International Journal of Advanced Engineering and Management Research

Vol. 5, No. 02; 2020

ISSN: 2456-3676

0/1 KNAPSACK PROBLEM: GREEDY VS. DYNAMIC-PROGRAMMING

Namer Ali Al Etawi [1], Fatima Thaher Aburomman [2]

[1] AL-Balqa Applied University, Princes Rahma College,
Al-Salt-Jordan

[2] AL-Balqa Applied University, Princes Rahma College,
Al-Salt-Jordan

ABSTRACT

Knapsack Problem (KP) is one of the most profound problems in computer science. Its applications are very wide in many other disciplines liken business, project management, decision-making, etc. In this paper we are trying to compare between two approaches for solving the KP, these are the Greedy approach and the Dynamic Programming approach. Each approach is explained by an algorithm. Then results are obtained by implementing the algorithm using Java. The results show that DP outperforms Greedy in terms of the optimized solution, while greedy is better than DP with respect to runtime and space requirements.

Keywords: Knapsack Problem, Greedy Algorithm, Dynamic-Programming Algorithm.

1. INTRODUCTION

The Knapsack Problem (KP) is an example of a combinatorial optimization problem, which seeks fora best solution from among many other solutions [1]. KP can be formalized in the following manner [2]:

- Assume we have a knapsack with a capacity C.
- Assume we have $n$items to be filled in the knapsack in a way that maximizes the profit while still bounded to the capacity of the knapsack.
- Each item has a weight $w$and a profit p.
- Let the selection of an item i expressed by x_i, such that $x_i \in \{0, 1\}$ and $i = 1, 2, \dots, n$
- The objective is to:

$$maximize \sum_{i=1}^{n} p_i x_i \ \ subject\ to \sum_{i=1}^{n} w_i x_i \leq C$$

The sense of the knapsack problem is that we have a set of decisions, aka solutions, each decision has a certain value (weight) which can be looked at as the feasibility of that solution, the aim is to select all set of feasible solutions. In other words, a decision-making process must be initiated to compare between the alternatives [8].

Vol. 5, No. 02; 2020

ISSN: 2456-3676

We can think of the 0/1 Knapsack problem as an investment-decision making problem. An investor who has an amount of money C, and has n investments; for each investment there is a cost w_i and some expected profit p_i that will return from investing in an investment. The investor will strive to invest in the investments that will maximize his profit.

In this paper, we are trying to compare between two well-known algorithms that are used to solve the knapsack problem, these are the Greedy and the Dynamic-Programming algorithms. We implement the algorithms in Java and compare the results of both algorithms together.

The remainder of this paper is organized as follows: in section 2, we present some work related to the Knapsack problem. The methodology is illustrated in section 3. The Greedy and Dynamic Programming approaches are discussed in sections 4 and 5 respectively. Results are discussed in section 6. Finally, we conclude our research and highlight areas of future work.

2. RELATED WORK

KP is a hot research area. It's classified as an NP-Problem [1]. A lot of work done to solve the problem and to make comparisons between different approaches that tried to solve this approach. Greedy method, dynamic programming, branch and bound, and backtracking are all methods used to address the problem.

Maya Hristakeva and Dipti Shrestha [3] started a similar work in 2005 to compare not only between these two types of algorithms but also with other algorithms including genetic algorithm, backtracking, and others.

In this paper, we try to overcome the limit reached by [3] which is 1000 items; we start with 100 items and increase the number until we reach 1 million items.

3. Methodology:

We first give a formal definition to the knapsack problem. Then, we give a brief on both the greedy and dynamic programming approaches. We show the algorithm used in each approach to solve the KP. Runtime and space complexity are shown for each algorithm. We then implement the algorithms in Java, obtain the results and compare between then and draw the conclusions.

4. The Greedy Algorithm

Greedy algorithm can be classified as blind. In its simplest way, it looks always for the future and doesn't look back to the past. All what greedy is trying to achieve is to try to collect the best benefit from the solutions in hand, regardless of whether some other solutions can be more beneficial at another moment of time in the future.

