MANAGEMENT SCIENCE
Vol. 17, No. 5, January, 1971
Printed in U.S.A.

THE LOADING PROBLEM*

SAMUEL EILON AND NICOS CHRISTOFIDES

Imperial College of Science and Technology, London

The loading problem is defined as the allocation of given items with known magnitude to boxes with constrained capacity, so as to minimize the number of boxes required. Two methods of solution are considered: The first is by a zero-one programming model, for which the solution procedure is described; the second is by a heuristic algorithm. Fifty problems were solved by the two methods and in all but two the second method yielded the optimal solution with significantly less computing time than that needed by the first method.

Introduction

A particular type of loading problem, which is discussed in this paper, is akin to the knapsack problem and may be formulated as follows: Allocate n objects or items of given magnitude Q_i $(i = 1, 2, \cdots, n)$ to boxes, each box having a capacity C, in such a way that the capacity constraints are not violated and the number of boxes required is a minimum. An assumption is made that the capacity requirements are additive, so that the capacity needed to accommodate two items of magnitude Q_1 and Q_2 is $Q_1 + Q_2$, as is the case when Q_i and C are measured in units of weight, length, money or liquid volume.

This is only one of several types of loading problems. More generally, if an array of items of magnitude Q_i is specified, where the value of item i is given by u_i $(i = 1, 2, \cdots, n)$, and if an array of N boxes of capacity C_j and their corresponding values v_j $(j = 1, 2, \cdots, N)$ are given, then two situations can arise (denoted as S_1 and S_2):

S_1 : $\sum_j C_j \geqq \sum_i Q_i$ and all the items can be accommodated in the array of boxes

S_2 : either $\sum_j C_j \geqq \sum_i Q_i$ but not all items can be accommodated or $\sum_j C_j < \sum_i Q_i$.

Several objectives can be stated for the allocation of items to boxes, as suggested below (the objectives are denoted as O_1, O_2 and O_3):

O_1 : Objective based on boxes

(a) Minimize the number (or value) of boxes required

(b) Minimize the unused space (or its value) of the partially used boxes

O_2 : Objective based on items

Minimize the number (or value) of items not accommodated in the boxes

O_3 : Combined objective

Minimize the combined (weighted) value of boxes chosen and value of items not accommodated.

These situations and criteria define the following problem matrix:

	O_1	O_2	O_3
S_1	1	—	—
S_2	2	3	4

Problem 1 is a more general formulation of the problem described at the beginning of this paper. In situation S_1, where all the array of boxes has adequate capacity to accommodate all the given items, objectives O_2 and O_3 are irrelevant. Problem 2 is an ex-

* Received March 1969; revised December 1969.

tension of problem 1, except that $O_1(b)$ becomes the relevant objective. Problem 3 is the classical multidimensional knapsack problem, except that there are several boxes rather than a single one. This problem can be solved by methods somewhat similar to those suggested by Gilmore and Gomory [4], [5] for cutting stock problems. The methods of Gilmore and Gomory, which are based on dynamic programming, can also be employed to solve problem 3 in two (or more) physical dimensions where the two dimensions are interrelated such as in the problem of packing several small rectangles into a larger rectangle. It should be pointed out that the two physical dimensions problem does not, strictly speaking, belong to this matrix of problems, because the additivity conditions for Q_i mentioned earlier do not apply, for example in the problem of cutting rectangles. Problem 4 involves an objective O_3 which is based on a combination of objectives O_1 and O_2.

This paper is concerned with a particular case of problem 1, where all the C_j have the same value, say, C. Two methods of solution are examined and their efficiency is compared: the first is by zero-one programming and the second by using a heuristic algorithm.

The methods are also tested against a problem which is more general than problem 1, in that each item is characterized not by a single magnitude Q_i, but by several magnitudes $Q_{i1}, Q_{i2}, \cdots$, each representing a particular attribute, for example weight, liquid volume, money value, etc. Similarly, the boxes have corresponding capacities C_1, C_2, etc.

