**Lab 2 : Genetic Algorithm Report**

**12 August 2024, Monday 2-5 PM**

Team Members

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **ID** | **Year/Trimester** | **Program** |
| Lee Zhan Jun | 2103098 | Y3T1 | SE |
| Cheng Jia Han | 2101869 | Y3T1 | SE |
| Eric Lim Jun Kit | 2105156 | Y2T3 | AM |
| See Thau Ke Qi | 2106306 | Y2T3 | AM |

**TODO1**

The below codes read a set of cities from the filename to be used for creating an initial population.

**city\_list = []**

An empty list ‘city\_list’ is initialized to store the cities read from the file.

**with open(filename, 'r') as file:**

**# Skip header**

**next(file)**

While opens the file specified by filename in read mode, skip the header of the file.

**for line in file:**

**city\_id, x, y = line.strip().split(', ')**

**city\_list.append((int(city\_id), int(x), int(y)))**

After that, start a loop to iterate over each remaining line in the file. The first line strips any whitespace and splits the line into city\_id, x, and y coordinates with a comma. Next, it converts the city ID and coordinates to integers and appends them to city\_list.

**cities = random.sample(range(len(city\_list)), 10)**

The ‘random.sample’ function is used to select 10 different indices from a list of cities. This ensures that the initial population is chosen at random from the provided data set of cities.

**for index in cities:**

**cityList.append(City(x=city\_list[index][1], y=city\_list[index][2]))**

This code creates city objects using the x and y coordinates from the city\_list and adds them to the ‘cityList’. Each city object represents a city with specific coordinates, and this list of cities (cityList) will be used in the genetic algorithm to solve the TSP.

**TODO2 (Tournament Selection)**

Tournament selection is a method for choosing an individual from a population of individuals in a genetic algorithm. Below shows the codes’ explanation line by line:

**for i in range(0, poolSize):**

**tournament = random.sample(population, 3)**

A loop is initiated to run poolSize times, ensuring that enough parents are selected. In each iteration, a "tournament" is conducted by randomly choosing 3 individuals from the population using random.sample(population, 3). This subset of individuals will compete to determine which one is the best candidate for mating.

**best\_individual = min(tournament, key=lambda ind: Fitness(ind).routeDistance())**

The purpose of this line of code is to select the best individuals in the tournament. The individual with the smallest route distance is selected as the best individual. The lambda function ‘key=lambda ind: Fitness(ind).routeDistance()’ evaluates each individual’s fitness by calculating its route distance.

**matingPool.append(best\_individual)**

The selected best individual is added to the ‘matingPool’. This chosen individual has been recognized as one of the fittest in terms of shortest route distance, making it a great fit for passing on its traits to the next generation.

**TODO3 (Proportional Selection)**

Proportional selection, also known as Roulette Wheel Selection, is a genetic algorithm used to select individuals based on the proportion of their fitness relative to the entire population. Below shows the codes’ explanation line by line:

**S = sum(1.0 / Fitness(ind).routeDistance() for ind in population)**

This line calculates the total "fitness" of the population. Since the problem is a Travelling Salesman Problem (TSP), where a lower routeDistance() is better, the inverse of the route distance (1.0 / Fitness(ind).routeDistance()) is used to make the fitness value higher for shorter routes. The sum of all these fitness values (S) represents the overall fitness of the population.

**for i in range(0, poolSize):**

**pick = random.uniform(0, S)**

**current = 0**

This loop represents the Parent Selection Process. It runs pool Size times to select that many individuals for the mating pool. A random number is generated to determine which individual gets selected, with the probability being proportional to fitness.

**for ind in population:**

**current += 1.0 / Fitness(ind).routeDistance()**

**if current > pick:**

**matingPool.append(ind)**

**break**

The code iterates through the population, adding every individual's fitness (inverse of route distance) to current. When current exceeds pick, the corresponding individual is selected and added to the ‘matingPool’.

This step ensures that individuals with higher fitness (lower route distances) are more likely to be selected, but it still gives a chance to less fit individuals.