The algorithm shown in Figure 1 describes the solution of the KP using the greedy approach [3].

Vol. 5, No. 02; 2020

ISSN: 2456-3676

```
ALGORITHM GreedyAlgorithm (Weights [1 … N], Values [1 … N])
// Input:  Array Weights contains the weights of all items
           Array Values contains the values of all items
// Output: Array Solution which indicates the items are included in the knapsack ('1') or
           not ('0')

Integer CumWeight
Compute the value-to-weight ratios r_i = v_i / w_i, i = 1, …, N, for the items given
Sort the items in non-increasing order of the value-to-weight ratios
for all items do
        if  the current item on the list fits into the knapsack then
              place it in the knapsack
        else
              proceed to the next one
```

Figure 1: Pseudocode for knapsack 0/1 using a greedy algorithm

Table 1 shows the time complexity computation for the greedy method by dividing the algorithm show in Fig. 1 to 3 components: (1) Ration Computation, (2) Sorting, and (3) Decision Making.

Table1: Time complexity calculation of the 0/1 greedy algorithm

Component	*Discussion*	*Effort*
Ratio Computation	*Takes only one loop of n (the number of items)*	$O(n)$
Sorting	*One of the efficient well-known sorting algorithms. e.g. Merge Sort.*	$O(nLgn)$
Decision Making	*The greedy algorithm for fitting the items into the knapsack. Contains only one loop of n items.*	$O(n)$

As shown in Table 1, the time complexity of the algorithm shown in Fig. 1 is $O(nLgn)$. This means the algorithm is an efficient in terms of runtime complexity.
Space complexity is measured according to (1) and the analysis is shown in Table 2:

$$Space\ Complexity = O(inputs) + O(outputs) + O(DS) \quad (1)$$

Table 2: Space complexity calculation of the 0/1 greedy algorithm

Component	*Discussion*	*Effort*
Inputs	*Two arrays, weights and values, each of size n.*	$O(n) + O(n) = O(2n)$
Outputs	*An array of knapsack contents. As a worst-case scenario, all items will be selected.*	$O(n)$
Data Structures	1. *One array to calculate ratios.* 2. *Space complexity of the merge sort.*	$O(n) + O(n) = O(2n)$ $O(2n) + O(n) + O(2n)$ $O(n)$

5. Dynamic-Programming Algorithm Dynamic programming (DP) is different than greedy in the way in which the optimized solution is selected [7]. As mentioned earlier, greedy always seeks the maximum available profit without looking for the future or the past. DP generates all feasible solutions from the solutions in hand, then iterates again through all of them to select the best solution.

The pseudo code of DP is divided into two parts; the first part is the evaluation of alternatives which is shown in Figure 2 [3].

```
ALGORITHM Dynamic Programming (Weights [1 … N], Values [1 … N],
                                Table [0 … N, 0 … Capacity])
// Input:  Array Weights contains the weights of all items
           Array Values contains the values of all items
           Array Table is initialized with 0s; it is used to store the results from the dynamic
           programming algorithm.
// Output: The last value of array Table (Table [N, Capacity]) contains the optimal
           solution of the problem for the given Capacity

for i = 0 to N do
        for j = 0 to Capacity
            if  j < Weights[i]      then
                    Table[i, j] ← Table[i-1, j]
            else
                    Table[i, j] ← maximum { Table[i-1, j]
                                            AND
                                    Values[i] + Table[i-1, j – Weights[i]]
return Table[N, Capacity]
```

Figure 2: Part 1 of DP, alternative evaluation

Vol. 5, No. 02; 2020

ISSN: 2456-3676

Starting from the maximum value reached from the algorithm shown in Fig. 2, a decision-making phase starts to select the optimized solution as shown in Figure 3 [3].

```
n ← N          c ← Capacity
Start at position Table[n, c]
While the remaining capacity is greater than 0 do
        If Table[n, c] = Table[n-1, c] then
                Item n has not been included in the optimal solution
        Else
                Item n has been included in the optimal solution
                Process Item n
                Move one row up to n-1
                Move to column c – weight(n)
```

Figure 3: Decision-Making phase

The runtime complexity of the first part of the algorithm can be calculated using equation (2):

$$complexity = \sum_{i=0}^{n}\sum_{j=0}^{C} C1 = C1\sum_{i=0}^{n}\sum_{j=0}^{C} 1 = C1\sum_{i=0}^{n} C = C1nC \qquad (2)$$

Whereas C1 is constant. Thus, the complexity of the algorithm shown in Fig. 2 is $O(nC)$.

