The Knapsack problem: Optimization Approaches

Shawn Yap

School of Information Technology  
734 E North 14th St. 79601 Abilene, TX  
Apt. D  
+1(325)232-2559

scy12a@acu.edu

Aldo Anaya

School of Information Technology

1207 Pueblo Rd  
Alamo, TX 78516  
+1(956)429-2598

axa13a@acu.edu

**ABSTRACT**

In this paper, we present several attempted algorithms to further optimize the speed of the classic Knapsack problem. The Knapsack problem is a combinatorial problem that searches for the sum of the highest values from a list of items while constrained within the cost of the knapsack is able to carry. The optimizations that we have applied on the knapsack problems are some greedy approximation algorithm which consist of four different sorts to find the *upper bound* (The highest value obtainable within the cost limit determined from the list of given set of items) and the *lower bound* (The lowest values obtainable within the cost limit determined from the list of given set of items). Besides greedy algorithms, we have also developed a state space that consists of all possible combination obtainable from the given set of items in the list. The primary goal of this paper is go explore and present a comparative study of an exhaustive search, greedy algorithm and other ways. This paper provides insights of the complexity of each algorithm in terms of time and memory.

**CCS Concepts**

• **Symbolic and algebraic manipulation➝ Optimization Algorithm; Combinational Algorithm;**

**Keywords**

Combinatorial optimization; the Knapsack problem;

# INTRODUCTION

The Knapsack problem is a classical problem that searches the highest combinational values from a list of items that consist of cost and values. The knapsack problem is a decision problem such that given a set of items that each consists of a cost and a value, find the highest possible value from each of the item while remaining in the range of the cost that the problem is being constrained. Which means that the cost may only remain less than or equal to the constrained cost limit. The decision form of the Knapsack problem is a NP-complete problem such that a precise solution for a huge input is nearly practically impossible to obtain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference’10*, Month 1–2, 2010, City, State, Country.

Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

DOI: http://dx.doi.org/10.1145/12345.67890

# The Knapsack Problem

In this section, we will define the Knapsack problem, then we will provide the details of the attempted approaches in which we have tried to get the most optimal results.

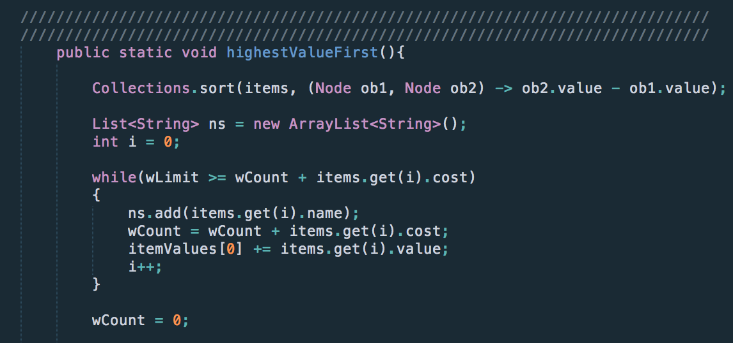
## Method Descriptions

### Greedy Solutions

The greedy approach are the primary methods that we have used to obtain the upper bound and the lower bound of the Knapsack problem. There are four type of sorts used to sort the list of items: Sort by highest-value first, sort by lowest-cost first, sort by highest ratio first, and partial knapsack. However, since it is impossible to determine if the result returned from one of the approach is the best one, all four approaches mentioned above were used to obtain the most optimal result with the cost and value being at its maximum.

