Research Journal of Applied Sciences, Engineering and Technology 9(11): 922-925, 2015
DOI:10.19026/rjaset.9.2584
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: September 13, 2014 Accepted: September 20, 2014 Published: April 15, 2015

Research Article

Volume Constraint Model and Algorithm for the 0-1 Knapsack Problem

[1]M.A. Ofosu, [1]S.K. Amponsah and [2]F. Appau-Yeboah
[1]Department of Mathematics, Kwame Nkrumah University of Science and Technology,
[2]Department of Mathematics and Statistics, Kumasi Polytechnic, Kumasi, Ghana

Abstract: Today's economic environment has highlighted the importance of properly optimizing every organization's asset including space and facilities portfolio. Reducing space cost by using space more efficiently can release funds for other more important activities according to the National Audit Office (NAO) space management in higher education. Utilization rate is a function of a frequency rate and an occupancy rate. The frequency rate measures the proportion of time that space is used compared to its capacity. In this study, we have proposed a new model formulation and algorithm design for the 0-1 knapsack problem. Our proposed algorithm considers volume or space occupancy of the items as paramount, since no matter how profitable the item is to the camper if the volume or the size is bigger than that of the volume of the knapsack since every knapsack also has a volume, there would be no need to force it into the knapsack. The most interesting thing about the algorithm is that, it eliminates symmetric branching tree and the lazy sorting used by most algorithms in the literature. Computational experiments provided also show that our proposed algorithm can be among the most efficient algorithms available in the literature.

Keywords: 0-1 knapsack problem, algorithm development, knapsack model, volume constraint

INTRODUCTION

The classical Knapsack Problem is the problem of choosing a subset of the *n* items such that the corresponding profit sum is maximized without having the weight sum to exceed the capacity c.

The general problem is formulated as integer linear programming model:

Maximize $\sum_{j=1}^{n} p_j x_j$
Subject to $\sum_{j=1}^{n} w_j x_j \leq$ c
$x_j \in \{0, 1\}^N$

where, x_j is a binary variable equalling 1 if item j should be included in the knapsack and 0 otherwise, thus a single constraint model.

From practical experience it is known that many KP instances of considerable size can be solved within reasonable time by exact solution methods. This fact is due to several algorithmic refinements which have emerged over the last two decades. This include advance dynamic programming recursions, the exact methods, heuristics method, the hybrid methods, the concept of solving a core and the separation of cover inequalities to tighten the formulation.

LITERATURE REVIEW

Sahni (1975) presented the first approximation scheme for KP which made use of a greedy-type procedure which finds a heuristic solution by filling in order of decreasing $\frac{P_j}{w_j}$ ratio, that part of c which is left vacant after the items of a given set M have been put into the knapsack. This gave a time complexity of O (n) for the procedure GS and the number of times it is executed is O (n^k).

Ibarra and Kim (1975) obtained a fully polynomial-time approximation scheme, i.e., a parametric algorithm which allowed one to obtain any worst-case relative error (note that imposing ε is equivalent to imposing (γ) in polynomial-time and space and such that the time and space complexities grow polynomially also with the inverse of the worst-case relative error ε. The basic ideas in the Ibara-Kim algorithm are:

- To separate items according to profits into a class of large items and one of small items
- To solve the problem for the large items only, with profits scaled by a suitable scale factor δ, through dynamic programming. The dynamic programming list is stored in a table T of length:
$\left\lfloor (\frac{3}{\varepsilon})^2 \right\rfloor + 1$

T (k) = undefined or is of the form (L (k), P (k), w (k)), where L (k) is a subset of {1, ..., n}, P (k) = $\sum_{j \in L(k)} P_j$, w(k) = $\sum_{j \in L(k)} w_j$ and k = $\sum_{j \in L(k)} \bar{P}_j$ with $\bar{P}_j = \left[\frac{P_j}{\gamma}\right]$.

Corresponding Author: M.A. Ofosu, Department of Mathematics, Kwame Nkrumah University of Science and Technology, Kumasi, Ghana

Balas and Zemel (1980) developed a heuristics method for the KP. The author's heuristics was a simple exchange algorithm for the core problem of KP, which successfully removes an item i and replaces it by one or two other items j, b in order to obtain a filled knapsack. In spite of the simple structure, Balas and Zemel (1980) were able to prove some interesting properties on the quality of the solution found by their heuristic. The authors procedure assumed that the core items are indexed by 1, ..., m and that the capacity of the core problem is $\bar{c}$. The break item in the core is denoted by b and the residual capacity by filling the knapsack with items j< b is r = $\bar{c} - \sum_{j=1}^{b-1} w_j$.

