RESEARCH ARTICLE OPEN ACCESS

Exploring Traditional Approaches for Solving 0-1 Knapsack Problem

Stephen Opoku Oppong [1], Evans Baidoo [2]
Department of Information Technology [1], Academic City College, Ghana
Department of Computer Science [2], KNUST, Ghana

ABSTRACT

The rationale behind this paper is to identify among the various traditional algorithm design paradigms the optimal approach to be applied to the 0/1 Knapsack problem. The Knapsack problem model is a general resource allocation model in which a single resource is assigned to a number of alternatives with the aim of maximizing the total return. The Knapsack problem is a combinatorial optimization problem. It is a traditional problem with a single limitation, which is an NP-complete problem, and as such, it is not possible to attain an accurate solution for large input.

The main objective of the paper is to present analyses of the several traditional algorithm designs that can be applied to solve the knapsack problem i.e. greedy, branch and bound and dynamic programming. The paper explores the complexity of each algorithm in terms of memory requests, as well as the programming efforts required.

Keywords: — knapsack, greedy, branch and bound, dynamic programming

I. INTRODUCTION

This paper presents a complete utilisation of the traditional approach to solving the knapsack problem. The traditional approach to solving the knapsack problem includes the Greedy method. Branch and Bound and Dynamic Programming. The Knapsack problem is a classic problem with a single constraint. Different types of Knapsack Problems occur, depending on the distribution of the items and knapsacks and also partly due to their wide range of applicability. This paper focuses mainly on the 0/1 knapsack problem. The ultimate aim of this paper is to evaluate the results of traditional algorithms and find the best one.

II. THE 0/1 KNAPSACK PROBLEM (KP)

A problem where an optimal solution has to be identified from a finite set of solutions is a combinatorial optimisation problem of which the knapsack problem is an example, thus the knapsack problem, seeks for a best solution from among many other solutions. It restricts the number x_i of copies of each kind of item to zero or one and the corresponding sum is maximized without having the weight sum to exceed the capacity.

Mathematically the Binary knapsack problem is: assuming without loss of generality that all input data are positive integers, given a set of a set of
n items and a *knapsack* with

$P_j =$ value of item j,

$W_j = weight$ of item j,

C= $capacity\ of$ the knapsack,

select a subset of the items so as to

$$Maximize \sum_{j=1}^{n} P_j X_j \quad (1)$$

$$expose\ to\ \sum_{j=1}^{n} (w_j x_j) \leq C \quad (2)$$

$x_j = 0$ or 1, $j \in N = \{1, \dots, n\}$,

Where $x_j = \begin{cases} 1 & if\ item\ j\ is\ picked; \\ 0 & Otherwise \end{cases}$

The 0/1 Knapsack Problem has an extensive array of applications which includes internet security, industrial, financial, aerospace, computer science, naval, etc.

III. THE TRADITIONAL APPROACHES

a. Greedy Algorithm

Greedy is a strategy that works well on optimization problems. It is an approach that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. It works by making the decision that seems most promising at any moment without worrying about the effect these decisions may have in the future.

Two sets are sustained by the Greedy Algorithm. One contains chosen items and the other contains rejected items. It is made up of four (4) functions.

1. A function that verify whether selected set of items present a solution.
2. A function that verify the viability of a set.

3. A chosen function that tells which of the items shows potential.
4. An ideal function, which does not overtly, provides the value of a solution.

The structure of the greedy algorithm is:

- the set of selected items is unfilled at the first stage i.e., solution set.
- At each step
  - using selection function, an item will be added in a solution set.
  - IF the set would no longer be feasible
    - discard items under consideration (and is never regard again).
  - ELSE IF set is still feasible THEN
    - add the current item.

ALGORITHM GreedyAlgorithm (Weights [1 … N], Values [1 … N])
// Input: Array Weights holds the weights of all items
Array Values holds the values of all items
// Output: Array Solution which points out the items are integrated in the knapsack (‘1’) or
not (‘0’)
Integer CumWeight
Calculate the value-to-weight ratios $r_i = V_i / W_i$, i = 1, …, N, for the items provided
Sort the items in non-increasing order of the value-to-weight ratios
for all items do
if the present item on the listing fits into the knapsack then
put it in the knapsack
else
continue to the next one

Complexity

1. Sorting by any complex algorithm is O (n log n)
2. $\sum_{i=1}^{2^n} 1 = [1+1+1 \ldots 1](n\ times)$ = N ≈ O(N)

From (1) and (2) above, the complexity of the greedy algorithm is, O (n log n) + O(N) ≈
O (n log n). with respect to memory,

b. Branch and Bound

Branch and Bound is a class of exact algorithm for various optimization problems especially integer programming problems and combinational optimization problems (COP). It separates the solution space into smaller sub problems that can be solved independently (branching). Bounding discards sub problems that cannot contain the optimal solution, thus decreasing the size of the solution space. Thus it is proven that this approach is an enhancement over exhaustive search, partly due to the fact that branch and bound build up supposed solutions one component at a time and assesses the partly constructed solutions. If there prove to be no probable values of the residual components, which can result to the solution, the residual components are not produced at all. Solving very large instances of difficult combinatorial problems using this approach is much achievable even though, at the worst case, it still has an exponential complexity.

