Homework 6 Advanced Analytics and Metaheuristics

Group 1: Nicholas Jacob

April 2, 2024

1. Strategies for the problem

   (a) Two differing initialization solutions for the knapsack could be random or all zeros.

   Empty We could also just ask that initially your knapsack is empty. We simply make the initial all zeros. We know this will be in the feasible set initially. This will aide in the completion of a legitimate solution since the first task will not be to discover a solution in the feasible set.

   ```
   for i in range(n):
       x.append(0)
   ```

   Random Random has the benefit of being just that but it also has the serious issue of not being in the feasible set (nor any where close to the feasible set). This could be dealt with by re-seeding the random until it was inside the feasible regime. We see in the code below that we seed randomly, this causes an expected value of $\frac{n}{2}$ to be included. We expect that only 25 items will be in any feasible set. We trim randomly too based on index.

   ```
     for i in range(n):
         x.append(myPRNG.randint(0,1))  #initial seeding

   i = myPRNG.randint(0,n-1) #random index to make zero if needed
   while evaluate(x)[1] > maxWeight: #winnowing down randomly
     x[i] = 0
     i=myPRNG.randint(0,n-1)
   ```

   (b) Neighborhood

1 Flip We really like the 1-flip neighborhood that was provided in the sample code. We bring it up here to mention that using the 1-flip, zero initial vector and best improvement local search, we are doing a greedy search. We will take the highest value item each time until we fill the knapsack. When the knapsack is full, we will take the next highest object that can still fit. We provide no code sample as it was already provided for us.

Sliding window Take the x_{curr} and slide all the 1's and 0's to the left. Each time you move one unit, add to the neighborhood. This will add $n-1$ items to the neighborhood. Unfortunately it will not add or subtract any items to our knapsack. It will not work with all initially 0 and will not work if we just have too many items to start.

```
for i in range(0,len(x)-1):
    newlist = x[i+1:]
    newlist.extend(x[:i+1])
    nbrhood.append(newlist)
```

Slide and Flip We combine the slide and flip method. This will create n^2, neighbors. We do insist that the slide return the original flip to make this work. This does give us many chances to check the same neighbor but we did not worry about that based on how our code was constructed. Here is the code sample

```
for i in neighborhood_flip(x): #do the flip and look at each item
   nbrhood.extend(neighborhood_slide(i)) #add all the slide elements
```

Permutation We attempted to do permutation but could not execute without new package and some runtime errors on the google colab server. Size would have depended on number of 1's, n_1, and would be $\binom{n}{n_1}$.

(c) Infeasible

Small Value If the weight is outside of allowed, simply make the value small (negative). While this works, the infeasible will match and end the while loop in the infeasible region.

```
if totalWeight > maxWeight:
     totalValue = -1000
```

This did not work well. If you were far outside of the feasible region to start, you stayed outside of the feasible and returned an infeasible solution.

Random Tried random value, that didn't work either. Again if you were far outside of the feasible, it would randomly find a low feasible value and get stuck there.

Negative of Total Value What does seem to work okay for us is using the opposite of the value when you were in the infeasible region. Since you would want to increase this value, you would in turn decrease the absolute value of this. It allowed for us to escape most times when we were far from the infeasible region.

```
if totalWeight > maxWeight:
    totalValue = -totalValue
```

We also note that the random generator with utilizing the randomness to get us into feasiblity helped us get away from the problem areas.

2. Local Search with Best Improvement: We apply this method using the all zero's as the initial, the Negative of Total Value to deal with in-feasibility, and the Slide and Flip neighbors. We include the details of this run in the table at the end of the document.

3. Local Search with First Improvement: We repeat the methods from above (zeros, negative, and slide/flip). We report results below in table. Only change in logic was introduction of the 'break' command to the for loop. This allowed the exit of the for loop when the first improvement of the model was found.

```
if evaluate(s)[0] > f_best[0]:
            x_best = s[:]                #find the best member and keep track o
            f_best = evaluate(s)[:]      #and store its evaluation
            break
```

We include the results at the end of the document.

4. Random Restarts Best Improvements: To implement this we had to use one of our random initializers. We used the truly random one, knowing that the first objective would be to get into the feasible regime. This did not preform very quickly. We attempted again with the initializer that guaranteed a solution in feasible region to start.

