Knapsack Problem with Category Constraint

1. Introduction
The Knapsack Problem is a classic problem in Computer Science and a good example of a problem that is NP-complete and thus, probably cannot be solved in polynomial time. It involves selecting items from a list which have all been prescribed a given weight and value. The solution to the problem would be the selection of items with the highest cumulative value, but whose cumulative weight is below an upper bound. One way to think about the problem is as a collection of objects with values that you are trying to fit into a knapsack that can only hold up to a certain weight. This problem can be extended to many different situations such as packing things in a suitcase, buying items on a budget, and many other types of resource allocation problems.

There are multiple ways to solve this problem, including 2D dynamic programming and branch-and-bound algorithms. In addition, there are several approximation algorithms such as the greedy algorithm solution. However, none of these algorithms take into account what the final collection of items will be. In real-life applications of this problem it is possible to imagine other kinds of constraints on the solution besides the weight threshold. For instance, if you were packing a suitcase and needed to fit in items in order to get maximum value, it could happen that all the most-valued, but least-weight items would be shirts, so the end result would be a collection of shirts, which does not make sense if you’re trying to pack a full wardrobe for a trip. Thus we can imagine a category constraint on the solution to the Knapsack problem. If each item in the list is given a category (e.g., shirts, pants, socks, etc.) then you could add a requirement that the solution try to allocate approximately equal amounts of the weight capacity to each type of item. The way you restrict the categories could vary, but the basic idea is to require that the solution to the problem try to give some amount of weight to each category.

2. Background
The Knapsack Problem has been studied extensively over the years. One important book on the subject, titled Knapsack Problems by Hans Kellerer, Ulrich Pferschy, and David Pisinger, delves into the complexities of the problem, as well as multiple algorithms – some approximate and some more precise – that can be used to solve it (e.g., greedy algorithms, dynamic programming). In addition the book examines extensions of the Knapsack Problem such as a situation in which there are multiple knapsacks and items can only be placed in certain knapsacks.¹ Other study

of added constraints to this problem includes the Multi-Dimensional Knapsack Problem where there other factors in addition to weight, like volume, that have to be considered when trying to fit items into the knapsack. In addition, the multi-dimensional problem has been studied as it specifically applies to grid resource scheduling of applications running on the same machine (in this paper it is titled the “Temporal Knapsack Problem”).

3. Implementation

3.1 DP Solution to Normal Knapsack Problem

I began this project by writing an initial dynamic programming solution to the normal Knapsack Problem. I decided that I wanted to keep track of which items were in my solution, so my 2D matrix of possible solutions ended up requiring a lot of storage. This matrix had the dimensions of number of items by amount of capacity of the knapsack, so when I attempted to store it entirely on the stack, I quickly ran out of space on the stack. This indicated to me that memory space was going to be a significant constraint for this algorithm. As a result, I rewrote the algorithm to dynamically allocate space on the heap instead. In order to test my algorithm for bugs I generated a list of items with random values and weights (within the range of 1 and 100, inclusive). Then, to test the accuracy of the solutions my algorithm generated, I found two dynamic programming solutions online. These algorithms were essentially the same, but I tested with both of them just in case either had a small bug. These solutions can be found in outsideSolution.cpp and outsideSolution2.cpp, and both files include citations to the websites on which I found the code as well as main() functions that I wrote in order to run these solutions on my randomly generated sets of data. Testing my algorithm against these outside solutions verified that my solution reported the correct final value of the items in the knapsack. However, in order to test that my solution included the correct items as well I tested against data published by Florida State University’s Department of Scientific Computing.

3.2 Approximation with Greedy Algorithm

Another way to evaluate the performance of the dynamic programming algorithm I wrote was to compare it to an approximation algorithm. I chose to implement the greedy algorithm solution to this problem. This algorithm first calculates the ratio of value to weight for each item and then sorts the items in decreasing order of this ratio. Then the algorithm goes through each item in sorted order and if that item’s weight fits in the remaining amount of capacity, then the item is added to the solution. I initially implemented a version of this algorithm


titled Greedy-Split, which halted after the first time it encountered an item that did not fit in the remaining capacity. However, I later decided that it was more important to maximize the value of the solution than to shorten the runtime of the algorithm. As a result, I modified the algorithm so that it considered every item from the input, even after it came across some items that did not fit in the remaining capacity.

