5

TREE-SEARCHING METHODS WITH AN APPLICATION TO A NETWORK DESIGN PROBLEM

R. M. BURSTALL
EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

SEARCH TREES

We will talk about problems with the following characteristics:

(i) We could recognise a solution to the problem if given one.
(ii) The solution is known to lie in some finite set of objects which is nevertheless too large to enumerate in practice.
(iii) We have no straightforward means of calculating the solution or solutions, without searching through at least part of the set of objects.

Let us call the set of objects which is known to contain the solutions the 'candidates'. We include cases where there is more than one solution or where an optimal solution is required.

Some examples are:

(i) In playing chess there are only a finite number of possible strategies (candidates) but the number is far too large to enumerate.
(ii) In industrial scheduling we might have to assign some jobs to some machines to satisfy certain conditions. Any assignment is a candidate and there are a finite, but usually large, number of assignments.
(iii) Given a set of points and a table of distances between them we might want to find the shortest route passing through each point just once (the 'Travelling Salesman' problem). The candidates are the $(n-1)!$ permutations of the points omitting the starting point.

In this section we will describe in an abstract way two approaches to problems of this type. We will give examples of their use, first in 'Some problems about sets' (p. 69 some examples of special search algorithms for problems about choosing subsets of a set, and then in 'An electricity

network design problem' (p. 75) applications of the approaches in the solution of a network design problem. Although both the approaches have often been used before, the discussion may help to clarify those features common to the different applications.

Our two approaches are both search techniques (partial enumeration techniques). The first we will call 'the method of neighbours' and the second 'the method of decision trees'. We claim no special merit for these names, but they will serve for the present.

In our first method we define a relation called neighbourhood between pairs of candidates. This should have the property that from any candidate, or at least from some known starting set of candidates, it should be possible to get to any candidate which might be a solution by means of a chain of candidates each of which is a neighbour of the last. For example, if the candidates were permutations the neighbours of a permutation might be all permutations obtainable from it by interchanging a pair of elements. Beginning with the candidate or candidates of the starting set we may take all neighbours of each, then all their neighbours, etc.

In order to save computing time we may not wish to explore all neighbours, or at least if we do explore them all we wish to do it in an order which is likely to bring the desired solutions more quickly. To eliminate some neighbours or to choose the order we may use heuristic methods, i.e., methods which are supposed to be helpful but not guaranteed to be so, or we may use some properties of the candidates, if we know any useful ones, to rule out certain candidates or sets of candidates. A common device is to assign a value to each candidate by means of an evaluation function and explore only neighbours with better values, or at least explore these first. In the extreme case we take only the best valued neighbour and if this has a value worse than the last candidate we simply give up, or start at random somewhere else (Reiter & Sherman 1963). We may call this the linear case. If on the other hand we sometimes take more than one neighbour we will be searching a tree of candidates (compare the contribution to these proceedings by J. E. Doran). Another slightly different method is to define more than one kind of neighbourhood relation and then decide which kind of neighbour is likely to be more promising. For example, in the case given above, permuting two elements which lie far apart might be more favourable for some reason. If there are a lot of neighbours this may save us generating them all and evaluating the merit of each.

Our second method is to construct a decision tree. It is similar to, but rather more general than, the method of 'backtrack programming' which has just been reviewed by Golomb & Baumert (1965). In this case we start off with an open mind, considering any candidate as a possible solution and then make the hypothesis that the solution satisfies some condition. We must of course remember to consider also the alternative that it fails to satisfy the hypothetical condition (or if there are many solutions we must consider in turn both those satisfying and those not satisfying it). For each

case we then impose a further condition, considering also the opposite case and so on. If we go on long enough, the accumulated conditions will have specified a unique candidate, in which case we look to see whether it is a solution; or possibly we can see from them that no candidate satisfying the conditions could possibly be a solution, in which case we can terminate this branch of the search. In general we will branch not just into two (a condition holding or failing) but into any number of mutually exclusive conditions.

Instead of talking about conditions it may be easier to talk about the corresponding sets of candidates, saying that each node of the tree corresponds to a set of candidates. The initial node, where we impose no conditions, corresponds to the set of all candidates. If a node corresponds to a set S of candidates then either S is a unit set or the node has successors which correspond to sets $S_1, S_2, \ldots, S_n$ forming a partition of S, i.e.,

$$S_1 \cup S_2 \cup \ldots \cup S_n = S \quad \text{(they cover } S\text{)}$$
$$S_i \cap S_j = nil \quad \text{for all } i, j \quad \text{(they are disjoint)}$$

We demand that they cover S so as not to ignore any possible solutions. We demand that they are disjoint so that we can be sure not to waste our time considering the same set (i.e., $S_i \cap S_j$ if this is non-empty), more than once. Disjointness is sometimes very difficult to arrange without laborious checking or carrying a great deal of information and it may be the major problem in defining a suitable decision tree. The dilemma is that if we characterise the sets S_i too loosely they will not be disjoint but if we do it too restrictively they will not cover S. This is reminiscent of the difficulties met in choosing an axiom set for some mathematical system; if the axioms are too loose we may be able to prove contradictions, if they are too restrictive we may not be able to prove statements which we would like to have as theorems.

Let us take a simple example to make this more concrete. Suppose that four players a, b, c and d have taken part in a championship with the following results

d beat a	a beat b	c beat b
c beat d	b beat d	a beat c.

We wish to rank the four players in the sequence with the smallest number of discrepancies, defining a discrepancy as: x precedes y but y beat x. The candidates are all possible permutations, and we may construct the decision tree shown in Fig. 1, partitioning the set of all permutations into those in which d precedes a ($d > a$) and those in which $a > d$, then continuing with other similar partitions, until we eventually partition into unit sets.

