The 0/1 Knapsack Problem

A Comparison of Two Algorithms

Alexander Rosati  
 Computer & Information Science  
 University of Michigan-Dearborn  
 Dearborn MI USA  
 ajrosati@umich.edu

ABSTRACT

In most introductory data structure courses, the 0/1 knapsack problem is discussed. Most courses show how you can solve the problem using dynamic programming. The dynamic programming algorithm is the most well-known algorithm for this problem. However, this algorithm is not very efficient. There is a simpler, efficient algorithm but it approximates the solution. This algorithm uses a greedy approach. For this study, I set out to determine when the dynamic programming algorithm should be used and when the greedy algorithm should be used. This study did not end up covering every possible scenario. However, I did draw some interesting conclusion. If capacity is large compared to the number of items, then the greedy algorithm is more accurate. Since it is always faster, you would want to use it in this scenario. If the capacity is small compared to the number of items, then the greedy algorithm is less accurate. In this scenario, the dynamic programming algorithm is efficient enough and is more accurate than the greedy algorithm so it should be used. Each item K has a ratio rK = vK / wK. For example, item A has a value of 5 and a weight of 2. Thus, it has a ratio of 2.5. Assume we have a problem with N items. Let item A be the item among the N that has the lowest ratio rA. Let item B be the item among the N that has the highest ratio rB. If rA-rB is a small range, then the greedy algorithm is more accurate. If rA-rB is a large range, then the greedy algorithm is less accurate.

CCS CONCEPTS

• Design and analysis of algorithms •

KEYWORDS

greedy, dynamic programming, knapsack, approximation algorithm

1 0/1 Knapsack Problem

Suppose you have N items and each item K has a value vK and a weight wK. Suppose you have a knapsack that can hold weight W. You would like to put items in the knapsack such that the value in the knapsack is maximum. You can put an item in the knapsack only once. Imagine you have the following items:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | 5 | 10 | 2 | 4 | 5 |
| W | 5 | 2 | 9 | 2 | 3 |
|  |  |  |  |  |  |

Imagine you have a knapsack that can hold a weight of 10. Which items would you put in the knapsack to maximize the value in the knapsack? The answer is below:

**Knapsack**

|  |
| --- |
| Item 1: V=5, W=5  Item 2: V=10, W=2  Item 3: V=5, W=3 |

Value in Knapsack = 20

2 The Algorithms

I will assume that the reader is familiar with the dynamic programming algorithm for the knapsack problem. If you are not, you can read about it here [1]. I will assume that the reader is not familiar with the greedy algorithm. The algorithm is simple so I will describe it in plain English. For each item K, calculate a ratio rK = vK / wk, like so:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| V | 3 | 6 | 2 | 7 |
| W | 3 | 2 | 4 | 5 |
| R | 1 | 3 | 0.5 | 1.4 |

Next, sort the items by ratio, like so:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| V | 6 | 7 | 3 | 2 |
| W | 2 | 5 | 3 | 4 |
| R | 3 | 1.4 | 1 | 0.5 |

Assume the capacity of the knapsack is six. Next, we will go through the items in the order shown above. We will try to put each item in the knapsack. If it does not fit, then we will move on to the next item. So, in this instance, we will put the first item in the knapsack. The second item will not fit so we will move on to the third. The third item fits so we put it in the knapsack. We cannot put the fourth item in the knapsack. We end up with the following:

**Knapsack**

|  |
| --- |
| Item 1: V=6, W=2  Item 2: V=3, W=3 |

Value in Knapsack = 9

For more information, visit this resource [1]. Let N be the number of items and let W be the capacity. Then, the time complexity of the dynamic programming algorithm is O(N\*W) [1]. The time complexity of the greedy algorithm is O(NlogN) [1]. It is O(NlogN) in the worst, best, and average case. The previous statement is supported by the following graphs. Note that the worst-case occurs when all items are included, the best-case occurs when none of the items are included, and the average-case occurs when half the items are included.



**Figure 1: Worst Case Run Time of the Greedy Algorithm**



**Figure 2: Best Case Run Time of the Greedy Algorithm**



**Figure 3: Average Case Run Time of the Greedy Algorithm**

The curves in Figures 1, 2, and 3 appear to be of the form A\*NlogN + C.

3 Problem Generation

Generating problems for the algorithms to solve was fairly simple. Several files were randomly generated. For each file, each line is an item. Each line contains a number, a comma, and then another number. The first number is the weight of the item and the second number is the value of the item. A snippet from one of the files is given below:

53,35

26,21

29,21

The line “26,21” above is an item. 26 is the weight of the item and 21 is the value of the item. To generate a problem, a certain number of lines were randomly selected from one of the files. Once a capacity for the knapsack is arbitrarily selected, the problem has been generated.

