes can it sensibly comment upon them.

This requirement has occasionally been explicitly recognised. For example, a major motivation for Carbonell's use of semantic networks was that "experimental evidence indicates that this information structure [of the teacher] is a semantic network very much like that utilised in SCHOLAR" (Carbonell, 1970).

However, it has to be said that AI research viewed as theoretical psychology has not had much direct influence on the design of CAI systems, since any significant results have generally been with respect to somewhat artificial situations for which it is difficult to see any immediate
relevance to practical teaching systems. Occasionally, the jargon of information processing theories of psychology (e.g. problem behaviour graphs (Sleeman, 1975), production systems (O'Shea, 1973)) creeps into descriptions of CAI systems but this usually reflects an expository or programming convenience rather than any conviction as to a concept's psychological validity.

All CAI programs make assumptions about how a student will respond to a given teaching action — that is, they have a 'model of the student'. Usually, this model is implicit; occasionally it can be used to predict how the student will respond, e.g. Laubsch and Chiang (1974) used mathematical models of paired associate learning. Since it is a CAI system's objective to modify students' responses, it is desirable for the system to not merely predict responses but to understand how responses are derived. An explicit descriptive model of the student's responding processes ought to be of help to a system aiming to effectively initiate steps to modify those processes.

This is most clearly seen with systems to teach programming. After a student has written a program to, say, find the roots of a quadratic, we can make reasonable predictions about how he would respond to a "What are the roots of . . ." question by executing this program. If we just had the student's response we could do little more than say "right" or "wrong". However, from the program, which is an approximation to an "explicit descriptive model of the student's responding process", it may be possible, for example, to determine the immediate cause of any errors and to comment upon it. In most teaching situations, of course, the student is not required to specify his 'program' or responding process. The CAI system has to hypothesise what it is.

Kirbach and Schmidt (1976) give a detailed discussion of the uses of models in education. In particular, they distinguish between models used by a student (i.e. simulators) and those used by a selectional element for performing a teaching strategy (i.e. models of learning and instruction). We would like to emphasise the fundamental role played by one component of the latter kind of model — this is a further model, a model of the student, surely the key to 'individualisation' (Self, 1974).

Learning

Since the student's mental processes will change during a teaching session (one hopes), it is desirable that a CAI system's descriptive model of these processes should also change. As current student models are so weak, it is not surprising that representations of the student's learning are extremely simplistic.

Most teaching systems adopt an approach analogous to adaptive pattern recognition in Al (Lorton and Killam, 1976). They maintain a vector of 'measurements' intended to describe the student and on the basis of this vector attempt to classify the student into a category sufficient to determine the next teaching operation. As the lesson progresses, this vector is modified according to some model of learning, intuitive or mathematical.

CAI systems have yet to make full use of the methods of structural learning (Winston, 1975). In those systems which have structured knowledge representations the student's learning is interpreted with respect to the representations rather than in terms of reorganizing the structure. For example, Carbonell (1970) uses tags for each node of the semantic network to indicate whether the student knows the information associated with that node. Similarly, Koffman and Perry (1976) use a 'concept tree', which indicates the pre-requisite structure of the course, and maintain a student model which determines the level of the tree from which the next concept to be taught is to be selected. The actual selection of a concept is made using an evaluation polynomial (after Samuel, 1959) although the coefficients of the polynomial are not changed during a teaching session.

Any methods that we may have for representing learning are not restricted to being applicable only to our student models — they may also be applied to our model of the teaching process itself. We might try to develop CAI systems which not only taught but, as a result of their teaching experience, gradually taught better.

Smallwood (1962) described a mechanism whereby a branching program of the programmed instruction type could determine the 'best' branch to take by collecting statistics about previous students' paths through the program (cf. attempting to learn how to play a game by memorising paths through the game tree). Weber and Hagamen (1969) suggested similar methods for enabling programs to branch as a function of previous students' actions. Kimball's (1973) system learned from students' responses by replacing archived solutions by better student ones. 'Better' was defined as 'containing fewer steps', although in general one would obviously have difficulty in determining whether one problem-solving process were an improvement on another. Since the solution archive was used to determine appropriate problems and hints to present, changes to the archive also, as a side-effect, changed the system's teaching performance.