Having specified a set of candidates S at some node it may be possible by looking at its defining conditions to reduce it to a set S' such that $S' \subset S$ and S' contains all solutions in S. We then proceed by partitioning only S'. In the extreme case we can reduce S' to the empty set and stop exploring the branch altogether. To do this for the tree of permutations in Fig. 1 we must agree to explore it in a certain sequence, e.g., always taking the upper

branch first. If we then keep a record of the best sequence found to date and its number of discrepancies we can frequently reject a whole sub-tree by noticing that *any* permutation in the set corresponding to this sub-tree must have more discrepancies than the best to date. The circled numbers

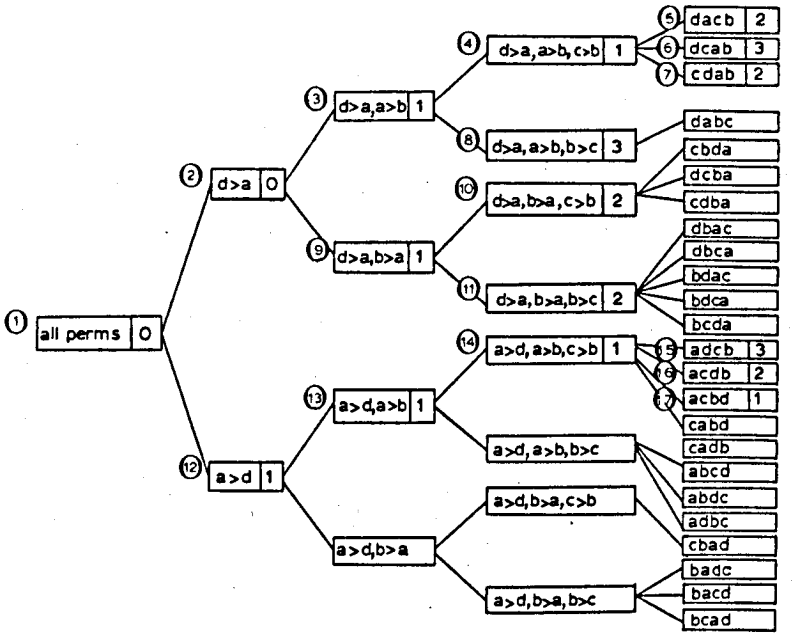


FIG. 1. Decision tree for ranking problem. The numbers in circles show the sequence in which the tree is explored. The numbers in the boxes show the minimum possible number of discrepancies, or the exact number if the box is a tip of the tree.

show the sequence of exploration, unnumbered boxes being ruled out as just explained. The number to the right of each box shows the least number of discrepancies which could possibly occur in any permutation of the set corresponding to that box.

This is an example of a logical exclusion rule which enables us to rule out certain parts of the tree without sacrificing any chances of getting the desired solution. We may also use heuristic exclusion rules, and heuristic principles may also be used to guide us in choosing the sequence of exploration. Another paper in this collection, by J. E. Doran, discusses in detail a particular class of heuristic sequencing rules.

It is often important to choose a useful definition of the decision tree for a given problem, i.e., a useful method of partitioning the candidates. For example, in the permutation problem, instead of the partitioning given above we could have first partitioned according to the first letter of the permutation (a 4-way split), and then according to the second letter (a 3-way split) and so on. This might have led us to some more powerful exclusion rule and reduced the amount of exploration still further. Some problems such as board games with a sequence of moves have a natural tree structure; specify

the first move, then given this specify the second move and so on. Before basing the algorithm exclusively on this it may be worth examining other decision tree structures which might be imposed, e.g., specifying that certain states will or will not be reached, or specifying the last move first.

Before leaving the permutation example we may note that it would also be amenable to our first method, the method of neighbours. For example, we could define the neighbours of a permutation as all permutations which can be obtained from it by exchanging a pair of adjacent elements. Thus from 'abcd' (2 discrepancies) we could move to any of its neighbours 'bacd' (3 discrepancies), 'acbd' (1 discrepancy) or 'abdc' (3 discrepancies). The best of these being 'acbd' we examine its neighbours, and so on until we fail to secure any improvement. Notice that in this method we would not know whether we had found the optimal solution, whereas using the decision tree method we would be sure of this, an important advantage.

Decision trees for optimising problems which use the current best value to exclude parts of the tree, on the ground that they cannot contain solutions any better than this, have been called 'Branch and Bound' algorithms (Little, Murty, Sweeney & Karel 1963). Little and his colleagues give an algorithm for the 'Travelling Salesman' problem (shortest route through a number of points, visiting each point just once). A number of people have used the method of neighbours for this problem (Reiter & Sherman 1963). The paper by Reiter and Sherman gives a discussion of the general method. Amongst other applications of the decision tree method we may mention assembly line balancing (Tonge 1960), scheduling (Lomnicki 1965), job-sequencing (Burstall 1966*b*) and solving integer inequalities (Burstall 1965*b*).

SOME PROBLEMS ABOUT SETS

Successor families

We will now show how the decision tree method can be used for some problems about choosing sets of things. A natural way of going about such problems is to choose the elements of the set one at a time. This is a particular way of imposing a decision tree structure on the problem. We will investigate some problems where it is possible to construct such a decision tree satisfying the essential conditions of the previous section: (i) $S_1 \cup S_2 \ldots \cup S_k = S$ and (ii) $S_i \cap S_j = nil$.

We suppose that we are given a set U and the solution to our problem is a family of subsets of U. We have a means of recognising whether a given subset belongs to the family or not, and the problem can be solved by generating all subsets of U and testing each with our recogniser. However, the number of subsets to be tested grows rapidly with the number of elements in U and may become hopelessly uneconomic, so we seek a means of generating only the ones belonging to the required family. In some cases we can do this quite easily, namely, cases where we can find a generating rule such that each set of the family (except the empty set) can be obtained by

adding a point to some other set of the family. We will call such a family a 'successor family'. Of course, not all problems about families of sets can be characterised in this way. We give below a more formal definition of 'successor family' and five examples of problems which can be formulated in this way. This definition is *constructive*, allowing us to find an algorithm for building up the required sets, as opposed to a merely *predicative* definition which allows us to recognise the required sets.

However, although the constructive definition can easily be turned into an algorithm we may seek a more efficient algorithm. The difficulty in building up sets of the family element by element is to avoid the twin dangers of omitting a set and of generating the same set twice. We can avoid omissions by considering all possible elements which could be added. But to avoid duplication of sets we will generally have to keep some sort of list of the sets already encountered and test each new set against this list, an expensive business in both computing time and storage. Now under certain conditions we can manage without doing this testing, and we will describe a decision tree algorithm for generating successor families which works quickly and uses very little storage.

Under certain further conditions this algorithm which calls itself recursively twice is equivalent to one with only one recursive call and a loop. It is this latter algorithm which seems to be more obvious when first approaching problems of this type. However, it is a trap (into which I fell at one time) because although plausible it fails for some cases and it is not easy to see at first exactly what has gone wrong.

We will give a proof that the more general decision tree algorithm does generate the sets of a successor family. This may be more convincing than computing results alone, and it may also be of interest in connection with the paper in these proceedings by D. C. Cooper, in which he discusses some proofs that programs are correct.

Let us now return to the definition of successor families of subsets of the given set U.