The model that this paper is concerned with has applications in solving problems of loading vehicles (or other containers) with consignments that cannot be split, or in solving some apportioning problems where large units need to be subdivided into several smaller units of given or constrained magnitude.

In Appendix I it is shown that the zero-one programming method can also solve the general problem 1 with boxes of differing capacities. In fact, this programming approach can be used to solve all the problems listed in the problem matrix, although this is not discussed further here.

Method I: The Zero-One Programming Method

Let x_{ij} denote the allocation matrix of items to boxes, so that $x_{ij} = 1$ if item i is allocated to box j ($j = 1, 2, \cdots, N$), otherwise $x_{ij} = 0$. The total capacity taken up by all the items in box j must not exceed the capacity of the box, and this constraint is expressed for each box by

$$\sum_{i=1}^{n} Q_i x_{ij} \leqq C \tag{1}$$

where

$$C \geqq Q_i > 0. \tag{2}$$

Each item can be allocated to one box only, hence

$$\sum_{j=1}^{N} x_{ij} = 1 \qquad (x_{ij} = 1 \text{ or } 0). \tag{3}$$

A penalty V_j is assigned to the allocation of items to box j, and this penalty increases with j so that

$$V_{j+1} \gg V_j > 0. \tag{4}$$

The total penalty function when the allocation is completed is then

$$z = \sum_{j=1}^{N} \left(V_j \sum_{i=1}^{n} x_{ij}\right) \tag{5}$$

and provided V_{j+1} is sufficiently greater than V_j, the minimization of the penalty function z will ensure that the smallest number of boxes is used in the allocation process. If the largest number of items that can be loaded into a box cannot exceed p, then the penalty values in equation (3) can be chosen as

$$V_{j+1} > pV_j \quad \text{and} \quad V_1 = 1. \tag{6}$$

The objective function to be minimized is then given in equation (5), subject to the constraints (1) and (3). There may be additional constraints that have to be incorporated, for example if item i_1 cannot be allocated to box j_1 this may be expressed as

$$x_{i_1 j_1} = 0, \tag{7}$$

or if two items i_1 and i_2 must not be allocated to the same box then for all j

$$x_{i_1 j} + x_{i_2 j} \leqq 1. \tag{8}$$

The tree search algorithm of Balas [1] was reformulated for the solution of the loading problem and the method suggested by Glover [6], which was found to reduce the required computer storage, was also incorporated.

A flow diagram of the programme for solving the loading problem is given in Figure 1 and some explanations are given below:

An initial upper bound $\bar{z}$ on the objective function is determined, either by inspection or by using the heuristic algorithm which is referred to later, and whenever a better solution is obtained, it replaces the upper bound.

Of the nN variables x_{ij} a set S is formed, consisting of k variables, to each of which the value of 0 or 1 is assigned, the other $nN - k$ variables remaining "free." Thus, the set S defines a subproblem involving $nN - k$ free variables. This subproblem is resolved either when the best values for the free variables are determined (and these values together with those in set S then form a feasible solution that minimizes z), or when no feasible solution is obtainable, i.e. when a set T (defined below) is empty. In the former case, the solution is recorded for future reference. Note that since all $V_j > 0$, the best values (although not necessarily feasible) for the free variables are always 0 and the resulting solution we denote by X. In either case, k is reduced and the computation procedure is then repeated.

When a solution X is not feasible, a subset T is defined. This subset consists of all the "free" variables representing nodes in the decision tree that are permissible (i.e. that will not violate the capacity constraints and the upper bound $\bar{z}$). One member of T is then transferred to S, thereby defining a new subproblem. The member of T chosen for transfer is such that it produces the allocation of the item with the largest Q_i to the box with the smallest spare capacity. One way to exclude whole sections of the tree at this point (sections that lead to infeasible solutions) is to resort to an LP formulation of the subproblem, allowing the variables to assume any value between 0 and 1. If the solution to this LP problem exceeds the upper bound $\bar{z}$ or if there is no solution that conforms with all the constraints of the problem, then further search of the branch may be abandoned. When T is empty, no further subproblems in the branch under consideration are possible. Various other suggestions [4], [5] that increase the efficiency of the algorithm outlined in Figure 1 were used in the actual programme.

