**ISE-DSA 5113 Homework 7**

**Group 12 Team Members:**

* Orkhan Khankishiyev
* Anvitha Reddy Thummalapally
* Tahsin Tabassum

**Question 1.**

**Initial temperature**

In Simulated Annealing, the initial temperature plays a critical role in determining the algorithm’s capacity to explore the solution space. A higher initial temperature increases the likelihood of accepting worse solutions early in the search, allowing for better global exploration. Based on empirical testing, two initial temperature values were chosen for this problem: 1500 and 2000. These values were selected after preliminary runs indicated they allowed sufficient exploration without overshooting the solution space. Among these, 1500 paired with an appropriate cooling rate consistently yielded better results, striking a balance between exploration and convergence speed.

**Cooling schedule**

Two cooling schedules were tested to observe their effects on the convergence behavior and quality of solutions:

1. Geometric Cooling Schedule

This schedule updates the temperature according to the formula:

where 𝛼=0.99

This exponential decay gradually reduces the temperature, promoting more refined searches over time. The geometric approach maintained better control over temperature reduction and showed superior performance, especially when used with Mk = 50 and initial temperature = 1500.

1. Reciprocal Cooling Schedule

The second method follows the equation:

​

where T0 is the initial temperature, and k is the iteration counter.

This approach reduces the temperature more aggressively as the search progresses. While computationally efficient, it frequently converged prematurely and failed to discover high-quality solutions, especially in configurations with Mk = 50 or 100.

**Acceptance Probability for Non-Improving Moves**

To escape local optima, Simulated Annealing allows the acceptance of non-improving moves based on a probability function. The acceptance probability is computed as:

where Δ is the loss in objective function value (i.e., the difference between the current best and the candidate solution), and T is the current temperature.

This logic was implemented using the following Python code:

delta = f\_best[0] - evaluate(s)[0] #If not, calculate the difference

ep = myPRNG.random() #pick a random number on [0.1]

if ep < np.exp(-delta/tk):

x\_best = s[:]

f\_best = evaluate(s)[:]

break

A random number is generated and compared with the computed probability. If the random number is less than the probability, the algorithm accepts the worse solution.

**Stopping Criterion**

The algorithm uses a minimum temperature threshold as its stopping criterion. Once the temperature drops below this threshold, the search terminates. In this implementation, the minimum temperature was set to 5, which ensures the algorithm stops when the probability of accepting worse solutions becomes negligible. This approach prevents excessive computation while ensuring adequate exploration.

minTemp = 5

while tk > minTemp:

**Results Summary**

The performance of each configuration was recorded based on key parameters, including cooling schedule, number of iterations per temperature (Mk), and initial temperature. The table below summarizes the outcomes:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Cooling Schedule |  | Initial Temperature | Temperature Checked | Iterations | # of Items | Weight | Value |
|  | 100 | 2000 | 597 | 30851 | 41 | 2498.6 | 21928.4 |
|  | 50 | 1500 | 568 | 16574 | 42 | 2494.7 | 22016.7 |
|  | 100 | 2000 | 400 | 35742 | 36 | 2479.7 | 15274.0 |
|  | 50 | 1500 | 300 | 13340 | 32 | 2495.7 | 13989.7 |

The best solution was obtained using the geometric cooling schedule with Mk = 50 and initial temperature = 1500, resulting in a knapsack value of 22016.7 with a total weight of 2494.7. This configuration offered an efficient balance between exploitation and exploration and proved to be the most effective strategy among all tested combinations.

**Question 2**

**1. createChromosome() logic**

The createChromosome(n) function generates a chromosome for a genetic algorithm, specifically addressing the knapsack problem. It initializes an empty list x to hold the chromosome and a list selected to track chosen items. The function randomly selects items from all\_items until the total weight surpasses the knapsack's limit. It then constructs the chromosome by marking selected items with a 1 and the rest with a 0. The function returns the created chromosome x, yet it's important to note that despite its name implying a continuous-valued chromosome, it produces a discrete one. Each index in the list represents an item, simplifying solution space exploration for the genetic algorithm, aiding in iterative evolution towards optimal knapsack solutions.