Time complexity of the second part of the algorithm, Fig. 3, is calculated in the same manner; only one loop of the number of items (n), thus the time complexity of it is $O(n)$.

Thus, the runtime complexity of DP is $O(nC) + O(n)$, which is $O(nC)$, and this complexity is less efficient than the $O(nLgn)$ of the greedy approach.

Table 3 represents the analysis of the space complexity of DP in the same manner used to analyze the space complexity of the greedy algorithm.

Table 3: Space complexity calculation of the 0/1 DP algorithm

Component	*Discussion*	*Effort*
Inputs	*Two arrays, weights and values, each of size n.*	$O(n) + O(n) = O(2n)$
Outputs	*An array of knapsack contents. As a worst-case*	$O(n)$

Vol. 5, No. 02; 2020

ISSN: 2456-3676

	scenario, all items will be selected.	
Data Structures	*A matrix of n x C items.*	$O(nC)$ $O(2n) + O(n) + O(nC)$ $O(nC)$

Compared to the greedy algorithm, DP costs more memory$O(nC)$. This means, DP requires more memory space to perform.

6. Results

To perform the required testing and comparisons between the two algorithms, an implementation of the two algorithms was done in Java. All results are test on an Intel Core(TM) i3 M-380 CPU with 2.53 GHz and 3 MB cache with 2 cores [4]. Memory size of the computer in use is 4 GB of RAM. The PC runs windows 7 Enterprise edition 64-bit.

6.1 The Application

The application is developed using Java JDK 8.0; the tool used to design the application is Net Beans IDE 8.0.2. It's a Graphical User Interface (GUI) application that uses the java x. swing package.

Item weights are generated randomly using JavaRandom.next() method which has$O(1)$ complexity [5]. Thus, the total complexity of the item generation shown in Fig. 5 is $O(n)$since it uses only one loop that iterates $n$times, such that$n$ is the number of elements (items).

```
Random rnd = new Random();
int lowerBound = 1;
//Generating Items, Runtime Complexity = O(n)
for (int i = 1; i <= parent.getProblem().getItems(); i++){
    // Weight generation, such that 1 <= w <= maxWeight
    int w = rnd.nextInt(parent.getProblem().getMaxWeight() - lowerBound + 1) + lowerBound;
    // Profit generation, such that 0 <= p <= maxProfit
    int p = rnd.nextInt(parent.getProblem().getMaxProfit() + 1);
    Item item = new Item("w" + i, w, p);
    parent.getProblem().addItem(item);
    ctrlProgress.setValue(i);
    lblPercentage.setText((int)(ctrlProgress.getPercentComplete()*100) +"%");
    item = null;
}
```

Figure 4: Item generation and weight randomization

Vol. 5, No. 02; 2020

ISSN: 2456-3676

6.2 Greedy Results

Greedy algorithm is implemented based on the same algorithm shown in Fig. 1. Here, the Collections.sort() method of Java is used to sort the items based on the ratios. According to Java documentation [6] it takes $O(nLgn)$ runtime complexity.

Knapsack capacity is set to 30, maximum weight and profit are both set to 1000, and items are generated as show in table 4.

Table 4: Items generated and results of greedy method

Items	*Items Selected*	*Knapsack Value*	*Used Capacity*	*Available Capacity*	*Elapsed Time (nSec)*
100	*3*	*1577*	*27*	*3*	*0.000306044*
1000	*6*	*4211*	*28*	*2*	*0.000573912*
10000	*18*	*13060*	*30*	*0*	*0.003385042*
100000	*30*	*25914*	*30*	*0*	*0.031406762*
1000000	*30*	*29646*	*30*	*0*	*0.467525163*

6.3 DP Results

Algorithm shown in Figure 2 and Figure 3 is implemented using Java based on table 5 using the same knapsack capacity of 30 with a maximum weight and profit of 1000.

