A tree search approach to the M-partition and Knapsack problems*

A. Albano and R. Orsini

Istituto di Scienze dell'Informazione, Corso Italia 40, 56100 Pisa, Italy

Given r positive integers $s_1, s_2, \ldots, s_r$ with an associated profit p_i two problems are at the root of several interesting applications. The 'M-partition problem', that is determining all possible combinations of these numbers which sum to M, and the 'Knapsack problem', that is determining a combination of these numbers maximising the p_i sum subject to the condition that the s_i sum is not greater than M.

A solution is proposed for both the problems based on a tree search approach which leads to algorithms with simple structure, linear storage requirements and an average computation time lower than other proposed algorithms.

(Received July 1978; revised February 1979)

1. Introduction

Given r positive integers $s_1, s_2, \ldots, s_r$ we wish to find all possible combinations of these numbers which sum to M (M-partitions). This problem appears in several interesting applications. For example in number theory the problem 'given N find all the sets of integers S_K such that $\sum_{s_i \in S_K} s_i = N$' can be reduced to the previous one assuming $s_i = i$ and $s_r = N = M$. Another example is shown by Musser (1971), where the partition problem appears in the polynomial factorisation where M is the degree of the given polynomial and the s_i are the suspected degrees of its irreducible factors. Finally, an interesting relationship can be found with operations research if we associate to each number s_i a 'measure of desiderability' or 'profit' p_i and we wish to find a combination of them maximising the p_i sum subject to the condition that the s_i sum is not greater than M. This is a well known integer programming problem called the Knapsack problem.

Both the knapsack and partitions problems belong to the class of nondeterministic polynomial-time complete problems which includes, among others, the travelling salesman problem, the Hamilton circuit problem and the satisfiability of Boolean expressions. All the problems in this class can be solved with a nondeterministic algorithm in polynomial time and if a polynomial deterministic algorithm is found for one of them, then this will be possible for all the problems in the class.

At the moment it is not known if a polynomial deterministic algorithm can be found for both the partitions and knapsack problems. It is however interesting to develop solutions with a good average behaviour because of their wide use in the applications. Horowitz and Sahni (1974) have summarised the performances of the algorithms that have been proposed and they present new solutions leading to important improvements.

In the next section we will formulate the M-partition problem using the concept of multiset and we will present a tree search approach to the solution. In Section 3 we will develop a recursive algorithm which, as will be shown in Section 4, has better performances than the others previously known. In Section 5 we will use the same approach to solve the knapsack problems and in Section 6 we will develop a recursive algorithm. The iterative version of the program has the same structure as the one proposed by Horowitz and Sahni but the empirical studies reported in Section 7 indicate that the heuristics used give significant improvements.

Our original motivation for this topic arose from an investigation of the allocation problems with one, two and three dimensional objects and resource (Adamowicz and Albano, 1976; Orsini, 1976).

2. The M-partitions problem

In order to give the mathematical formulation of the problem, let us present some useful definitions.

Definition 1

A *multiset* S is a collection of positive integer numbers s_i denoted by $S = \{s_i\}$.

Definition 2

A set S is a multiset whose elements satisfy $s_i \neq s_j$ if $i \neq j$.

Definition 3

The cardinality of a multiset S, denoted by $|S|$, is the number of elements in S. If $|S| = r$ then S will be written as S_r.

Definition 4

An M-partition of a multiset $S_r = \{s_1, \ldots, s_r\}$ is an r-tuple $X = (x_1, \ldots, x_r)$ where

$$x_i \in \{0, 1\}\,, \quad 1 \leqslant i \leqslant r \text{ and } \sum_{i=1}^{r} x_i s_i = M \tag{1}$$

Definition 5

An algorithm computes the M-partitions of S_r, if it generates all r-tuples X satisfying (1) and no other X's.

The problem of computing the M-partitions can be solved either with iterative or tree search techniques. In what follows we will formulate the problem as a search in a binary tree while for a discussion on the iterative approach see Horowitz and Sahni (1974).

Let B be a binary tree with $|S| + 1$ levels. Each node at level i, except the leaves, has two sons corresponding to the i-th element inclusion or exclusion into the partition. With every node of the tree is associated:

a) the path from the root up to it, represented by a binary i-tuple $(x_1, \ldots, x_i)$, i indicating the length of the path, and

b) a partial sum defined by $\sum_{j=1}^{i} x_j s_j$.

In Fig. 1 an example is shown for $S_4 = \{6, 5, 3, 1\}$ and $M = 9$. The i-th element inclusion and exclusion is shown by associating with the arcs 1 or 0 respectively. A path from the root to a leaf represents a binary r-tuple corresponding to a partition. With this representation in mind the problem of generating all the M-partitions is seen as a search process for finding all the paths labelled with M (M solution path). In the example there are two 9 partitions, that is (1010) and (0111), and the solution paths are given by the dark arcs.