#create a binary valued chromosome

def createChromosome(n):

#this code as-is expects chromosomes to be stored as a list, e.g., x = []

x = [] # i recommend creating the solution as a list

selected = []

while sum(all\_items[item][1] for item in selected) < maxWeight:

item = random.choice(list(all\_items.keys()))

if sum(all\_items[item][1] for item in selected) + all\_items[item][1] <= maxWeight:

selected.append(item)

else:

x = [1 if i in selected else 0 for i in range(1, n+1)]

break

return x

**2. crossover() logic**

In a genetic algorithm, the crossover(x1, x2) function makes breeding easier by executing crossover with a specific probability, indicated by crossoverRate. First, to decide if crossover should be done, a random number is chosen. Crossover happens when the random number is less than or equal to the crossover rate. The next step is to randomly select a crossover site (assuming that the chromosomes are 150 nm long) between indices 1 and 150. After that, the segments of the chromosomes x1 and x2 are switched at the crossover site to produce the two new offspring chromosomes, offspring1 and offspring2. Breeding does not take place and the progeny chromosomes, x1 and x2, are just copies of their parent chromosomes if the random number is greater than the crossover rate. Finally, the function returns the two offspring chromosomes, offspring1 and offspring2, regardless of whether crossover was performed or not.

#implement a crossover

def crossover(x1,x2):

#with some probability (i.e., crossoverRate) perform breeding via crossover,

rand\_num = myPRNG.random()

if rand\_num <= crossOverRate: # Perform a cross over

cross\_over\_point=myPRNG.randint(1,150)

offspring1=x1[:cross\_over\_point]+x2[cross\_over\_point:]

offspring2=x1[cross\_over\_point:]+x2[:cross\_over\_point]

else:

# if no breeding occurs, then offspring1 and offspring2 can simply be copies of x1 and x2, respectively

offspring1=x1[:]

offspring2=x2[:]

return offspring1, offspring2 #two offspring are returned

**3. Logic to compute chromosome fitness, e.g., any modifications to the evaluate() function**

The evaluate(x) function assesses the fitness of a given chromosome x in the context of the knapsack problem. It calculates the total value and total weight of the items selected in the knapsack based on the provided chromosome. Firstly, the chromosome x is converted into NumPy arrays for efficient computation. Arrays a, b, and c correspond to the binary representation of the chromosome, the values of the items, and the weights of the items, respectively. The total value of the knapsack selection is computed by taking the dot product of arrays a and b, representing the selected items and their respective values. Similarly, the total weight is computed by taking the dot product of arrays a and c, representing the selected items and their respective weights. If the total weight exceeds the maximum weight allowed for the knapsack, a penalty is applied to the total value. The penalty is set as 500 times the difference between the total weight and the maximum weight. This discourages selecting solutions that exceed the weight constraint. Finally, the fitness of the chromosome is determined by its total value, which is then returned. The higher the fitness value, the better the chromosome performs in terms of maximizing the value of items selected while adhering to the weight constraint.

#function to evaluate a solution x

def evaluate(x):

a=np.array(x)

b=np.array(value)

c=np.array(weights)

totalValue = np.dot(a,b) #compute the value of the knapsack selection

totalWeight = np.dot(a,c) #compute the weight value of the knapsack selection

if totalWeight > maxWeight:

penalty = 500\*(totalWeight - maxWeight) # set a large penalty for exceeding the max weight

totalValue = totalValue-penalty

fitness = totalValue

return fitness #returns the chromosome fitness

**4. rouletteWheel() logic**

The rouletteWheel(population) function applies a selection process called roulette wheel selection in genetic algorithms. This method randomly picks individuals from the population to construct a mating pool, where selection probabilities are proportional to each individual's fitness. Initially, an empty list named matingPool is initialized to collect the chosen individuals. The total fitness of the entire population is computed by summing up the fitness values of all individuals. Following this, selection probabilities for each individual are determined based on their relative fitness within the population. These probabilities are calculated by dividing each individual's fitness by the total fitness sum of the population. To execute roulette wheel selection, cumulative probabilities are generated by iteratively accumulating the selection probabilities. Subsequently, a random spin of the roulette wheel is simulated for each individual in the population. During each spin, a loop iterates over the cumulative probabilities until it identifies the first cumulative probability greater than the random spin value. The corresponding individual is then selected and appended to the mating pool. This process repeats until the desired number of individuals is chosen. The resulting mating pool comprises individuals selected with probabilities that correlate with their fitness levels, thereby favoring the selection of fitter individuals. Ultimately, the mating pool is returned for subsequent genetic operations such as crossover and mutation.