The algorithm is constructive in nature. The operation of a branch and bound algorithm may be visualized in terms of the construction of a state space tree whose nodes represent the subclasses of solutions.

State space tree is a rooted tree which at each level presents options in the solution space that is dependent on the level above and every promising solution is represented by some path beginning at the root and terminates at the leaf. By definition, the root has a zero level and stands in for the state where no partial solution has been made. A leaf has no children and represents the state where every options leading to the solution have been made.

To explain further, in the state space tree, the nodes is index by n = 1, 2, … in the order in which they are generated by the algorithm. B(n) is denoted as the upper bound associated with node n. A node that has not been branched from is a terminal node. Branching takes place at the terminal node which has the highest value of B(n) and is accomplished by creating two new descendent nodes. A node is described by listing the items which are explicitly included in the solutions contained in the sub-class, and the items which are explicitly excluded from the solutions which are contained in the subclass. Thus at a given node n, three categories of items are conceived. These are

a) the set of items which are explicitly included in the solutions contained in node n
b) the set of items which are explicitly excluded from the solutions contained in node n
c) the items which belong to neither I_n or to E_n have not yet been specifically assigned.

When the set of unassigned items at any node is empty, the node contains only one solution and further branching from that node is impossible.

ALGORITHM BranchAndBound (Weights [1 … N], Values [1 … N])
// Input: Array Weights holds the weights of all items
Array Values holds the values of all items
C, Total capacity
// Output: An array which holds the best solution and its MaxValue

//Stage 1 (Preliminary)

(a) Test the nontrivial feasibility of the problem by verifying at least one index i = 1,2,... ,N, $w_i \leq C$
If the problem is nontrivially feasible, proceed to (b), if not, stop.

(b) If $\sum_{i=1}^{n} w_i \leq C$, then all items may be loaded and the problem is trivial. If not, proceed to (c)

(c) Order the items by decreasing magnitude of v_i/w_i. In all that follows we assume that the items have been so ordered. Proceed to (d)

(d) For node 1 set B(1) = 0, $I_i = \varphi$, $E_i = \varphi$. Proceed to Stage 2.

//Stage 2 (Selection of Node for next Branching)

(a) Find the terminal node with the largest value of B(n). This is the node at which the next branching will take place.

(b) Test if at the current node k, $(I_k \cup E_k)^c = \varphi$. If so, an optimal solution is given by the indices contained in I_k . If not, select the new pivot item i^* by,

$i^* = i$ *such that*

$\frac{v_i}{w_i}$ *is a* max *for* $i\varepsilon(I_k \cup E_k)^c$

Proceed to stage 3(a)

//Stage 3 (Computation of Upper Bounds)

(a) Set n = n + 1, $I_n = I_k$, $E_n = E_k \cup (i^*)$. Proceed to (c)

(b) Set n = n + 1, $I_n = I_k \cup (i^*)$, $E_n = E_k$. Proceed to (c)

(c) Test the feasibility of the solutions contained in node n by verifying if $\sum_{i\varepsilon I_n} w_i \leq C$.
If the test fails, set B(n) = -999, otherwise proceed to compute the upper bound B(n) by first loading all items in the set I_n and then proceeding in sequence, i = 1, 2,… , N, loading as much as possible of each item belonging to $(I_n \cup E_n)^c$ until the total weight loaded is exactly C. The total value so loaded is B(n). Test if the node index n is even. If yes, proceed to 3(b); if no, proceed to Stage 2.