*This work has been partially sponsored by the Consiglio Nazionale delle Ricerche.

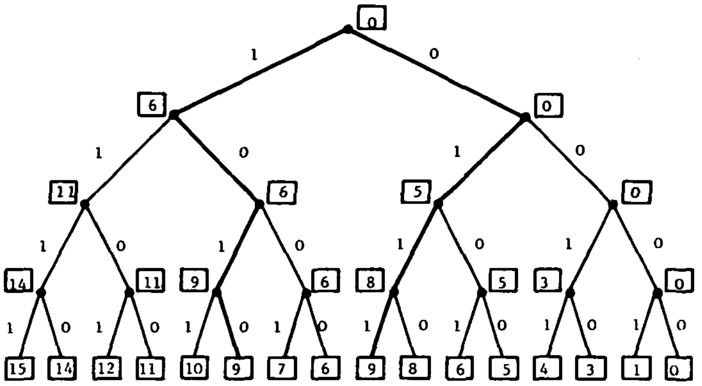

Fig. 1

To solve the problem efficiently an algorithm is needed which finds the solution paths visiting a subset of the nodes making use of heuristic tests for avoiding exploring fruitless paths deeply. In this paper by heuristic test we mean a test to stop a path visit without loosing the algorithm admissibility, that is it will always terminate with the set of solution paths whenever a solution exists.

Such an algorithm can be organised in the following steps.

1. Put the root on a list called GENERATED.
2. If GENERATED is empty, exit.
3. Select a node from GENERATED according to some rule R and put it on a list called EXPANDED. Call it n.
4. If a solution has been found, print it.
5. If from n a solution cannot be obtained go to Step 2.
6. Put the two successors of n on GENERATED and go to Step 2.

The selection rule R used in step 3 distinguishes two classes of algorithms:

(*a*) by choosing the first node on the GENERATED list and by putting the successors at the *end* of it. This means that the nodes are expanded in the order in which they are generated and therefore the tree is visited level by level (breadth-first methods).

(*b*) by choosing the first node on the GENERATED list and by putting the successors at the *beginning* of it. Now the most recently generated node is expanded first and therefore the tree is visited path by path (depth-first methods).

The successors generation at Step 6 consists in the inclusion or exclusion of the n-th element in the partial sum associated with the n-th node. Using a depth-first methods the storage requirements are linear with r because during the search process only the nodes on a path, which are at most r, need to be stored.

With the other method the storage requirements grow exponentially with $|S|$ because expanding the tree by levels all the nodes generated need to be stored. In the next section we will present a depth-first algorithm which using heuristics will be faster than the ones of the dynamic programming type (Horowitz and Sahni, 1974).

3. The M-partitions algorithm

A recursive formulation for the algorithm can be derived as follows. Let us use the notation $P(s_i, s_{i+1}, \ldots, s_r; m)$ to denote the set of m-solution paths by using numbers from $(s_i, \ldots, s_r)$. We can split the set P in two classes according as the number s_i is used or not. If it is not then we reduce the original problem to that of finding the set of m-solution paths with numbers from $(s_{i+1}, \ldots, s_r)$. If s_i is used, then the problem is reduced to that of finding the set of $(m - s_i)$-solution paths with numbers from $(s_{i+1}, \ldots, s_r)$. We thus have the relation:

$$P(s_i, \ldots, s_r; m) = P(s_{i+1}, \ldots, s_r; m) \cup P(s_{i+1}, \ldots, s_r; (m - s_i))$$

The heuristics used to speed up the search are:

1. The elements $(s_i, \ldots, s_r)$ are ordered $(s_j \geqslant s_{j+1})$, $i \leqslant j < r$
2. If $\sum_{j=1}^{r} s_j < m$ $\quad P(s_i, \ldots, s_r; m) = \emptyset$
3. If $\sum_{j=1}^{r} s_j = m$ $\quad P(s_i, \ldots, s_r; m) =$ {the path from the current node to the leaf including all the elements}
4. If $s_i > m$ $\quad P(s_i, \ldots, s_r; m) = P(s_{i+1}, \ldots, s_r; m)$
5. If $s_i < m$ $\quad P(s_i, \ldots, s_r; m) = P(s_{i+1}, \ldots, s_r; m - s_i) \cup P(s_{i+1}, \ldots, s_r; m)$
6. If $s_i = m$ $\quad P(s_i, \ldots, s_r; m) = P(s_{i+1}, \ldots, s_r; m) \cup$ {the path from the current node to the leaf including the first element only}

This approach to the M-partition problem is reported also in Horowitz and Sahni (1974). But the algorithm based on the above ordering of the elements leads to more powerful heuristics. Rubin (1976) came to the same conclusion independently.

