

# A tree-search algorithm for mixed integer programming problems

By R. J. Dakin\*

This paper describes a new algorithm for finding solutions to optimization problems in which some of the variables must take integral values. The algorithm appears to offer some advantages over a similar algorithm proposed by Land and Doig, from which it was developed.

The paper describes computational procedures which result in modest storage requirements. The method has been programmed and used to solve several problems.

Land and Doig (1960) have proposed an algorithm for solving linear-programming problems in which some of the variables are required to be integers. The procedure consists of a systematic search of continuous solutions in which integer variables are successively forced to take integral values; the logical structure of the set of solutions is that of a tree.

As it stands the algorithm gives rise to substantial practical difficulties for computer implementation, since the recording of the tree could involve excessive storage requirements.

This paper presents an algorithm which is similar in concept, but is easier to implement than the Land and Doig algorithm. It appears likely to be more efficient, although this has not been conclusively demonstrated. The algorithm employs a tree search procedure similar to that described by Little, Murty, Sweeney and Karel (1963) in their section entitled "Throw away the tree"; a somewhat more complicated version of this procedure could also be used for the Land and Doig algorithm. We also note that the procedure falls into the general classification of "branch and bound" techniques, as described by Little *et al.*

The algorithm is applicable to both linear and non-linear problems; however, computational experience has so far been obtained for the case of linear programming only.

## The problem

We are required to find values for the variables

$$(x, y) = x_1, x_2, \dots, x_r, y_1, y_2, \dots, y_n \quad (1)$$

which minimize the function

$$z(x, y) \quad (2)$$

and satisfy the conditions:

$$f_i(x, y) = 0 \quad (i = 1, 2, \dots, m), \quad (3)$$

$$\text{the elements of } (x) \text{ are integral} \quad (4)$$

$$\text{and} \quad x \geq 0, y \geq 0. \quad (5)$$

The problem (1), (2), (3) and (5) (without the integrality condition) is referred to as the *continuous problem*. A

solution to the continuous problem which also satisfies (4) is called an *integer solution*; if (4) is not satisfied it is a *non-integer solution*.

We assume that an algorithm exists (which we call the *sub-algorithm*) for finding solutions to problems consisting of the continuous problem with the addition of upper or lower bounds on any of the integer variables. The existence of such an algorithm will almost certainly limit us to problems in which  $z(x, y)$  is concave and (3) and (5) specify a convex region.

The integer algorithm consists of a number of applications of the sub-algorithm to different problems. We shall show that, if the sub-algorithm is finite and the integer variables are bounded, then an optimal integer solution will be reached after a finite amount of computation.

## Outline of the algorithm

Suppose  $x_j$  is an integer variable and  $k$  is an integer. Then, since the range  $k < x_j < k + 1$  is inadmissible, we may divide all solutions to the constraints (3) to (5) into two non-overlapping groups, viz.:

(i) solutions in which

$$x_j \leq k \quad (6)$$

(ii) solutions in which

$$x_j \geq k + 1. \quad (7)$$

The algorithm starts by finding a solution to the continuous problem. If this solution is integral, then it is the solution to the complete problem. If it is non-integral, then at least one integer variable — $x_j$ , say—is non-integral, and takes the value  $b_j$ , say, in this solution.

We now divide  $b_j$  into integral and fractional parts  $[b_j]$  and  $f_j$ , respectively, defined by:

$$b_j = [b_j] + f_j \quad (8)$$

where  $[b_j]$  is integral and  $0 < f_j < 1$ .

We now substitute

$$k = [b_j] \quad (9)$$

into (6) and (7); it is apparent that neither of these relationships is satisfied by the current non-integer

\* *Basser Computing Department, School of Physics, University of Sydney, N.S.W., Australia.*

solution. We now add each of these constraints in turn to the continuous problem and solve these augmented problems.

We then repeat the procedure for each of the two solutions so obtained. The logical structure of this process is that of a "tree"; a typical tree is shown in Fig. 1.

Each node represents a solution to an augmented continuous problem, and is shown with the additional constraint which was applied in order to reach the solution.

The tree will always terminate in one of two ways; we may reach an integer solution (denoted by "I" in Fig. 1) or we may find that the current set of constraints has no solutions (denoted by "NS"). The solution to the complete problem will be the best integer solution reached in this way. Hence the complete problem is solved by searching this tree.