Table 5: Items generated and results of DP method

Items	*Items Selected*	*Knapsack Value*	*Used Capacity*	*Available Capacity*	*Elapsed Time (nSec)*
100	*2*	*377*	*25*	*5*	*0.00017*
1000	*5*	*6542*	*27*	*3*	*0.00026*
10000	*16*	*91193*	*29*	*1*	*0.009256*
100000	*29*	*367864*	*29*	*1*	*0.093495*
1000000	*29*	*402900*	*29*	*1*	*1.448606*

6.4 Comparison

6.4.1 Items Selected

Vol. 5, No. 02; 2020

ISSN: 2456-3676

As a result of comparison between the two tables 4 and 5, we can see that the greedy method selects more items than the dynamic programming.
An explanation of that could be clear from the algorithm itself; the greedy method, tries to find the items with higher ratios, thus it'll generate higher number of items selected. While the DP is not too much interested of the number of items compared to their overall profit after a deep analysis level.

Figure 6 shows a comparison between items selected from both methods.

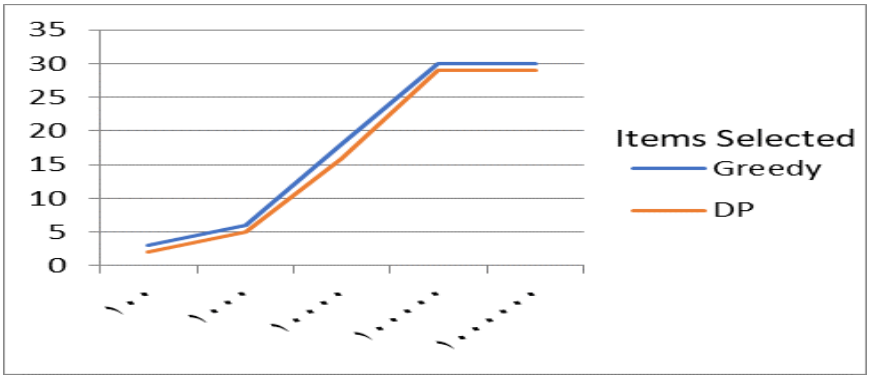

Figure 6: Comparison between Greedy and DP based on items selected

The difference between the two methods is not significant and could be neglected as shown in tables 4 and 5 and Fig. 6. So, it can't be used as a factor to decide which method is better than the other one.

6.4.2 Knapsack Value

Except for a large amount of inputs, the DP method records higher knapsack value than greedy. This is show in Figure 7.

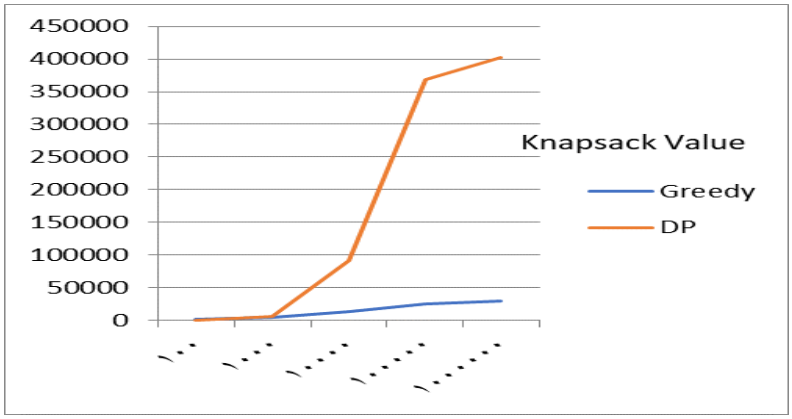

Fig. 7: Comparison between Greedy and DP based on the knapsack value

Vol. 5, No. 02; 2020

ISSN: 2456-3676

As shown in Fig. 7, DP method outperforms greedy in terms of the knapsack value; the difference between both of them is significantly high.

