Algorithms Design and Analysis: Solving the 0-1 Knapsack Problem Using Greedy and Dynamic Programming‘ Algorithms

Arnold Aryeequaye, Innocent Chikwanda, Princess Asiru-Balogun
Department of Computer Science
Ashesi University
Email: {arnold.aryeequaye, innocent.chikwanda, princess.balogun}@ashesi.edu.gh

***Abstract*— The knapsack problem, a combinatorial optimization problem that has intrigued mathematical and computer science academicians and problem solvers for a long while, is a puzzle that has existed since the earlier civilizations. Originating in the early 20th century within operations research, it presents a fundamental question: how to the capital of the small space efficiently come up is the question which may be asked? This essay takes us on a chronological path of the knapsack problem dating back to the problems of resource allocation leading to its present status as a crucially important research question in the fields of computational complexity and algorithm design research. As the solution of this problem has progressed, two approaches, namely, greedy algorithms and dynamic programming, have led the way ahead. Greedy algorithms presents the problem in a convenient and efficient way by making calculations that lead to the locally optimal choice, while dynamic programming returns global optimality by methodically decomposing the subproblems. But they bring different period complexities into account, such as timing and space. Greedy algorithms show the virtues of simplicity and lower time complexity but they may compromise optimality, while dynamic programming is accompanied with a strict guarantee of optimality but on the other hand, at a cost of the input space growth. This paper discusses the advantages and disadvantages of two solving techniques. It selects a computationally-efficient method for decision makers, researchers, and practitioners to solve the knapsack problem and other types of combinatorial optimization problems.**

I. Introduction

Combinatorial optimization constitutes an arena where the Knapsack Problem proved to be an everlasting dilemma to mathematicians, computer scientists and problem solvers alike for truly decades. It is a concept which roots can be traced in the early 20th century when it was established as a major issue for performance analysis in the field of operations research. Stemming from practical dilemmas encountered in resource allocation and logistics, the Knapsack Problem poses a deceptively simple question: which strategies are most likely to meet the storage requirements given that the space for a container is limited?

A rich tapestry of mathematical investigation and algorithmic invention is revealed by following the history of the Knapsack Problem. The problem was first presented by mathematician Tobias Dantzig in 1897. During World War II, it gained prominence in the operations research community and was used in resource management and military logistics. Ever since, it has grown beyond its practical roots and established itself as a fundamental issue in the field of computational complexity and algorithm design research.

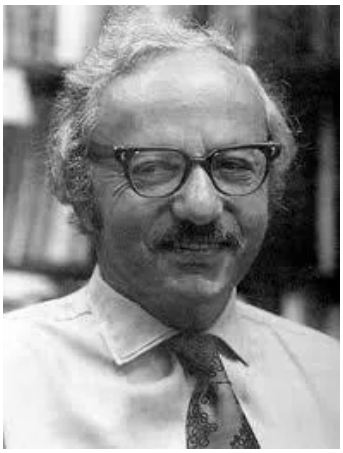

Fig. 1. Tobias Dantzig

The search for an optimal solution to the Knapsack Problem has resulted in the development of various algorithmic approaches, each of which is characterized with a distinct operation principles and efficiency/effectiveness trade-offs. Taking the greedy algorithms and dynamic programming in consideration; the two of them appeared to be the most popular strategies so far, having distinct advantages and disadvantages.

Local optimality and the general principal of Greedy Algorithm is to make the best choice all the time, under the hope that it will lead to the global optimal. The simplicity and efficiency in them that they get to use to solve the Knapsack Problem are one of the attributes that make them appealing, especially in large dataset scenarios and situations with limited resources. On one hand, they have the characteristics of close-mindedness which occasionally results in inefficient planning. On the other hand, this makes it necessary to be careful while using these metrics.

However, while Dynamic programming has a systematic approach, which involves separation of the problem to smaller subproblems and optimization of best solutions that can apply to solving larger issues, The first use of Dynamic Programming algorithms for the Knapsack Problem dates from the 1950s by Richard Bellman, who presented a potent

methodology able to face different complexity levels of the constraints and to develop a solution of high quality. The fact they are using memoization and recursion though may incur temperature cost when handling larger instances which may entail extra memory and time overheads.