#### Sort by Highest-Value First

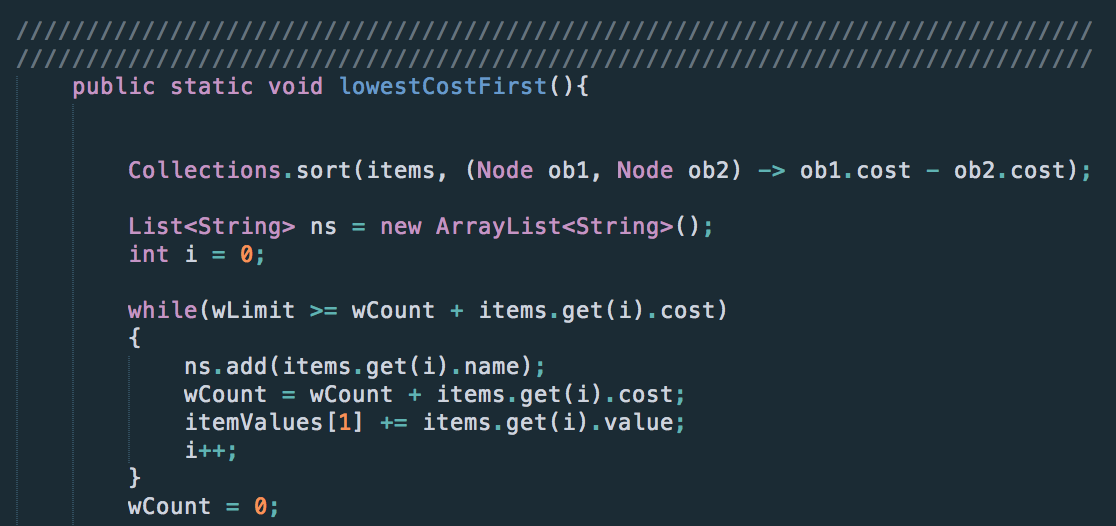
*Code Snippet:*



This solution first sorts the given set of items into a collection of highest values first and then sums the values from the top of the list with the highest values while keeping the cost within the range of the limit that the problem is constrained to. Although this algorithm prioritizes the highest values in the list first, there were cases where a more optimal answer was found by looking at the items with lower values with lower cost.

#### Sort by Lowest-cost First.

*Code Snippet:*



This solution sorts the given list of items from the lowest cost to the highest. By doing so, we are able to keep the cost the lowest for as many iterations as possible, allowing more space for more items to be included in the knapsack. Although this method would work for a list of items with low cost and high values, this method does not produce the optimal answer when the list of items contains both low cost and low values. There were cases where a single item would have a significantly higher value than the result obtained from this greedy approach.

#### Sort by highest ratio first

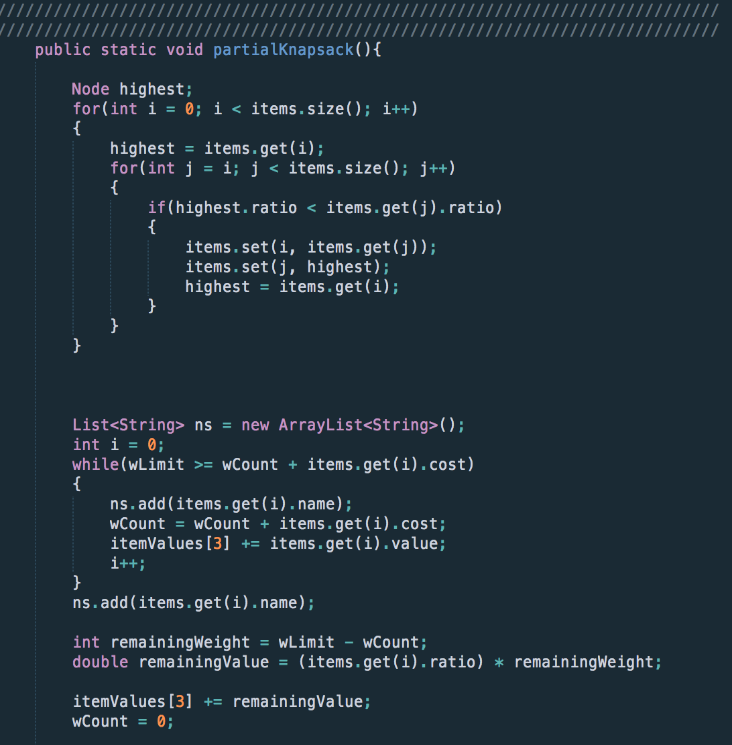
*Code Snippet:*



This greedy approach sorts the given inputs by the highest ratio first allowing the algorithm to grab the item with the highest cost to value ratio.