Early papers which specialized on Dynamic Programming for the 0-1 Knapsack Problem includes Bellman (1957), Dantzig (1957) and Bellman and Dreyfus (1962). The idea is to first fill a small knapsack optimally and then, using this information, fill larger a knapsack optimally. This process is repeated until the original problem is solved completely. Some computational improvements were proposed by Toth (1980).

In his award winning PhD-thesis, Pisinger (1995) devised a dynamic programming recursion, which, although the worst-case time complexity was *O* (*bn*) as for the Bellman recursion, solves most relatively large problem instances without enumerating too many variables. The algorithm starts from the break solution and at each stage either inserts or removes an item. Strong upper bounds are used to limit the number states in the recursion. The enumeration process terminates due to some bounding tests, in which case it is possible to prove that the current incumbent solution is optimal.

Martello *et al*. (1999) studied a model that incorporated cardinality constraints into a very efficient Dynamic Programming algorithm. Although the worst-case time complexity of their algorithm was *O* (*bn*), they solved most instances quite quickly due to the tight bounds produced by the cardinality constraints.

Kolesar (1967) presented the first branch-and-bound approach to the exact solution of KP. The authors algorithm consists of a highest-first binary branching scheme which:

- At each node, we select the not-yet-fixed item j having the maximum profit per unit weight and generate two descendent nodes by fixing x_j, respectively to 1 and 0.
- We continue the search from the feasible node for which the value of upper bound U_1 is a maximum. This however required a large computer memory and processing time.

Due to the large computer memory and time requirement of Kolesar's algorithm Greenberg and Hegerich (1970) presented a method which greatly reduced Kolesar approach, differing from that of the Kolesar in two main respects, namely:

- At each node, the continuous relaxation of the induced sub-problem is solved and the corresponding critical item $\bar{s}$ is selected to generate the two descendent nodes (by imposing $x_{\bar{s}}$ = 0 and $x_{\bar{s}}$ = 1).
- The search continues from the node associated with the exclusion of item $\bar{s}$ (condition $x_{\bar{s}}$ = 0). When the continuous relaxation has an all-integer solution, the search is resumed from the last node generated by imposing $x_{\bar{s}}$ = 1, i.e., the algorithm is of depth-first type.

Horowitz and Sahni (1974) and independently, Ahrens and Finke (1975) derived from the previous scheme a depth-first algorithm in which:

- Selection of the branching variable x_j is the same as that of Kolesar.
- The search continues from the node associated with the insertion of item j (condition x_j = 1), i.e., following a greedy strategy.

However the algorithm presented by Horowitz-Sahni was the most effective, structured and easy to implement and has constituted the basis for several improvements. The authors underlying assumption is that, the items are sorted as in ascending order of profit to weight ratio. A forward move consists of inserting the largest possible set of new consecutive items into the current solution. A backtracking move consists of removing the last inserted item from the current solution. Whenever a forward move is exhausted, the upper bound U_1 corresponding to the current solution is computed and compared with the best solution so far, in order to check whether further forward moves could lead to a better solution. If so, a new forward move is performed; otherwise a backtracking follows. When the last item has been considered, the current solution is complete and possible updating of the best solution so far occurs. The algorithm stops when no further backtracking can be performed.

As an improvement upon Horowitz-Sahni algorithm, Martello and Toth (1977a, b) presented a method which differs from that of Horowitz and Sahni in the following respect:

- Upper bound U_2 was used.
- The forward move associated with the solution of the j^{th} item is split into two phases, namely: building of a new current solution and saving of the current solution. In the first phase, the largest set N_j of consecutive items which can be inserted into the current solution starting from the j^{th} is defined and the upper bound corresponding to the insertion of the j^{th} item is computed. If this bound is less than or equal to the value of the best solution so far, a backtracking move immediately follows. If it is greater, the second phase, that is insertion of the items of set N_j into the current solution, is

performed only if the value of such a new solution does not represent the maximum which can be obtained by inserting the j^{th} item. Otherwise, the best solution so far is changed, but the current solution is not neglected, hence useless backtracking on the items in N_j is avoided.