These methods of automatically changing teaching performance involve making statistical variations to the parameters of a teaching algorithm rather than changing the algorithm itself. A system developed by O'Shea (1973) can be better viewed in the latter light. Here, the teaching strategy was expressed as a set of production rules which were modified in a manner derived from Waterman (1970). The modifications were determined by a deductive procedure, using an informal modal logic, operating on a set of assertions, some pre-specified and some created by the system. An assertion, for example, might be "increasing the rate of encouragement may increase the post lesson score". The crux of the system lies in the ease (or difficulty) with which a satisfactory set of assertions can be specified or developed. O'Shea reports that "an improvement in teaching performance was obtained". However, there are obviously immense general difficulties confronting the designer of a self-improving teaching system.

Communicating

Developing complex descriptive student models, as suggested above, without an equivalently complex communication component is neither possible nor worthwhile. It is not possible because the student would not be able to communicate with the system well enough to provide it with the information it needs to create the model and it is not worthwhile because the system would not be able to communicate to the student the benefits of its understanding.

There is some reason to hope that enabling free dialogue in CAI systems may not be as difficult as the work on general natural language understanding systems might suggest. CAI systems will for some time be constrained to
Artificial Intelligence Techniques

deal only with very restricted subject domains and it may therefore be feasible to develop special-purpose understanding systems making use of knowledge of the particular subject. Also, CAI dialogues are not 'free' ones—they have some overall purpose or direction. While we have little knowledge at the moment about discourse structures, and even less about how this knowledge could be incorporated into language processors, it should eventually prove an additional means of eliminating misunderstandings and resolving ambiguities.

It is this overall direction of CAI dialogue which gives the high degree of predictability in student inputs which makes key-word matching techniques surprisingly effective. Typically (and especially with author language systems), anticipated words in the response are pre-specified and looked for under various matching options. Carbonell (1970) relied heavily on key-word matching techniques and Taylor (1969) used the ELIZA approach (Weizenbaum, 1966) which could be said to involve key-phrase rather than key-word matching.

However, as control is given to the student, the predictability of student input falls, making key-word matching less appropriate. In the SOPHIE system, Brown and Burton (1975) retain the ability of key-word matchers to ignore some of the input words by defining a context-free grammar the rules of which also specify how tightly controlled the search for constituents is to be. The grammar itself is defined in terms not of general syntactic categories (noun-phrase, verb, etc.) but of 'semantic categories' (e.g. measurements, circuit elements) of special relevance to the domain of discourse. This approach is reasonable only because the number of such categories is relatively small for the SOPHIE domain. The rules of the grammar are encoded as LISP procedures much as in the SHRDLU system (Winograd, 1972) and they perform similar functions. Partial phrases lead to predictions about following constituents, e.g. recognising "the voltage at" as a 'measurement', leads to an expectation that a 'circuit location' will follow. Similarly, the problems of pronoun reference "Set the voltage control to .8".

"What is the current thru R9?".

"What is it with it set to .9"?

and ellipsis

"What is the voltage at node 5?"

"At node 1?" are handled by determining the class restrictions of possible referents. It is obvious from the illustrative dialogues given by Brown and Burton that the SOPHIE system provides a linguistic capability beyond anything that a key-word matcher could feasibly provide. The system is both robust (it "handles nearly all sentences generated by users who have had a few minutes of exposure to the system") and efficient (it "parses a typical student statement consisting of 8 to 12 words in around 150 milliseconds"). That the approach also has some generality is indicated by the fact that an adapted version has been used by Grignetti, Hausmann and Gould (1975).

Formal grammars have not been widely used in CAI natural language systems but one example is in EXCHECK (Smith and Blaine, 1976) where the topic, mathematical theories, is sufficiently formalised that the specification of context-free grammars is possible. For example, a sentence such as

"For every x there is a y such that x is a subset of y."

is recognised by a context-free grammar characterising the language of axiomatic set theory and converted into the desired internal representation by macro templates specified, like the grammar, by the course author. One of the research objectives of this project is to develop the system to accept more natural statements, such as "Everything is a subset of something".

—an ambitious goal which will not be attained without the use of considerably more advanced techniques.