We will use lower-case letters for elements of U, upper-case letters for subsets of U and script letters for families of subsets of U (we say 'families of subsets' rather than 'sets of subsets' in the hope that it is less confusing). We use '*nil*' for the empty set. Given a function ϕ from subsets of U to subsets of U such that $\phi(S) \cap S = nil$ we say that a set P is an *immediate successor* of R ($R < P$) if $P = R \cup \{t\}$ for some $t \in \phi(R)$. We say that Q is a *successor* of R ($R \ll Q$) if either $Q = R$ or there is a P such that $R < P$ and $P \ll Q$. We now say that $\mathcal{F}$ is a *successor family* generated by ϕ if

(i) $nil \in \mathcal{F}$;

(ii) for all non-empty $P \in \mathcal{F}$ there is an $R \in \mathcal{F}$ such that $R < P$, i.e., every set in the family is empty or it is the successor of another set in the family.

We say that a successor family $\mathcal{F}$ is 'accessible' if whenever $P \subseteq Q$ and $P \in \mathcal{F}$ and $Q \in \mathcal{F}$ then $P \ll Q$. The decision tree algorithm will apply only to accessible successor families. Accessibility just means that any set of the

family can be derived from any of its subsets which are in the family by successively adding elements.

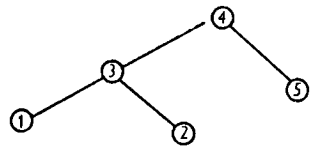
We will now give some examples of accessible successor families.

Example 1: Cost function

Suppose C is a function which assigns a cost to a set R and is monotonically non-decreasing, i.e., if $P \subseteq R$ then $C(P) \leqslant C(R)$, and we wish to generate all sets with cost less than α. We set $\phi(P) = \{t \in U - R \mid C(R \cup \{t\}) \leqslant \alpha\}$ and the successor family is just the family of sets required. More generally, if A is a predicate meaning acceptable, and implies $P \subseteq R$ and $A(R)$ AC (P), then we can generate all acceptable sets by taking $\phi(R) = \{t \in U - R \mid A(R \cup \{t\})\}$.

Example 2: Connected subsets of a graph

Suppose that Γ is a symmetric graph over U, i.e., there is a relation $\gamma(r, t)$ between members of U such that $\gamma(r, t)$ implies $\gamma(t, r)$. We say that $S \subseteq U$ is connected if either it is empty or it cannot be partitioned into two sets S_1 and S_2 such that for all $s_1 \in S_1$ and $s_2 \in S_2$ not $\gamma(s_1, s_2)$, e.g., in the graph below {1, 2, 3} is connected but {1, 2, 4} is not.



The problem is to generate all the connected subsets of Γ. We do this by taking

$$\phi(R) = \{t \in U - R \mid R = nil \vee \exists\, r \in R\ [\gamma(r, t)]\}$$

We will use this example in the 'Electricity network design problem, (p. 75).

Example 3: Strongly connected subsets of a graph

Suppose Γ as in the previous example, we call a subset S strongly connected if it is empty or for all $s \in S$ and $t \in S$, $\gamma(s, t)$. To generate all the strongly connected subsets of Γ we take

$$\phi(R) = \{t \in U - R \mid R = nil \vee \forall\, r \in R\ [\gamma(r, t)]\}$$

Example 4: Non-redundant covers of a set

Let σ be a set and U be a family of subsets of σ. We define

$$sumset(S) = \{\alpha \mid \exists\, s \in S\ [\alpha \in s]\}$$

i.e., the members of members of S.

We say that S is a *cover* of σ if $\sigma \subseteq sumset(S)$, and that S is a *non-redundant cover* of σ if no proper subset of S is a cover of σ. We wish to generate all

the non-redundant covers of σ. All such sets (and some that are not covers) are contained in the successor family with respect to

$$\phi(R)=\{t \in U-R \mid sumset(R) \subset sumset(R \cup \{t\})\}$$

For an example of the use of an algorithm similar to this in a job sequencing problem see Burstall (1965*b*).

Example 5: Linear combinations of points

Let U be a set of points in n-space. We say that a point $q \in U$ is between points p and $r \in U$, '*Between* (p, q, r)', if q is in the line segment joining p and r. Given a basis set of points $B \subseteq U$ we say that a set $R \subseteq U-B$ is derived from B if any point in R is between two other points in $R \cup B$. To obtain the family of sets derived from B we take

$$\phi(R)=\{t \in U-R-B \mid \exists\, s \in R \cup B\ \exists\, u \in R \cup B\ [Between\ (s, t, u)\ ]\}$$

Between is a particular three-place relation, but the method generalises quite easily to any other three-place relation and, indeed, to any k-place relation.

The Algorithm

What is the connection between successor families and decision trees? How can we use it to get an algorithm?

Suppose that we have already chosen some elements of U, say a set R, and wish to add further elements. A natural approach would be to associate a node of the decision tree with R. We could say that it corresponds to that set of candidates (family of subsets of U) defined as $\mathscr{F}=\{P \subseteq U \mid R \subseteq P\}$. However, this is not good enough since two different sets R and R', would have overlapping families, both containing $R \cup R'$, for example, and this contradicts the condition of 'successor families': $S_i \cap S_j = nil$. The trick is to associate with a node a pair of disjoint sets R and S, R being the elements of U we have chosen and S being the elements of U which we may choose. We then associate with the node labelled '(R, S)' the set of candidates $h(R, S)=\{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup S\}$, i.e., sets obtained by adding to R elements of S. Now if we choose some element $t \in S$ we may either add it to R or not. This partitions $h(R, S)$ into two subsets:

$$h(R \cup \{t\}, S-\{t\}) \text{ i.e. } \{P \subseteq U \mid R \cup \{t\} \subseteq P \wedge P \subseteq [R \cup \{t\}] \cup [S-\{t\}]\}$$

and $$h(R, S-\{t\}) \text{ i.e. } \{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup [S-\{t\}]\}$$

It is easy to verify that these sets are disjoint and that their union is $h(R, S)$.

Now $h(nil, U)$ contains all subsets of U and $h(R, nil)$ contains just the set R so that the function h defines the nodes of a complete decision tree.