Algorithm PRIC (i, m, X)

Let:

i = index of the next s_i to be examined;

m = the remaining sum: M-(sum of all the elements used up to this point);

X = $\{j | s_j$ is included in the path from the root to the current node$\}$; and

$$\mathrm{SUM}(i) = \sum_{j=1}^{r} s_j.$$

The algorithm is initially invoked as PRIC(1, M, NULL). For the example in Fig. 1 the algorithm will traverse the subtree shown in **Fig. 2.** The nodes are labelled with the generation order number.

4. Empirical results

The algorithm has been programmed and tested extensively to determine its average performance. Both the recursive (PRIC) and the iterative program (PITER) have been written in ALGOL W for the VM/370 running on a IBM 370/168 and are reported in Albano and Orsini (1977). The results have been compared with the ones obtained by the dynamic programming type algorithm proposed by Horowitz and Sahni (1974)(2(b) in Table I), which proved to give the best results at that time.

The tests were performed using several data sets. In **Table 1** the computing time for the following case is reported:

$$s_i = i\,, \quad 1 \leqslant i \leqslant r\,, \quad M = r, 2r, 3r, r(r+1)/4\,.$$

The other results present a similar pattern and are described in Albano and Orsini (1977). As Table 1 shows, the recursive

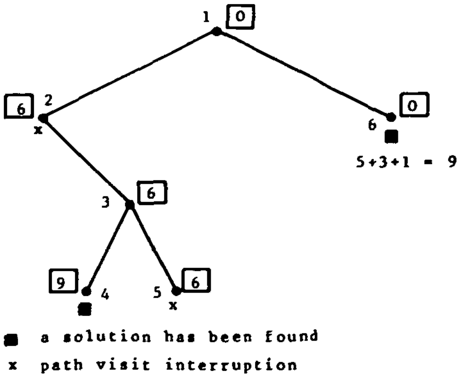

Fig. 2

Table 1 **Times in milliseconds**

M	r	$2(b)$	PITER	PRIC
max	15	7	3	6
	20	20	6	13
	25	54	10	26
	30	151	19	50
	35	351	35	96
	40	862	62	174
2max	10	5	2	6
	12	9	5	11
	14	18	7	19
	16	36	12	35
	18	69	23	65
	20	127	40	113
3max	10	6	3	7
	12	16	7	17
	14	37	13	45
	16	91	36	110
	18	212	83	251
	20	471	187	538
$\Sigma s_i/2$	10	6	3	6
	12	16	7	18
	14	44	20	56
	16	140	59	182
	18	478	203	613
	20	1679	727	2102

version of the algorithm already presents an interesting behaviour. Of course the iterative one reduces the computation time significantly improving in almost all circumstances the algorithm performances proposed by Horowitz and Sahni. Moreover our algorithm has a linear storage requirement while the dynamic programming type algorithm has an exponential storage requirement which limits the problem size that it can handle.

5. The knapsack problem

Let us consider a classical integer programming problem. Let S be a set of r positive integers s_i with an associated profit p_i. We wish to solve:

$$\max \sum_{i=1}^{r} p_i x_i \text{ subject to:}$$

$$\sum_{i=1}^{r} s_i x_i \leqslant M,\ x_i = 0{,}1\ (i = 1, \ldots, r)$$

The knapsack problem has a strong analogy with the M-partition problem: we must select among the possible combinations of S elements the ones which are an M-partition in one case or which maximise the profit in the other. Moreover both problems can be solved either with iterative or tree search techniques. In the sequel we shall be concerned with the last approach and we will present a recursive algorithm which is a natural extension of the one proposed for the partitions. For the iterative approach see Gilmore and Gomory (1966), Horowitz and Sahni (1974) and Orsini (1976).

Let us consider the following tree search process:

1. Put the root on a list called GENERATED.
2. If GENERATED is empty, exit.
3. Select a node from GENERATED according to some rule R and put it on a list called EXPANDED. Call it n.
4. If the optimal solution has been found, exit. Otherwise continue.

4'.If the present solution is better, update the current solution.

5. If from n the current solution cannot be improved, go to step 2.
6. Generate the n-successors and add them to GENERATE. Go to step 2.

In this process there are two choices to be made which lead to different kinds of algorithms:

(*a*) Step 3. If we select the last generated node we proceed depth-first on the tree with a branch-and-search or backtracking algorithm, while if we select the node on the basis of an 'evaluation function' we have a branch-and-bound algorithm. The evaluation function is an estimate that a node can be on the optimal solution path.

