Louis Zhang
Yusuf Khan
CS 170 - Fall 2017
2 May 2017

Project Writeup

1. **Phase 1 Instances**

   For phase one, the basis for our difficult inputs was splitting up all the classes into certain groups. Each class of items can only be placed into the sack with other items from its class. Thus, an algorithm must select the most optimal group. Many groups with the highest sack value go over the weight limit, or cost limit or both. Thus greedy solutions that select items individually will end up choosing a class that is not optimal, and thus will not be able to fill the sack to its ideal amount. Each test file then had a few different complexities. Our second file had more groups than the first; however the second set of groups was almost identical to the first. Thus many more optimal individual items constrain the algorithm to a class that is not optimal. In the third test file, there are more groups and more constraints. The groups are broken into two large groups, and each of those is split into 5 individual groups. One must choose from the larger group, then two of the smaller groups. Again, there are many individual items that seem appealing, which lead to suboptimal groups.

2. **Strategy**

   We first attempted to implement a dynamic programming solution similar to the Knapsack without repetition problem in the textbook. This solution attempted to create a three-dimensional table instead, due to the added dimension of having only M dollars to spend. It seemed to work well, but its space and time complexity became too large for the official test instances provided. As a result, we turned to a greedy solution. We kept the constraint map from the DP solution, which created a dictionary, given the constraint list, that mapped every class to a list of all of the other classes that it was incompatible with. This made it simple to check whether or not adding an item to the current GargSack violated a constraint.

   Our greedy solution begins by sorting the list of items by a heuristic in decreasing order: the item's resale value subtracted by its cost, all divided by the item's weight. With this, the list of items is arranged in order from highest to lowest net gain per unit weight. We then parsed the list in this order and added each item in the list to the GargSack if it satisfied all of the following conditions: its weight was not more than the remaining weight capacity, its cost was less than the remaining spending money, and it did not violate any of the constraints given. After parsing through the entire list of items while updating the remaining weight and money in each iteration, we return the resulting GargSack.

3. **Runtime Analysis and Performance Bounds**

   The runtime of our algorithm can be seperated into 3 parts: sorting the N items, creating the constraint map, and checking whether or not current item violates a constraint. We assume the worst case of our sort using either Merge Sort or Quicksort, which gives a runtime at O(N log(N)). Because we know that each of the C constraint lists has a maximum length of 199,999 as defined in the spec, we can hash each item into a map at O(199, 999 * C) with accessing each class in the map being constant time. The worst case for checking whether or not an item violates a constraint, however, is when the GargSack holds N items and we had to check at every step, providing a worst case O(N^2) runtime. Thus the total runtime is bounded by O(N^2) + O(N log(N)) + O(199, 999 * C)