Fig. 2. Richard Bellman

In solving the Knapsack problem as well as by studying the different approaches of Greedy and Dynamic programming, we discover their specific strengths. Through deconstructing of domain concepts, considering essence, looking at application areas, we lay the groundwork for understanding combinatorial optimization better. It is our hope that this will open the door to even greater algorithmic advances in the future

II. Literature Review

As aforementioned the KnapSack problem is a frequently recurring problem in many practical situations. Therefore, the approaches to solving it, chiefly Dynamic Programming and the Greedy Approach, have been the subject of many researchers for theoretical exploration as well as for their practical utility[3]. It is observed that for large inputs Dynamic Programming typically performs much better than the Greedy Algorithm as the global maximum becomes more differentiated with local minima[4]. However, researchers such as Wu make a case that both the Dynamic programming and Greedy Algorithm have their best case scenarios in the real world. For example, considering you need to optimize an investment portfolio DP is the best option as it can yield the most optimal solution after a complete simulation of all possible combinations. On the other hand in logistics where a little accuracy can be sacrificed for a prompt accomplishment of the task, the Greedy Algorithm is the best bet. For example, if you are moving out and transporting furniture, the fastest strategy would be to pick the best known option at each step and fit it in rather than try to test all possible combinations for the most optimal.[2]

III. Description of Algorithm

Greedy Algorithm Description: The Greedy Algorithm for the knapsack problem works by the process of making near optimal selections at each step, hopefully leading to an optimal solution. It goes throw the available objects and picks the one, which gives me the best short term happiness regardless of the damage for the future. Continuation of this procedure may finish when the knapsack is full or no more items to consider are upcoming.

Dynamic Programming Description: The Dynamic Programming solution approach for the knapsack problem operates on solving the sub problems of either excluding or including the last item in the knapsack. Our base case is identified to be zero for zero items and zero weight knapsack, and the weight of the knapsack is recursively broken down into smaller sub problems where each unit of weight is considered as the maximum weight. The matrix is populated with the formula; m[i,W] = max(m[i-1,w] , m[i-1,w-w[i]]+P[i]) The solution to the problem is the value in the cell intersecting the last row and last column. For the selection process, if the maximum value occurs in your current row and the row before, you do not select the current item to add to the knapsack. However if the value in the row before is different, the item is selected. The table is then backtracked to determine the total number of optimal subsets possible

IV. Pseudocode for Greedy Algorithm

Greedy Approach:

Algorithm 1 GreedyKS

procedure GREEDYKS(Options, MaxWeight)
  Input: Options, a 3-D array of size n, the number of items to be selected
  MaxWeight, the maximum allowed weight of items
  Output: Choices, a 1-D array of size n with the chosen elements
  for $i = 0$ to n **do**
    $Options[2] \leftarrow \frac{v_i}{w_i}$ ▷ populate the array's third column with the value to weight ratio of the items
  end for
  QUICKSORT(Options) ▷ sort Options based on the ratios
  $weight, j, k \leftarrow 0, 0, 0$
  $choices \leftarrow []$
  while ($weight < MaxWeight$ and $j < n$) **do**
    if ($weight + w_j < MaxWeight$) **then**
      $weight \leftarrow weight + w_j$
      $choices[k] \leftarrow Options[j]$
      $k \leftarrow k + 1$
    end if
    $j \leftarrow j + 1$
  end while
  return $choices$
end procedure

V. Implementation of Greedy Algorithm

Summarize the findings of your research.