(*b*) Step 6. Once a tree traversing method has been chosen the process efficiency depends on the strategy used to generate successors, ie the strategy used to select the variable to be assigned the value 0 or 1.

Let us show how the most common solutions proposed in the literature can be described on the basis of the outlined general tree search process.

Branch-and-bound algorithms

The next node to be expanded is chosen using an evaluation function. A possible one can be defined as follows.

Let us assume the elements of S are in decreasing order of profit densities ($p_i/s_i \geqslant p_{i+1}/s_{i+1}$). Let t the least integer ($0 < t \leqslant r$) such that $\sum_{i \leqslant t} s_i \geqslant M$.

Dantzig (1957) has shown that the fractional solution of the 0-1 Knapsack problem given by:

$$\begin{cases} x_i = 1 & \text{if } i < t \\ x_i = 0 & \text{if } i > t \\ x_t = (M - \sum_{i<t} s_i)/s_t \end{cases} \qquad (2)$$

has the following properties:

(*a*) If $x_t = 0$, then it is the optimal solution.

(*b*) If x_t is fractional the value $Z(1) = \sum_{i<t} p_i + p_t x_t$ is an upper bound to the optimal solution value.

(*c*) If no t exists we have all $x_i = 1$.

(*d*) At node n of the tree, $Z(n)$ represents the optimal fractional solution with assigned variables added as constraints.

If we assume Z as an evaluation function, the next node to be expanded will be the one with the largest value of Z.

Let us illustrate the algorithm with the following example:

$$\max(63x_1 + 10x_2 + 15x_3 + 25x_4)$$

$$\text{subject } 6x_1 + 1x_2 + 5x_3 + 3x_4 \leqslant 9$$

$$x_i \in \{0, 1\}, \quad 1 \leqslant i \leqslant 4.$$

Solving (2), we obtain $Z(1) = 91$ with $x_1 = 1, x_2 = 1, x_3 = 2/5$ and $x_4 = 0$. We have only one node in the list GENERATED and to proceed we must decide how to generate successors (see