### 3.3 Addition of Category Constraint

Initially, I considered implementing a category constraint in terms of the number of items chosen from each category, rather than the amount of weight capacity taken up by each category. *Knapsack Problems*, by Hans Kellerer, Ulrich Pferschy, and David Pisinger, describes one variation of the Knapsack Problem that considers the number of items. This problem, called the Multiple-Choice Knapsack Problem, can be solved in pseudopolynomial time. In this problem, each item is given a class and the goal is to select exactly one item out of each class of items and maximize the total value. The specification of the general Knapsack Problem is not suited to constraints on the number of items. The Multiple-Choice variation requires exactly one item of each class, so it gets around this issue. However, for my idea of a category constraint it became clear that it did not make sense to try to require a certain number of items from each category. Because the only variables on which the Knapsack Problem depends are capacity, item value, and item weight, I chose to instead allocate a certain percentage of the capacity to each category. This category constraint is better suited to the Knapsack Problem and allowed me to break up the whole problem into sub-problems based on category.

The solution to the Knapsack Problem with a category constraint that I came up with involved running the dynamic programming algorithm on each subset of items, where the subsets were split up by category, with a capacity for each subset that depended on the percentage of the capacity that was allocated to the corresponding category. Thus, if the capacity were 100 and there were two categories that were given 3% and 7%, then the dynamic programming algorithm would be run on the subset of items from category 1 with a capacity of 3 and on the subset of items from category 2 with a capacity of 7. In order to maximize usage of the full capacity given, my solution kept track of the leftover capacity after the solutions for each category were combined. Then I ran the dynamic programming algorithm on the leftover items with the leftover capacity. Thus, if a category did not use up its whole allocated capacity or if the percentages allocated to each category summed to less than 100%, this algorithm would still attempt to fill the rest of the knapsack in order to maximize the value of the solution.

I approached the addition of the category constraint to the greedy algorithm in the same way as I did for the dynamic programming solution. I ran the greedy algorithm on each subset of items determined by category, with the corresponding

---


category’s allocated capacity, and then combined the solutions for all the categories. Then the algorithm kicked off a final run of the greedy algorithm on just the leftover items and the leftover capacity.

4. Results

4.1 Generating Test Data

In order to evaluate the four algorithms a lot of test datasets were needed. I made the decision to include exactly 100 items in each dataset because there were many variables over which these algorithms could be tested so it made sense to keep this one constant. Each of my generated datasets fell under one of the following categories:

- Random Data with Equal Ranges
- Random Data with High Value Range and Low Weight Range
- Random Data with Low Value Range and High Weight Range
- Parallel Increasing Sorted Data with Equal Ranges
- Parallel Decreasing Sorted Data with Equal Ranges
- Parallel Increasing Sorted Data with High Value Range and Low Weight Range
- Parallel Decreasing Sorted Data with High Value Range and Low Weight Range
- Parallel Increasing Sorted Data with Low Value Range and High Weight Range
- Parallel Decreasing Sorted Data with Low Value Range and High Weight Range
- (increasing value, decreasing weight) Sorted Data with Equal Ranges
- (decreasing value, increasing weight) Sorted Data with Equal Ranges
- (increasing value, decreasing weight) Sorted Data with High Value Range and Low Weight Range
- (decreasing value, increasing weight) Sorted Data with High Value Range and Low Weight Range
- (increasing value, decreasing weight) Sorted Data with Low Value Range and High Weight Range
- (decreasing value, increasing weight) Sorted Data with Low Value Range and High Weight Range

To generate random data over a given range, I used a seeded random number generator in python and gave it 1 as the minimum and the range as the maximum. When generating sorted data over a range, I produced a list of uniformly distributed numbers between 0 and the range (excluding 0, but including the range) and rounded them all to the nearest integer. Thus, when generating parallel increasing sorted data with equal ranges of 100 for value and weight, the (value, weight) pairs for the 100 items would be (1, 1), (2, 2), ..., (100, 100).

For each category, I ended up with 10 data points for each algorithm, where each point was a (weight, value) pair from the solution returned by the algorithm. For the categories with equal ranges, the data points were based upon generated data over the ranges of 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. For the categories with high and low ranges, the data points were based upon data generated with low ranges of 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 and high ranges of 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. Thus, the high range was always 10 times the low range for a given data point. In addition, for all of the categories with random data, I generated 10 datasets for each range possibility and then later averaged the solutions over those 10 datasets for get the final (weight, value) pair for that range.

