Homework 7 Advanced Analytics and Metaheuristics

Group 20: Nicholas Jacob

April 16, 2024

1. Simulated Annealing

Initial Temperature Looking at our original knapsack problem, we see that the best increase we could expect in the value is 1000. We test with that value as our initial temperature extensively. After tweaking this some we recognized that a higher initial temperature gave some nice results. Temperature tweaking is really what this part of the assignment was all about so doing so felt natural. I was even so bold as to try 10,000 for an initial temperature run. I did not see much difference although to be honest this method does not allow for seeing much as to what is happening along the way (at least as I have coded it...)

Cooling We use both an exponential and Cauchy cooling scheme. Exponential cooling was simply:

```
T = 0.99*T
```

While the Cauchy cooling required the knowledge of how many temperatures we had used (gotten from the for loop) and the initial temp (reserved as it's own variable, T_0).

```
T = T0/(1+j)
```

Probabilities For the probabilities, we did a random selection of the neighbor using the index. We then assigned a probability using the following code:

```
def probability_assignment(x,T):
  if x>0:
    return 1
```

```
else:
  return np.exp(x/T)
```

We compared this to a random value between 0 and 1. We see that if the x (which was the difference of the neighbor and the current) was positive, we accepted that as the new value. If not, there was a chance that we would take the other value.

Stopping Criteria We first attempted to just run through a total number of iterations with a for loop, just allowing it to continue until it exhausted all possibilities in the for loop. This had a few draw backs: while simple, it could get stuck and take a long time. It could also hit a piece of the logic and not find an acceptable new solution. I quickly made an edit to the annealing code, if it could not find a suitable neighbor in 150 tries (length of the neighbors), I exited the loop looking for an acceptable neighbor. It would then return to that loop and again attempt to find a suitable neighbor. Since there are probabilities involved, perhaps it would not find a neighbor to move to. Soon I added another chance to break the loops. If the code failed to find an acceptable neighbor after so many iterations, I just wanted to break the outside loop and return what it had. Again I just accomplished this with break rather than coding it into whiles. Here is the essential part of that code

```
      elif howLongInWhile >150:
        failedToExit +=1
        break
  if failedToExit > 1000:
    break
```

2. Genetic Algorithm

createChromosome I originally thought about creating random but keeping the solutions in the feasible set. I decided against this as I would just punish those outside of the feasible set in my evaluation. The literature makes it clear that since the optimal solutions should occur near the boundary, it is best to consider those values in some way. To do this random process the code is very simply repeated below.

```
def createChromosome(n):
```

```
    x = []   #i recommend creating the solution as a list
    for i in range(n):
      x.append(myPRNG.randint(0,1)) #pick a random 1 or 0 until you fill the

    return x
```

crossover For the crossover, I implemented two random processes. One determines if you mate at all. This utilizes the crossOverRate. If you do not mate, you just go into the offspring pool as is. If you do mate, I slice you at a random index. I insist this index is between 1 and 2 less than the length so that you are indeed sliced.

```
def crossover(x1,x2):
  p = myPRNG.random()
  if p>crossOverRate: #just go back into the gene pool as is
    offspring1 = x1[:]
    offspring2 = x2[:]
  else: #have some children
    cutIndex = myPRNG.randint(1,n-2) #cut at a place where you indeed make n
    offspring1 = x1[0:cutIndex] + x2[cutIndex:]
    offspring2 = x2[0:cutIndex] + x1[cutIndex:]


  return offspring1, offspring2  #two offspring are returned
```

evaluate We modified this from our earlier assignments. With the strength of near the edge solutions, we wanted them to be considered even if overweight but penalized. We decided on a penalty of 100 in value for each pound overweight. We expected we might get some optimal solutions that were outside of the feasible set but did not observe any so long as the number of generations was high enough.

```
def evaluate(x):

    a=np.array(x)
    b=np.array(value)

    totalValue = np.dot(a,b)     #compute the value of the knapsack selectio
    totalWeight = calcWeight(x) #compute the total weight
```

```
    fitness  = totalValue

    if totalWeight > maxWeight:
      fitness = fitness - 100*(totalWeight - maxWeight)#penalty of 100 per p

    return fitness   #returns the chromosome fitness
```

