CS 154 Final Project

PICK_ITEMS: An Approximation for NP-COMPLETE problems

Victor Phung

I. ABSTRACT

This report examines an NP-Complete problem PICK_ITEMS, which is a variation of the multi-objective knapsack problem. To find a solution, a polynomial time approximation solution is constructed using greedy selection, memoization, and randomization. Finally, we discuss further areas of fine-tuning such as hill-climbing algorithms and using alternative criteria for selecting the knapsack.

II. INTRODUCTION

Background

NP is a category of problems in computer science complexity theory. The term refers to the set of decision problems that can be solved in polynomial time by a non-deterministic Turing Machine. More importantly, it has the key attribute that correct solutions to NP problems can be efficiently verified. The category of complexity problems are as follows[1]:

- P: problems that can be solved in polynomial time
- NP: problems that can be *verified* in polynomial time.
- NP-hard: problems to which all other problems in NP reduce. In other words, if one could efficiently solve an NP-hard problem, one could efficiently solve any problem in NP.

An NP-Complete problem is a subset of problems that are both NP and NP-hard. To prove a problem is NP-complete, one should show that the solution is verifiable in polynomial time, and that a proven NP-hard problem can reduce to it. This is done by solving a known NP-hard problem using the input constraints of the given problem, and showing a solution to this problem will also provide a solution to the NP-hard problem.

There are many instances of NP-Complete problems, including the Circuit SAT, knapsack, and Subset Sum to name a few. These problems have important application, such as finding optimal protein folding pattern (clique), and minimizing costs for network coverage (vertex cover). The difficulty is that it is costly to find optimal solutions to NP-Complete problems: brute-force or naive algorithms require exponential running time on deterministic machines. Therefore, approximation algorithms using heuristics are necessary to deal with these difficult problems. The following report will examine one specific instance of an NP-complete problem and a proposed approximation algorithm for it.

Problem Set-up

The NP-Complete problem for this exercise is a multi-constraint knapsack problem. An example application of this problem is choosing stocks to add to an investment portfolio, with the goal of maximizing resale value and minimizing risk. We will define our problem as PICK_ITEM. The objective is to come up with a list, or “portfolio”, of items that will generate the most profit given a set of constraints, defined below:

- Total weight of items cannot exceed P, maximum capacity of the portfolio
- Total cost of items cannot exceed M, the budget

- Items cannot be duplicated and items must be integral (no fractions).
- Each item has a class to which it belongs. Certain classes are incompatible and cannot be grouped in the portfolio. For example, suppose an item “AAPL” belongs to class 1, and an item “INTL” belongs to class 2, and class 1 is specified to be incompatible with class 2. (The extrapolation here is to minimize selection of stocks in complementary industries so we can mitigate risk across the portfolio).

The input data is a list of items that can be chosen for the portfolio, along with the attributes of each item provided in the following format.[2]

```
[P weight]
[M dollars]
[N number of items]
[C number of constraints]
[item_name]; [class]; [weight]; [cost]; [resale value]
...
[incompatible_class1, incompatible_class2, incompatible_class3]
[incompatible_class4, incompatible_class5]
...
```

Therefore, problem PICK_ITEMS is defined as: Given a list of *n* items with corresponding weights and constraints, is there a portfolio with valid classes of items that will yield profit of at least *K*.

Proving NP-Complete

We can prove that the problem PICK_ITEMS is NP-complete. First, demonstrating NP is straightforward. In order to verify a given solution, one can check that the solution satisfies the constraint and objective requirements. This includes a check that the total weight of chosen items does not exceed the maximum weight capacity P, and that the cost of these items is less than or equal to M, the budget. To verify class constraints, iterate through the items of the solution set to make sure that incompatible classes are not included. Finally, it is a constant time operation to verify that the solution’s profit satisfies the objective condition (yield profit at least K). These calculations can be performed in polynomial time to the size of the input.

To prove NP-hard, we will reduce the generalized 0-1 knapsack problem to PICK_ITEMS. This shows that solving PICK_ITEMS will solve 0-1 knapsack, implying PICK_ITEMS is just as hard as the proven NP-Complete problem. 0-1 knapsack has the decision criteria:

- Given: list of n items, where item i has value $v_i > 0$ and weight $w_i > 0$; a knapsack that can hold maximum weight of W
- Decision: Is there a set of items that will yield value of at least X?