4 Problem Sizes

Before we get into the analysis, let us define the following problem sizes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Problem Size | Capacity | Num. Items | Value Range | Weight Range |
| 1 | 100 | 10 | 1-38 | 1-25 |
| 2 | 500 | 50 | 1-75 | 1-50 |
| 3 | 1000 | 100 | 1-75 | 1-50 |
| 4 | 2000 | 200 | 1-150 | 1-100 |
| 5 | 5000 | 500 | 1-150 | 1-100 |

If we generated a problem of size 1, then the capacity of the knapsack would be 100 and the number of items would be 10. The weight of each item would be in the range 1-25 and the value of each item would be in the range 1-38. In addition, for each item K, ceil(0.5wK) ≤ vK ≤ ceil(1.5wK). That is, if an item has a weight of 3 then its value can be at most 5 and must be at least 2. Let us label this property “property P”. Property P will apply for a portion of the analysis and then we will remove the property later on. Here is an example of a problem of size 1:

|  |  |
| --- | --- |
| 2,3 | 3,4 |
| 5,4 | 22,15 |
| 15,18 | 20,30 |
| 25,36 | 1,1 |
| 25,14 | 2,2 |

W=100

5 Analysis

Below is a table and following it is an explanation of its contents:

|  |  |
| --- | --- |
| **Problem Size** | 1 |
| **Num. Problems** | 100,000 |
| **Total Time for Greedy** | 196ms |
| **Total Time for DP** | 703ms |
| **Avg Error of Greedy** | 2.3385 |
| **Avg Time for Greedy** | 1960ns |
| **Avg Time for DP** | 7030ns |

**Figure 4: Results for Problem Size 1**

100,000 problems were generated all of size 1. Each algorithm solved each problem. Various things were recorded. The time it took the greedy algorithm to solve all of them was recorded. The time it took the dynamic programming algorithm to solve them all was recorded. The average amount of time it takes the greedy algorithm to solve a problem of this size was calculated. The average amount of time it takes the dynamic programming algorithm to solve a problem of this size was also calculated. How far off the greedy algorithm was for each problem was recorded. This information was used to calculate the average error of the greedy algorithm for this problem size. The following tables are presented with no explanation. The tables speak for themselves.

|  |  |
| --- | --- |
| **Problem Size** | 2 |
| **Num. Problems** | 100,000 |
| **Total Time for Greedy** | 566ms |
| **Total Time for DP** | 21.67s |
| **Avg Error of Greedy** | 1.95 |
| **Avg Time for Greedy** | 5660ns |
| **Avg Time for DP** | 216,700ns |

**Figure 5: Results for Problem Size 2**

|  |  |
| --- | --- |
| **Problem Size** | 3 |
| **Num. Problems** | 100,000 |
| **Total Time for Greedy** | 1.157s |
| **Total Time for DP** | 89.406s |
| **Avg Error of Greedy** | 1.04 |
| **Avg Time for Greedy** | 11,570ns |
| **Avg Time for DP** | 894,060ns |

**Figure 6: Results for Problem Size 3**

|  |  |
| --- | --- |
| **Problem Size** | 4 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 1.289s |
| **Total Time for DP** | 2.75mins |
| **Avg Error of Greedy** | 1.27 |
| **Avg Time for Greedy** | 25,780ns |
| **Avg Time for DP** | 3.3ms |

**Figure 7: Results for Problem Size 4**

|  |  |
| --- | --- |
| **Problem Size** | 5 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 3.31s |
| **Total Time for DP** | 23.27mins |
| **Avg Error of Greedy** | 0.447 |
| **Avg Time for Greedy** | 66,200ns |
| **Avg Time for DP** | 27.92ms |

**Figure 8: Results for Problem Size 5**

One would figure that the average error of the greedy algorithm would increase as the problem size increased. But, if anything, the average error of the greedy algorithm appears to be decreasing as the problem size increases. Meanwhile, the greedy algorithm outperforms the dynamic programming algorithm in terms of time for every problem size. For problem size 5, the greedy algorithm solved all of the problems in ~3 seconds but the dynamic programming algorithm took ~23mins! For each problem size talked about thus far, the greedy algorithm is the far superior algorithm, assuming property P is true. But they do not teach the greedy algorithm in introductory data structure courses. So it can’t perform well in all cases. Under what conditions does the greedy algorithm become unusable? I came up with two hypotheses. The first one was that if we removed property P then the algorithm would become less accurate. The second hypothesis was that if capacity was small compared to the number of items, then the greedy algorithm would become less accurate. With these two hypotheses in mind, I created five new problem sizes. They are listed in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Problem Size | Capacity | Num. Items | Value Range | Weight Range |
| 6 | 100 | 10 | 1-38 | 1-25 |
| 7 | 100 | 50 | 1-75 | 1-50 |
| 8 | 100 | 100 | 1-75 | 1-50 |
| 9 | 100 | 200 | 1-150 | 1-100 |
| 10 | 100 | 500 | 1-150 | 1-100 |