The tree is not, in general, unique for a given problem since at any stage we are at liberty to form the next constraint using *any*  $x_j$  which is non-integral; choice of different  $x_j$  will lead to different trees. As we shall see, the choice can have a large effect on the amount of computation required.

Note that only two branches, and hence two sub-trees, emanate from any one node—contrasting with the Land and Doig algorithm where any number of branches may emanate from a node (e.g. Fig. 8 of Land and Doig (1960) where six branches emanate from the node labelled  $\gamma^0$ ).

More than one constraint may operate on a variable at once. For example, at node 9 in Fig. 1 the constraints  $x_1 \geq 1$  and  $x_1 \geq 5$  both apply. The number of these constraints which may operate on a variable at once is limited by the range of values which the variable may take. Thus, if the original constraints (3) restrict  $x_j$  to the range  $0 \leq x_j \leq v$  then only  $v$  integer bound constraints within this range are consistent with  $x_j = k$ ; viz.

$$x_j \geq 1, x_j \geq 2, \dots, x_j \geq k, x_j \leq k, x_j \leq k + 1, \dots, x_j \leq v - 1.$$

### Computational search procedure

If this algorithm is used in hand computation, then it is sensible to proceed, at each stage, from the lowest cost terminal node which has been reached; this will minimize the searching of unlikely (i.e., high cost) sub-trees. One can use this nodal solution as a starting-point for the optimization if the sub-algorithm permits it, since this is likely to be near to the next solution, and should require less computation than if we started at an arbitrary point (this appears to be true in the case of linear programming).

Such a procedure is impracticable for implementation on an automatic computer, if we are to deal with problems of any size—since the recording of the required information for each node\* involves an excessive

\* For the case of linear programming by the straightforward Simplex method, we would require a complete copy of the tableau for every node.

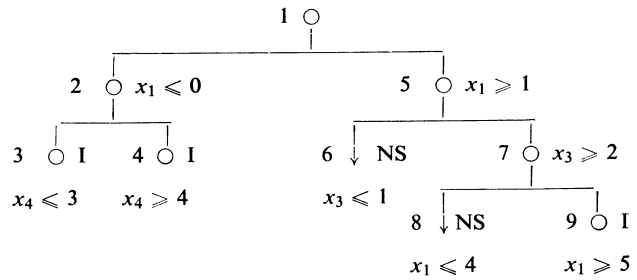


Fig. 1.—Tree representation of search procedure

amount of storage. The use of a backing store (e.g., magnetic tape) for this purpose would involve an excessive penalty in transfer time for most present-day computers.

This difficulty may be overcome if we restrict the order in which the tree is searched so that each branch is followed until an infeasible or integer solution is reached. In this way we may use the solution from the preceding node as starting-point for most optimization steps, but we may search some of the tree which would otherwise be avoided.

All necessary information regarding the current status of the search can be reduced to a list, each entry of which contains details of a node in the tree. At any stage the list represents the chain of nodes leading from the continuous solution node to the current node; the position of the entry in the list is the same as that of the corresponding node in the chain.

Each list entry carries a "list marker" which is set when we have explored one of the two sub-trees emanating from the previous node. Since an alternative constraint is never tried until we have explored the sub-tree emanating from it, the presence of a list marker indicates that the corresponding constraint has been swapped for its alternative.

The amount of information in a list entry—variable name, value of the bound, whether the bound is upper or lower, and list marker—is not excessive and in both computer codes so far written it was possible to pack each list entry into one computer word.

The computational procedure is summarized in the flow diagram in Fig. 2. Referring to the steps in the flow diagram, we may describe the sequence of events in terms of the tree structure as follows:

Step 2 (first time). Solve the continuous problem.

Steps 2, 3, 4, 6. Proceed down a chain in the tree until either an integral solution is reached or the problem becomes infeasible. In practice step 3 will usually be part of step 2.

Step 5. If the solution is integral, we record this solution if it is the best so far. This information, together with the list, is the only storage requirement additional to that required for the sub-algorithm (step 2). We may avoid the need for storing this solution internally by printing it out at this stage.