Note, however, that we have defined h *predicatively* by a property of the sets belonging to it. This is simple on paper but computationally inefficient. The equivalent constructive definition is (see Appendix 1 for notation):

$$\begin{aligned} g(R, S)=&\ null(S) \rightarrow \{R\},\\ &[\text{let } t=any(S);\\ &\ g(R \cup \{t\}, S-\{t\}) \cup g(R, S-\{t\})] \end{aligned}$$

Now we can see how the successor families can be used. Instead of choosing any element $t \in S$ we can restrict ourselves to choosing an element $t \in \phi(R) \cap S$ since this is the way to get the successors of R. Does it matter which element t we choose? If we are dealing with an *accessible* successor family the answer is no; this is precisely why we introduced the idea of accessibility. To put it another way, provided the successor family is accessible we can build up any member of it element by element *in any order*. Before justifying these assertions we will present our algorithm as a recursive function (see Appendix 1 for notation). We define a function $f(U, \phi)$, whose value is the accessible successor family over U with respect to ϕ. The definition uses an auxiliary function $g(R, S)$.

$$
\begin{aligned}
&\textbf{let} \quad f(U, \phi) = g(nil, U) \\
&\textbf{where rec } g(R, S) = [\textbf{let } T = S \cap \phi(R); \\
&\qquad\qquad null(T) \rightarrow \{R\}, \\
&\qquad\qquad [\textbf{let } t = any\ (T); \\
&\qquad\qquad g(R \cup \{t\}, S - \{t\}) \cup g(R, S - \{t\})]\ ]
\end{aligned}
$$

Theorem

If $\mathscr{F}$ is the successor family over U with respect to ϕ and $\mathscr{F}$ is accessible then $f(U, \phi) = \mathscr{F}$.

Proof of theorem

We first of all note that $g(R, S)$ is defined for all R and S. If S has no elements then for any R, $g(R, S) = \{R\}$ since T is empty. If S is not empty then either T is empty, in which case $g(R, S) = \{R\}$ or $g(R, S)$ is defined in terms of $S - \{t\}$ which has fewer elements than R. Since the 'function' *any* is supposed to pick any element of T it is not immediately clear that the value of $g(R, S)$ is independent of the element chosen. However, this is in fact the case as can be seen from a lemma which we now prove:

Lemma: If $R \in \mathscr{F}$ and $R \cap S = nil$ then $g(R, S) = h(R, S) \cap \mathscr{F}$, where as above

$$h(R, S) = \{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup S\}.$$

Proof of lemma: Suppose $R \in \mathscr{F}$ and $R \cap S = nil$.

Case (*i*). Suppose $S \cap \phi(R) = nil$. Then $g(R, S) = \{R\}$ by the definition of g.

Now $R \in \mathscr{F}$ by hypothesis and $R \in h(R, S)$ by the definition of $h(R, S)$. Therefore $\{R\} \subseteq h(R, S) \cap \mathscr{F}$.

We now show that $h(R, S) \cap \mathscr{F} \subseteq \{R\}$.

Suppose $P \in h(R, S) \cap \mathscr{F}$. Then $R \subseteq P \subseteq R \cup S$ by definition of h.

But since $\mathscr{F}$ is accessible, $R \subseteq P$ implies $R \ll P$, i.e., $P = R$ or else there is a Q such that $R < Q \ll P$.

But by definition of immediate successor $Q = R \cup \{t\}$ where $t \in \phi(R)$.

It also follows easily from the definition of successor that $Q \ll P$ implies $Q \subseteq P$ which gives $Q \subseteq R \cup S$ and hence since $R \cap S = nil$, $t \in S$.

We have that $t \in S \cap \phi(R)$ but this is empty by hypothesis, therefore there is no such Q and hence if $P \neq R$ then $P \notin h(R, S) \cap \mathscr{F}$, i.e., $h(R, S) \cap \mathscr{F} \subseteq \{R\}$.

We now have $h(R, S) \cap \mathscr{F} = \{R\} = g(R, S)$.

Case (ii). Suppose $S \cap \phi(R)$ is not empty. We now prove the lemma by induction on the number of elements in S.

If S is empty then $S \cap \phi(R)$ is empty and Case (i) above applies.

We now show that the lemma holds for $g(R, S)$ for any R, assuming as induction hypothesis that it holds for $g(R', S-\{t\})$ for any R' and any $t \in S$.

Suppose $t \in S \cap \phi(R)$, $R \in \mathscr{F}$ and $S \cap R = nil$.

Then $g(R, S) = g(R \cup \{t\}, S-\{t\}) \cup g(R, S-\{t\})$.

But for both these g expressions the conditions of the induction hypothesis hold, i.e., $R \cup \{t\} \in \mathscr{F}$ since $R < R \cup \{t\}$), and $[R \cup \{t\}] \cap [S-\{t\}] = nil$, also $R \in \mathscr{F}$ and $R \cap [S-\{t\}] = nil$.

Therefore

$$\begin{aligned} g(R, S) &= [h(R \cup \{t\}, S-\{t\}) \cap \mathscr{F}] \cup [h(R, S-\{t\}) \cap \mathscr{F}] \\ &= [h(R \cup \{t\}, S-\{t\}) \cup h(R, S-\{t\})] \cap \mathscr{F} \\ &= h(R, S) \cap \mathscr{F} \end{aligned}$$

by the identity mentioned when h was first defined.

This completes the proof of the lemma.

Now we had $f(U, \phi) = g(nil, U)$.

But this is $h(nil, U) \cap \mathscr{F}$ and we observed before that $h(nil, U)$ is the set of all subsets of U and hence $h(nil, U) \cap \mathscr{F} = \mathscr{F}$.

This concludes the proof.

In all the five examples given above the solutions could be characterised as an accessible successor family with respect to some stated ϕ, so we can use the function f to generate the solutions to any of these examples. In Appendix 2 we give an Algol program corresponding to f. We also give a slightly simpler Algol program corresponding to a variant of f, which may be used when a certain extra condition holds, namely: if $P \subseteq R$ then $\phi(R) \subseteq \phi(P)$. This condition holds for Examples 1, 3 and 4, but it does not hold in general for Examples 2 or 5.

We can make a useful generalisation of the function f to deal with what we may call 'collections' or unordered lists. These are like sets in having no order of the elements, but elements are allowed to occur more than once, e.g., $\{a, b, b, d, d\}$ is a collection and the same as $\{b, a, d, d, b\}$. Another representation would be $\langle 1, 2, 0, 2\rangle$ associated with the alphabet $\langle a, b, c, d\rangle$. We assume operations analogous to unit set and union in the natural way, e.g., $\{a, b, b\} \cup \{a\} = \{a, a, b, b\}$. Then the analogue of $f(U, \phi)$ is $f'(U, \phi)$ where f' is a new function, U is a set and ϕ is a successor function carrying a collection into a set; f' produces a set of collections. The definition of f' is formally the same as that of f except that the term $g(R \cup \{t\}, S-\{t\})$ is replaced by $g(R \cup \{t\}, S)$ since t may be added more than once to the collection R. Example 1 above carries over in a natural way to collections and

we shall use this in the 'Electricity network design problem'. Example 4 also carries over to the case where σ is a collection.

AN ELECTRICITY NETWORK DESIGN PROBLEM

The problem

As an example of the use of tree-searching methods I have developed an algorithm for designing electricity distribution networks. This research was assisted by discussions with engineers of the Central Electricity Generating Board, particularly U. G. Knight of the H.Q. System Planning Branch. For a more detailed account see Burstall (1965*a*).