Method II: A Heuristic Algorithm

The heuristic algorithm for the loading problem suggested here involves comparatively little computation and yields a feasible solution N' for the required number of

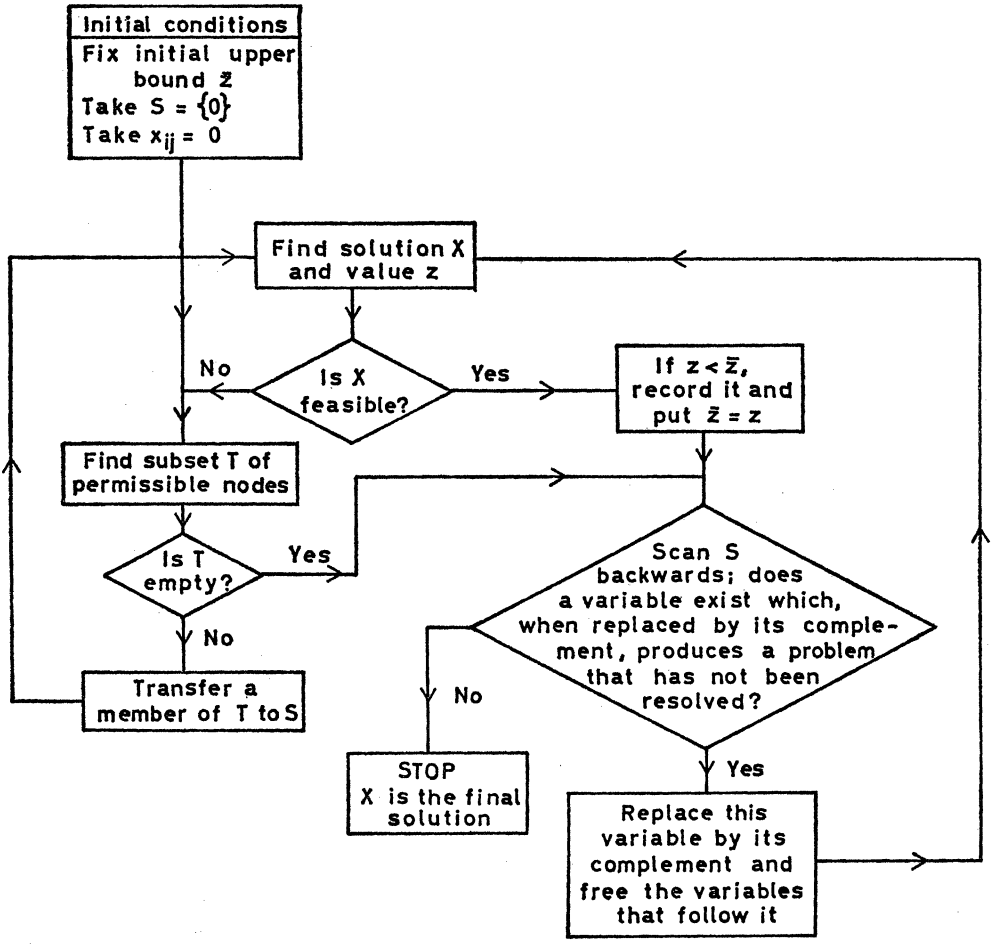

FIGURE 1. Outline of the Solution to the Zero-One Model

boxes. A lower bound N_0 for N' is then found from

$$\sum_{i=1}^{n} Q_i/C \leqq N_0 < \sum_{i=1}^{n} Q_i/C + 1, \tag{9}$$

N_0 being the smallest possible number of boxes dictated by the total capacity requirements of the n items. If the solution $N' = N_0$ then N' is optimal. If, however,

$$N' > N_0 \tag{10}$$

then the solution N' may not be optimal and a reshuffle routine is resorted to, which allows for the allocation of the next item to a box to depart from the allocation that the algorithm would produce, at one point in the procedure. Unlike the zero-one model, this algorithm does not ensure that the optimal solution is always obtained.