- A particular formal procedure, based on dominance criteria, is performed whenever, before a backtracking move on the i^{th} item, the residual capacity $\hat{c}$ does not allow insertion into the current solution of any item following the i^{th}. The procedure was based on the following consideration: the current solution could be improved only if the i^{th} item is replaced by an item having greater profits and a weight small enough to allow its insertion, or by at least two items having global weight not greater than w_i+$\hat{c}$. By this approach it is generally possible to eliminate most of the useless nodes generated at the lowest level of the decision tree.

The upper bounds associated with the nodes of the decision tree are computed through a parametric technique based on the storing of information related to the current solution. Supposing the current solution has been built by inserting all the items from the j^{th} to the r^{th}, then, when performing a backtracking on one of these items, (say the i^{th}, j≤i<r), if no insertion occurred for the items preceding the j^{th}, it is possible to insert at least items i+1, ..., r into the new current solution. To this end, we store in $\bar{r}_i$, $\bar{p}_i$ and $\bar{w}_i$ the quantities r+1 ; $\sum_{k=i}^{r} p_k$ and $\sum_{k=i}^{r} w_k$ respectively for i = j, ..., r and in $\bar{r}$ the value r-1 (used for subsequent updating). Following are detailed description of the algorithm.

METHODOLOGY

Mathematical formulation of our proposed problem: The classical Knapsack Problem is the problem of choosing a subset of the *n* items such that the corresponding profit sum is maximized without having the weight sum to exceed the capacity c. The general problem is formulated as integer linear programming model:

$$\begin{aligned}&\text{Maximize } \sum_{j=1}^{n} p_j x_j\\ &\text{Subject to } \sum_{j=1}^{n} w_j x_j \le \text{c}\\ &x_j \in \{0, 1\}^{N}\end{aligned}$$

where, x_j is a binary variable equalling 1 if item j should be included in the knapsack and 0 otherwise, thus a single constraint model.

However in real cases, the knapsacks as well as the items to fill the knapsacks have volumes or sizes and hence consideration should be given to the volume constraints when modelling and developing an algorithm to solve the knapsack problem, with the mathematical formulation:

$$\begin{aligned}&\text{Maximize } \sum_{j=1}^{n} P_j\, x_j\\ &\text{Subject to } \sum_{j=1}^{n} w_{ij}\, x_j \le C_i \text{ , i = 1, ..., m}\\ &\sum_{j=1}^{n} v_{ij}\, x_j \le V_i \text{ , i = 1, ..., m}\\ &x_j \ge 0 \text{ and integer, j = 1, .., n} \qquad (1)\end{aligned}$$

where, x_j is a binary decision variable equalling 1 if item j should be included in the knapsack and 0 otherwise, thus a two constraint model.

Algorithm: Given the knapsack problem with a total of N items, let the number of items type be represented by n mutually exclusive subsets of N, thus n∈N. Let the n subsets also have members n_p that represent the number of items that can be taken from each subset or not taken from each subset, up to the total number of items in each n subset and volume v_p of each item. The input to this algorithm is the feasible region of a binary integer program {x∈ {0, 1}^{n}: Ax≤b}, mutually exclusive n sets that represents the number of item types of the binary integer program, the positive integers of the weights (w_i), the profits (p_i), the (v_i) volume and the knapsack capacity (b). The basic idea of the algorithm is to find non-cover optimum solutions for z^*, W_c, P_c, V_C, the number of items from the various n sets item types that will be selected to obtain the optimum solution to the optimization 0-1 knapsack problem x_j.

Initialization:

```
S: = b; V;
L: = 0, $Max_{Val}$: = 0, $Max_{VOl}$: = 0, $T_W$: = 0;
k: = n
x (k): = {0, ...., n}
```

Main step:

```
$Max_{n_p}$ = 1
For each subset item $i_s \in n_p = 0$ to $Max_{n_p}$
  Solve the following
      L $= i_s \in n_p: \sum_{i_s=1}^{n_p} i_s V_{i_s}$
      $W_{cal} = i_s \in n_p: \sum_{i_s=1}^{n_p} i_s W_{i_s}$
$V_{cal} = i_s \in n_p: \sum_{i_s=1}^{n_p} i_s V_{i_s}$
```

Check for feasibility:

```
  While $W_{cal}$ < S, $V_{cal} < S_V$ and $Max_{Val}$ < L
        $Max_{Val}$ = L; $T_W = W_{cal}$; $Max_{VOl}$ = $V_{cal}$
            For j = 1, k
              x (j) = {$i_s$ }
          End For
  End While
End For
```

Termination:

Output: x (j), Max_{Val}, Max_{VOl} and T_W as the solutions.