**Steps 7, 9, 10.** We now proceed to look for a part of the tree which has not so far been explored. We start inspecting earlier nodes of the tree (corresponding to higher list entries) until we find a node—node  $j$ , say—such that one of the sub-trees emanating from it has not so far been explored, i.e., an unmarked list entry. Since we have had to go back as far as node  $j$  to find an unexplored sub-tree, the current sub-tree from node  $j$  must be completely explored. Hence we attach a list marker to the node  $j$  list entry and start going down a chain of the alternative sub-tree from node  $j$ , using steps 2, 3, 4 and 6 as before.

**Step 8.** The search terminates when there are no unexplored sub-trees.

Reverting to the tree shown in Fig. 1, the nodes have been numbered in the same order in which they would be reached by this search procedure. This order is not in general unique, since the choice of which branch to follow at each stage is quite arbitrary. **Table 1** shows how the list would appear at each stage of the process. The table has been laid out such that the list, as it is at any node, consists of the entries appearing next to the node number and extending up to the next horizontal line in the table. It should be noted that the node number and the nature of the solution obtained do not actually appear in the list but are included here as explanation. We notice that although there are nine different solutions to keep track of, the list never contains more than three entries (at nodes 8 and 9).

#### Comparison with Land and Doig method

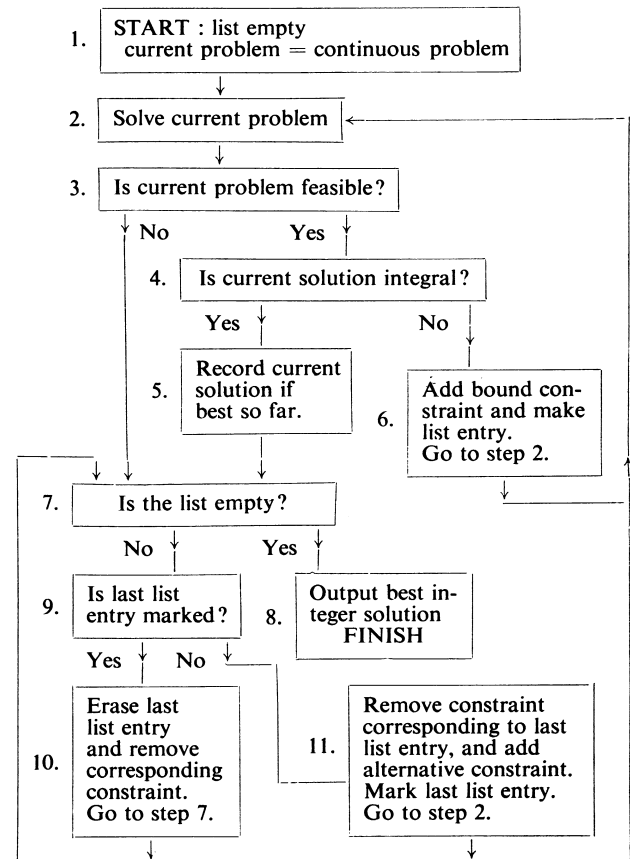
The close connection between this algorithm and that proposed by Land and Doig (1960) has already been noted. The basic difference is that the Land and Doig algorithm forces variables to take exact integral values rather than applying bounds, so that it is necessary to search over the range of integer values for which  $z$  is less than the value it takes at the best known integer solution. If all integer variables are restricted to the values 0 and 1 then the two methods are essentially the same, and the only contribution of this paper is to provide a convenient computational procedure for carrying it out.

For the general case in which integer variables are not so restricted the two methods are quite distinct, and it is not altogether clear which one will produce a solution more rapidly. Indeed, the convergence of both methods is strongly dependent on the choice of integer variable to constrain at each stage.

There are, however, some grounds for expecting more rapid convergence for our algorithm in many cases. This algorithm will, in some circumstances, search over a smaller range of values for an integer variable than the Land and Doig algorithm, since we can stop as soon as an integer solution is reached. Moreover our method will, in some cases, skip over some values of an integer variable which would be tried in turn by the Land and Doig algorithm. Thus, in Fig. 1,  $x_1$  goes from 1 at node 5 to 5 at node 9.