There has been little attention paid in CAI systems to the generation of sentences—output sentences are generally pre-specified, perhaps with slots to be filled appropriately. Simmons (1971) proposed that a program to generate sentences under control of a semantic network (Simmons and Slocum, 1972) be incorporated in a CAI system but this seems not to have eventuated. Stansfield (1974), however, makes some more specific suggestions in this direction. He adopts a procedural representation of knowledge and emphasises that these procedures must be available, unvaluated, to the output routines. Since the structure of the reason for a fact follows the structure of the subroutine calls which prove it true, a 'why-function' is associated with each function to explain why the function evaluates as it does. Explanations can, in principle, be made to depend on any contextual information, such as a student model, and given to any required depth by evaluating why-functions of sub-expressions. As Stansfield remarks, pre-specifying why-functions is not a satisfactory general approach: why-functions should be generated from the functions themselves. In simple cases this is possible but usually it would demand rather deeper understanding of programs than is currently attainable.

Other modes of communication have yet to make use of AI techniques. Speech recognition has not been attempted in CAI systems and speech generation has been regarded as a technological problem. Sanders, Benbassat and Smith (1976) describe a system which allows audio messages to be created dynamically. The text representation of the desired message initiates a search for each word's speech data, previously established by analysis of human speech. The data generates a pulse code modulation version of the speech which is then converted into analog speech to be transmitted.

Similarly, visual communication is achieved by applied computer graphics, with AI playing no role. Displays to the student may be pre-stored or generated, to display a static or moving picture. Students may typically point to parts of a displayed picture but cannot themselves input visual information.

Programming languages

One of the more concrete examples of the use of AI techniques in CAI is the adoption of programming languages originally designed for AI research for developing CAI systems. This is seen by simply listing the languages in which some of the systems mentioned above were written: CONNIVER (Ruth, 1976), INTERLISP (Brown, Burton and Bell, 1975), SAIL (Barr and Beard, 1976) and LISP (most of the remainder). Moreover, some of these systems are immense, even by AI standards. For example, Smith and Blaine's EXCHECK and Brown, Burton and Bell's SOPHIE systems occupy about 250K and 300K (36-bit) words respectively.

This trend reflects both that the problems which are
being presented to students are becoming more difficult and hence require the systems themselves to have more powerful problem-solving capabilities and also that it is increasingly recognised that good teaching cannot be simulated if the program has only a superficial understanding of the teaching process.

However, the philosophy underlying the design of AI programming languages has not yet had much influence on the writing of teaching programs themselves. To explain what is meant here, we will briefly indicate how some of the components of a teaching system may be expressed in terms of new AI language concepts.

A teaching program’s objectives may be expressed as goals to be achieved. A goal may, for example, be specified in terms of the desired contents of a data base representing the student’s knowledge. One or more procedures may be defined to express possible teaching strategies to achieve the goals (and sub-goals). Which of these procedures is executed may be decided to depend on any relevant information, such as a student model. A strict hierarchial structure may be broken by, for example, a realisation that a topic previously presented has not been understood as well as the program had assumed, thus initiating backtracking for some remedial teaching. The context may or may not be fully restored, depending on how much of the subsequent teaching the program considers has been effective. The context may provide a dynamic record of the student’s state of knowledge (and any other relevant information about the student). Student questions may lead to temporary contexts within which answers and further questions may be processed. Laubsch (1975) draws some analogies between curriculums (which identify situations, their interrelationships, the questions they give rise to, the pre-requisite concepts, etc.) and frame-systems (Minsky, 1975). A related way of looking at pre-requisites is to view them as demons or issues (Burton and Brown, 1976) which are activated only if events indicate they need to be.

Philosophical approach

Apart from applications of specific AI techniques one can also see that an AI philosophy is becoming more common in CAI. The AI researcher is not content to accept ‘solutions’ which circumvent problems, e.g. to use only multiple-choice questions to avoid having to understand natural language. The AI approach is to try to teach ‘properly’, rather than, say, economically. Since the relative costs are changing so rapidly, it may in the long run be wise to concentrate on general principles.