The procedure is as follows:

1. Make two lists: one for boxes in ascending order of the available space in each, the other for items in descending order of magnitude.

2. Scan the list of items to see if any single item will precisely fill the first box in the list of boxes; if so, transfer the item to the box and remove both the item and the box from their respective lists.

3. Repeat 2 for each box in turn.

4. Rearrange the list of boxes, if necessary, to have them in ascending order of available space.

5. Take the first item in the list of items and allocate it to the first box that will accommodate it.

6. Repeat 2 for the box that has been loading by step 5.

7. Repeat 4.

8. Repeat 5, 6 and 7 until one of the two lists is exhausted; if the list of items is exhausted first, then total number of boxes required is given by the fully and partially loaded boxes used by this procedure, but if the list of boxes is exhausted first, the remaining items are those that cannot be accommodated in the limited number of boxes available.

In a large number of cases it was found that the algorithm's efficient performance was not impaired when steps 2, 3 and 6 were eliminated from the procedure. The algorithm has been tested for many numerical examples, and three cases are cited in Appendix 2 to illustrate its application.

A reshuffle routine

When the heuristic algorithm yields a solution such that $N' > N_0$ then it is possible that the solution is not optimal. Under these circumstances, the following reshuffle routine may be employed:

1. Start with the two lists and perform the first four steps outlined in the algorithm.

2. Continue with the algorithm as before up to the point when a box is to be loaded with its second item.

3. Instead of allocating the item to this box, reallocate it to the next available box (or the next but one, etc.).

4. Continue allocating the items to boxes according to the algorithm, until the list of items is exhausted.

5. Find the new value of N'. If now $N' = N_0$, the new solution is optimal, but if $N' > N_0$ then the reshuffle routine may be repeated for other alternatives in step 2.

The "reshuffling" of items in the boxes occurs at the point of departure from the algorithm, since at this point the new allocation of an item does not conform to step 5 in the algorithm. However, except for this single point of departure, the algorithm is adhered to.

During the loading process there may clearly be several alternative points of departure from the algorithm. Step 2 in the routine identifies a point at which one box in the list of boxes is to be loaded with its second item. Other possible alternatives for departure from the algorithm are:

—the point at which a second box in the list is to be loaded with its second item, or

—the point at which a box is to be loaded with its third item, or in general:

—the point at which the first (or second, third, etc.) box is to be loaded with its second (or third, fourth, etc.) item. This simply means that the point of departure may be at any point on the list of items, so that the reshuffle procedure may be examined for up to $n - 2$ alternative points of departure (since there are n items, but no meaningful departure from the algorithm can occur when the allocation of the first or the last item is considered); in practice the number of different alternatives is often much smaller.

The results of a large number of numerical examples suggest that the routine is effective in improving on the initial solution of the algorithm, when that solution involves

$N' > N_0$. Furthermore, when an optimal solution $N' = N_0$ exists, the routine is generally effective in producing that solution with a comparatively small number of alternative departure points considered by the routine. It is, of course, possible that a solution $N' = N_0$ exists, but that the routine fails to produce it. A more elaborate routine may then be pursued involving more than one departure point and the iterations repeated for all possible combinations of departure points. In practice, however, the performance of the algorithm, coupled with the reshuffle routine having a single departure point, is very effective in yielding optimal solutions, so that further computations with multi-departure points may be rarely justified.

Multi-Dimensional Constraints

The constraints in (1), as well as those in (7) and (8), refer to constraints in a single dimension. In practice, problems have to be solved involving constraints in more than one dimension, for example when items have an array of values $Q_i(1)$ to denote weight, an array $Q_i(2)$ to denote money values, etc. In general, each item may be specified by r values, each relating to one particular dimension, and similarly each box will have r "capacity" constraints, and constraints (1) and (2) must each be repeated r times. The objective function in equation (5) and the conditions (3), (4) and (6) remain unchanged.

