Spring 2017 CS170 Algorithms Final Project Write-Up

Team:
Gan Tu (26070763)
Chris Hinrichs (25159702)

Phase I: Input File Generation Strategy

We wrote a script called "create.py" to automatically generate the input file. We added algorithmic logic to randomly generate each input variables, but generate files strictly according to the following 3 boundary decision cases: (1) when item_1 is more costly than item_2 and item_1 has a higher profit margin than item_2, but item_1 is also heavier weight than item_2 – this is the situation of higher cost for higher profit; (2) when item_1 is less costly than item_2 and both items have same profits, but item_1 is heavier than item_2 – this is the situation of a trade of using more weight capacity or using more money; (3) completely random files. We implemented this logic because it will therefore require users to think more about what to do at each trade off positions. Also, instead of having one item conflicting with a lot of other items, we made it that each constraint contains a small amount of classes only, but a lot more constraints. Therefore, we won't make the problem actually easier by accidently reducing the choice space. Lastly, we made the properties of each other items really close and the value/cost relatively small compared to the capacity limits, so it will take users longer time to solve the problem.

Phase II: Algorithms

1. Data Structures

We made several data structures up front to make the solver code run easy.

a. We implemented a bag class in "bag.py," where the instance itself contains all the properties resembling a bag, like total weight, weight limit, total value, value limit, constraints, items contained, etc. We implemented the logic to check if we can take another item in the bag as well here.
b. We implemented an item class in "item.py," where it contains the properties of each item (i.e. name, class number, weight, etc.). We also calculate useful ratios and heuristics that will be used later here, so other methods can directly use these values without re-computing them.
c. We implemented a constraint class in "constraint.py" that helps check if two items have conflicts with each other. These relationships are pre-processed and saved in a

hashmap, so we later don't have to re-run the loop to check constriants. It helps to speedup the algorithm mlater.

d. We implemented an auto-grader class in "autograder.py" that does sanity check to see if our algorithm later actually generates a valid output solution given an input file.
e. We implemented a "grade.py" file to quickly calculate the score of a solution. This is used to compare solutions or see how an algorithm is doing.
f. We implemented a UnitTest to test unit functions of different structures. We implemented a python file to generate various bash scripts to automatically run specific code needed at different stages of development

2. Algorithms

We made a solver class in "solver.py" with various algorithms. We attempted many different algorithms, and they are all encapsulated as a separate class extending the "Sovler" class, which takes care of all the initializing, input, and output work.

Algorithm I: Simple Greedy Algorithm:

We sort the items by their **(resale - cost) / (weight)** ratio in a list. We keep taking the item from the list with the highest ratio, which does not conflict with any existing items in the bag, and stop when the bag is full. Then, we add the solution to a list, and randomly delete some items from the bag, and added back the items that didn't get used due to conflicts. We repeat this process until a certain number of solutions are found, or the list of items is empty. We return the solution with the max total value (profit + remaining money).

Algorithm II: Hybrid Greedy Algorithm:

This is a similar idea like Algorithm I, but we run multiple runs of the greedy algorithms but based on different priorities value (profit/cost, profit/weight, profit/(cost+weight)), which is used for sorting the order of items to take. We also run two slightly randomized algorithms that reorder the list obtained by the above priorities. We choose the solution with best value among all five greedy algorithms.

Algorithm III: Approximation Knapsack:

We implemented the approximation algorithm in the book for Knapsack by scaling the profits for each item by an epsilon value. (For more, refer to the book). We tried various versions of this constant, and chose the one that works the best based on its effects on the runtime and the optimality of the solution. We randomly selected a set of items using greedy algorithm above so that none of them conflict with each other. Then, we formulated it as a Knapsack problem and used **Google's python Knapsack solver** to solve it. ("from ortools.algorithms import pywrapknapsack_solver).

Algorithm IV: Linear Programming:
We formulated the problem entirely using linear programming. Our objective value is to maximize the sum of profit of chosen items. We set each item as a binary variable, and we set constraints to be (1) sum of the weights of chosen items < weight limit; (2) sum of the cost of chosen items < money limit. We set each class number as a binary variable. We set the constraints that (1) if any item of that class is chosen, this class variable has to be 1; (2) the sum of class binary values for each constraint <=1 – this ensures that there is at most one class number of items from each constraint is chosen, so as to incorporate the constraint factor to the integer linear programming problem. We then used **Gurobi Optimizer** (recommended by Google optimizer as the fastest solver out there) to solve this integer linear programming problem. We chose the items whose binary values are 1 in the solution returned. We later found out that it is much more efficient and we get better results by setting the objective function to its dual. We also experimented with different parameters of the optimizer, tuned the model a lot, and set a time out for the algorithm. This is our MAIN algorithm solver we landed on later, since it gets pretty good results.

Algorithm V: Scaled Linear Programming:
It is pretty much the same as the last algorithm, but we scaled the profits of each item to speed up the process and runtime in the same way we did for algorithm 2. We used this algorithm when certain input takes too long to solve or takes too much memories and computational resources.

3. Improvements

We coded several files to help improve out solution. First, the "improve.py" file takes a solution file and tries to put each unchosen item to the bag again to see if there are still items that we can put to the bag that make our total value bigger but without breaking the constraints and limits. The order of the items are sorted by profit / (k1* cost + k2*weight) ratio, where k1 and k2 are chosen either randomly or handpicked. Second, our "merge.py" file merges the solution of two output files we've generated in the past, and then merge them according to the greedy algorithm described above, attempting to get a better result.

Runtime:

Hard 21 Input Files: we set a time out of 1600 seconds for each input file, so it's about 30 minutes per input file, and 10 hours for all 21 inputs.

Extra Credit Files: it turns out that most extra credits files are easy inputs and can be solved with an optimal solution within 40 – 100 seconds.