Optimizing for NP-Hard problems

Problem

We have a file full of items that we have to "buy" so we can "resell" the items for a profit. Each item has a weight, a cost, a resale value, and a class. We have a previously defined M dollars and can only fit P pounds worth of items into our "bag". Additionally there are constraints on items based on their classes. For example, you cannot buy a desk with class "Wood" and buy a bag of termites with class "Wood Destroyer"

Main Idea

This problem is quite similar to a more general the Knapsack problem with more constraints. The problem is also reminiscent of an Independent Set problem because the constraints can be represented by a graph and the solution will only have classes that are an independent set of nodes of classes, which means that classes, that are represented in a constraint graph, that can be placed in the same bag cannot be directly connected to each other. To approach this problem with naive dynamic programming would yield exponential time and therefore to process 21 input files in a time-efficient manner, we implemented a greedy scheme, inspired by the independent set and knapsack problem, which runs much quicker than exponential time. Depending on the input file, the runtime was anywhere from a seconds and occasionally up to a few minutes.

Our algorithm is as follows. Once we read in the input file and setting variables and lists accordingly, we created an array full of ones which represent each class and iterate through each constraint. For each class in this particular constraint, we added the length of the list to its value in the array. This array basically represents how many classes this one particular class constrains and mathematically represents the degree of the class node if all constraints are represented as a graph. This took roughly O(CN) time where C is the number of constraints N is number of items.

We sorted the items by adding them to a priority queue with a heuristic (Resale Value - Cost)/(Degree in connectivity)^{exp}, where exp is the variable we change to find the most profitable set of items. This heuristic made the algorithm pick items that are more profitable and items that do not constrain us from picking other items before any other item. In the process of picking, wherein we simply returned the highest heuristic value, we checked constraints and overweight and overspending along the way. A crucial part of this algorithm was to be able to check the constraints of each additional item fast. We did this by writing a checkCons() method that took a restrictedSet and recursively checked if the constraints were viable. The more the algorithm runs the faster it becomes since it is designed to become faster by storing previously seen constraints in the restrictedSet. This specific method takes almost constant time when restrict set is filled and O(C) time when you initially start. This step took the most resources and combined, it takes roughly O(NC) at the beginning speeding up to O(N) time as we start seeing more constraints.

We ran values ranging from zero to two for the variable exp as different problem input had different optimal values for exp. The degree of connectivity improved the profit from a slight 10%

to sometimes 150%.

Code

```
def solve(P, M, N, C, items, constraints):
  """
  Returns: a list of strings, corresponding to item names.
  """
  maxer={}
  exp = 0.65

  itemProfit = 0
  itemWeight = 0
  itemCost = 0
  itemlist = []
  priorityQ = queue.PriorityQueue()
  random.shuffle(items)
  allconstraints = []
  restrictSet = set()

  # Counts the number of connections to other classes
  connections = np.ones(N)
  if constraints:
    for constraint in constraints:
      for c in constraint:
        connections[c] += len(constraint) - 1


  for i in items:
    # If this item restricts many other items if picked, decrease priority by division
    #exp = 0.4
    if i[4] - i[3] > 0:
      heuristic = (i[3]-i[4])/(connections[i[1]]**exp)
      priorityQ.put((heuristic,i))

  for i in range(N):
    maxItem = priorityQ.get()[1]
    cond, temp = checkCons(maxItem[1], restrictSet, constraints)
    if cond:
      if itemWeight + maxItem[2] <= P and itemCost + maxItem[3] <= M:
        restrictSet = temp
        itemlist.append(maxItem[0])
        itemWeight += maxItem[2]
        itemCost += maxItem[3]
        itemProfit += maxItem[4]-maxItem[3]
  return itemlist
```

```
def checkCons(lastItemAdded, restrictSet, constraints):

  if lastItemAdded in restrictSet:
      return (False, restrictSet)
  for constraint in constraints:
      if lastItemAdded in constraint:
        for item in constraint:
          if item != lastItemAdded:
            restrictSet.add(item)
  return (True, restrictSet)
      #check set to see if clash
```