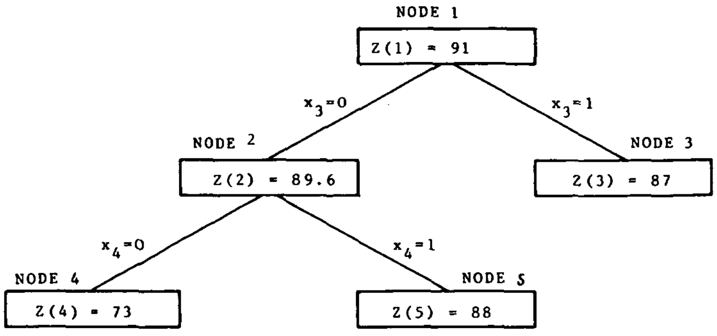

Fig. 3

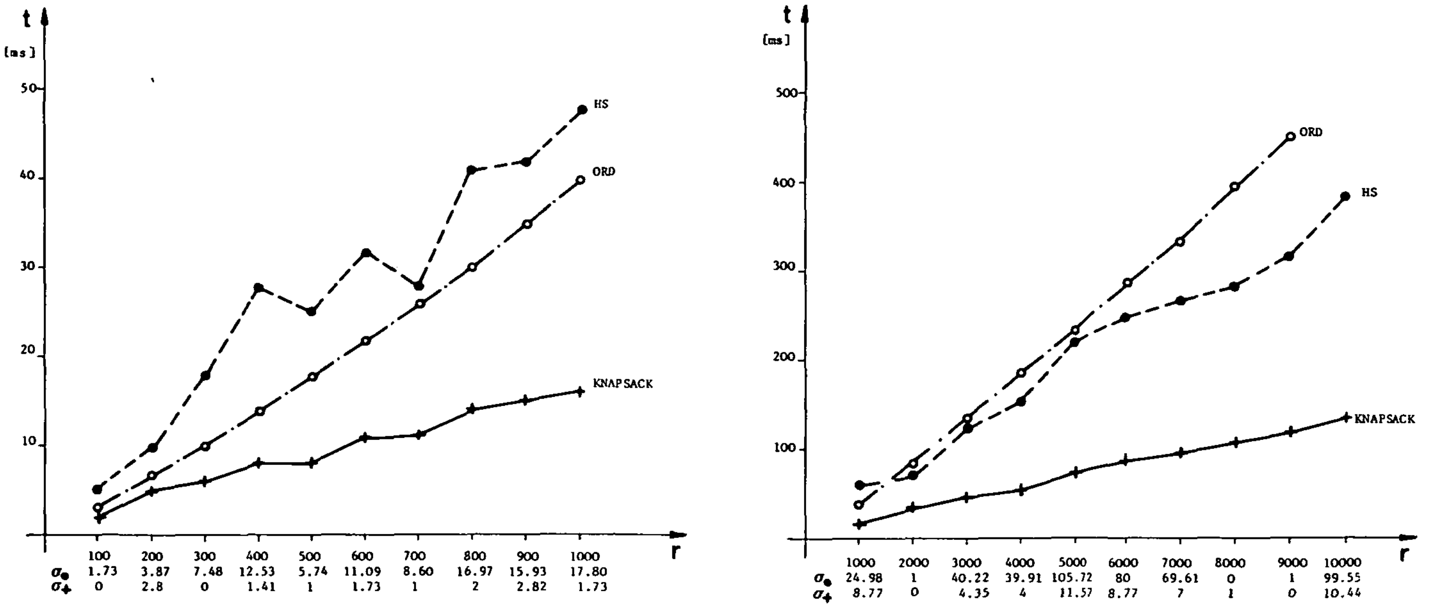

Fig. 4 Mean computing times and standard deviation with M = 2 . (max s_i)

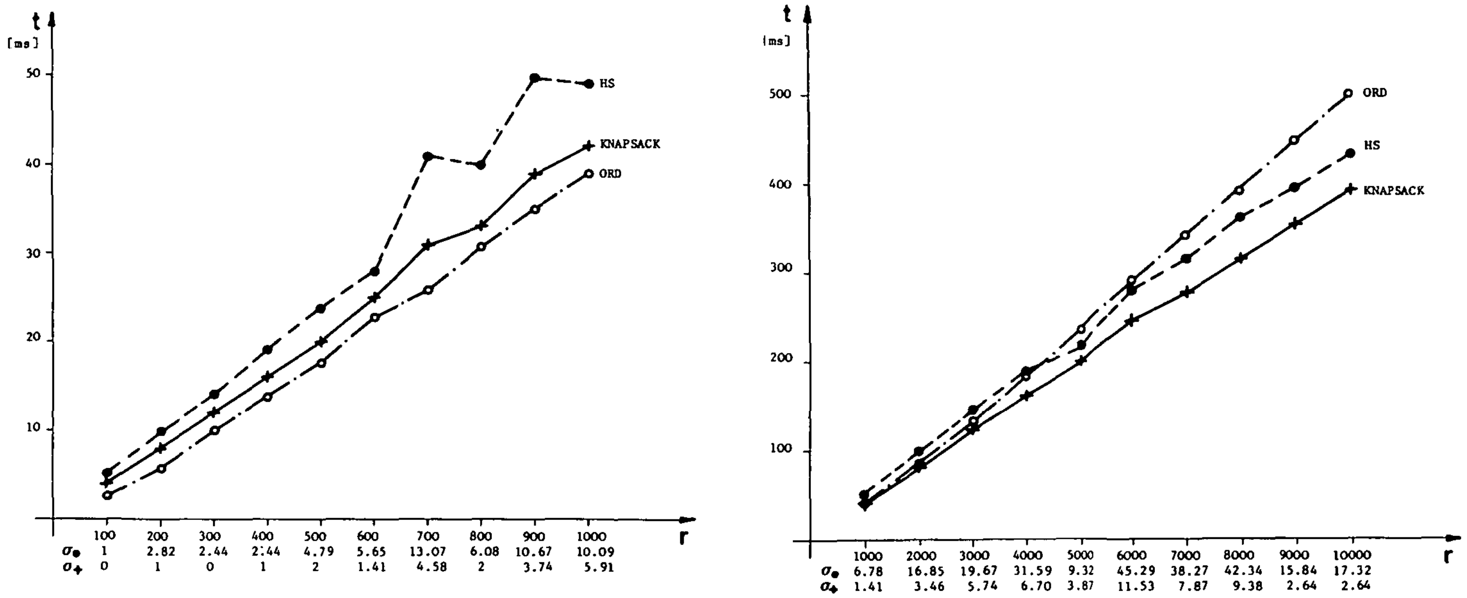

Fig. 5 Mean computing times and standard deviation with M = $\Sigma s_i/2$

Fig. 3). Kolesar's algorithm (1967) considers the unassigned variable with smallest index. In this way we have a tree similar to the one for the M-partition: the i-th level corresponds to the i-th variable assignment. Greenberg and Hegerich (1970) proposed another strategy: the next variable is the fractional one at the selected node.