This work is concerned with the design of national grid and city supply networks, a complex task in which computers at present play only an ancillary role. My program is able to create designs to meet the fundamental loading and security requirements. These designs can then be scrutinised by engineers and tested further by other methods. A number of problems have been run on an Elliott 503 computer with encouraging results.

The aim is to design a network which will supply a number of points consuming power from other points which produce it, by specifying the number of standard circuits to be laid down between each pair of points, usually between 0 and 6 circuits. The points may be individual stations or they may be whole areas considered as an approximation to be single points. The network must also have a very high probability of functioning even if some circuits break or some points consume more than the expected amount. This security requirement poses a difficult problem in probability theory. I have followed a method used by the CEGB. This is to specify the number of circuits required by each subset of the points which is a net consumer of power, using a simple formula based on the estimated power requirements of the individual points in the set and including a safety margin. This gives a constraint for each consumer subset.

We are given an estimate of the capital cost per circuit between each pair of points and asked to produce a solution at minimum cost. In practice we seek only to produce solutions close to this minimum.

For computational convenience we neglect losses and let one of the points be a slack. We assign to this slack point consumption or production so that the total consumption and total production just balance. This means that half of the subsets will be net consumers of power and their complements net producers. In what follows we will by convention associate the constraint with either the consumer set or its complement, whichever does not contain the slack point.

We denote the slack point by 0 and the set of all other points by U. If S is a subset of U we write S^0 for $S \cup \{0\}$.

Let x_{ij} be the number of circuits between points i and j and c_{ij} be the cost of each circuit between i and j.

Let $g(s)$ be the number of circuits required by the set S, i.e., circuits between S and $U^0 - S$.

Then the constraints are

$$\sum_{i \in S} \sum_{j \in U' - S} x_{ij} \geqslant g(S) \quad \text{for each } S \subseteq U \tag{1}$$

$$x_{ij} \geqslant 0 \quad \text{for each } i \in U, j \in U \tag{2}$$

The total cost to be minimised is

$$Z = \sum_{i,j \in U'} c_{ij} x_{ij}$$

In some practical cases a non-linear cost function occurs. The program can treat this but we do not discuss it here.

This is an integer linear programming problem and has been tackled in this way by U. G. Knight of CEGB who first formulated it in these terms. Using an IBM 7094 and including only a sample of the constraints he has successfully tackled some problems with 9 points; on another problem of this size, however, the integer linear programming program seemed to be making no progress and was stopped after about 5 minutes. The limitations of the integer linear programming method should be apparent from the following table:

Number of Points	Number of Variables	Number of Constraints
5	10	15
10	45	511
15	105	16383
20	190	524287

(The number of constraints does not include the non-negativity constraints.)

The computer program using tree search techniques has been run successfully on programs up to 16 points in size (32767 constraints), for which it takes 2 to 3 hours on an Elliott 503 (7.5 microseconds for fixed point add and similar simple operations). On a 9-point (255 constraint) problem it obtained the same solution as integer linear programming. For the larger problems no exact standard of comparison was available but the results were quite consistent when repeated with variations in the strategy, and they agreed well with the kind of network previously arrived at by engineers using intuitive design methods.

The optimising method

The difficulty of the problem lies mainly in recognising whether a solution is feasible. In an ordinary network flow problem we only have to check that each consumer point has enough circuits into it with sufficient total flow to satisfy its demand. In our problem, however, each consumer *subset* of points has to have enough circuits. Instead of checking say $\frac{1}{2}n$ points we must check $\frac{1}{2}2^n$ subsets. A solution can look very convincing until a careful and tedious check uncovers a single deficient subset. On problems with 15 or so points even to check 100 per cent of the subsets is virtually

impossible without a computer; the engineer must use his intuition to diagnose any weak spots. He supports this by some analytical checks.

The basic method which we use is the one we have called the method of neighbouring solutions. We start with a feasible solution, either one found by hand experimentation or one formed by joining each point to the slack with a sufficient number of circuits (mostly a very expensive solution but one which can easily be shown to be feasible). We then find a succession of solutions, each one feasible and cheaper than the previous one, until further efforts yield no more improvement. Each solution is a neighbour of the previous one in the following sense: it is obtained from it by adding 0, 1 or 2 circuits and deleting some other circuits.

Even with as few as 10 points in the problem it is utterly impracticable to examine all the neighbours of a given solution. For n points there are $\frac{1}{2}n(n-1)$ pairs of points and therefore $\frac{1}{2}[\frac{1}{2}n(n-1)][\frac{1}{2}n(n-1)+1]$ ways of adding 2 circuits. Each of these could lead to several deletions, say half a dozen different combinations of deletions to be conservative. For each of these we must check 2^{n-1} subsets for feasibility. The total number of subsets to be checked is thus of the order of $6n^4 2^{n-4}$. If $n=10$ this gives nearly 4 million subsets, if $n=15$ it gives about 600 million (the 503 takes a millisecond or so to check a subset). All this is just to make one slight improvement in the solution!

Consequently we decide to look at a limited selection of the neighbours and try to simplify the checking process so as not to go through the whole business afresh for each neighbour. To do this we choose some subset of U, say F, which we will call the 'focus set', since we intend to focus attention on it for the time being. We restrict the neighbouring solutions which we consider to those which involve adding or deleting circuits within F^0, i.e., only involving points of the focus set or the slack point. We take the best of these neighbours and repeat the process considering its neighbours. When no improvements occur we choose another focus set, and so on until a sufficient number of focus sets have been tried to cover each point several times over and no further improvement seems likely. By doing this we can also simplify the checking by setting up a small number of special constraints for the set F, which, so to speak, synopsise all the constraints of the whole problem. Once set up, these enable us to make changes at will within F^0 disregarding the constraints of the whole problem and checking only these special constraints. This idea is crucial for the success of the program. We call it 'constraint mapping'. If U^0 has n points and F^0 has k we can temporarily replace the 2^{n-1} constraints associated with subsets of U by 2^{k-1} constraints associated with subsets of F, and these are all we need to worry about for additions and deletions within F^0.

We now show how these special constraints are to be set up. Suppose that x_{ij} is the current solution. We first define a quantity for each $T \subseteq U$

$$spare(T, U) = \sum_{i \in T} \sum_{j \in (U^0 - T)} x_{ij} - g(T)$$

This is the number of circuits that T has over and above its requirement. For feasibility $spare(S, U) \geqslant 0$ for all S (by equation (1) above).

We now associate with each subset $S \subseteq F$ a family of sets,

$$\mathscr{F}(S, F, U) = \{T \mid T \subseteq U \quad \text{and} \quad T \cap F = S\},$$

i.e., sets consisting of S together with some other points outside the focus set.