The zero-one programming method can easily cope with the multi-dimensional problem and requires no modifications. The heuristic algorithm can be extended in several ways to handle this problem, and the following two methods are noteworthy:

(i) Consider each dimension separately and solve the problem with the items ordered relative to that dimension, checking at each step that all the constraints for each of the r dimensions are satisfied. r solutions are obtained and the one with the smallest number of boxes is then selected.

(ii) Define a new "equivalent" dimension by a weighting procedure, for example by $Q_i = \sum_{u=1}^{r} Q_i(u)/C(u)$ where $u = 1, 2, \cdots, r$ and then proceeding as for (i) above but with the items ordered according to the Q_i.

Method (ii) was found to be generally more efficient than (i), both computationally and as concerns the quality of the solution.

Computational Results and Conclusions

Fifty one-dimensional problems with up to 50 items were solved by the zero-one programming method and by the heuristic algorithm. The capacity of the boxes was taken as $C = 100$ and the magnitude of the items was randomly generated with uniform distribution. In all these problems the only constraints (in addition to the conditions (2), (3) and (4)) were of that of box capacity, as expressed in equation (i); constraints of the type described in (7) or (8) were not considered. A comparison between the results of the two methods is summarized in Table 1.

In every case Method I (the zero-one programming model) took longer to produce the result than Method II (the heuristic algorithm), and as the programme compilation time (on an IBM 7094) in either case was about 0.5 min., the difference in net computing time was often quite appreciable.

In almost 90% of the cases (44 problems out of 50) Method II yielded the optimal solution without resorting to the reshuffle routine. For the remaining six problems, Method II was effective in yielding optimal solutions in four cases, leaving only two problems out of fifty for which it failed to produce the optimum, but it produced the next best solution.

TABLE 1

Comparison of results for two methods of solution for one-dimensional problems

Problem	No. of items n	Solution N for no. of boxes			Computing Time		
		Method I	Method II		Method I	Method II	
			II(a)	II(b)		II(a)	II(b)
1	10	5	5	—	1.3	.6	—
2	10	4	4	—	1.9	.6	—
3	10	4	4	—	2.5	.6	—
4	10	6	6	—	2.5	.6	—
5	10	4	4	—	1.4	.6	—
6	10	5	6	5	1.5	.6	.8
7	10	5	5	—	1.1	.6	—
8	10	6	6	—	1.9	.6	—
9	15	4	4	—	1.7	.6	—
10	15	3	3	—	1.2	.6	—
11	15	5	5	—	2.4	.6	—
12	15	5	5	—	2.0	.6	—
13	15	5	6	5	2.1	.6	.9
14	15	6	6	—	2.9	.6	—
15	15	3	3	—	1.5	.6	—
16	15	4	4	—	1.8	.6	—
17	20	6	6	—	3.0	.7	—
18	20	6	6	—	5.0	.7	—
19	20	5	5	—	3.1	.7	—
20	20	5	5	—	2.8	.7	—
21	20	6	6	—	4.6	.7	—
22	20	7	7	—	6.1	.7	—
23	20	6	6	—	4.3	.7	—
24	20	7	8	8	6.0	.7	1.0
25	25	5	5	—	3.9	.7	—
26	25	6	6	—	5.5	.7	—
27	25	5	5	—	5.1	.7	—
28	25	6	6	—	6.6	.7	—
29	25	7	7	—	9.0	.7	—
30	25	6	6	—	5.3	.7	—
31	30	5	5	—	4.2	.7	—
32	30	5	5	—	4.8	.7	—
33	30	4	4	—	3.7	.7	—
34	30	6	6	—	8.8	.7	—
35	30	7	8	8	13.2	.7	1.2
36	35	4	4	—	6.0	.7	—
37	35	4	4	—	7.1	.7	—
38	35	3	3	—	4.3	.7	—
39	35	3	3	—	4.1	.7	—
40	35	4	4	—	5.9	.8	—
41	40	3	3	—	6.2	.8	—
42	40	4	4	—	8.1	.8	—
43	40	3	3	—	5.3	.8	—
44	40	4	4	—	7.2	.8	—
45	40	4	5	4	6.6	.8	1.0
46	50	3	3	—	5.9	.8	—
47	50	3	3	—	6.1	.8	—
48	50	3	3	—	7.8	.8	—
49	50	3	3	—	5.0	.8	—
50	50	3	4	3	4.1	.8	1.0

Method I—The zero-one programming method. Method II—The heuristic algorithm. II(a)—Method II without the reshuffle routine; II(b)—Method II including the reshuffle routine.