**Table 1**  
**Lists associated with Fig. 1**

NODE	VARIABLE	CONSTRAINT	MARKER	SOLUTION
1	List empty			Non-integer
2	$x_1$	$\leq 0$	no	Non-integer
3	$x_4$	$\leq 3$	no	Integer
4	$x_1$	$\leq 0$	no	Integer
	$x_4$	$\geq 4$	yes	
5	$x_1$	$\geq 1$	yes	Non-integer
6	$x_3$	$\leq 1$	no	Infeasible
7	$x_1$	$\geq 1$	yes	Non-integer
	$x_3$	$\geq 2$	yes	
8	$x_1$	$\leq 4$	no	Infeasible
9	$x_1$	$\geq 1$	yes	Integer
	$x_3$	$\geq 2$	yes	
	$x_1$	$\geq 5$	yes	



**Fig. 2.—Flow diagram for integer algorithm**

On the other hand our method may require the use of two constraints rather than one in order to force a variable to an integral value. Thus the argument is indecisive.

Both A. G. Doig and J. M. Bennett suggested to the author that the Land and Doig algorithm should be amenable to a search procedure similar to that described here. Such a procedure is the following. Instead of a list marker, it is necessary to incorporate in each list entry the range of integer values for which the corresponding sub-trees have been explored, together with two markers which indicate that the value of the variable has been increased or decreased as far as it can be (without incurring infeasibility). All sub-trees emanating from a node would then be regarded as exhausted when both markers are set.

### Speed considerations

Experience to date indicates that the convergence of the algorithm is, to a very considerable extent, dependent on how certain details of the algorithm are applied. The amount of computation involved in finding a solution depends on two principal factors: the number of minimizations and the amount of computation involved in each minimization. The latter depends on the particular type of problem and the sub-algorithm used, and will not be dealt with here. The number of minimizations is strongly dependent upon the number of entries in the list, since if for all integer or infeasible solutions the list were of length  $K$ , then  $2^K$  such solutions would be reached, involving  $2^{K+1} - 1$  minimizations in all. In practice, the list length will not be the same for all infeasible or integer solutions, but the need for limiting it is apparent.

### Bound constraints on $z$

Land and Doig (1960) point out that once we have reached an integer solution with  $z = z_0$ , say, then at any later stage where  $z > z_0$  it will be pointless to add any further constraints which will make  $z$  still greater. Hence we may add the constraint  $z \leq z_0$ , which makes the condition  $z > z_0$  appear as an infeasibility. This constraint will be adjusted each time an integer solution is reached.

Land and Doig suggest setting  $z_0$  at some initial value at the start of the procedure—a useful device, provided we know enough about the problem to be able to set a reasonable value.

Another device is to replace  $z_0$  by  $z_0 - e$ , where  $e$  is a constant, whenever an integer solution is reached. This will have the effect of producing a solution which, though not necessarily optimum, is known to be within  $e$  of the true optimum. If this leads to a substantial reduction in computation then it may very well compensate for obtaining only an approximate result.

### Choice of constraints

After any application of the sub-algorithm several of the integer variables may take non-integral values. A number of selection criteria have been tried in linear

programming examples with widely varying results. By far the best (in terms of Simplex iterations) has been to look for the constraint which will lead to the largest increase in  $z$  during the first Dual Simplex iteration performed after adding the constraint (this can be determined without actually performing the iteration), and to add either this constraint or its alternative. In order to determine the change in  $z$  we first add the constraint and go through the motions of pivot selection for the Dual algorithm—a procedure similar to the “parametric programming” method described by Land and Doig (1960).

The aim of the criterion is to violate the  $z \leq z_0$  constraint as quickly as possible, and hence limit the length of the list. The choice of the alternative constraint will alter only the order of search, not the tree itself; we would, however, expect to reach a low cost solution more rapidly with the latter choice.

Other criteria which we have tried include the selection of upper bounds on the non-integral  $x$  variable with the smallest non-zero fractional part, the one with the largest fractional part and the one which is nearest to the top of the tableau. All of these criteria required at least twice as many Simplex iterations as the “largest change in  $z$ ” criterion for the problems tried.

If, in the course of selecting a constraint by the “largest change in  $z$ ” criterion, we find a constraint which would immediately lead to infeasibility or to  $z_0$  being exceeded, then the alternative constraint must apply. We may therefore add any such constraints, and their alternatives need not be considered. In practice this has made only slight improvements to the performance of the algorithm.

### Computational schemes for linear programming

Although we have regarded the constraints (6) and (7) as additional to the original constraints (3) and (5) it is possible, in the case of linear programming, to incorporate them without increasing the size of the problem.