VI. Pseudocode for Greedy Algorithm

Dynamic Programming Approach:

Algorithm 2 Knapsack Algorithm

function KNAPSACK(weights$[1..n]$, values$[1..n]$, W, n)
    // Create a 2D array to store the results of subproblems
    // Initialize with 0s, where dp$[i][j]$ represents the maximum value for
    // a knapsack with capacity j and first i items.
    Let dp$[0..n][0..W]$ be a 2D array, initialized with 0

    // Build the table bottom-up
    for i from 1 to n **do**
        for w from 1 to W **do**
            if weights$[i-1] \leq w$ **then**
                // If the current item's weight is less than or equal to the
                // current capacity w, we have two choices:
                // 1. Include the item
                // 2. Exclude the item
                dp$[i][w] \leftarrow \max($values$[i-1]$ + dp$[i-1][w-$ weights$[i-1]]$, dp$[i-1][w])$
            else
                // If the current item's weight is more than the current capacity w,
                // we cannot include the item in the knapsack.
                dp$[i][w] \leftarrow$ dp$[i-1][w]$
            end if
        end for
    end for
    // The maximum value will be at dp$[n][W]$
    return dp$[n][W]$
end function

VII. Implementation

To test our two implementations we used the the two inputs shown below:

TABLE I
OPTIONS1

Item	Value	Weight	Initial Value/Weight
1	10	2	0
2	20	5	0
3	30	10	0
4	40	15	0

TABLE II
OPTIONS2

Item	Value	Weight	Initial Value/Weight
1	5	2	0
2	21	3	0
3	10	10	0
4	40	8	0

Greedy Algorithm:

We implemented our Greedy Algorithm solution using Java. The full code can be found on our github: click here to view. Below are snapshots to present implementation and execution of the Greedy Algorithm program implemented using Java.

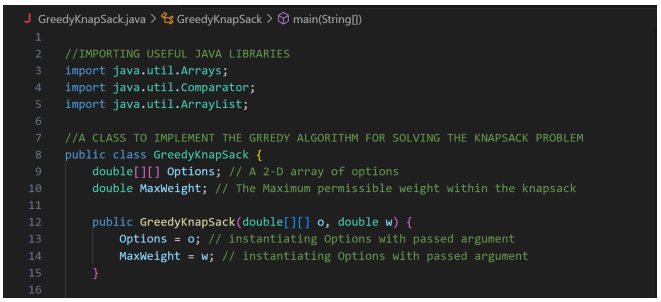

Fig. 3. Snippet 1

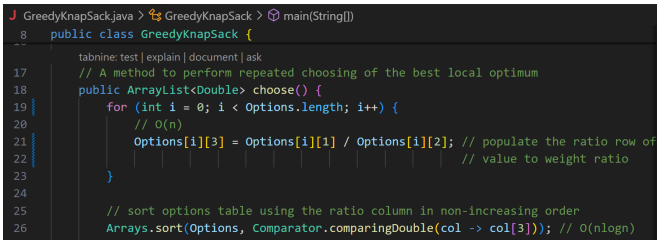

Fig. 4. Snippet 2

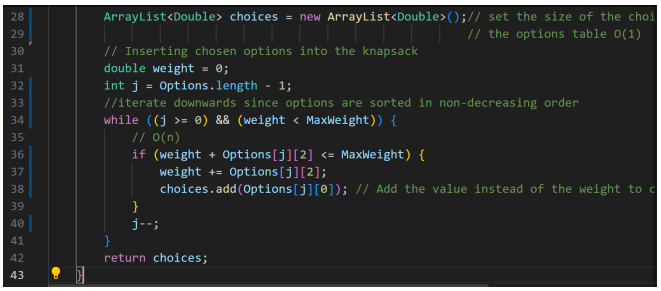

Fig. 5. Snippet 3

Experiment 1 results

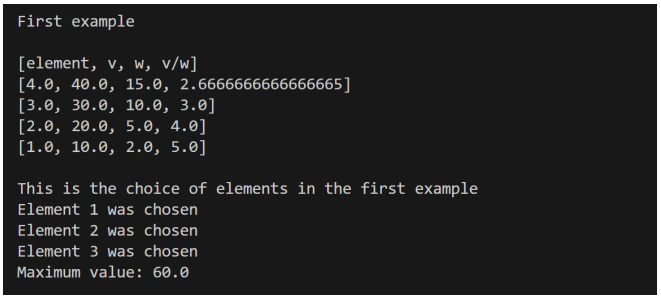

Fig. 6. Example Picture

Experiment 2 results

```
Second example

[element, v, w, v/w]
[3.0, 10.0, 10.0, 1.0]
[1.0, 5.0, 2.0, 2.5]
[4.0, 40.0, 8.0, 5.0]
[2.0, 21.0, 3.0, 7.0]

This is the choice of elements in the second example
Element 2 was chosen
Element 4 was chosen
Element 1 was chosen
Maximum value: 66.0
```