The primary difference is that PICK_ITEMS has an additional class compatibility constraint. The transformation is to add a class constraint to 0-1 knapsack, using an empty list with zero incompatible items (meaning all classes are compatible with all other classes). This is a polynomial time conversion of the inputs for 0-1 knapsack into the input of PICK_ITEMS. We will then proceed to solve PICK-ITEMS with the modified inputs of 0-1 Knapsack. If PICK_ITEMS finds a valid solution that gives at least profit K, then there is a solution to 0-1 Knapsack of value at least equal to X. As such, PICK_ITEMS is indeed NP-Complete. The next portion will examine an approximation algorithm to solve this difficult problem.

III. ALGORITHM & PROCEDURE

Greedy Selection

The initial approach was to construct a greedy algorithm SOLVE which picked the items with the highest profit proportional to weight. To preprocess the given input list, the items were sorted in decreasing order by their value to weight ratio, calculated as (resale value - cost)/weight. Any items that had negative or zero ratio (sells at a loss or no profit) were removed, filtering out trivial cases from the data set to increase efficiency.
The algorithm iterates through the sorted list of items using the decision criteria:

- If the items cost does not exceed M budget remaining, and the item is compatible with items already in the portfolio, include the item in the portfolio.
- If any of those conditions are violated, ignore that item.

SOLVE runs this process until the list of n items is exhausted, or there is no more budget, or adding another item would exceed the maximum capacity P.

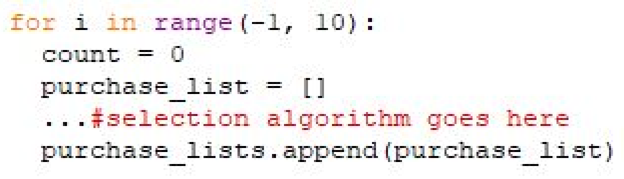
Memoization

An efficiency is introduced for the process of checking class constraints. Because this function is called each time the greedy algorithm tries to pick a new item, memoization was used to save intermediate results and reduce overall runtime. This was done by a function CREATE_CONSTRAINT_DICTIONARY, which constructs a dictionary to quickly find incompatible classes. Each class is denoted by a key, and the value is a list of other class numbers that are incompatible with it. The function builds the mapping by iterating through the list of classes provided in the input. When it encounters a new class, it creates a key value pair and continually updates the list of class constraints.

```
def create_constraint_dictionary(constraints):
    constraint_dictionary = {}
    for constraint in constraints:
        for class_num in constraint:
          for unpairable in constraint:
              if (class_num != unpairable):
                  if (class_num not in constraint_dictionary):
                      constraint_dictionary[class_num] = []
                      constraint_dictionary[class_num].append(unpairable)
                  else:
                      if (unpairable not in constraint_dictionary[class_num]):
                          constraint_dictionary[class_num].append(unpairable)
    return constraint_dictionary
```

CREATE_CONSTRAINT_DICTIONARY will show whether or not items could be placed together in a portfolio. The memoization process eliminates redundant work that would be done each time we search through an item's attribute. In this case, it is trading off space efficiency (to store the intermediate dictionary values) for time efficiency. As memory is considerably cheap, this seems an acceptable tradeoff to produce a faster algorithm.

Increasing Compatibility
An additional enhancement was added to diversify the item groups that were looked at. This is to defend against a case where high value items may be incompatible with many other items. For example, suppose item X has the highest value to weight ratio and will be looked at first by the greedy algorithm. Item X is selected into the portfolio, but if it's incompatible with the remaining items, it could potentially prevent better solution sets. To correct this, the greedy algorithm is run 10 times, and looks at a modified input list each time. The input list is altered in this manner: for every subsequent call to SOLVE, ignore one more item from the beginning of the input list.



```
for i in range(-1, 10):
    count = 0
    purchase_list = []
    ...#selection algorithm goes here
    purchase_lists.append(purchase_list)
```