6.4.3 Runtime

Figure 8 summarizes the comparison between the two methods in terms of runtime.

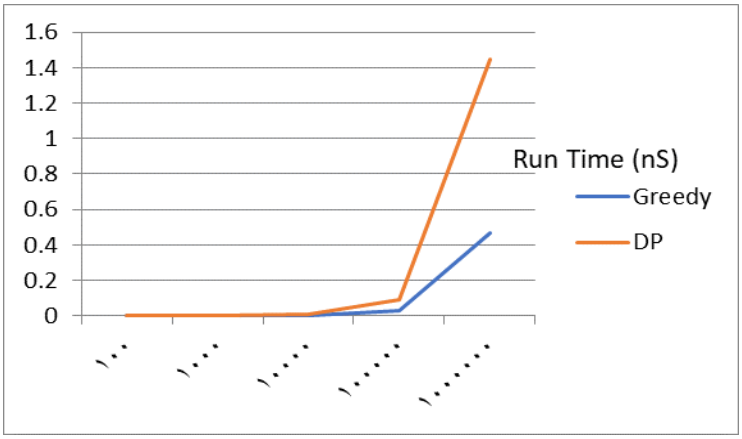

Fig. 8: Comparison between Greedy and DP based on runtime

It's clear from Fig. 8, that DP is much slower than Greedy. Analytically, DP takes $O(nC)$runtime complexity, while Greedy takes$O(nLgn)$ time which is smaller than that of DP especially on higher inputs.

6.4.4 Summary:

Table 6 summarizes the differences between the two algorithms:

Table 6: Overall comparison

	Greedy	*DP*
Runtime	*Higher on small inputs, and better on larger inputs*	*Lower on small inputs and worse on larger inputs*
Space	$O(n)$	$O(nC)$
Items Selected	*More items are selected*	*Less items are selected*
Optimality of Solution	*Less*	*Higher*

Vol. 5, No. 02; 2020

ISSN: 2456-3676

7. Conclusion and Future Work

In this paper, two methods are explained, Greedy and Dynamic-Programming Methods. For that, a Java application has been developed to generate the items and perform the two algorithms and get the results.

According to the results, each method has its advantages and disadvantages; Greedy is has lower runtime and memory requirements but it gives less knapsack value. This indicates that the DP method is near to optimal than the Greedy method.

On the other hand, DP takes more runtime and its memory requirements are higher. Greedy is less expensive than DP either in time or space requirements, whereas DP's cost get higher and higher depending on the size of knapsack and the number of items.

As a future work, further solutions can be compared together and compared to the results we obtained in this paper.

References

Maya Hristakeva and Dipti Shrestha. Solving the 0-1 Knapsack Problem with Genetic Algorithms. Midwest Instruction and Computing Symposium April 16 – 17, University of Minnesota at Morris. 2004.

A. Al-Shaikh, H. Khattab, A. Sharieh and A. Sleit. Resource Utilization in Cloud Computing as an Optimization Problem. International Journal of Advanced Computer Science and Applications (IJACSA). vol. 7, no. 6, pp. 336-342, 2016.

Maya Hristakeva, Dipti Shrestha. Different Approaches to Solve the 0/1 Knapsack Problem. Midwest Instruction and Computing Symposium. 2005.

Your Source for Intel® Product Information, Intel Corp., http://ark.intel.com/products/50178/Intel-Core-i3-380M-Processor-3M-Cache-2_53-GHz cited on 01/12/2014.

Class Random, Java Doc, https://docs.oracle.com/javase/6/docs/api/java/util/Random.html#next%28int%29, cited on 01/12/2014.

Class Collections, Java Doc, http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html, cited on 01/12/2014.

Ágnes Erdősné Németh, László ZSAKÓ. The Place of the Dynamic Programming Concept in the Progression of Contestants' Thinking. Olympiads in Informatics. 2016. Vol. 10, 61–72

Wesley Kerr. Investigation into Knapsack. ,2014