The AI approach to developing programs also encourages a different attitude to teaching to that which seemed to follow from the behaviourist tradition and led to programmed instruction. An AI program is a success if it performs some task better than its predecessors or if it provides further understanding of some aspect of the task domain. But even successful AI programs have bugs; there are some situations in which the program does not perform as well as it was hoped. These bugs are, however, not regarded as causes of despair but as sources of enlightenment, as the places to start looking if one wants to develop a better theory. An AI worker then will not be too perturbed by evidence of a bug in a student’s thought processes. His mistakes can provide both the teacher (or teaching program) and student with information leading to a deeper understanding. This is most clearly demonstrated in systems to teach programming, or thinking in general, especially the LOGO work (Papert, 1973).

Teaching AI

Finally, at the risk of becoming lost in a recursive haze, we mention an attempt to write a program to teach AI. “The task was to make it easy to construct and play with simple, laboratory-sized instances of AI programs ... the effort transmitted into one of building a program that would understand AI — that would be able to explain and run programs, ask and answer questions about them, and so on, at some reasonable level” (Moore and Newell, 1974). And therein lies the course of development for all CAI systems: to write a program to teach something one must first write a program that would understand it.

Summary

Several AI techniques have been borrowed and put to use in CAI systems. In many cases, the technique has not been fully capitalised upon and the implications of what AI has to say have not been followed through. However, it would be unfortunate for the future of CAI if it were the case that the possibilities of applying AI research to CAI were “unpromising” (Lighthill, 1973), since CAI is bound to become more AI oriented as the limitations of ‘traditional’ CAI are revealed and it becomes increasingly recognised that these may only be overcome by developing programs with more understanding of the teaching/learning process. As would be expected, problems on the AI/CAI border are particularly difficult ones but ones for which any progress at all towards solutions promises much for practical CAI. Already we have, as Brown, Burton and Bell (1975) put it, “existence proofs that there are many exciting possibilities for innovative uses of AI type systems in education.”

References


Short Communication

Attribute Contributions to a Classification

By G.N. Lance* and W.T. Williams†

This paper describes a method for ranking the attributes, used in a numerical classification, so that their contribution to the grouping is displayed. The three most common types of attributes, qualitatives, multistates and numericals can be incorporated in this ranking.

Key words: Numerical classification, attributes, groups, taxonomy.

CR Categories: 5.4 and 5.5.

Introduction

One objective of all classificatory procedures is to determine the relative importance of the attributes in the definition of inter-group structure. For strictly hierarchial classifications the program GROUPER (Lance et al. 1968) has been used for this purpose; but GROUPER compares only two groups at a time, and has no provision for determining relative importance over a complete set of classificatory groups. It is therefore inapplicable to a program such as REMUL (Lance and Williams 1975) in which, as a result of terminal reallocation of individuals, the hierarchical structure is lost. What is required is some measure for each attribute that can be calculated over all groups, and is such that the values for different attribute-types are compatible. In this paper we define such a measure. The attribute-types with which we shall be concerned are multistate nominal ("exclusive disordered multistates"), binary nominal ("qualitatives") and numeric (i.e. continuous variables); all other types of attribute in general use can be treated as one of these. In particular the results given here are also valid for non-exclusive multistate nominal variables.

Notation

We shall use the following notation for nominal multistate attributes:

<table>
<thead>
<tr>
<th>Group/State</th>
<th>1</th>
<th>2</th>
<th>S</th>
<th>Row Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_{11}$</td>
<td>$a_{12}$</td>
<td>$a_{13}$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>2</td>
<td>$a_{21}$</td>
<td>$a_{22}$</td>
<td>$a_{23}$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>n</td>
<td>$a_{n1}$</td>
<td>$a_{n2}$</td>
<td>$a_{n3}$</td>
<td>$a_n$</td>
</tr>
<tr>
<td>Col. totals</td>
<td>$a_{1}$</td>
<td>$a_{2}$</td>
<td>$a_{3}$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

so that $a_i$ is the number of individuals in the $i$th group, and $N$ is the total number of individuals in the whole population. For the study of nominal binary attributes it will be convenient to use the table in the following reduced form:

<table>
<thead>
<tr>
<th>Group/State</th>
<th>1</th>
<th>2</th>
<th>Row Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$r_1(=a_1 + b_1)$</td>
</tr>
<tr>
<td>2</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$r_2(=a_2 + b_2)$</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>n</td>
<td>$a_n$</td>
<td>$b_n$</td>
<td>$r_n(=a_n + b_n)$</td>
</tr>
<tr>
<td>Col. totals</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Measures

The tables above are in effect contingency tables, for which the conventional test statistic would be $X^2$ defined as

$$X^2 = N \left[ \sum \frac{a_{ij}^2}{(a_i)(a_j)} - 1 \right] \ldots \ldots \ldots \ldots (1)$$

However, this is unsuitable for our purpose as it increases indefinitely as $N$ increases. A more convenient measure is that proposed by Cramer (1946):

$$C = \left[ \frac{X^2}{N \min (s-1, n-1)} \right]^{1/2}$$

which is constrained between 0 and 1. For binary attributes it reduces to $(X^2/N)^{1/2}$.

We can obtain a measure constrained between 0 and 1 for numeric attributes by appeal to the analysis of variance. Let $B$ be the between-group sum of squares and $T$ be the total sum of squares over the whole population; then an appropriate measure would be simply $(B/T)^{1/2}$, which we shall denote by $S$.

Compatibility

It is possible to investigate the compatibility of $C$ and $S$ for a binary attribute by changing the numerical model for the latter. Let such an attribute represent, not a categorisation, but a numeric attribute which, in the population under study, happens to take only the values 0 and 1. Then, in the binary table above, $a_i$ is the number of individuals in the $i$th group with value 0, and $b_i$ is the...
Attribute Contributions to a Classification

number with value 1; the population mean is then $c_2/N$. However for qualitatives (1) shows that

$$\chi^2 = N \left[ \sum_i \left( \frac{a_i^2}{c_1 r_i} + \frac{b_i^2}{c_2 r_i} \right) - 1 \right]$$

so that

$$C = \left[ \sum_i \left( \frac{c_2 a_i^2 + c_1 b_i^2}{c_1 c_2 r_i} \right) - 1 \right]^{1/2}.$$

Hence C (obtained by regarding the attribute as binary nominal) and S (obtained by regarding it as numeric) take the same value; and C is of course also applicable to nominal multistates. C and S may therefore be regarded as fully compatible; if C is calculated for multistates, S for numerics, and either for qualitatives, we have a measure which can be used to rank all attributes in order of their importance to the classification.

References

Book Review

(Continued from page 111)
programming at the present time. Winograd’s system, for example, which manipulates objects in a “black world” while conversing with a human, is brought in on the grounds that it is relevant to programming in natural language.

This interesting and readable, but rather breathless article, is concluded with a list of over one hundred references. The second article, by John R. Rice, describes his own research into the problem of selecting algorithms for solving given problems. He regards as its starting point a mathematical theory which has been intensively studied, the theory of approximation of functions, and generalises this to provide a model of the algorithm selection process. To show how his approach works in practice he applies it to two apparently unrelated problems: the selection of quadrature algorithms and the selection of operating system schedulers.

From the earliest days of computing, it has been understood that one may increase the speed of a computer by increasing hardware parallelism. This has been exploited in computer designs which specialise in array processing (ILLIAC IV and STAR for example), but less has been done in the area of processing ordinary programs. However, with hardware costs continually decreasing it is clear that this expedient must be more fully investigated. In the third article, David J. Kuck gives a comprehensive survey of the subject. He shows that (contrary to common opinion), it is possible with appropriate compilation techniques to get theoretical speedups which grow almost linearly with the number of processors used. On the other hand, proper hardware is necessary to exploit this parallelism, so that he also discusses the associated problems in machine design. Anyone who is interested in compilation, optimisation or computer architecture will find this a worthwhile article.

Human language is something that we use with such ease and effectiveness that at first sight it may seem simple. In reality it is very complex; but, given our natural aptitude, we all too easily look at major problems in linguistics without recognising them as such. Thus, because it forces explicitness, computer modelling has played and will continue to play an important part in linguistic research. Larry H. Reeker, in his article, discusses the learning of language. After briefly discussing the background linguistic theory and noting some of the open problems in the area he describes several previous computational models of language acquisition. He then describes a simplified version of his own model in considerable detail. Although the work still has a long way to go, it is clear that it represents a major advance on pre-computational theorising.