The branch and bound algorithm, in the worst case scenario will produce all intermediate stages and all leaves. Thus, completed tree will have $2^{n-1} - 1$ nodes, i.e. will have an exponential complexity. However, comparism to other algorithm specifically the brute force and backtracking algorithms proves that it is better because averagely it will not produce all possible solutions. The required memory depends on the length of the items or tree.

c. Dynamic Programming

This is an approach for responding to an unpredictable problem by reducing it into a set of simpler sub-problems. It is appropriate to problems displaying the properties of overlying sub-problems and optimal substructure. Dynamic Programming (DP) is an effective procedure that permits one to take care of a wide range of sorts of problems in time O(n^2) or O(n^3) for which an innocent methodology would take exponential time. Dynamic Programming is a general way to deal with a sequence of interrelated choices in an optimum way. This is a general approach to taking care of problems, much like “divide-and-conquer” aside from that unlike divide-and-conquer, the sub-problems will normally overlap. Most in a general sense, the approach is recursive. Once ceased, the solution is solved by expelling data from the stack in the best possible sequence. With a specific end goal to take care of a given problem, dynamic programming method tackle different parts of the problem (sub-problems), and after that consolidate the results of the sub problems to achieve a general solution. Regularly when utilizing a more guileless approach, a considerable lot of the sub-problems are created and solved many times. The dynamic programming methodology tries to take care of every sub-problem just once, therefore diminishing the quantity of calculations: once the answer for a given sub-problem has been registered, it is kept or "memoized": whenever the same solution is required, it is basically looked up. This methodology is particularly valuable when the quantity of rehashing sub-problems develops exponentially as a size's function of the input. Dynamic programming algorithms are used for optimization (for instance, discovering the most limited way between two

ways, or the speediest approach to multiply numerous matrices). A dynamic programming algorithm will look at the earlier tackled sub-problems and will consolidate their answers for give the best answer for the given problem.

```
// Input:
// Values (stored in array v)
// Knapsack capacity (C)
// z = space left or storage limit,
// n= # items still to choose from,
// i= # items,
// w_n = size or data weight of item
{
if (n == 0) return 0;
if (w_n > z) result = Value(n-1,z); // can't use nth item
else result = max{ Value(n-1, z), v_n + Value(n-1, z-w_n), };
return result;
}
```

From the equation, this takes exponential time. However, there are simply $O(n^C)$ dissimilar couples of values the arguments can probably take on, hence ideal for "memoizing".

```
Value(n,C)
{
if (n == 0) return 0;
if (arr[n][z] != unknown) return arr[n][z]; // <- added this
if (w_n > z) result = Value(n-1,z);
else result = max{ Value(n-1, w) , v_n + Value(n-1, z-w_n) };
arr[n][z] = result; // <- and this
return result;
}
```

Given that any known couple of arguments to Value can go through the array test just once, and in doing so generates at most two recursive calls, we have at most 2n(C +1) recursive calls summation, and the sum time is $O(n^C)$.

IV. DISCUSSION

A simple program (Knapsack Optimizer) in java was written to run the analysis. This is shown in Fig 1. The methods included are the greedy method, dynamic programming and branch and bound. The weights and profits are randomly generated using the number of items. The optimal weight and profit are calculated and the memory taken for the results is also calculated.

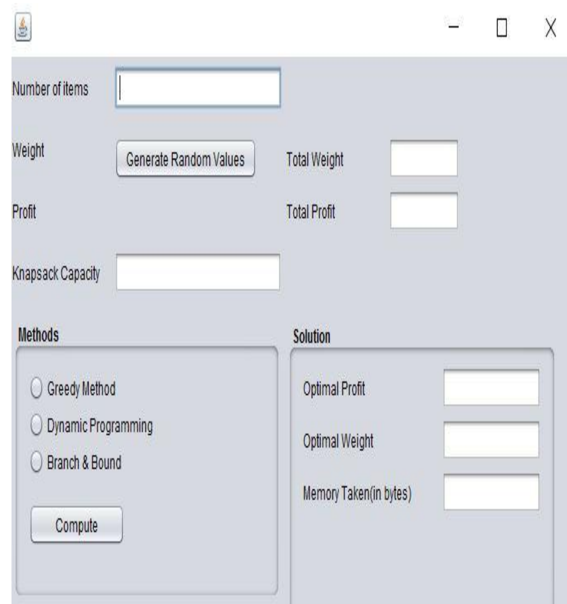

Fig 1: Knapsack Optimizer

The table below summarizes the outcome of the analysis

Table 1 shows the results obtained when the number of items are 50, total weight is 248, total profit is 534 and the capacity of the Knapsack is 200

Table 1:
Analysis of Knapsack Approaches

	Optimal Profit	Optimal Weight	Memory Taken (in bytes)
Greedy Method	498	192	2819600
Dynamic Programming	505	200	2485392
Branch and Bound	505	200	2398280

Table 2 shows the results obtained when the number of items are 100, total weight is 604, total profit is 1047 and the capacity of the Knapsack is 500

Table 2:
Analysis of Knapsack Approaches

	Optimal Profit	Optimal Weight	Memory Taken (in bytes)
Greedy Method	1009	496	2144616
Dynamic Programming	1011	500	2761352
Branch and Bound	1011	500	2345632