For the case where the integer variables may take only the values 0 or 1, provided a bound constraint  $x_j \leq 1$  which is equivalent to  $x_j + x'_j = 1$  is included for all integer variables  $x_j$ , then  $x_j$  may be forced to either 0 or 1 (equivalent to the “additional constraints”  $x_j \leq 0$  and  $x_j \geq 1$ ) by forcing either  $x_j$  or  $x'_j$  out of the basis.

For the more general case in which the integer variables, though restricted in range, may take on values other than 0 or 1, we make use of the “bounded variables” technique (Charnes and Lemke, 1954; Dantzig, 1955) which provides for the inclusion of the effects of bound constraints such as:

$$x_j \leq u_j \quad (10)$$

without actually including (10) in the constraint matrix. We set initial bounds on all of the integer variables, and employ an adaptation of this technique in order to insert further bounds. Thus the constraint

$$x_j \leq k \quad (6)$$

is effected by changing the value of the upper bound to  $k$ , while the constraint

$$x_j \geq k + 1 \quad (7)$$

is effected by a change of origin: we substitute

$$x_j = x_j'' + k + 1 \quad (11)$$

and replace the previous bound— $u_j$ , say,—by  $u_j - k - 1$ . The implicit condition  $x_j'' \geq 0$  will then ensure that (7) is satisfied.

#### Storage requirements

The only data storage requirement, apart from the requirements of the sub-algorithm, is that required for the list. As we have seen, the number of additional constraints, and hence the length of the list, cannot exceed  $\sum_j v_j$ , where  $v_j$  is the range of values for the integer variable  $x_j$ . The list usually is a small proportion of the total.

In the computer codes so far written for linear programming, using the Simplex method as sub-algorithm and updating the complete tableau, the total data storage requirement has been

$$\sum_j v_j + (m + 2)(n' + 2) \text{ words,}$$

where  $m$  is the number of constraints in the continuous problem (excluding upper-bound constraints if the bounded variables procedure is used), and  $n'$  is the number of non-basic variables.

#### Computational experience

A summary of the performance of the algorithm in solving six problems is given in Table 3. These calculations were carried out on an English Electric-Leo KDF9 computer, using the bounded variables procedure. Details of the problems are given in Table 2. The iteration counts include only iterations in which the tableau is updated in the normal manner, and exclude cases in which a bounded variable goes from its lower to its upper bound or vice versa.

Performing Simplex iterations is the major part, but not the only part, of the computation; the handling of bounds and selection of new constraints are also substantial items. Analysis of the computation time indicates that about half of the time is spent on performing Simplex iterations by the KDF9 program.

A number of other problems have been solved by the algorithm; the results quoted are typical. Two problems similar to Nos. 5 and 6 have, however, failed to yield optimal or near-optimal solutions within a reasonable time.

The computation involved in solving problems 1 and 4 appears to be similar to that required by Healy (1964) using Multiple Choice Programming, although Healy's results are given in a form which makes exact comparison difficult. Problem 3 was solved faster by Gomory's mixed integer algorithm (Gomory, 1960),

Table 2

Details of integer programming examples

NO.	SIZE*	INTEGER VARIABLES		DATA STORAGE (words)	DESCRIPTION
		NO.	RANGE		
1	$7 \times 15$	15	0-1	168	Fixed charge problem (Healy, 1964)
2	$26 \times 9$	8	0-1	316	Phase problem (Freeman <i>et al.</i> , 1963)
3	$31 \times 39$	44	0-4	1528	Power systems problem (Dakin, 1961)
4	$15 \times 30$	30	0-2	574	Liquids and containers problem (Healy, 1964)
5	$12 \times 99$	91	0-1	1505	Phase problem (Dakin, 1964)
6	$17 \times 93$	80	0-1	1885	Phase problem (Dakin, 1964)

\* (Linear inequality constraints, excluding objective and upper bounds)  $\times$  (non-basic variables)

Table 3

Performance of integer programming algorithm

PROB. NO.	CUMULATIVE SIMPLEX ITERATION COUNTS				TOTAL TIME* MIN., SEC.
	CONTINUOUS SOLUTION	FIRST SOLUTION	FINAL SOLUTION	END OF SEARCH	
1	7	10	10	18	0, 5
2	16	64	189	201	0, 11
3	24	36	50	75	0, 12
4	23	210	210	484	0, 28
5†	27	440	3155	>9700	14, 50
6†	90	445	460	>4600	7, 28

\* Central processor time for KDF9, including input/output conversions but excluding peripheral transfer times.

