CS170 Spring 2017 — Project Writeup

Brent H Yi
brentyi@berkeley.edu
SID: 26858652

William M Lu
wml@berkeley.edu
SID: 26105385

---

1 Algorithm: Main Idea

Presented with the NP-hard problem PickItems, we propose a fast greedy solver using a statistically pruned set of heuristic functions, which are procedurally generated as functions of 12 hand-selected features. For each input file, we consider all pruned heuristic functions' outputs and retain the list of items with the maximum sum of resale values and remaining principal.

Our algorithm is built on a foundation consisted of three core principles:

1 – Deterministic greedy algorithms with constant-time heuristic functions are simple, but run extremely *quickly* and *efficiently* in the context of runtime and space complexity.

2 – Solutions to separate input files can be best generated by using heuristics that compare select item properties, such as the weight, spending allotments, and class incompatibility constraints. Each property varies in importance based on the specific input's content.

3 – Given a set of heuristic functions, we can leverage the law of large numbers to statistically rank the utility of each function. The relative strength of a heuristic can be roughly extrapolated by examining its average performance over a limited number of inputs.

From these principles, we decided to develop a large variety of usable heuristic functions. By generating one output per heuristic for each file and keeping only those that generated the most favorable item combinations, our algorithm is able effectively tackle extremely different input files, using heuristics that emphasize certain favorable item properties over others. Expanding our collection of heuristic functions, the algorithm starts by recursively generating 400 linear and multiplicative combinations from a dozen handselected heuristics and item features (profit-to-weight ratio, profit-to-cost ratio, constraint count, etc).
Running every generated heuristics on the 21 hard input files would yield better results but would have taken over 10 hours to run on the pool of 980 extra credit problems. Therefore, we rely on heuristic pruning to solve larger datasets within our project's time limitations. Of our 400 generated heuristics, we used SET COVER to select the fewest number of functions that effectively solved the largest number of problems within a threshold of the maximum score achieved of all heuristics for each problem. Using 50 general extra credit input files for training and the 21 hard input files for validation, we found that roughly a dozen heuristic functions could be used instead of the full set of 400 without a significant drop in the output's score.

2 Algorithm: Run-time Analysis

Given P pounds, M dollars, N items, and C constraints in a problem, running a greedy algorithm with K constant-time heuristic functions has a runtime consisted of the following steps:

- $O(1)$ – File read: the upper bound on file read times is capped by the file size, which can be no more than 4 megabytes.
- $O(N + C * N)$ – Setup: to speed up our algorithm, we memoize some commonly used data. A list of valid items is constructed in N time, as is a map that associates a class to a list of relevant constraint list pointers (worst case: C constraints multiplied by N classes).
- $O(K * N \log N)$ – Sorting every item with a heuristic function as the key. This is performed once per heuristic function using Python's default sorting function.
- $O(K * N)$ – Iterating through all N items in the sorted order, adding the item to our set of items to keep if class compatibility constraints permit. This is performed once per heuristic function.

From this list, we can see that our asymptotic runtime evaluates to $O(K * N \log N + C * N)$. This means that our algorithm is largely limited by the $O(K * N \log N)$ term.

To evaluate speedup after pruning, consider the number of heuristic functions before and after

pruning. $K = 400$ initially, but is then reduced to around 12. The improvement to the dominating term is $\frac{400-12}{400} = 97\%$!

3 Input Files

To generate our input files, we used a Python script that started by generating a large number of random items and constraints. To increase difficulty, our strategy:

- Limits the number of classes per constraint. Increasing the number of incompatible classes decreases the number of possible solutions. More possible solutions = wider range of outputs from different algorithms.
- Limits the range of weight, cost, and resale values. By keeping them similar but not overtly identical, the relative qualities of each item from naively analyzing the weight, cost, or resale becomes more difficult to discern. Complex inter-class relationships and constraints therefore become more important.
- Adds a handful of "trap" items designed specially to fool simple greedy algorithms. These can appear very attractive from simple features like profit-to-weight and profit-to-cost ratios, but in reality result in suboptimal solutions. If constraints are not considered, for example, adding any one of these items will automatically eliminate the possibility of adding any non-trap items.