In order to reproduce the generation of these datasets, which I did not include in the submission of code, one can go into the generateData directory and run the command `python generate.py`. All of the test datasets will be created and
placed in the directory generateData/test (which must already exist before the command is run).

In order to run all four algorithms on each of these datasets, one can compile and run knapsack.cpp and withConstraint/knapsackConstraint.cpp, which both take one command line argument for the capacity. If the capacity given to knapsack.cpp is 10, 50, 100, or 500, this code will read in all of the test datasets and write the results to the terminal and into text files in the directory generateData/0010, generateData/0050, generateData/0100, or generateData/0500. If the capacity given to withConstraint/knapsackConstraint.cpp is 10, 50, 100, or 500, this code will read in all of the test datasets and write the results based on the test category percentages (see Table 1) to the terminal and into text files in the directory generateData/0010/cat, generateData/0050/cat, generateData/0100/cat, or generateData/0500/cat. If any other capacity is given to either knapsack.cpp or knapsackConstraint.cpp, then the solutions given by the algorithms will be printed in the terminal, but not written to any files.

I later generated the results of running the Knapsack Problem algorithm with the category constraint on the same test datasets, but with the different category allocations described in Table 1. These test datasets all had the categories determined randomly for each item. The results of running knapsackConstraint.cpp on the same test datasets with different category allocations, but all with the capacity 100, can be found in the directories generateData/0100/c01, generateData/0100/c02, and generateData/0100/c03.

<table>
<thead>
<tr>
<th>Category Constraint Sets</th>
<th>Category</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic test allocation (results found in directories named &quot;cat&quot;): Category Set 0</td>
<td>Category 1</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Category 2</td>
<td>16%</td>
</tr>
<tr>
<td></td>
<td>Category 3</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>Category 4</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td>Category 5</td>
<td>10%</td>
</tr>
<tr>
<td>Equal priority to two categories (results found in directory named &quot;c01&quot;): Category Set 1</td>
<td>Category 1</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 2</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 3</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 4</td>
<td>45%</td>
</tr>
<tr>
<td></td>
<td>Category 5</td>
<td>45%</td>
</tr>
<tr>
<td>Priority given to one category (results found in directory named &quot;c02&quot;): Category Set 2</td>
<td>Category 1</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 2</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 3</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 4</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Category 5</td>
<td>90%</td>
</tr>
<tr>
<td>Small allocation for every category (results found in directory named &quot;c03&quot;): Category Set 3</td>
<td>Category 1</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Category 2</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Category 3</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Category 4</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Category 5</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 1: Percentage of Capacity Allocated to the 5 categories
4.2 Using Plotly to Visualize Results

I used the Python graphing library, Plotly, in order to generate line graphs of the solutions given by each algorithm for each category of generated data. There are 10 data points connected by a colored line for each algorithm in each graph. The html files for all these graphs can be found in graphs/0010, graphs/0050, graphs/0100, and graphs/0500 where the directory name signifies what capacity was used to generate the data displayed. By hovering over a data point on one of these graphs, the exact x-axis and y-axis values can be seen as well as a letter label that is a letter from “a” through “j” that indicates where it is in the order of data points, where “a” represents the first in the list. For instance in the category “Random Data with Equal Ranges,” the ranges 100, 200,…, 1000 correspond to the points a, b,…, j.

The graphs showing the differences in performance of the dynamic programming solution for different sets of percentages allocated to the five categories can be found in the directory graphs/c100.

4.3 Effect of Category Constraint on Solution Value and Weight

One interesting trend that can be observed from looking at the graphs with random data (see Figures 1 through 4) is that the data points tend to be distributed from the lower-right hand corner to the upper left-hand corner. By hovering over the data points we can see that the points in the lower right-hand corner, which have weights closest to the possible capacity, are labeled with an “a” which means that the data that the algorithms were run on had the lowest range at those points. Therefore, it makes sense that those points would be the lowest, as the values in the datasets they were run on were lowest, but it is interesting that they were the right-most on the graph because that means that when the data is generated randomly over larger ranges, the algorithms solving the Knapsack Problem find solutions with weights that are farther from the capacity of the knapsack. This could be because when there is a higher weight range, then more items will have higher weights that will be hard to fit into the knapsack. In addition, if a solution returned by the dynamic programming algorithm has a weight of 88 when the capacity is 100, then that must mean that there were no items in the input that had a weight less than or equal to 12, because if such an item had existed, it would have definitely been added to that solution.
**Figure 1:** Graph for capacity 10

**Figure 2:** Graph for capacity 50
It is also important note that there was not a clear trend within any of the other graphs, which were produced from datasets where the weight and value were...
sorted. Most graphs displayed data that jumped around a lot. However, when comparing the results across different graphs it can be seen that the best value results came from the datasets that had high value ranges and low weight ranges and where the values and weights were sorted in opposite directions. Conversely, the worst value results came from the datasets where the values and weights were sorted in the same direction, but where there was a low value range and a high weight range. Figures 5 and 6 show the results from these datasets after the four algorithms were run with a capacity of 500.