rouletteWheel This was by far my favorite part of the code. I utilized rank as I do believe that survival of the fittest is about ranking rather than evaluating potential mates. I created a list of the ranks (starting from the largest number n and counting down). I then divided by the sum of all those digits $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Next a did a cummulative of that array (using numpy cumsum). This gives me the steps of the probability. Then I jump into the first while loop that will continue until I fill the mating pool. The second while loop finds the index where the random number falls. Now that we have the index, we add that chromosome into the matingPool.

```
def rouletteWheel(pop):

    matingPool = [] #initialize an empty array
    rank = [i/(len(pop)*(len(pop)+1)/2) for i in range(len(pop),0,-1)] #get
    cumrank = np.cumsum(rank) #needed cummulative probability of those ranks

    while len(matingPool)<populationSize: #fill up the whole mating pool
      p = myPRNG.random() #get a rando between 0 and 1
      i = 0 #start with the most likely
      while p>cumrank[i]: #grab the random int when finally surpass the p va
        i+=1
      matingPool.append(pop[i]) #add it into the pool


    return matingPool
```

mutate For mutate, I chose to simply change 1 bit. If you get a random low enough, you will randomly change one index to a different value.

```
def mutate(x):
```

```
    p = myPRNG.random() #p is random

    if mutationRate > p: #only go here if you are lower than the small numbe
      i = myPRNG.randint(0,n-1)
      if(x[i]==0):#change one random bit
        x[i] = 1
      else:
        x[i] = 0

    return x
```

insert I simply kept the best solutions from the previous generation by appending them onto the kids.

```
def insert(pop,kids):

    kids = kids[:]

    for i in range(eliteSolutions):
      kids.append(pop[i]) #these have been ordered so keep the first few


    return kids
```

This method of coding allowed us to follow the code closely (at least statistically). We saw the best candidate normally appear in the 25-55 range which is why most of the tests have 100 generations. Increasing the mutation rate did help a bit as well as increasing the population size. It was honestly unclear how to properly tune the parameters or if we were just getting randomly lucky.

Method

SA	T_0	Cooling, t_k	M_k	# of temps	Iterations	Items	Weight	Value
	100	$0.99t_{k-1}$	50	1000	126710	39	2487.2	23405
	1000	$0.99t_{k-1}$	50	1000	236504	43	2471	24456.6
	1500	$0.99t_{k-1}$	25	500	625724	44	2499.8	24898.2
	2500	$0.99t_{k-1}$	50	1000	249256	43	2481.0	24747.5
	1000	$\frac{T_0}{1+k}$	50	1000	170655	41	2494.5	23247
	2000	$\frac{T_0}{1+k}$	100	1000	106265	45	2498.2	22999.4
	2000	$\frac{T_0}{1+k}$	100	1000	231663	45	2496.1	24459.5
	3000	$\frac{T_0}{1+k}$	100	500	244890	42	2498.3	24147.5
	10000	$0.99t_{k-1}$	100	500	500448	42	2452.4	23704.5
	10000	$\frac{T_0}{1+k}$	100	1000	363743	42	2464.4	23288.7
GA	Gens	PopSize	CrosOvr	Mutation	Elitism	Items	Weight	Value
	100	150	0.8	0.05	top 10	29	2487.8	28852.9
	50	100	0.8	0.05	top 10	29	2496.8	27670.9
	50	200	0.8	0.25	top 20	33	2499.4	32342.3
	100	200	0.95	0.25	top 5	32	2468.9	33976.1
	100	200	0.95	0.50	top 25	32	2487.5	38201.4
	100	2000	0.7	0.05	top 200	32	2480.8	37589
	100	2000	0.95	0.50	top 200	34	2491.8	40874.5

Finishing this assignment, we note that the results of the methods cannot be compared. They used a slightly different set up so the values in the GA method were allowed to be higher than those in the SA (and previous assignment). We did not notice this issue while running our initial tests.