Table 1: Computational average time/sec for weakly correlated data instance

Item size	Optimal value	Optimal weight	Optimal volume	Item selected	Time
5	45	26	14	(1, 1, 0, 1, 1)	05.12
10	479	492	310	(1, 1, 0, 1, 1, 0, 1, 1, 1, 1)	05.16
15	463	517	264	(1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1)	05.21
20	571	562	341	(1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1)	05.57
25	703	899	767	(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1)	20.47

Table 2: Computational average time/sec for strongly correlated data instance

Item size	Optimal value	Optimal weight	Optimal volume	Item selected	Time
5	112	112	64	(1, 0, 0, 1, 1)	05.14
10	503	503	119	(0, 0, 1, 1, 1, 1, 1, 0, 1, 1)	05.20
15	1391	1391	394	(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1)	05.29
20	1830	1830	855	(0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1)	06.11
25	1683	1683	966	(1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0)	21.10

Computational experiments: The presented algorithm was implemented in FORTRAN 95 personal edition and a complete listing is available from the author on request. The following results have been achieved on an HP Pavilion g series Laptop machine. The operating system is windows 7 ultimate edition. The system rating is 5.5 windows experience index. The processor is Intel (R) Core (TM) i5-2430m CPU at 2.40 GHz speed. Installed Memory (RAM) is 6.00 GB. The system type is 32-bit operating system. We considered how the algorithm behaves in computational time for different problem sizes and test instances. Two of the three types of randomly generated data instances were considered. Each of the two set of data type instance considered were tested for different problem size of up to 25 for the sake of resource constraint to the researcher, especially time constraint.

The presented results are average computational time values. The results also give the optimal values computed within the average computational time. These are shown in the various tables for the various data instances.

RESULTS

The average computing time for the two types of data instances are given in Table 1 and 2. It can be seen that our proposed algorithm is able to solve the data selected for the two data instances within seconds. It can also be seen from the computational times for both data instances that, a part of the number of data size, which determines the computational time, the weight of the items also plays a significant role in determining the computational time.

CONCLUSION

From the above, our presented results show that our proposed model and algorithm can be among the most efficient algorithms available in the literature for solving the 0-1 Knapsack Problem. The symmetric branching tree and the lazy sorting and reduction used by most algorithms are eliminated. The algorithm is so simple to implement (with the number of lines depending on the problem size) and should be an attractive alternative to other algorithms.

REFERENCES

Ahrens, J.H. and G. Finke, 1975. Merging and sorting applied to the 0-1 knapsack problem. Oper. Res., 23: 1099-1109.

Balas, E. and E. Zemel, 1980. An algorithm for large zero-one knapsack problems. Oper. Res., 28: 1130-1154.

Bellman, R., 1957. Dynamic Programming. Princeton University Press, Princeton, NJ.

Bellman, R. and S.E. Dreyfus, 1962. Applied Dynamic Programming. Princeton University Press, Princeton, NJ.

Dantzig, G.B., 1957. Discrete variable extremum problems. Oper. Res., 5: 266-277.

Greenberg, H. and R.L. Hegerich, 1970. A branch search algorithm for the knapsack problem. Discrete Appl. Math., 2: 21-25.

Horowitz, E. and S. Sahni, 1974. Computing partitions with applications to the knapsack problem. J. ACM, 21: 277-292.

Ibarra, O.H. and C.E. Kim, 1975. Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM, 22: 463-468.

Kolesar, P.J., 1967. A branch and bound algorithm for the knapsack problem. Manage. Sci., 13: 723-735.

Martello, S. and P. Toth, 1977a. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. Eur. J. Oper. Res., 1: 169-175.

Martello, S. and P. Toth, 1977b. Computational experiences with large-size unidimensional knapsack problems. Presented at the TIMS/ORSA Joint National Meeting, San Francisco.

Martello, S., D. Pisinger and P. Toth, 1999. Dynamic programming and tight bounds for the 0-1 knapsack problem. Manage. Sci., 45: 414-424.

Pisinger, D., 1995. A minimal algorithm for the multiple-choice knapsack problem. Eur. J. Oper. Res., 83(2-3): 394-410.

Sahni, S., 1975. Approximate algorithm for the 0-1 knapsack problem. J. ACM, 22: 115-124.

Toth, P., 1980. Dynamic programming algorithms for the zero-one knapsack problem. Computing, 25: 29-45.