def rouletteWheel(population):

#randomly select k chromosomes; the best joins the mating pool

matingPool = []

fitness\_sum=sum(i[1] for i in population) # Calculate the sum of fitness values

probabilities=[i[1]/fitness\_sum for i in population] # Calculate the probability of selection for each individual

cumulative\_probability=[]

temp=0

for j in range(len(population)):

temp=temp+probabilities[j]

cumulative\_probability.append(temp)

#create sometype of rouletteWheel selection -- can be based on fitness function or fitness rank

#(remember the population is always ordered from most fit to least fit, so pop[0] is the fittest chromosome in the population, and pop[populationSize-1] is the least fit!

for i in range(len(population)):

spin = random.random()

for i in cumulative\_probability:

if i>spin:

try:

pop=population[cumulative\_probability.index(i)+1][0]

except:

pop=population[cumulative\_probability.index(i)][0]

break

matingPool.append(pop)

return matingPool

**5. mutate() logic**

The mutate(x) function applies controlled mutations to a chromosome x within a genetic algorithm, where the mutation rate is regulated by the parameter mutationRate. Mutations play a crucial role in maintaining population diversity and exploring different sections of the solution space. Initially, a random number is generated using a pseudo-random number generator (myPRNG) to determine whether a mutation will happen. If this number falls within the specified mutation rate, a mutation is triggered. To perform the mutation, a random point within the chromosome is selected using the randint function, ensuring that the mutation doesn't affect the first or last element. The value at this point is then toggled between 0 and 1, effectively modifying the chromosome's genetic makeup. The resulting mutated chromosome, mutated\_x, is then returned. If no mutation occurs, an exact copy of the original chromosome x is returned. This cautious approach ensures that mutations are applied sparingly, preventing excessive disruption to the genetic information contained within the chromosome.

#function to mutate solutions

def mutate(x):

#create some mutation logic -- make sure to incorporate "mutationRate" somewhere and dont' do TOO much mutation

rand\_num = myPRNG.random()

mutated\_x=x[:]

if rand\_num <= mutationRate: # Perform a cross over

mutation\_point=myPRNG.randint(1,n-2)

mutated\_x[mutation\_point]= 1-mutated\_x[mutation\_point] # perform mutation by flipping bits

return mutated\_x

return mutated

**6. insert() logic**

The insert(pop, kids) function incorporates offspring into the population, but the current implementation lacks proper elitism, potentially discarding valuable solutions from the previous generation. A suggested improvement involves retaining the top solutions from the previous population (pop) to maintain diversity and preserve promising candidates. This ensures that the algorithm doesn't lose potentially superior chromosomes. Depending on preference, it's advisable to keep a certain number of elite solutions from the previous generation, such as the top 5 or 10 solutions. The updated function inserts elite solutions from the previous population (pop) into the new population along with offspring (kids). Then, the combined list is sorted based on fitness, ensuring the best solutions are retained. Finally, the function returns the updated population list. This approach safeguards against losing valuable genetic material and maintains the evolutionary progress of the genetic algorithm.

#insertion step

def insert(pop,kids):

#this is not a good solution here... essentially this is replacing the previous generation with the offspring and not implementing any type of elitism

#at the VERY LEAST evaluate the best solution from "pop" to make sure you are not losing a very good chromosome from last generation

#maybe want to keep the top 5? 10? solutions from pop -- it's up to you.

new\_list=pop[:eliteSolutions]+kids[eliteSolutions:]

new\_list = sorted(new\_list, key=lambda x: x[1], reverse=True)

return new\_list

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Generations | Pop size | Crossover | Mutation | Elitism | Items | Weight | Value |
| 500 | 150 | 0.7 | 0.5 | 5% | 21 | 2418.7 | 23626.19 |
| 500 | 200 | 0.7 | 0.5 | 5% | 23 | 2447 | 25580.81 |
| 1000 | 200 | 0.7 | 0.5 | 5% | 22 | 2497.5 | 25834.4 |
| 1000 | 150 | 0.85 | 0.1 | 10% | 23 | 2410 | 25406.1 |
| 1000 | 150 | 0.85 | 0.15 | 10% | 22 | 2479.8 | 2594.85 |