Fig. 7. Example Picture

Dynamic Programming Algorithm:

```
GreedyKnapSack.java > GreedyKnapSack > main(String[])
1
2   //IMPORTING USEFUL JAVA LIBRARIES
3   import java.util.Arrays;
4   import java.util.Comparator;
5   import java.util.ArrayList;
6
7   //A CLASS TO IMPLEMENT THE GRREDY ALGORITHM FOR SOLVING THE KNAPSACK PROBLEM
8   public class GreedyKnapSack {
9       double[][] Options; // A 2-D array of options
10      double MaxWeight; // The Maximum permissible weight within the knapsack
11
12      public GreedyKnapSack(double[][] o, double w) {
13          Options = o; // instantiating Options with passed argument
14          MaxWeight = w; // instantiating Options with passed argument
15      }
16
```

Fig. 8. Snippet 1

```
1   // implementing a tabulation (bottom up) algorithm to solve the 0-1 knapsack problem
2   //author : princess asiru-Balogun
3
4   import java.util.*;
5
6   class DPKnapSack {
        tabnine: test | explain | document | ask
7       public int choice(int knapsack_Capacity, int[] item_values, int[] item_weights) {
8           /* number of items given is assigned to n */
9           int n = item_values.length;
10          /* table to store values of subporblems as calculated */
11          int[][] table = new int[n + 1][knapsack_Capacity + 1];
12
```

Fig. 9. Snippet 2

```
6   class DPKnapSack {
        tabnine: test | explain | document | ask
7       public int choice(int knapsack_Capacity, int[] item_values, int[] item_weights) {
8           /* number of items given is assigned to n */
9           int n = item_values.length;
10          /* table to store values of subporblems as calculated */
11          int[][] table = new int[n + 1][knapsack_Capacity + 1];
12
13          // Fill the table in bottom up manner
14          for (int i = 0; i <= n; i++) {
15              for (int w = 0; w <= knapsack_Capacity; w++) {
16                  // base case is when there are no items and no knapsack weight, so we f
17                  // first column and row with zero
18                  if (i == 0 || w == 0) {
19                      table[i][w] = 0;
```

Fig. 10. Snippet 3

```
20                      // checking first taht the item is not heavier than the knapsack w
21                      // populate the cell with the maximum or optimal value from either
22                      // last item or excluding the last item.
23                  } else if (item_weights[i - 1] <= w) {
24                      table[i][w] = Math.max(item_values[i - 1] + table[i - 1][w - item_
25                  } else {
26                      table[i][w] = table[i - 1][w];
27                  }
28              }
29          }
30          // the maximum value will be in the cell intersectng the last row and last
31          // column.
32          return table[n][knapsack_Capacity];
33      }
```

Fig. 11. Snippet 3

Experiment 1 results

```
Maximum value is 70
Maximum value is 66
```

Fig. 12. Example Picture

THEORETICAL ANALYSIS

Greedy Approach:

Time Complexity:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} 5 + n\log n + 4 + \sum_{j=0}^{n} 13 \\ &= 5\sum_{i=1}^{n} 1 + n\log n + 4 + 13\sum_{j=0}^{n} 1 \\ &= 5\sum_{i=1}^{n} 1 + n\log n + 4 + 13\sum_{j=0}^{n} 1 \\ &= 5n + n\log n + 4 + 13n \\ &= \Theta(n\log n) \end{aligned}$$

Therefore, the overall time complexity of the greedy approach is $\Theta(n \log n)$.

Space Complexity:

The options table has size n where n in the number of elements. The overall space complexity of the algorithm never changes, therefore it is $\Theta(n)$.

Dynamic Programming approach: Tabulation method:

Time complexity: This algorithm has a time complexity of $O(N \times W)$, where N is the number of elements and W is the weight of the knapsack. Therefore the time complexity is $\Theta(NW)$.