Notice that if we delete a circuit which joins a point in S to a point in $F^0 - S$ then for *each* $T \in \mathscr{F}(S, F, U)$, $spare(T, U)$ decreases by 1. This is the reason for defining the family $\mathscr{F}$, all sets in it are equally affected by changes within the focus set. To find how many such circuits we can safely delete we define

$$spare(S, F) = \min_{T \in \mathscr{F}(S, F, U)} spare(T, U)$$

So long as $spare(S, F) > 0$ we have not introduced an infeasibility by such deletions. Similarly adding a circuit from a point in S to a point in $F^0 - S$ gives an extra circuit to each $T \in \mathscr{F}(S, F, U)$ and hence increases $spare(S, F)$ by 1.

We see now that we can add or delete circuits within F^0 and it is sufficient to watch the effect on $spare(S, F)$ for each $S \subseteq F$.

More formally if $spare(S, F)$ is calculated from x_{ij} and we change x_{ij} to $x_{ij} + \delta x_{ij}$, keeping $\delta x_{ij} = 0$ unless $i, j \in F^0$, feasibility is maintained if and only if

$$\sum_{i \in S} \sum_{j \in F - S} \delta x_{ij} \geqslant -spare(S, F) \quad \text{for all } S \subseteq F \tag{3}$$

To maintain non-negativity of x_{ij} we also need

$$\delta x_{ij} \geqslant -x_{ij} \quad \text{for all } i, j \in pairs(F^0) \tag{4}$$

The increase in cost is

$$\delta Z = \sum_{i, j \in F^0} c_{ij} \delta x_{ij}$$

To find the best neighbour we minimise this cost, hoping for a negative value, i.e., an improvement.

These constraints again define an integer linear programming problem. In fact we restrict ourselves to changes involving not more than two additions and tackle the problem by a decision tree method. Since we are on a subproblem even a guaranteed optimal solution to the subproblem would not be sure to give an optimal solution to the whole problem, hence our preference for the less powerful but we hope faster method. Another reason is that we feel the decision tree method is more flexible in dealing with extra *ad hoc* constraints or complications in the cost function. Having got a better solution with not more than two additions we try again, and again, until the possibilities of the focus set are exhausted. We must then set up the constraints for another focus set and continue.

Using decision trees to find the best neighbour

To find the best neighbour, restricting ourselves to changes within the focus set, we use one of the algorithms described in 'Some problems about

sets' (p. 69). The first example in that section gave a method for generating a family of sets which each had a property of 'acceptability'. Now for 'sets' read 'collections' and for 'acceptability' read 'feasibility'; we can apply the algorithms to our current problem.

If $+ij$ means add a circuit between i and j, and $-ij$ means delete a circuit between i and j we can describe a combination of changes by a collection, e.g., $\{+12, +12, -13, -24, -15\}$. The solution before alteration, x_{ij}, corresponds to the empty collection.

We shall use italic bold type for symbols denoting collections to distinguish them from sets. If $\boldsymbol{C}$ is a collection of additions and deletions, we denote the collection of all additions in it by $\boldsymbol{C}^+$ and that of all deletions in it by $\boldsymbol{C}^-$. Thus $\boldsymbol{C}=\boldsymbol{C}^+ \cup \boldsymbol{C}^-$. Similarly for sets of additions or deletions. We denote the set of all additions and deletions in F^0 by V, i.e., V contains two elements for each pair of points in F^0, one with a $+$ and one with a $-$.

Now if we allow up to m additions (in practice $m=1$ or 2) we can characterise the set of possible changes by an accessible successor family with

$$\phi(R) = \text{size}\,(R^+) < m \rightarrow V^+, \{v \in V^- \mid \mathit{Feasible}(\boldsymbol{R} \cup \{v\}) \wedge \neg v \in \boldsymbol{R}^+\}$$

where *size* is a function which counts the number of elements in a collection and *feasible* is a predicate which takes the value true if its argument is a feasible collection of changes, i.e., it satisfies the constraint inequalities 3 and 4 above. If $\boldsymbol{R}$ contains less than m additions we may add any circuit, otherwise we can always delete any circuit provided that the deletion is feasible and we do not simply delete a circuit which has just been added (a waste of time).

We notice that ϕ has the property: if $P \subseteq R$ then $\phi(R) \subseteq \phi(P)$. This means that we can use the slightly simpler algorithm described in Appendix 2. Since a lot of the computation is taken up in testing for feasibility a further refinement is to keep a record of the successor elements $\phi(P)$ and pass this set on as an extra parameter of g; since we know that $\phi(R) \subseteq \phi(P)$ we can determine $\phi(R)$ more quickly by ignoring all points not in $\phi(P)$. We mention this to show how the general algorithm can be improved by adapting it to the circumstances of the particular case.

Checking only connected subsets

It still happens that each time we start optimising a new focus set we have to set up the special constraints for its subsets by examining the constraints for every subset of U. This only happens say ten or twenty times instead of many thousands in the naïve method originally discussed, but it is still tedious. It is easy to prove that if the function g which specifies the number of circuits needed has the following property we need only check those subsets which are connected (in the sense of Example 3 of 'Some problems about sets' (p. 69):

$$g(S \cup T) \leqslant g(S) + g(T)$$

By the nature of the security rules this condition does in fact hold.

The argument is that if a subset P is not connected it must be divisible into two sets S and T such that there are no circuits between S and T. The number of circuits into P is then the sum of the numbers into S and into T; but its requirement is less than or equal to the sum of the requirements of S and T if the above condition holds. Therefore to check P it is sufficient to check S and T. Repeating the argument if S or T is not connected we see in the end that it is sufficient to check only connected subsets.

Now using Example 3 of 'Some problems about sets' (p. 69) we know how to generate just the connected subsets of the symmetric graph defined by '$\gamma(i, j)$ if and only if $x_{ij} > 0$', and we do not need to examine all the disconnected subsets. On a 16-point problem this saves a factor of about 10 in the checking process. The saving increases with the problem size.

Results obtained with the program

Design engineers supplied data for several design problems, and a number of other problems have been invented using random numbers to generate problems broadly similar to those supplied.

Some experimenting was done to find the effect of different methods of choosing the sequence of focus sets. One method was to choose them at random, another was to choose intuitively a sequence which seemed to be suitable bearing in mind the following criteria:

(i) The points in a focus set should be fairly close together.
(ii) Each point should occur at least twice at different stages in the optimising process.