#### Partial Knapsack

*Code Snippet:*



This greedy approach first sorts the given list of inputs by the highest cost to value ratio then grabs as many items as possible until the cost is near or equal to the limit that the problem is being constrained to. If the cost is less the constrained limit then we will continue to add part of the cost and value from the next item until we have the cost at its maximum.

### State Space Search

The state space is an approach that we have used to obtain every possible combination of the given list of items. Thus, allowing us to find the best possible combinations of the items that has its cost at its maximum and the values at its highest.

#### “Dumb” Exhaustive Search

Using a “dumb” exhaustive search, we looked at every possible leaf node of the tree. While looking through each leaf, we calculated the total cost and total value. The last row in the tree consists of all possible combination of items so if we look for the highest total value within the cost limit, clearly the best solution will be found.

#### “Smart” Search

The way we built the smart search will probably be a little different from how most people did it and it involves the way the tree was built. Initially, the tree was built with each node being a certain combination of items. However, with the smart search, the tree was built so that nodes that had already gone over capacity would receive a flag that would indicate that no calculation was needed to be done. This decreased would decrease the time taken to search through leaf nodes for the best solution to the knapsack problem.

# Additional Optimizations and Results

In this section, we will discuss the improved speed on some other method that we have tried to shortened the search time.

## Arrays of Strings Versus Nodes

As mentioned in section 2, the entire state-space was planned to be built with the linked-list nodes approach. However, there were issues that we encountered from that approach: memory and time issues.

By following this approach, a given set of input items with the size of N, produces a tree with the size of . An enormous amount of memory would be consumed when N > 18. Consider a regular node in a linked list, a regular node that contained a left and a right pointer takes up about 8 bytes. If we include a string and 2 integers into the node, it all adds up to (number of character in the string \* 2 + 2 + 2 + 8) about 14 bytes per node which totals up to about 469762020 bytes = 0.46976202 GB when N = 24.

No calculations were included yet on what had mentioned above. Considering the function we have built has a complexity of O(), time is the other issue that this approach produces.

### Alternative to Trees with Nodes in Linked-list

*Code Snippet:*



The approach that we have taken in alternative to build a tree which consist of linked-list nodes, we chose to build the tree using an array that consist of only ‘chars’ in each index of the array. Each index of the array represents different nodes in a visualized form of the tree. We are able to move to the left sub-tree by determining if the index is odd (left branch) or even (right branch) as shown in the code included below:

By doing so, it significantly reduced the amount of time taken to compile as compared to building a tree with nodes.

### Calculations After Tree is Built

To shortened the time spent on calculating the cost and values in each of the sub-trees, we’ve decided to perform the calculations after the tree is successfully build. From that, we are able to only look at the leaf indexes of the state-space tree which consists of all combinations since the root index, reducing more than half of the calculations needed to be done as compared to building the tree while calculating at every index.

# Results

Table 1. Results of All Optimizations

|  |  |  |
| --- | --- | --- |
| **Optimization Methods** | **Time taken** | **Memory issue** |
| “Dumb” Exhaustive Search | 25(seconds) | None |
| Greedy | < 1 seconds | None |
| Tree with Nodes (Including Search) | Over a minute | Compile Error; insufficient  ram |
| Tree with Array (Including Search) | 22.43(seconds) | None |

As we can see from Table 1, the time taken to load trees with nodes was significantly longer than the tree with array. The Node class takes up quite a bit of memory, especially when making a tree with Nodes. The Greedy method is seen to have considerably fast search times. The reason being that no tree need be built nor does it search through a large state space.

# Future Optimization

Due to the time constraint of this Knapsack research project, there were optimization ideas that we could not implement in time but thought that it might be worth the time to dig-deeper into. To further optimize the huge memory usage used in order to build a -2) sized tree, while building the tree, we could cache the two most recent rows as row A and row B then build the next row using row B and then assign the new row as row A. Continue to iterate until all combinations are found. By doing so, we are not wasting more space than the program needs to. Therefore, opening up more space for larger lists of items. Another method that would prove to be even more efficient in terms of memory usage would be to delete one row as we are building the other row. For example, as we are building row B using the information from row A, we are also deleting used nodes from row A. This would greatly shorten the cost of memory by 50%. However, the time taken to run the program would significantly increase as well.