**TODO4 Survival Selection Explanation**

* **elites =[]**

This line initializes an empty array that will eventually hold the selected elite individuals.

* **merge\_population = population + population**

This assumes that population represents both parents and offspring

* **merge\_population.sort(key=lambda x: Fitness(x).routeFitness(), reverse=True)**

The combined population is sorted by fitness values. Presumably, Fitness(x).routeFitness() calculates the fitness of an individual x. The sorting takes place in descending order (reverse=True), which means that the fittest specimens are at the top.

* **elites = merge\_population[:eliteSize]**

After sorting, the population is truncated to the top eliteSize individuals, which are then considered as elites.

* **return elites**

Finally, the elites are returned.

**Result**

**A screenshot of a computer program

Description automatically generated** **A screenshot of a computer program

Description automatically generated**

**Crossover Explanation**

* **size = len(parent1) cx\_point1 = random.randint(0, size - 1) cx\_point2 = random.randint(0, size - 1) if cx\_point1 > cx\_point2: cx\_point1, cx\_point2 = cx\_point2, cx\_point1**

In the parent permutation, two crossover sites are arbitrarily chosen from the range of indices. To make sure that cx\_point1 is less than cx\_point2, they are switched if the second point is smaller than the first.

* **child1 = [None] \* size**
* **child2 = [None] \* size**

Two of the child permutations have None values initialized, meaning they are initially empty.

* **child1[cx\_point1:cx\_point2] = parent1[cx\_point1:cx\_point2]**
* **child2[cx\_point1:cx\_point2] = parent2[cx\_point1:cx\_point2]**

Each parent's segments between the crossing locations are duplicated straight into the matching child. The process of producing the offspring begins with this.

* **def pmx\_fill(child, parent, cx\_point1, cx\_point2): for i in range(cx\_point1, cx\_point2): if parent[i] not in child: pos = i while cx\_point1 <= pos < cx\_point2: pos = parent.index(child[pos]) child[pos] = parent[i] for i in range(size): if child[i] is None: child[i] = parent[i]**

This function is for filling the remaining positions in each child with the genes inherited by other parents, and to make sure that there are no duplicates, and the permutation constraints are respected. It transfers the gene to the correct position for each position in the kid's crossover segment if the parent's position is not yet present in the child. Following the crossover segment's mapping, if a parent gene is absent in the child, it is mapped in accordance with PMX guidelines and put in its proper location. The remaining genes from the other parent are then added to any None values that remain in the child.

* **pmx\_fill(child1, parent2, cx\_point1, cx\_point2)**
* **pmx\_fill(child2, parent1, cx\_point1, cx\_point2)**

To fill both children using the mappings from the opposite parent, the pmx\_fill function is called twice.

* **return child1, child2**

Finally, the two offspring created from the crossover of ‘parent1’ and ‘parent2’ is returned

**Results**

A screenshot of a computer program

Description automatically generated

**TODO 6 (Mutation)**

Mutation is the changes of sequence of the city in the route for this case.

**if (random.random() < mutationProbability):**

**geneIndex = random.randint(0, len(mutated\_route) - 1)  
gene = mutated\_route.pop(geneIndex)**

**insertIndex = random.randint(0, len(mutated\_route))  
mutated\_route.insert(insertIndex, gene)**

random.random() will generate a number and if the mutationProbability is larger than number that has been generated, mutation occurs. The higher the mutationProbability, the higher of chances that mutation occurs. A random gene will be selected and will be removed from the route. It will then randomly be inserted into the route.

A screenshot of a computer program

Description automatically generated

From this sample we can see the mutation will 100% occurs where mutate(route, 1), mutationProbabilty is 1. All of the city in the route will move to a random location.

**Performance Evaluation Explanation**

* parentSelection\_functions = [parentSelection, parentSelectionTournament, parentSelectionProportional]
* parentSelection\_names = ['Random Selection', 'Tournament Selection', 'Proportional Selection']

The two lines is to label a name for each of the parent selection method, which are random selection, tournament selection, and proportional selection.

* all\_best\_distances = []

This line initializes an empty array that will eventually hold the selected best distances individuals.