The computing time refers to the time (in min.) required on an IBM 7094, including about 0.5 min. of programme compilation time.

Some measure of the difficulty in devising optimal solutions to combinatorial problems, such as the loading-boxes problem, may be obtained by considering the success of random solutions. In the context of the loading problem a random solution is obtained by randomly allocating item i to box j and if the box j cannot accommodate item i because of the capacity constraints, another box is tried at random. After the list of items is exhausted, a feasible solution is obtained, and from several such solutions the best can then be selected. Random solutions were tried for the fifty problems referred to in Table 1 and the results are summarized in Table 2. Clearly, some of the randomly generated problems had a great deal of slack and must have been easy to solve, if some 60% of the problems could be solved optimally by no more than five random trials. The difficult problems are represented in the lower part of Table 2, where the random method of solution became increasingly ineffective.

Twelve additional problems with multi-dimensions were also solved and the results are shown in Table 3. Of the five problems for which Method II(a) did not yield the

TABLE 2

Random solutions to fifty one-dimensional problems

No. of trials	No. of problems for which optimal solution was obtained
1	24
2	27
3	30
4	30
5	31
10	36
15	39
20	40
25	41
40	42
70	44
100	45

TABLE 3

Multi-dimensional problems

Problem	No. of items n	No. of dimensions	Solution N for no. of boxes			Computing Time		
			Method I	Method II		Method I	Method II	
				II(a)	II(b)		II(a)	II(b)
1	10	2	4	4	—	1.5	.6	—
2	10	4	3	3	—	1.5	.6	—
3	10	6	5	6	6	1.6	.7	1.0
4	20	2	6	6	—	2.9	.7	—
5	20	4	5	5	—	2.9	.8	—
6	20	6	6	7	7	3.0	.9	1.1
7	30	2	5	5	—	8.2	.7	—
8	30	4	6	7	7	8.1	.8	1.2
9	30	6	5	6	6	4.4	1.0	1.4
10	40	2	4	4	—	7.3	.8	—
11	40	4	4	4	—	7.0	.9	—
12	40	6	4	5	4	5.8	1.0	1.6

optimum, one was solved optimally by using the reshuffle routine. The weighting procedure using the "equivalent" dimension Q_i was used in conjunction with the heuristic in these experiments.

Clearly, the algorithm in Method II is not as effective in solving multi-dimensional problems as it is in handling one-dimensional ones, but even when it did not produce the optimal solution the next best solution was obtained.

Appendix 1. Loading Boxes of Different Capacities

A large number N of boxes is given and what is required is the minimum number of boxes that can hold all the n items, namely the attainment of objective O_1(a).

Let $[x_{ij}]$ be the allocation matrix as defined previously

$y_j = 1$ if box j is loaded with any one object,

$= 0$ if box j is completely empty,

$L =$ any large number greater than n.

The problem, then, can be formulated as follows:

Minimize

$$z = \sum_{j=1}^{N} y_j \tag{11}$$

subject to

$$\sum_j x_{ij} = 1, \tag{12}$$

$$\sum_i x_{ij}Q_i \leqq C_j \qquad (\text{for all } j = 1, 2, \cdots, N), \tag{13}$$

$$\sum_i x_{ij} \leqq Ly_j. \tag{14}$$

Constraint (14) simply ensures that no items are accommodated in a box whose $y_j = 0$.

Thus the case of dissimilar boxes requires the introduction of the variables y_j because the boxes cannot be penalized as in the case of boxes of equal capacity. Moreover, N must now be the total number of boxes, all of which will have to be considered, whereas previously an upper bound on the number of boxes was sufficient.