These problem sizes are similar to the ones we saw before. Capacity remains constant here and property P has been removed. We can now have items like the following:

1,75

150,1

1,38

No other changes have been made. The following tables are presented with no explanation because they speak for themselves.

|  |  |
| --- | --- |
| **Problem Size** | 6 |
| **Num. Problems** | 100,000 |
| **Total Time for Greedy** | 188ms |
| **Total Time for DP** | 679ms |
| **Avg Error of Greedy** | 0.9137 |
| **Avg Time for Greedy** | 1880ns |
| **Avg Time for DP** | 6790ns |

**Figure 9: Results of Problem Size 6**

|  |  |
| --- | --- |
| **Problem Size** | 7 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 301ms |
| **Total Time for DP** | 1.372s |
| **Avg Error of Greedy** | 5.51 |
| **Avg Time for Greedy** | 6020ns |
| **Avg Time for DP** | 27,440ns |

**Figure 10: Results of Problem Size 7**

|  |  |
| --- | --- |
| **Problem Size** | 8 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 571ms |
| **Total Time for DP** | 2.68s |
| **Avg Error of Greedy** | 5.2 |
| **Avg Time for Greedy** | 11,420ns |
| **Avg Time for DP** | 53,600ns |

**Figure 11: Results of Problem Size 8**

|  |  |
| --- | --- |
| **Problem Size** | 9 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 1.211s |
| **Total Time for DP** | 4.676s |
| **Avg Error of Greedy** | 10.05 |
| **Avg Time for Greedy** | 24,220ns |
| **Avg Time for DP** | 93,520ns |

**Figure 12: Results for Problem Size 9**

|  |  |
| --- | --- |
| **Problem Size** | 10 |
| **Num. Problems** | 50,000 |
| **Total Time for Greedy** | 3.276s |
| **Total Time for DP** | 20.058s |
| **Avg Error of Greedy** | 8.91 |
| **Avg Time for Greedy** | 65,520ns |
| **Avg Time for DP** | 401,160ns |

**Figure 13: Results for Problem Size 10**

Since capacity is 100 for all the problem sizes, the dynamic programming algorithm performed much better. Its performance is comparable to the greedy algorithm for every problem size except 10. With the exception of problem size 1, the average errors of the greedy algorithm are much higher here. For instance, the only differences between problem size 4 and 9 are that property P was removed and capacity is smaller than number of items rather than larger. Clearly, these two changes had a significant impact. The average error of the greedy algorithm went from 1.27 to 10.05. My hypotheses appear to be correct. The average errors do not really give you a sense of how inaccurate the greedy algorithm is. For a problem of size 9, there’s a 20.5 percent chance that the algorithm is off by 20 or more. For a problem of size 10, there’s a 16.73 percent chance that the algorithm is off by 20 or more. These statistics are both point estimates. The following histograms show what percent of the time the error of the greedy algorithm is within a certain range.



**Figure 14: Error of Greedy Algorithm (Problem Size 9)**



**Figure 15: Error of the Greedy Algorithm (Problem Size 10)**

For problem size 6, the greedy algorithm is clearly superior. However, it is not that clear which algorithm is better for the rest of the problem sizes. If accuracy is desired, then the dynamic programming algorithm should probably be used. If efficiency is desired, then the greedy algorithm should probably be used. (Note: Some of the bars in the above histograms are not visible because they are so small.)

6 Conclusions

More research could be done to determine when the greedy algorithm is accurate and when it is not accurate. Based off my findings, we can conclude that if the capacity is small compared to the number of items then the greedy algorithm is less accurate. We can also conclude that if the range of ratios is large, then the greedy algorithm is less accurate.

7 Code & Data

All of my source code and data is publicly available at https://github.com/AlexanderRosati/CIS405-PROJ. Everything was written in C#. All the data generated by my code was stored in CSV format. This was done so that I could work with the data in Excel and Minitab. Some of the CSVs are quite large. This is because for each problem I stored the weight and value of each item, which items the greedy algorithm included, and which items the dynamic programming algorithm included. For the greedy algorithm, merge sort was used to sort the items. Performance could be improved by using a more advanced algorithm, like quick sort.

REFERENCES

[1] Maya Hristakeva and Dipti Shrestha. 2005. Different Approaches to Solve the 0/1 Knapsack Problem. 15 pages. http://www.micsymposium.org/mics\_2005/ papers/paper102.pdfConference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00