* for i in range(3):
* population = initialPopulation(popSize, cityList)
* distances = [Fitness(p).routeDistance() for p in population]
* min\_dist = min(distances)
* print("Best distance for initial population: " + str(min\_dist))
* for j in range(iteration\_limit):
* population = oneGeneration(population, eliteSize, mutationProbability)
* distances = [Fitness(p).routeDistance() for p in population]
* index = np.argmin(distances)
* best\_route = population[index]
* min\_dist = min(distances)
* print("Best distance for population in iteration " + str(j) +
* ": " + str(min\_dist))
* print("Optimal path is " + str(best\_route))
* all\_best\_distances.append(min\_dist)

**Generally, the objective of code as follows:**

This is a nested for loop. We first set the loop to be run this generic algorithm for three times to find the optimal route which are the approximated result. Then, we initialize a population of potential solutions which are the routes. After, we evolve the population over a specified number of iteration limit, known as generation. Thus, we track the best solution found during each generation and after the entire run.

**Code Explanation for each line:**

We first set the outer for loop to runs the genetic algorithms for three times to generate three different solutions. Then, a population of ‘popsize’ route is created using the method of ‘initialPopulation’. In the next line, we use ‘Fitness(p).routeDistance() for p in population’ to calculate the fitness which are the distances of all the routes in the population. After that, we find the best initial route by using the function min(), because the shortest route is our target to find out as our result to solve the problem as state in this assignment at first, which is to solve the travelling salesman problem. Thus, we print out the minimum distances that we found.

Next, we had created an inner loop which a iteration limit of generations at first. Then, we are using the one generation function to update the population for each of the generation which meant that whenever the loop iterates one time the population will update for once. After, we update the best route too as each time of iteration. Therefore, the corresponding minimum distance will be update too. Lastly, we get the best distance per iteration for (iteration limit) n times, and print out the optimal path for all the best distance, then store into the array (all\_best\_distance) we initialize previously.

**Showing performance evaluation of all parent selection functions in a graph:**

* plt.plot(parentSelection\_names, all\_best\_distances)
* plt.xlabel('Parent Selection Function')
* plt.ylabel('Best Distance')
* plt.title('Performance Evaluation of Parent Selection Functions')
* plt.show()

These few lines of code are used to plot a line graph which investigate the difference of best distance output from three different parent selections.

**Performance evaluation of each parent selection functions:**

* all\_best\_distances = []
* all\_fitness\_over\_generations = {name: [] for name in
* parentSelection\_names}

The first line of code is to initialize a list to store the shortest distance found in each complete run of genetic algorithms. The second line is used as a dictionary where each key corresponds to a parent selection such as random selection, tournament selection, and proportional selection.

* for selection\_function, name in zip(parentSelection\_functions, parentSelection\_names):
* print(f"Evaluating {name}...")
* # Initialize population
* cityList = genCityList('cities500.txt')
* population = initialPopulation(popSize, cityList)
* for generation in range(iteration\_limit):
* distances = [Fitness(p).routeDistance() for p in population]
* min\_dist = min(distances)
* all\_fitness\_over\_generations[name].append(min\_dist)
* print(f"Generation {generation}: Best distance = {min\_dist}")
* # Evolve the population using the current selection function
* population = oneGeneration(population, eliteSize, mutationProbability)
* # Capture the final best distance
* distances = [Fitness(p).routeDistance() for p in population]
* min\_dist = min(distances)
* all\_best\_distances.append(min\_dist)
* print(f"Optimal path for {name}: {min\_dist}")

The first line is to assign the name of parent selection to the corresponding parent selection functions. Then, the for loop iterate over these pairs, to allow the genetic algorithm to evaluate for each selection function separately.

After, we are using the genCityList() method to read the text file which is ‘cities500.txt’. Then, we initialize the population by assigning the cityList and having flexibility to change the ‘popSize’ parameter which indicate the percentage of size to be randomly choose from the cityList.