Space Complexity: We make use of an additional 2D array $[n][W]$ to store the results of the computations. This implies space complexity to be $O(N \times W)$ as well.

VIII. DISCUSSION OF RESULTS

It can be seen from the results that both approaches to the solution may yield close results to the problem. However as it should be noted that the degree of error between the Greedy Approach from the Dynamic Programming optimal solution continues to grow as we approach infinity. Nonetheless, determining the best option between greedy algorithms or dynamic programming in the course of solving the knapsack problem depends on the application of a well-balanced analysis of their apparent as well as the hidden advantages. Conventional algorithms like greedy algorithms, known for their simplicity and good performance, always select a local optimal point at the current stage of execution, hoping to get a global optimum at the end of the algorithm. Although their simplicity in designing and their lower time complexity provide advantages, they may encounter some problems involving their short-sightedness. This occurs when these algorithms make short-sighted decisions due to the blind point of their decision-making process. Because of this restriction, they are not applicable to cases of complex

relationships where competitors are mutually dependent or where constraints exist.

While the top-down technique is deduced from the subproblems to the global solution, the bottom-up method is driven to the global solution with the complete information of all subproblems being fully dissected and stored as the search-table. This ensures optimality at the cost of exponential time complexiy though the space is well balanced for bigger sizes of problems The demand to store intermediate results into the table that repeatedly creates higher memory requirements and even worse exponential time complexity is urgently needed. And the key thing is that the implementation of dynamic solutions requires the problem dependencies to be investigated so that it could be added to the complexity of the approach.

In the real world, we pick up a methodology between these methodologies depending on the particular properties of corresponding issue. Having in mind the greedy algorithms, these algorithms may be sufficient while operating with simple problem instances where reasonably is possible to globally optimal solutions to be approximated through local decisions. Statically however, dynamic programming is more efficient for solving the problems of higher degree of complexity and state-of-art precision, where both the time efficiency and the optimal results are tolerable. Unraveling the tradeoffs amongst those methods is the leading feature that will help in choosing the best approach and for the knapsack problem solution providing the knapsack problem solution one must know the trade-offs among these methods.

IX. Conclusion

In summary, a competition between greedy algorithms and dynamic programming in solving this knapsack problem occurs depending on a fine-tuned knowledge of the stated advantages and drawbacks of the two. The greedy greedy algorithms have simple and good performance, which fits in problems that are easy to solve where locality optimality is sufficient. Though they can provide immediate solutions, their fixed approach can be limited by complex scenarios which include interaction among various variables that are interrelated or subject to some constraints. On the contrary, in dynamic programming absolute optimality is achieved by seeking all the possible solutions using a sequential processing. Nonetheless, it is at the expense of excessive computational uploading and memory expenditure.

Among these methods, the question of the most appropriate is posed, as the choice is based on the features of the problem and its need. The avarice algorithms that only work for simple instances and allow straight forward solutions, are never equal to dynamic programming which is created to shine when it comes to developing the most precise and scalable solutions that complexity demands. Being aware of all such trade-offs being the practitioners' endeavors to find best possible solutions to the problems similar to knapsack problem being the combinatorial optimization challenge is the key thing.

Eventually this work lays the emphasis on a complex evaluation of algorithms based on multiple criteria of efficiency, productivity and cost aspect, and so on. Through identifying attributes of greedy algorithms and dynamic programming, academics and practitioners can take knowledgeable actions tackling one-on-one just about any real-world optimization challenge.

References

[1] Bari, A. (2018). 4.5 0/1 knapsack - two methods - Dynamic Programming. [YouTube Video]. Retrieved from https://www.youtube.com/watch?v=nLmhmB6NzcM&t=155s&ab_chann

[2] Wu, Y. (2023). Comparison of dynamic programming and greedy algorithms and the way to solve 0-1 knapsack problem. *Applied and Computational Engineering*, 5(1), 631–636. DOI: https://doi.org/10.54254/2755-2721/5/20230666

[3] Hristakeva, M., Shrestha, D. (2005). Different Approaches to Solve the 0/1 Knapsack Problem. *Midwest Instruction and Computing Symposium*.