TABLE 1

Comparison of costs for intuitive and random choice of focus sets

Job number	210	220	230	240	250	260
No. of points	10	9	11	8	8	9
Intuitive short run	620	528	632	593	506	502
Intuitive short run	620	513	635	593	506	522
Intuitive longer run	620	513	625	593	506	509
Random short run	636	513	631	591	525	509
Random short run	632	524	657	593	506	509
Random longer run	620	513	637	581	506	509
Average intuitive	620	518	631	593	506	511
Average random	629	517	642	588	512	509

Running times varied between 3 minutes and 22 minutes.

Table 1 shows the results of trying six problems invented randomly. In these problems only one point was a producer and the others were consumers of power. Three runs were done choosing focus sets intuitively, and three

more choosing them at random. The focus sets had four or five points each. There is a slight indication that the intuitive selection was better than random on average but not enough to be of any significance.

A 9-point problem supplied by a design engineer was tried. He had run this by integer linear programming using two slightly different sets of consumption data. For the first data set, two runs were done using different sequences of intuitively chosen focus sets. The first produced a cost 0.5 per cent higher than the linear programming solution and the second obtained a solution identical with that obtained by linear programming. The times taken were 10 and 11 minutes. For the second data set only one run was done again using intuitively chosen focus sets and the results were identical with those found by linear programming. The time taken was 27 minutes. The results are shown in Fig. 2.

Fig. 3 shows the results of trying the program on a 16-point problem provided by system design engineers. Two runs were done using quite different intuitively chosen sets of focus points. Each point represents an area rather than an individual station. The results agreed to within 0.2 per cent in cost. No hand-produced solution was available for this set of data, but the solution was shown to engineers who remarked upon the similarity of the computer solution to their more detailed design studies for the same system. The times taken were 1 hour 48 minutes and 2 hours 53 minutes.

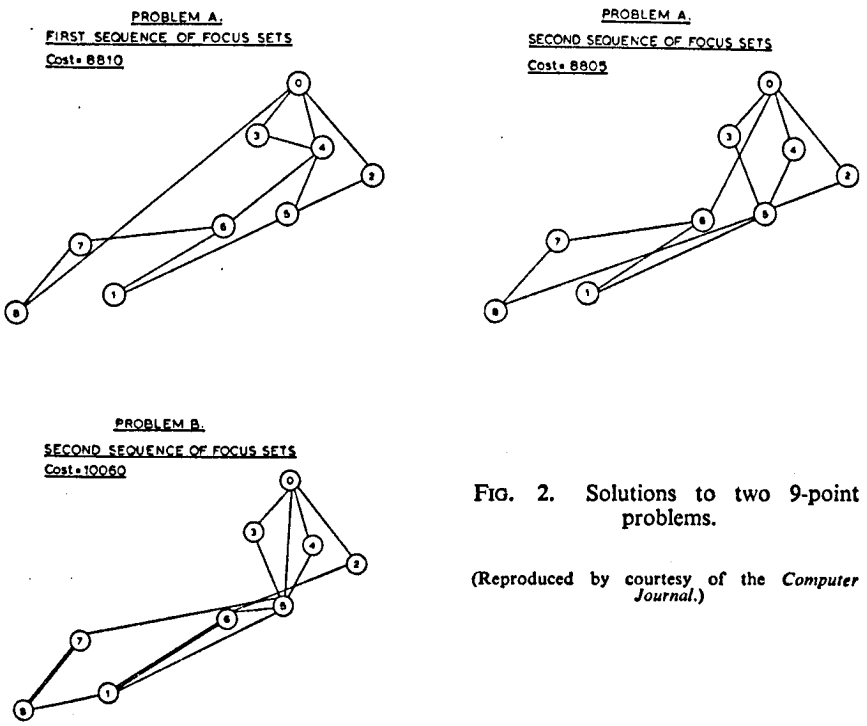


FIG. 2. Solutions to two 9-point problems.

(Reproduced by courtesy of the *Computer Journal*.)

Work is now proceeding to evaluate the performance of the program further and to improve it in various ways to enable larger problems to be tackled.

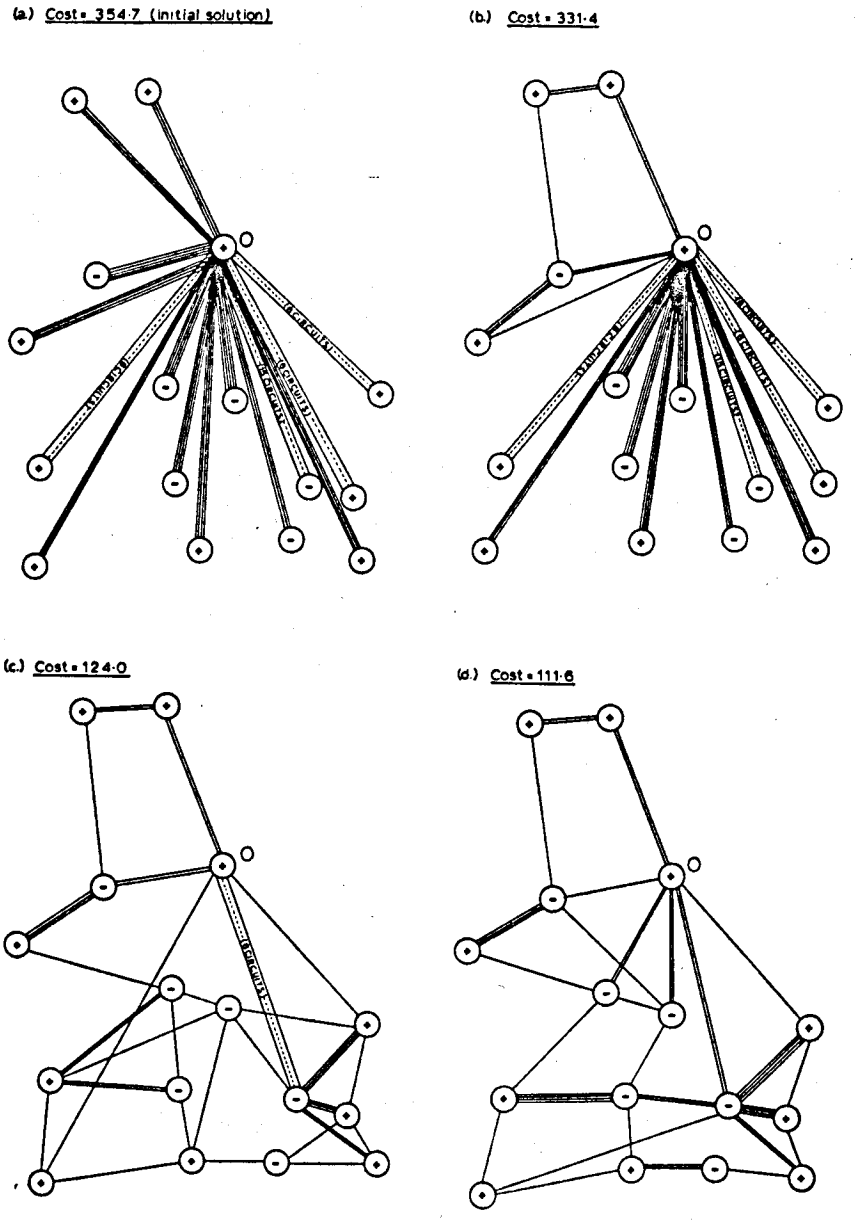


FIG. 3. Stages in the solution of a 16-point problem. − indicates a supply point, + indicates a demand point.