Next part, we loop the genetic algorithms by a specified number of generations (nth generation). When looping, the distance for each route in the population is being computed and the shortest distance is identified in every round of generation. Thus, the shortest distance in every generation will be append into the list associated with the parent selection used in every generation, must be tally before stored into the list. Then, we print out all the best distances in every generation.

Based on the ‘oneGeneration()’ method, we evolved the population by using current selected function in each generation. Then, we selected the number of top-performing individuals which preserved unchanged in every next generation (elite size) and probability that any given individual will undergo mutation (mutation probability).

After iterating all the generation, then we calculate the distance for the final population. Then, we found out the shortest route in the final population. After, we append the final best distance into the list named ‘all\_best\_distances’ and the list will keep track of the best results from all selection function. Lastly, we print all the best results of different selection method as optimal path.

* plt.figure(figsize=(12, 6))
* plt.plot(parentSelection\_names, all\_best\_distances, marker='o', linestyle='-', color='b')
* plt.xlabel('Parent Selection Function')
* plt.ylabel('Best Distance')
* plt.title('Performance Evaluation of Parent Selection Functions')
* plt.show()

This is to plot the best distances from all parent selection function. Then, we plot a line graph with best distance to visualize which of the following method is the best. Then, we label for x-axis as parent selection method, y-axis as best distance, and label title associated.

* plt.figure(figsize=(12, 6))
* for name in parentSelection\_names:
* plt.plot(all\_fitness\_over\_generations[name], label=name)
* plt.xlabel('Generation')
* plt.ylabel('Best Distance')
* plt.title('Fitness Over Generations for Different Parent Selection Functions')
* plt.legend()
* plt.show()

This is to plot a fitness value over generations for each selection function. Then, we plot a multilinear graph with three different methods with labelling. Then, we label for x-axis as generation, y-axis as the best distance and label the title as fitness over generations for different parent selection functions. Then, use legend to differentiate those three lines in a same graph and needed to be compared to check which is better.

**Best Distance to its next city**

best\_city = None

best\_distance = float('inf')

for i in range(len(best\_route)):

    from\_city = best\_route[i]

    to\_city = best\_route[(i + 1) % len(best\_route)]  # Loop back to the first city

    distance = from\_city.distance(to\_city)

    if distance < best\_distance:

        best\_distance = distance

        best\_city = from\_city

This code can effectively find the city in the route that perform the shortest distance to its next city by comparing all city to next city distances in the route. After the loop is completed, ‘best\_city’ will store all the city with the shortest distance to its next city.

# Variable to keep track of the best route

best\_route\_overall = None

best\_distance\_overall = float('inf')

for i in range(iteration\_limit):

    population = oneGeneration(population, eliteSize, mutationProbability)

    distances = [Fitness(p).routeDistance() for p in population]

    index = np.argmin(distances)

    best\_route = population[index]

    min\_dist = min(distances)

    print("Best distance for population in iteration " + str(i) + ": " + str(min\_dist))

# Update the overall best route and distance if the current best is better

    if min\_dist < best\_distance\_overall:

        best\_distance\_overall = min\_dist

        best\_route\_overall = best\_route

In order to identify the quickest path, this method iteratively improves a population of routes. It creates a fresh population, assesses route distances, and tracks the shortest path discovered for every iteration. The method modifies the overall best route and distance if a route in the current iteration shows to be greater to the best one discovered so far. In order to gradually optimize the route, this process is carried out until the loop reaches its maximum number of iterations.

city\_id\_map = {city: city\_id for city\_id, city in enumerate(cityList)}

This line is useful for quickly finding a city’s ID, based on reference cities by numeric index. Thus, we can search for the cities allocate faster, instead of we are using city name to allocate.

def extractCityIDs(route):

    return [city\_id\_map[city] for city in route]

This function is to extract the city’s ID from a route.

best\_route\_ids = extractCityIDs(best\_route\_overall)

This line is to assign the best route from overall to this variable named ‘best\_route\_ids’.

print("Optimal path (City IDs) is: " + str(best\_route\_ids))

print("Best distance: " + str(best\_distance\_overall))

This two line is to print out the optimal path and shortest path of city to its next city.