It is possible, of course, to formulate the problem with boxes of equal size (a special case of the problem discussed in this Appendix), but the number of variables will then be greater (because of the introduction of the y_j's) and the problem will take longer to solve.

The loading problem for objective O_1(b) can be treated in a similar manner.

Appendix 2. Examples to Illustrate the Heuristic Loading Algorithm

Example 1. Items (listed in descending order of magnitude):

68, 65, 40, 39, 32, 30.

Capacity per box $C = 100$.

Solution: Box 1 $68 + 32 = 100$

2 $65 + 30 = 95$

3 $40 + 39 = 79$.

Test: $\sum Q_i/C = 274/100 = 2.74$ or $N_0 = 3$.

The solution $N' = 3 = N_0$, hence this is an optimal solution.

Example 2. Fifty random numbers were taken to represent the magnitude of items and then arranged in descending order:

99,	99,	97,	95,	94,	90,	88,	88,	86,	75,
75,	75,	73,	73,	73,	71,	71,	68,	66,	64,
61,	60,	55,	55,	54,	53,	50,	49,	47,	45,
44,	42,	39,	39,	38,	37,	36,	35,	32,	28,
28,	23,	21,	18,	16,	12,	11,	8,	5,	3.

Capacity per box = 300.
Solution: Box 1 $99 + 99 + 97 + 5 = 300$
2 $95 + 94 + 90 + 21 = 300$
3 $88 + 88 + 86 + 38 = 300$
4 $75 + 75 + 75 + 73 = 298$
5 $73 + 73 + 71 + 71 + 12 = 300$
6 $68 + 66 + 64 + 61 + 39 = 298$
7 $60 + 55 + 55 + 54 + 53 + 23 = 300$
8 $50 + 49 + 47 + 45 + 44 + 42 + 18 + 3 = 298$
9 $39 + 37 + 36 + 35 + 32 + 28 + 28 + 16 + 11 + 8 = 270$.
Test: $\sum Q_i/C = 2664/300 = 8.9$ or $N_0 = 9$.
As $N' = N_0$, this solution is optimal.

Example 3. Items: 60, 50, 30, 20, 20, 20.
Capacity per box $C = 100$.
Solution: Box 1 $60 + 30 = 90$
2 $50 + 20 + 20 = 90$
3 20.
Test: $\sum Q_i/C = 200/100 = 2 = N_0$.
As $N' > N_0$, this solution may not be optimal.

However, when the reshuffle routine is employed, the optimal solution is obtained as follows:

Steps 1 and 2 of the algorithm result in loading the first two items to two boxes respectively and the point of departure is then defined as that of allocating the third item. According to the algorithm, this item should be allocated to box 1 (which contains item 1); instead, the item is now allocated to box 2, so that the position after performing step 3 of the routine is:

Box 1: 60
Box 2: $50 + 30$.

Proceeding with the algorithm the final result is

Box 1: $60 + 20 + 20 = 100$
Box 2: $50 + 30 + 20 = 100$

and now $N' = 2$, which is optimal.

References

1. BALAS, E., "An additive algorithm for solving linear programs with zero-one variables," *Oper. Res.*, Vol. 13 (1965), pp. 517–546.
2. ——, "Discrete programming by the filter method," *Oper. Res.*, Vol. 15 (1967), pp. 915–957.
3. FLEISCHMANN, B., "Computational experience with the algorithm of Balas," *Oper. Res.*, Vol. 15 (1967), pp. 153–154.
4. GILMORE, P. C. AND GOMORY, R. E., "Multistage cutting stock problems of two and more dimensions," *Oper. Res.*, Vol. 13 (1965), pp. 94–120.
5. —— AND ——, "The theory and computation of knapsack functions," *Oper. Res.*, Vol. 14 (1966), pp. 1045–1074.
6. GLOVER, F., "A multiphase-dual algorithm for the zero-one integer programming problem," *Oper. Res.*, Vol. 13 (1965), pp. 879–919.