Let us continue the example according to the last strategy which has been shown more effective (Greenberg and Hegerich, 1970). We select x_3 and generate nodes 2 and 3 (see Fig. 3). We compute $Z(2)$ with $x_3 = 0$ and we get $Z(2) = 89.6$. Similarly for node 3 we get $Z(3) = 87$. Now we select among the terminal nodes the node 2. The fractional solution is $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ and $x_4 = 2/3$. We select the variable x_4 and generate the successor nodes 4 and 5 labelled with $Z(4) = 73$ and $Z(5) = 88$. The next node to expand is 5 and because the solution is $x_1 = 1$, $x_2 = 0$, $x_3 = 0$, $x_4 = 1$ with no fractional variables the algorithm terminates. In fact when we select a node from the list GENERATED and the associated Z value has been obtained with no fractional variable then the optimal solution has been found.

With this approach the tree expansion is partly by depth-first and partly by breadth-first. This is a drawback because the storage requirements are exponential with the number of variables.

Branch-and-search algorithms

In this class of algorithms the tree search is developed depth-first. The algorithm first finds a simple solution which is a lower bound to the optimal one. Then a path of the tree is explored as far as it is possible to improve the current solution. Then it backtracks and develops a new one. The algorithm stops when all new paths are excluded and the solution is the current lower bound. Therefore Step 4 in the outlined general tree search process is not used.

The advantage of this kind of algorithm is that the storage requirements are linear with the tree depth, that is the number of variables. In order to specify completely an algorithm in this class we must decide both which variable to select and the value to assign to it.

Greenberg and Hegerich (1970) suggest selecting the fractional variable and excluding it first from the solution, while Horowitz and Sahni (1974) decided to select the next variable in order of decreasing profit densities and to include it first in the solution. Both assumed the solution $x_i = 0, 1 \leqslant i \leqslant r$ as the initial one.

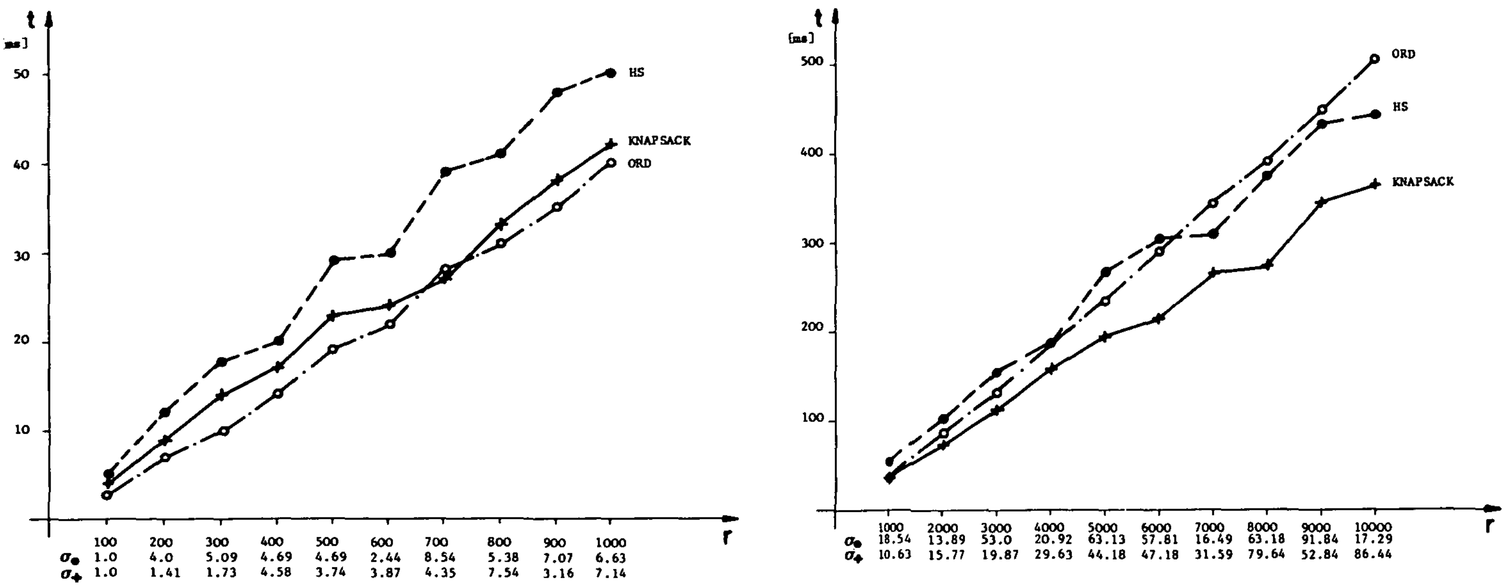

Fig. 6 Mean computing times and standard deviation with $M = \lfloor \alpha \Sigma s_i \rfloor$

6. The knapsack algorithm

The efficiency of a branch-and-search algorithm depends upon the possibility of directing as soon as possible the search on the tree portion where the optimal solution path is located because, in general, the selective power of the heuristic tests will improve. We propose a branch-and-search algorithm based on the following assumption.

