CS170–Spring 2017 — Final Project Writeup

Miles Daudistel (3032346669) and Jen-Yeu Wang (26739059)

To solve the **NP**-hard problem PickItems, we designed two candidate algorithms using different approaches. The first was a greedy algorithm, and the second was a genetic algorithm. We created the greedy algorithm based on what we learned in class, but we wanted to take a step further and look at other algorithms with potentially better accuracy and running time. The most promising source was a paper by Maya Hristakeva and Dipti Shrestha of Simpson College titled "Solving the 0-1 Knapsack Problem with Genetic Algorithms." While the PickItems problem differs slightly from the Knapsack problem, we wanted to test how a genetic algorithm would fare against a greedy algorithm given the additional constraints of the PickItems problem.

First, let's walk through our implementation of the greedy algorithm. PickItems is unique in that the constraints between classes has to be addressed first. To formulate the greedy heuristic, we first create a pool of classes that are not constrained by each other. Multiple heuristics were used to create this pool, and the heuristic with the highest profit was selected each time. Most of these class heuristics involved assigning a heuristic value to each item in the class, such as total profit: $resale - cost$, profit margin: $\frac{resale}{cost}$, and profit density: $\frac{resale-cost}{weight}$. After these values were calculated for each item, they were both summed, and averaged to give 2 heuristic values to each class per item heuristic. In addition, we used class heuristics based on the which ones had the least number of constraints, and also randomly. After assigning these heuristics and selecting pools of classes, we selected items greedily based on the above mentioned total profit, profit margin, and profit density. In the end, we had 8 class heuristics, and 6 item heuristics for each class, for a total of 48 possible solutions to each problem. We then selected the best from this pool of solutions.

While the greedy algorithm only yields approximate solutions to the PickItems problem, it does so in a significantly faster running time than some more accurate approaches like dynamic programming. Dynamic algorithms have to be ruled out because they cannot compute solutions in polynomial time. Solving PickItems in exponential time is not feasible given the project's time constraints and the number of items in the input files. The greedy approach takes polynomial time because we create adjacency lists in $\mathcal{O}(n^2)$ time.

A genetic algorithm involves several steps. First, a population of random individual solutions are generated. We call these individual solutions "chromosomes" that encode information. Items that are picked are represented with the value 1, and items that aren't picked are represented with the value 0. Then, these chromosomes are bred together and crossed over like in biology. A random point is chosen and both chromosomes are split at that point. The tails are then exchanged between the chromosomes to generate children chromosomes. To simulate biological evolution further, a mutation rate is chosen that determines whether a certain place on the chromosome is switched from 0 to 1 or 1 to 0. Individuals with the highest fitness from the previous generation are preserved to the new generation (a factor called "elitism"). By repeating this over many generations, the hope is that evolution will yield the fittest solution.

The biggest question we ran into when creating the genetic algorithm was what the parameters should be (i.e. population size, number of generations, number of elite individuals, mutation rate, etc.). The literature is unclear as to the optimal parameters, probably due to the breadth of problems it can be applied to. In addition, a huge factor of genetic algorithms is the randomness that allows potentially fit but currently unfit solutions be bred. Therefore, we realized that genetic homogeny could occur if the algorithm is heavily biased toward the fittest solutions, which is bad for genetic diversity and evolution as a whole. Since the parameters are so ambiguous, we did a lot of experimenting and tweaking.

The input files we submitted contained the maximum amount of data possible, to ensure a long computation time. We also kept the names of the items as small as possible so as to be able to include more items. However, we incorrectly believed that fewer constraints would result in a more difficult input file, since the solution space would be larger due to the maximization of compatible item sets. This led us to create files where each item was in its own class. With no class restraints, our input files boiled down to a 0-1 knapsack problem constrained only by the amount of weight we could carry, and the amount of money we could spend, which probably resulted in a relatively simple input file.