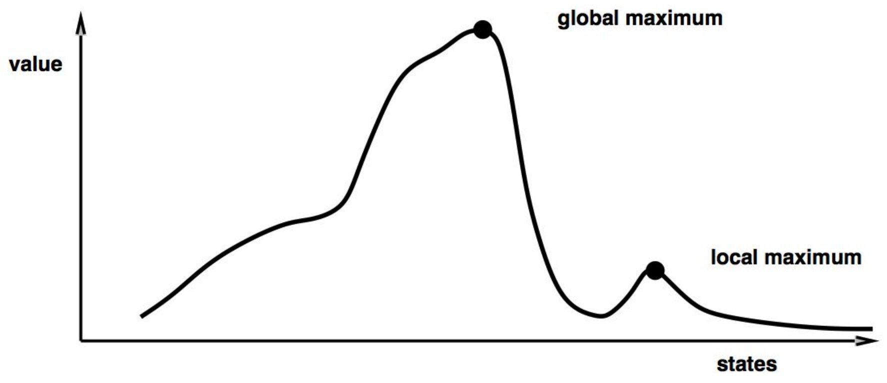
This means the first call uses every item, second call ignores the first item, third call ignores first 2 items, and so on. Once these 10 lists are compiled, the algorithm selects the one that yields the highest profit and returns that list of items. This approach increases the chance of finding a more compatible set of items, while remaining efficient.

IV. RESULTS & IMPROVEMENTS

Advantages
The advantages of this approximation solution is that the algorithm can be implemented efficiently. The program solves an input file with 80,000+ items in roughly 2 minutes, producing a result of 59% optimality. The runtime of the algorithm is calculated as O(n*logn) the time to sort the array + O(n) the time to iterate through the input list of items + O(C*l) cost of memoization, where l is the length of the longest constraint. All together this gives total runtime of O(nlogn + n + C*l) which is still polynomial in the input size of the problem.
Runtime aside, the solution concept is to start with an initial framework using a greedy algorithm, and apply improvements to find both more optimal solution sets and increase runtime efficiency. Therefore, it is a straightforward program to follow that provides good runtime.

Improvements
One pitfall of the proposed algorithm is that it could produce a sub-optimal solution - a profit that is significantly lower than the maximum. Greedy algorithms trade speed for accuracy. This is demonstrated in corner cases such as when the first items to pick are incompatible with other items. Although we attempt to address this by alternating input sets with different class constraints, the algorithm could be improved through induced randomization. This is built off the concept of hillclimbing[3]. If a solution set is a single hill among a series, each time the algorithm may only find a local maximum. The goal is to

determine the global optimum (or as close as possible to it). Selecting multiple starting points at random could help in that respect.



Therefore, the following improvement is to incorporate randomization:

Induced Randomization
Similar to the greedy algorithm, RANDOM preprocesses the data to remove items with value-weight ratios that are zero or lower. Instead of sorting the input list, items are randomly selected to add to the portfolio. This algorithm is run multiple times and the solution set with the highest profit is chosen. Applying randomization increases the chance of finding a “higher” hill, since it examines alternative combinations of items for the portfolio.**Attempt at implementing this Random method is included in Appendix files.*

Another method to maximize the occurrence of different item combinations is to find as many compatible class sets as possible. This could be done with the following implementation:
Run multiple iterations of the greedy implementation. For each subsequent iteration, modify the input list by removing items that share the same classes as items found in prior solution sets. This would allow greedy to test otherwise incompatible class sets. It could be that a portfolio with more compatible items, despite individually having lower value-weight ratios, could produce higher total profit.

V. CONCLUSION
An interesting final point is that the PICK_ITEMS problem has an inherent tension between its objective function and the class set constraint. The more class constraints examined, the more difficult it is to find a subset of items that satisfies all constraints. However, adding class constraints will make solving the knapsack problem easier.

This exercise provided quite a challenge. Finding an efficient approximation algorithm for an NP-hard problem required experiment with various approaches, although not all were successful. It was certainly a tremendous learning experience. Credit to Travis Dickey for guidance on improving algorithm runtime.

VI. APPENDIX
The following code and input files are attached with submission:

- ***Solver.py*** (Approximation Algorithm)

- ***solverRAND.py*** (Attempted randomization improvement)
- ***Input.in*** (Test Case)
- ***output.out*** (Generated Solution Set)

VII. REFERENCES

[1] Dasgupta, Sanjoy, Christos H. Papadimitriou, and Umesh Virkumar. Vazirani. "Chapter 8: NP-Complete Problems." Algorithms. Boston: McGraw-Hill Higher Education, 2011. N. pag. Print.

[2] Garg, Sanjam. Efficient Algorithms and Intractable Problems. NP-Complete Problems. 2017. Web

[3] Russell, Stuart J., and Peter Norvig. "Chapter 4: Beyond Classical Search."*Artificial Intelligence: A Modern Approach*. Boston: Pearson, 2016. 122-32. Print.