(*a*) the initial solution is the 'greedy solution' defined as follows:

Profit; $= 0$; G-solution := NULL;

(*b*) In order to avoid exploring fruitless paths deeply the heuristic tests make use both of the evaluation function Z and of the value of M still to be used;

(*c*) The variables are selected in order of decreasing profit densities;

(*d*) A selected variable will be included first in the solution.

A recursive formulation of the algorithm is the following:
Let us assume that:

$s_i \leqslant M, p_i, s_i > 0$ and $\Sigma s_i > M$:
i = index of next s_i to be processed;
m = M-(sum of all the elements used up to this point);
p = profit associated with the current sum;
x = set of j that yield this profit;
P = maximum profit obtained (current lower bound solution which at beginning is the greedy solution);
X = solution associated with P;
Z = the Dantzig function.
MINS$(1:r)$ = vector such that MINS$(j) = \min s_i, j \leqslant i \leqslant r$.

The algorithm is initially involved as KNAP(1,M,0,NULL)

7. Empirical results

Both the iterative version of the KNAPSACK algorithm and the one proposed by Horowitz and Sahni (1974), have been programmed in ALGOL W for the VM/370 running on a IBM 370/168.

For a number of variables $r = 100, 200, \ldots, 1000, 2000, 3000, \ldots, 10000$ three cases have been reported (**Figs. 4, 5** and **6**). In the first two cases, the s_i and p_i have been assumed independent pseudo-random numbers in [1, 100] with M equal to 2·(max s_i) and $\Sigma s_i/2$ respectively. In the third case, the s_i ($i \leqslant r/2$) are independent pseudo-random numbers in [1, 100] and in [800, 900] for the $s_i(i > r/2)$, M is equal to $\lfloor \alpha \Sigma s_i \rfloor$ and α being a pseudo-random number between 0 and 1.

The figures given, for each value of r, are the computing times in milliseconds, as the mean of times obtained for 10 tests, and the standard deviation. The inputs are assumed ordered according to decreasing profit densities and the time required for the initial ordering is reported separately. The computation times for both the algorithms include all the required initialisations with the exception of the ordering.

Although the iterative version of our algorithm has a similar structure to the one proposed by Horowitz and Sahni, the initial solution and the heuristic test to interrupt the search produce, in all the cases, a reduction in computing time as a function of the size of the problem. Moreover, the computation time, with r in the range of values considered, presents a behaviour remarkably similar to the one of the initial ordering. From the figures it is also clear that the improvements are particularly significant for $M = 2$. (max s_i) and they reduce in the other cases when the problem becomes more difficult to solve because of the increased number of feasible solutions to consider. However, from the value of the standard deviation, it also becomes evident that our algorithm is in general less sensitive to the data and it is therefore more stable.

Finally we should like to mention that in a recent paper Laurier (1978) has presented an improved method which preprocesses the input of a Knapsack problem in order to reduce the number of the variables before applying the actual algorithm. However, at least for a number of variables between 30 and 5000, as reported in his paper, the times required in the reduction stage are already significantly higher than the ones required by our algorithm to solve the original problem completely (see **Table 2**).

8. Conclusion

Two algorithms have been proposed to solve the M-partition and the Knapsack problem based on tree search techniques. This approach has been a natural way both to formulate efficient algorithms with simple structure and to present a general schema to compare other solutions. The algorithms have been tested thoroughly and their performances indicate that they compare favourably with respect to the others designed to solve these problems and reported in the wide survey on the subject presented by Horowitz and Sahni (1974).

Table 2

Laurier reduction method			*Knapsack and ordering*
n	*nrv*	*time*	*time*
100	9	50	7
200	9	50	14
300	12	250	23
500	12	320	38
1000	4	600	79
2000	6	1280	168
5000	0	2540	446