† These runs were terminated before completion. In both cases the first solution was substantially the same as the known optimal solution.

which required only 30 iterations. On the other hand problems 4 and 6 were unsolved by the Gomory algorithm after completing over 2000 iterations in each case.

#### Conclusions

One feature of the algorithm which could prove to be valuable is its ability to produce near-optimal solutions fairly rapidly, even in cases when it takes an unreasonably

long time to obtain an optimal solution and prove that it is optimal.

In common with other integer-programming algorithms it cannot be guaranteed to solve *all* problems within a reasonable time. A useful development in integer-programming theory would be the determination of classes of problem which can and cannot be readily solved. Our experience indicates that these classes may be different for different algorithms. In this connection, we have not had any failures with problems which have been solved by other methods, although our experience has been too limited to make any definite conclusions from this.

## References

- CHARNES, A., and LEMKE, C. E. (1954). "Computational Theory of Linear Programming I: 'Bounded Variables' Problem," *O.N.R. Research Memorandum* No. 10.
- DAKIN, R. J. (1961). "Application of Mathematical Programming Techniques to Cost Optimization in Power Generating Systems," M.Eng.Sc. Thesis, University of Sydney.
- DAKIN, R. J. (1964). "Some Integer Programming Formulations of the Phase Problem," *Basser Computing Dept. Tech. Rept.* No. 34.
- DANTZIG, G. B. (1955). "Upper Bounds, Secondary Constraints and Block Triangularity in Linear Programs," *Econometrica*, Vol. 10, p. 174.
- FREEMAN, H. C., SIME, J. G., BENNETT, J. M., DAKIN, R., and GREEN, D. (1963). "Linear Programming in the Solution of Crystal Structures," Paper presented at the Symposium on Crystallographic Computing Methods, International Union of Crystallography, Rome, 18 Sept., 1963.
- GOMORY, R. E. (1960). "An Algorithm for the Mixed Integer Problem," RAND paper RM-2597.
- HEALY, W. C. (1964). "Multiple Choice Programming," *Operations Research*, Vol. 12, p. 122.
- LITTLE, J. D. C., MURTY, K. G., SWEENEY, D. W., and KAREL, C. (1963). "An Algorithm for the Travelling Salesman Problem," *Operations Research*, Vol. 11, p. 972.

Although our experience, and some of the details of the algorithm, are limited to linear-programming problems, the basic principles of the algorithm apply to non-linear problems as well.

## Acknowledgements

I am grateful to Professor J. M. Bennett and Miss A. G. Doig for reading an early draft of this paper and making a number of helpful comments and suggestions, some of which are incorporated in this version.

This work was supported by Air Force Office of Scientific Research Grant AF-AFOSR-62-402.

## Book Review

*L'Automatisation des Recherches Documentaires: un modèle générale, Le SYNTOL*, by R. C. Cros, J. C. Gardin and F. Levy, 1964; 260 pages. (Paris: Gauthier-Villars, 30 F.)

One important question in information-retrieval research is how far document descriptions should be structured: should we describe a document on exports from Britain to America simply by the term list "exports, Britain, America," or by the encodement "exports  $r$  Britain  $r$  America," where  $r$  is some relation? The former is simpler but is indiscriminating: we get documents on exports in either direction. The latter is more selective, but makes retrieval more complicated: should the request " $Ar_1(Br_2C)$ " retrieve documents encoded by " $(Ar_1B)r_2C$ ," say? Most retrieval systems have some lexical organization of terms, to allow searches by associated terms; this already means some complexity, which is much increased if encodements have a syntactic structure. The initial document analysis is also much more work.

In this situation there is an obvious interest in finding the best combination of terms and syntax, and this book describes one of the most thorough and interesting attempts to do so. An encodement in Le SYNTOL consists essentially of terms linked by general logical relations; the terms are incorporated in lexical trees, and there are additional devices for marking the main theme, for connecting trees, and so on. Associated are bodies of rules, for making encodements, and for operating on them in retrieval. The system is described in detail, with chapters on the theory, programming, and experimental results. It is intended to be flexible, so that retrieval may be done by any combination of the aspects of the encodements. The important point is how effective is the detailed relational structure? Unfortunately, the answer seems to be that it is not effective enough, unless one is prepared to pay the cost of very heavily controlled and exhaustive searches.

KAREN SPARCK JONES