Table 3 shows the results obtained when the number of items are 200, total weight is 1045, total profit is 2049 and the capacity of the Knapsack is 800

Table 3:
Analysis of Knapsack Approaches

	Optimal Profit	Optimal Weight	Memory Taken (in bytes)
Greedy Method	1948	793	2354344
Dynamic Programming	1953	800	3435104
Branch and Bound	1953	800	2340776

Table 4 shows the results obtained when the number of items are 500, total weight is 2772, total profit is 5295 and the capacity of the Knapsack is 2000

Table 4:
Analysis of Knapsack Approaches

	Optimal Profit	Optimal Weight	Memory Taken (in bytes)
Greedy Method	4910	2000	2368592
Dynamic Programming	4910	2000	10203512
Branch and Bound	4910	2000	2338760

Table 5 shows the results obtained when the number of items are 1000, total weight is 5504, total profit is 10408 and the capacity of the Knapsack is 4000

Table 5:
Analysis of Knapsack Approaches

	Optimal Profit	Optimal Weight	Memory Taken (in bytes)
Greedy Method	9752	4000	2397200
Dynamic Programming	9752	4000	34486928
Branch and Bound	9752	4000	2168008

Fig 2 below shows the relationship between the number of items used in the knapsack analysis and the memory taken to provide the optimal result

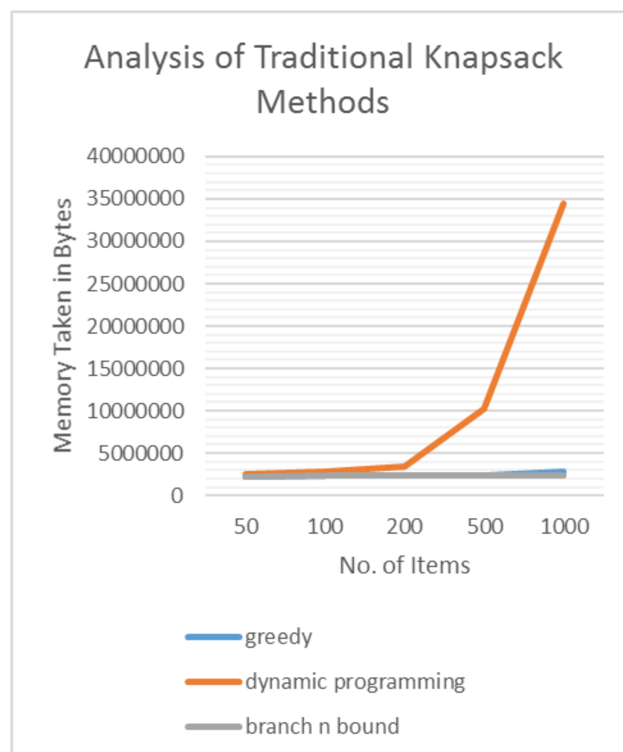

Fig 2: Summary of Analysis of Traditional Methods of Knapsack

From Table 1 to 5, it can be seen that the dynamic programming and branch and bound method produced the same optimal weight and profit but the greedy method provides a slightly lower profit. Since the knapsack problem tries to maximize profit, the dynamic programming and branch and bound are the best as compared to the greedy method.
Taking memory utilised into consideration from Figure 2, it can be seen that the memory utilised by the dynamic programming method increased exponentially as the

number of items increases. The Greedy method and Branch and bound shows an optimum memory utilisation but the memory taken for the branch and bound increase slightly as the number of items increases.

V. CONCLUSION AND FUTURE WORKS

The comparative study of the greedy, dynamic programming, branch and bound shows that while the complexities of these algorithms are known, the nature of the problem they are applied to makes some of them more suitable than others. As we have shown, the choice between the approaches depends on the size of the population.

For future work, we would like to implement some of the more advanced approximation schemes and compare their performance to the traditional methods discussed.

REFERENCES

[1] Bellman, R., E. (1957). Dynamic programming. Princeton University Press, Princeton, NJ.

[2] Hristakeva, Maya and Dipti Shrestha (2005) “Different Approaches to Solve the 0/1 Knapsack Problem” The Midwest Instruction and Computing Symposium, 2005 www.micsymposium.org/mics_2005/papers/paper102.pdf

[3] Levitin, Anany. The Design and Analysis of Algorithms. New Jersey: Pearson Education Inc., 2003.

[4] Matuszek, David (2002) Backtracking, Available at: https://www.cis.upenn.edu/~matuszek/cit594-2012/.../backtracking.html (Accessed: 16th April 2016).

[5] Kolesar, P., J. (1967). A branch and bound algorithm for the knapsack problem. Management Science, 13, 723-735