times in milliseconds
nrv: average number of residual variables after the reduction
p_i, s_i pseudo-random number in [1, 100]
$M = \lfloor \alpha \Sigma^n_{i=1} s_i \rfloor$, α being drawn at random between 0 and 1.

```
0. [All done]            if $i > r$ then return.
1. [Test heuristics]     if SUM($i$) $< m$ then return.
                         if SUM($i$) $= m$ then output
                         $X \cup \{i, i + 1, \dots, r\}$ return
2. [Try to include $s_i$]
                         if $s_i < m$ then PRIC($i$ + 1,
                                 $m - s_i$, $X \cup \{i\}$).
                                 else if $s_i = m$ then
                                 output $X \cup \{i\}$.
3. [exclude $s_i$]       PRIC($i$ + 1, $m$, $X$).
4. [return]              return.
```

```
  for $i = 1$ until $r$
   do if $s_i \leqslant M$ then begin
                         $M = M - s_i$;
                         Profit = Profit + $p_i$;
                         $G$-solution = $G$-solution $\cup \{i\}$;
                         end
```

```
Procedure KNAP($i$, $m$, $p$, $x$);

K1. [test on the path length]
      if $i = r$ then (if $s_i \leqslant m$ then ($p = p + p_i$;
                   $x = x \cup \{r\}$);
                   if $p > P$ then ($P = p$; $X = x$);
                   return)
K2. [Try to include $s_i$]
      if $s_i \leqslant m$ then KNAP($i$ + 1, $m - s_i$, $p + p_i$, $x \cup \{i\}$).
K3. [path visit interruption]
      if $Z(i + 1, m) + p \leqslant P$ or
          $m \leqslant$ MINS($i$ + 1) then return.
K4. [Try to exclude $s_i$]
      KNAP($i$ + 1, $m$, $p$, $x$)
K5. [return] return
```

References

Adamowicz, M. and Albano, A. (1976). A Solution of the Rectangular Cutting Stock Problem, *IEEE Trans. Syst., Man., Cybern*, SMC-6, pp. 302-310.

Albano, A. and Orsini, R. (1977). Metodi di ricerca su alberi per la soluzione dei problemi delle *M*-partizioni e dello zaino, Nota Scientifica S-77-17, Istituto di Scienze dell'Informazione, Università di Pisa, Maggio 1977.

Aho, A., Hopcroft, J. and Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, Mass.

Dantzig, G. B. (1957). Discrete Variable Extremum Problems, *Operations Research*, Vol. 5, pp. 266-277.

Gilmore, P. C. and Gomory, R. E. (1966). The Theory and Computation of Knapsack Functions, *Operations Research*, Vol. 14, pp. 1054-1074.

Greenberg, H. and Hegerich, R. L. (1970). A Branch Search Algorithm for the Knapsack Problem, *Management Science*, Vol. 16, pp. 327-332.

Horowitz, E. and Sahni, S. (1974). Computing Partitions with Applications to the Knapsack Problem, *JACM*, Vol. 21, pp. 277-279.

Kolesar, P. J. (1967). A Branch and Bound Algorithm for the Knapsack Problem, *Management Science*, Vol. 13, pp. 723-735.

Laurier, M. (1978). An Algorithm for the 0/1 Knapsack problem, *Mathematical Programming*, Vol. 14, pp. 1-10.

Musser, D. R. (1971). Algorithms for polynomial Factorization, Ph.D. Th., Techn. Rept 134, Computer Sciences Dept. of Wisconsin, Sept. 1971.

Orsini, R. (1976). Problemi di allocazione ad una, due e tre dimensioni, Tesi di Laurea, Istituto di Scienze dell'Informazione, Università di Pisa, 1976.

Rubin, F. (1976). Partition of Integers, *ACM TOMS*, Vol. 2, pp. 364-374.

Book review

Computer Security, by Peter Hamilton, 1979; 116 pages. (*Van Nostrand Reinhold*, £7·45)

On the day before this book reached me I heard that a passing acquaintance had been dismissed by the bank for which he worked. This person had been there for less than a year since leaving school, and had left amid rumours that dealings with the bank's computer had resulted in the misplacing of some tens of thousands of pounds. I harboured dark thoughts that this book might give me some hints as to how I could get in on these apparently widespread activities. In this way, I might be able to keep my wife in the manner to which she is accustomed, regardless of Professor Clegg's conclusion on the comparability of my remuneration.

In fact, the book is not about how fraud is achieved with the aid of a computer. It is a book written by a security expert about the security of computers. It contains some solid commonsense remarks about the means by which computer hardware and software can be made more secure than they usually are. While it is nice to know that an expert can set down commonsense, it seems to come rather expensive at nearly £8 for about a hundred pages of it. Additionally, the author's style is rather cluttered and difficult to read so that on occasion he has difficulty in expressing his ideas with precision.

The author tells us more than once that the computer is 'intellectually a moron, and morally permissive'. Perhaps, in the not too distant future, when the techniques of Artificial Intelligence enable us to give a computer a personality of its own, our computers will not only report syntax errors, but will also refuse to run programs because they are immoral.

G. Marshall (London)