![Graph for capacity 500](image1)

**Figure 5: Graph for capacity 500**

![Graph for capacity 500](image2)

**Figure 6: Graph for capacity 500**
Another thing to note when observing the graphs produced from sorted datasets is the way that the relationship between the solutions from the normal Knapsack Problem are closely related and the solutions from the Knapsack Problem with the category constraint are also closely related when the graph is based on data where the values and weights are sorted in opposite directions. When the values and weights are sorted in parallel, there is less of a clear difference between the solutions with or without the category constraint. However, we can see in Figure 5 that the orange and blue lines follow the same pattern and the red and green lines also follow their own same pattern. In many of the graphs this is so pronounced that the orange and blue lines overlap and the red and green lines also overlap. This can be observed in Figures 7 and 9, where there appears to be only two lines because the blue line is hidden behind the orange and the green line is hidden behind the red (Figure 8 shows the same graph as Figure 7, but with the orange and red lines “turned off” to show that the green and blue lines were behind them).

**Figure 7: Graph for capacity 500**

**Figure 8: Graph for capacity 500**
4.4 Effects of Different Percentage Distributions for Categories

To test out the effects of allocating different percentages of the capacity to five different categories, I created graphs visualizing the performance of the dynamic programming with categories algorithm on the four category allocations displayed in Table 1. In these graphs I also included the results from the dynamic programming algorithm that did not account for categories. Because the dynamic programming with categories algorithm always attempted to fit the leftover items in the leftover capacity after the sub-problems were solved for each category, I expected the category allocation that gave each category a very small allocation to each category (Category Set 3) to end up returning a very similar solution to the normal dynamic programming algorithm that ignores categories. Conversely, I expected the results of Category Set 1 and Category Set 2 to have a lower value than the others because they attempted to give priority to certain categories, which may have caused certain items to be chosen for the solution that would not have been chosen in the normal dynamic programming algorithm. Figures 10, 11, and 14 show the three graphs I produced to test these theories. As we can see from Figure 10, the blue and purple lines run very close to each other, as I expected. In addition, Figures 12 and 13 isolate the lines for the “no categories” and Category Set 3 lines from the graph from Figure 11 in order to show that they are identical. Similarly, Figures 15 and 16 do the same thing for the graph from Figure 14.
Figure 10

Figure 11
It is also important to note that for all of these graphs, the blue line always shows the highest possible value for each data point. If you open up one of the html...
files for these graphs and hover over the points you will see that for every point labeled 'a' the blue point is the highest with respect to the value axis and the same can be said for each subsequent letter. This is because the normal dynamic programming algorithm will always find the solution with the greatest value, disregarding categories. For this reason, the category constraint can never improve on the value of the solution to the Knapsack Problem, though it may be able to make the Knapsack Problem more applicable to certain real-world applications of this problem.

5. Future Work

Due to the high number of factors that go into testing the results of these algorithms, further study could definitely be done. I varied the ranges of the test datasets, the capacity input, the category allocations, and the ways that the values and weights of the items were generated, but there are still other ways to vary these factors that I did not get to examine. In addition, I did not use different total numbers of items. I also did not examine the effects of determining the categories of the different items input to the problem in a way other than random selection. For these reasons, a lot more testing of the algorithms I wrote could be done to get a full sense of how they perform in every situation.

Other future work that could be done in relation to this problem would be to find a real world application of Knapsack Problem with the category constraint and generate data for what the input would be and what the output should be. Then it would be possible to examine if I made the right decisions in designing the algorithm that I wrote. For instance, it would be interesting to know how best to handle the leftover capacity after the sub-problems for each of the categories are solved. In addition, it could turn out that there is a better way to approach this problem rather than splitting it into sub-problems based on category.

Acknowledgements

I would like to thank Professor Dana Angluin for all her advice and support throughout this semester. Our weekly meetings were so helpful in guiding me through the course of this project.

References