The last article is concerned with computer assisted instruction. After a flurry of activity in the 60s this area has settled down to a state of steady advancement. One of the most important systems in the field is PLATO, which is here described in detail by Donald Bitzer.

Taking the book as a whole, one may say that it puts into effect very well the intention of the series, namely that it should be an interesting and accessible vehicle for cross-fertilisation between the various specialisations within computing.

J.G. SANDERSON
For further information contact the IFIP Correspondent, A. Y. Montgomery, Department of Computer Science, Monash University, Clayton, Victoria, 3168.
COMPUTER SYSTEM TO SPEED UP WARFARE CONTROL

A new mobile information system which will enable formation headquarters command units to 'call up' battle information — such as locations and effective strength — on video screens will come into general service with the British Army in the early 1980s.

The computerised system, known as Wavell, is applicable to Australia whose Defence Departmental advisers have for some time been studying communication systems for the three service arms.

Plessey Australia announced recently that Plessey Radar, part of the United Kingdom group, was developing Wavell in conjunction with the British Ministry of Defence under a first phase contract valued at about $3 million.

An automatic data processing (ADP) system, Wavell will comprise data processors and terminals inter-connected through the trunks communication system serving a combat zone at corps, divisional and task force headquarters levels.

The system is designed to simplify much of the work of staffs at each plateau of command in logging, sifting, collating, recording and passing information for immediate analysis and evaluation.

It has been estimated that more than 50 percent of divisional operational staff time is at present taken up by using the telephone system to acquire, disseminate, confirm and handle location information.

Plessey Radar has informed Plessey Australia that the system's modular design can be tailored to suit the particular requirements of any country and it can also be integrated with existing communications networks.

DRANETZ MODEL 606
Power Line Disturbance Analyser

You know all about power line behaviour and your installation! The Dranetz Model 606 Line Disturbance Analyser will provide data to enable you to pinpoint the effects of irregular line quality of your equipment.

CONSIDER THESE FUNCTIONS:
- Complete round-the-clock record (up to 99 days) of power line disturbances and interruptions in permanent printed format.
- Monitors, classifies and records time, type and magnitude of each disturbance. Time clock may be set to real or system time.
- Switch selectable threshold values for impulse (Duration 0.5 & 100 μ. secs.), Cycle to cycle changes in RMS levels and slow changes in steady state RMS levels.
- Internal computer accumulates statistical data for daily summary read-out. Data available any time on command.

NOW CONSIDER THESE FEATURES:
- Isolated inputs for 115/230/460 V RMS. Single and three phase models.
- Operates only on disturbance or command—no wasted paper.
- Immediate printout of line disturbances.
- Excellent for permanent monitoring.
- Portable (mass 6.8 kg), weather-proof case.
- Easy to connect, easy to read, permanent record for analysis.

For further details contact:

ELECTRICAL EQUIPMENT LTD
86 Liverpool Street, Sydney, N.S.W. Telephone 61 8321
8 Remont Court, Cheltenham, VICTORIA. Telephone 550 4522
25 Maud Street, Newstead, QUEENSLAND. Telephone 52 3962
Contents:

92-100  Flowchart Optimisation in COPE a Multi-Choice Decision Table  
By B. DWYER and K. HUTCHINGS

101-106  A Course About Social Implications of Computers  
By G. K. GUPTA

107-111  Job Scheduling by a Front End Computer  
By B. J. AUSTIN

112-113  A Computer Program for Exploring the Homogeneity of Randomly Sampled Categorical Data  
By J. P. HIGGINS

114-117  Batch Submission, Demand Terminal and "Hands-on" Computing in a Computer Science Tertiary Education Environment  
By R. A. JARVIS and P. N. CREASY

118-127  Artificial Intelligence Techniques in Computer Assisted Instruction  
By J. A. SELF

Short Communication:

128-129  Attribute Contributions to a Classification  
By G. N. LANCE and W. T. WILLIAMS

Special Features:

100, 106, 111, 129  Book Reviews  
Forthcoming National and International Events