We used first improvement to juice the runtime a bit. We kept the neighbors to be the flip and slide.

```
k = 50
solns = [] #array for storing solutions values

for i in range(k):


  x_curr = initial_solution_random_start_feasible()  #x_curr will hold the curre
  x_best = x_curr[:]           #x_best will hold the best solution
  f_curr = evaluate(x_curr)    #f_curr will hold the evaluation of the current s
  f_best = f_curr[:]
...
  solns.append(solutionsChecked) #add pieces to solutions
  solns.append(f_best[0])
  solns.append(f_best[1])
  solns.append(np.sum(x_best))
  solns.append(x_best)
```

Then the coolest part of code found the best solution from this array:

```
weightsMax = []

for i in range(0,k*5,5): #five things added to the solution array each execution
  weightsMax.append(solns[i+1]) #objective is in 1 slot

iter = weightsMax.index(max(weightsMax)) #find iteration with best value

print ("\nFinal number of solutions checked: ", solns[5*iter]) #print results
print ("Best value found: ", solns[5*iter+1])
print ("Weight is: ", solns[5*iter+2])
print ("Total number of items selected: ", solns[5*iter+3])
print ("Best solution: ", solns[5*iter+4])
```

Three results are included in the table, shortened to be called 'RR'

5. Local Search with Random Walk: To implement this, we needed a p to compare to and a *ptest* that is randomly generated in each iteration of the while loop. We use the all zero initial solution and the slide

and flip neighborhood. The code that give us this result is a simple modification after the Neighborhood has been generated,

```
if p>ptest: #do the normal thing
  for s in Neighborhood:                #evaluate every member in the neighb
    solutionsChecked = solutionsChecked + 1
    if evaluate(s)[0] > f_best[0]:
        x_best = s[:]                 #find the best member and keep track o
        f_best = evaluate(s)[:]       #and store its evaluation
else: #set the best to a random
  randNbr = myPRNG.randint(0,len(Neighborhood)-1) #random integer smaller th
  x_best = Neighborhood[randNbr][:]#make it the best
  f_best = evaluate(x_best)[:] #do evaluations too
```

We saw the best results yet for $p = 90\%$. We did test some other values in this area about 90% but none came out better than our reported result.

6. Stochastic Hill Climb: We start with empty and used the flip and slide. This is neither best nor first improvement as we record all improvements and then make a random choice in which we choose (This may have been a copy/pasta error in the assignment document). First I create a list of all the improvements that occur in the neighborhood (I compare to the current not the best!). Then we attempt two executions of random. In the first we do a totally random draw based solely on index. This returned marginal results. Next we took the same idea of recording the improvements in a list but utilized the weight functionality in random.choices. This gave use away to still be random and improving and weight our decision on how much we were improving. The results here were also marginal.

```
    if evaluate(s)[0] > f_curr[0]:#compare to current not best
        improvements.append(s[:])                 #add to list
        improvements.append(evaluate(s)[:])       #and store its evaluation

if len(improvements)>0:
  w = []
  for i in range(int(len(improvements)/2)):
    w.append(improvements[2*i +1][0])
```

```
        whichone = myPRNG.choices(range(int(len(improvements)/2)), weights = w, k
        x_best = improvements[2*whichone]
        f_best = improvements[2*whichone +1]
```

Algorithm	Iterations	# Items Selected	Weight	Objective
Local Search (Best Improvement)	581100	25	2331.3	16654.5
Local Search (First Improvement)	37268	29	2497.9	15583.1
RR (Truly Random, FI, $k = 20$)	157279	34	2485.4	16366.8
RR (Random but Feasible, FI, $k = 20$)	26964	33	2496.1	17304.9
RR (Random but Feasible, FI, $k = 50$)	95386	33	2476	17442.7
Random Walk ($p = 0.90$)	630000	31	2497	22337.1
Random Walk ($p = 0.75$)	540000	29	2498.6	20423.3
Random Walk ($p = 0.5$)	405000	34	2490.0	16676.8
Stochastic (random by index)	945000	33	2498.7	15959.7
Stochastic (random value as weight)	742500	32	2492.7	16712.2

In all, we saw marginal returns on our model complexity, with random walk being the best method. We partly attribute this to our insistence on using the all zero vector as the initial in many of our attempts (we did not use it in the random restart algorithm). We wonder how much different neighborhood generators and infeasible requirements might change these results. We are interested in seeing how our peers approached these exercises in similar yet differing ways.