(Reproduced by courtesy of the *Computer Journal*.)

APPENDIX 1: NOTATION

The notation used for the recursive functions is broadly the same as in CPL (Barron, Buxton, Hartley, Nixon & Strachey 1963) or to a lesser extent that of the LISP meta-language (McCarthy 1960):

(i) Conditional expressions $P \rightarrow X,\ Y$ are equivalent to the Algol expressions **if** P **then** X **else** Y.
(ii) **let** $x=E;\ E'$ means that x has the value E in E'
let $f(y)=E;\ E'$ means that f has the value $(\lambda y.E)$ in E'.
(iii) E' **where** $x=E$ is the same as **let** $f(x)=E;\ E'$
E' **where** $f(y)=E$ is the same as **let** $f(y)=E;\ E'$.
(iv) Parentheses are used to enclose the parameters of functions and brackets for grouping terms.
(v) $\{\ \}$, $\cup$, $\cap$, $\in$, $-$, *nil* have their usual set theory meanings.
(vi) *null* (T) is true just if T is the empty set.
(vii) *any* (T) is an arbitrary member of the set T.
(viii) **rec.** in a function definition indicates recursion.

APPENDIX 2: ALGOL PROGRAMS

The following Algol program corresponds to the function f. The sets are represented by Boolean arrays. However the result is not a set of sets represented in this way, which would be wasteful of storage, but rather each set generated is presented as a Boolean array to the procedure *output* provided by the calling program to use the results. The parameter n is the dimension of U. The parameter function *phi*, is given the set R as a Boolean array and an element t; it takes the value **true** if $t \in \phi(R)$. We have taken advantage of some features of the auxiliary function g to save space by making its parameters into non-locals.

```
procedure f(U, n, phi) result: (output); value n;
  Boolean array U; integer n; Boolean procedure phi; procedure output;
  begin Boolean array R, S [1: n]; integer i;
    procedure g;
    begin integer t;
      for t:=1 step 1 until n do
        if phi (R, t) ∧ S [t] then goto Tnonnull;
      output (R); goto exit;
    Tnonnull: S[t]:=false; R[t]:=true; g;
      R[t]:=false; g; S[t]:=true;
  exit: end g;
    for i:=1 step 1 until n do begin R[i]:=false; S[i]:=U[i] end;
    g
  end f;
```

In some cases the following condition may be satisfied:

$$\text{if } P \subseteq R \text{ then } \phi(R) \subseteq \phi(P)$$

This enables us to use instead of f the function f_1 defined by:

let $f_1(U, \phi)=g(nil, U)$
where rec. $g(R, S)=null\ (S) \rightarrow \{R\},$
$[$**let** $t=any\ (S);$
$[t \in \phi\ (R) \rightarrow g(R \cup \{t\}, S-\{t\}), nil]\ \cup g(R, S-\{t\})]$

This function has the characteristic that the elements of S can be deleted one at a time in any order we choose. This means that S can be represented in Algol by

an integer s with the convention that $i \in S$ if and only if $i > s$. This in turn enables us to replace one of the recursive calls by an iteration without copying any arrays.

```
procedure f1(U, n, phi) result: (output); value n;
  Boolean array U; integer n; Boolean procedure phi; procedure output;
  begin Boolean array R[1: n]; integer i;
    procedure g(s); value s; integer s;
    begin integer t;
      for t:=s step 1 until n do
        if phi (R t) ∧ U[t] then begin R[t]:=true; g(t+1);
                                       R[t]:=false
                                end;
    output(R);
    end g;
    for i:=1 step 1 until n do R[i]:=false;
    g(1)
  end f1;
```

The procedures for dealing with collections can be coded in a very similar manner, replacing Boolean arrays by integer arrays.

REFERENCES

Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, E., & Strachey, C. (1963). The main features of CPL. *Comp. J.*, **6**, 134-143.

Burstall, R. M. (1965*a*). Computer design of electricity networks by a heuristic method. *Comp. J.* (in press).

Burstall, R. M. (1965*b*). A tree searching method for solving integer linear inequalities. *Experimental Programming Report.* No. 10. Edinburgh: Experimental Programming Unit, University of Edinburgh.

Burstall, R. M. (1966). A heuristic method for a job scheduling problem. *Opl. Res. Q.*, **17**, 291-304.

Golomb, S. W., & Baumert, L. D. (1965). Backtrack programming. *J. Ass. comp. Mach.*, **12**, 516-524.

Knight, U. G. W. (1960*a*). Logical design of electrical networks. *J. Instn elect. Engrs*, **6**, 228-230.

Knight, U. G. W. (1960*b*). The logical design of electrical networks using linear programming methods. *Proc. Instn elect. Engrs*, **107A**, 306-319.

Knight, U. G. W. (1963). Economic design and operation of power systems. *Elect. Rev., Lond.*, **92**, 124-130; **92**, 163-166.

Little, D. D. C., Murty, K. G., Sweeney, D. W., & Karel, C. (1963). An algorithm for the travelling salesman problem. *Ops. Res.*, **116**, 972-989.

Lomnicki, Z. A. (1965). A 'branch-and-bound' algorithm for the exact solution of the three-machine scheduling problem. *Opl. Res. Q.*, **16**, 89-100.

McCarthy, J. (1960). Recursive functions of symbolic expressions. *Communs Ass. comput. Mach.*, **3**, 184-195.

Newell, A., Shaw, J. C., & Simon, H. A. (1959). Report on a general problem-solving program. *Proc. of the International Conference on Information Processing.* pp. 256-264. Paris: UNESCO House.

Newell, A., & Simon, H. A. (1961). GPS, a program that simulates human thought, in *Lernende Automaten*. Munich: R. Oldenbourg KG.

Reiter, S., & Sherman, G. R. (1963). Discrete optimising. *Institute for Quantitative Research in Economics and Management, Purdue University, Institute Paper* No. 37.

Tonge, F. M. (1960). Summary of a heuristic line balancing procedure. *Management Science*, 7, 21-42.