**Lab 2: Genetic Algorithm Report**

**UECS2153 P2 August 2025, Monday 2-5 PM**

Team Members

|  |  |  |
| --- | --- | --- |
| **Name** | **ID** | **Program** |
| Sim Jing Jia |  | SE |
| Tan Kye Wen |  | SE |
| Thanirmalai |  | SE |
| Adrin Yong Yi Kai |  | SE |

**TODO1 POPULATION INITIALISATION**

The below code reads a set of cities from the filename to be used for creating an initial random population.

|  |
| --- |
| NUM\_RANDOM\_CITIES = 10      cities = [] |

Two variables are declared NUM\_RAMDON\_CITIES to specify how much cities are to be generated and a list to store the initial list of cities.

|  |
| --- |
| # Read cities from txt file      with open("./" + filename, "r") as f:          csv\_reader = csv.DictReader(f)          for row in csv\_reader:              cities.append(City(row["City"], int(row["X"]), int(row["Y"]))) |

The code above reads the text file that is pass to this function and splits the data into three columns which are “City”, “X”, and “Y”. “City” indicates the city name, “X” indicates the x-coordinate and “Y” is the y-coordinate

|  |
| --- |
| cityList = random.sample(cities, NUM\_RANDOM\_CITIES) |

Then a number of random cities are appended into the cityList.

**TODO2 TOURNAMENT SELECTION**

This section will explain the code used for Tournament Selection which is a type of Genetic Algorithm based on a tournament based system where the highest fitness of the selected specimen will be selected for the mating. The function named is changed from parentSelection to parentTournamentSelection.

|  |
| --- |
| def parentTournamentSelection(population, poolSize=None): |

|  |
| --- |
| k = 3        for i in range(0 , poolSize):          # Select k random chromosome from population          kChoices = random.sample(population, k)          # Find the best fit chromosome among k chromosome          highestFit = kChoices[0]          for choice in kChoices:              if Fitness(choice).routeFitness() > Fitness(highestFit).routeFitness():                  highestFit = choice            matingPool.append(highestFit) |

This function loops through the poolSize and 3 random specimens are chosen from the population, then another loop is used to find the highest fitness value from the 3 selected specimens. Following that the specimen with the higher fitness value is appended into the matingPool.

**TODO3 PROPORTIONAL SELECTION**

This section will explain the code used for 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. The function named is changed from parentSelection to parentProportionalSelection.

|  |
| --- |
| def parentProportionalSelection(population, poolSize=None): |

|  |
| --- |
| if self.fitness == None:              if self.routeDistance() == 0:                  self.fitness = math.inf                  return self.fitness |

This code is added into routeFitness(self) in the class Fitness to handles edge cases such as divide by 0

|  |
| --- |
| weights = []        # Get the fitness weights      for pop in population:          weights.append(Fitness(pop).routeFitness())        # Select the poolSize parents based on the roulette wheel      matingPool = random.choices(population, weights=weights, k=poolSize) |

This function will loop through the entire population and calculate fitness using routeFitness then appends the highest proportional fitness using a built-in python library for weighted random sampling based on the population and the weights provided by the aforementioned weights[] variable.

**TODO4 SURVIVAL SELECTION**

This section will explain the code used for Survival Selection which is a type of Genetic Algorithm based on a survival of the fittest system where a select few of the highest fitness value will be able to mate for the next generation. The function named is changed from parentSelection to survivalSelection.

|  |
| --- |
| def survivorSelection(population, eliteSize): |

|  |
| --- |
| popWithFitness = []      for pop in population:          popWithFitness.append({"pop": pop, "fitness": Fitness(pop).routeFitness() })        # Sort the chromosome by the highest fitness score      popWithFitness = sorted(popWithFitness, key=lambda pop: pop["fitness"], reverse=True)      # Select the top eliteSize chromosomes      for i in range(eliteSize):          elites.append(popWithFitness[i]["pop"]) |

The above code will append the list of cities route with the fitness level to a list named popWithFitness, then it will be sorted from highest fitness to lowest fitness value. Lastly it will then choose the highest fitness values based on the determined eliteSize into the elites[] list for the elites to mate and generate the next generation.

**TODO5 CROSSOVER**

This section will explain the crossover code. The crossover function is for the selected parents to breed for the next generation which will hopefully generate a better solution. Let’s split this function into a couple of steps.

|  |
| --- |
| total\_len = len(parent1)      crossover1, crossover2 = sorted(random.sample(range(1 , total\_len - 1), 2)) |

There is one assumption where both parents will be the same length, thus a total\_len will be declared to get the length of the parents. Following that the 2 crossover points will be randomly generated using random.sample which is a built in python library. The crossover points will only be after the first genetic and before the last genetic, Eg: assume genetic code = 1 2 3 4, let “|” be the crossover point, the code will only generate 1 | 2 3 | 4. And the code will ensure crossover2 > crossover1.

|  |
| --- |
| child1 = [-1] \* total\_len      child1[crossover1: crossover2 + 1] = parent1[crossover1: crossover2 + 1]      child2 = [-1] \* total\_len      child2[crossover1: crossover2 + 1] = parent2[crossover1: crossover2 + 1] |

The code above will initialize child1 and child2 with an empty gene pool and the gene between crossover1 and crossover2 will be copy from parent1 to child1 and parent2 to child2.

|  |
| --- |
| def pmx (child, childsParent, otherParent):          for i in range(0, crossover1):              city = otherParent[i]              while city in childsParent[crossover1: crossover2 + 1]:                  city = otherParent[childsParent.index(city)]              child[i] = city          for i in range(crossover2 + 1, total\_len):              city = otherParent[i]              while city in childsParent[crossover1: crossover2 + 1]:                  city = otherParent[childsParent.index(city)]              child[i] = city |

A function named pmx will be define and with arguments child, childsParent, and otherParent. Firstly it will loop through the start of the child to crossover1 and check if there are any repeated genes, if there is a repeat it will get the corresponding gene from the other parent to replace it, it will loop until all genes are unique.

Following that, it will do the same for crossover2 to the end of the gene pool of the child

|  |
| --- |
| pmx(child1, parent1, parent2)      pmx(child2, parent2, parent1) |

Then it will call the pmx using child1 followed by child2

**TODO6 MUTATION**

This section will explain the mutation code. The mutation function will randomly mutate a gene based on a set probability defined later on.

|  |
| --- |
| for i in range(len(mutated\_route)):          if (random.random() < mutationProbability):              # mutationProbability is the probability of a gene undergoing mutation                # Replacement starts here              # Get a random index from the route array              idx = random.randint(0 , len(mutated\_route) - 1)              # Remove the selected              item = route.pop(idx)              # Insert              idx2 = random.randint(0 , len(mutated\_route) - 1)              mutated\_route.insert(idx2 , item) |

The function will loop through the route and generate a random seed and check if the seed is lower then the mutation probability, if seed < mutation probability then it will start the mutation. Firstly, a random index is generated and defined as idx. Following that, we will remove the gene from said index into item. Furthermore, another gene will be randomly selected and defined as idx2. Lastly it will replace idx2 with the original idx.

**TODO7 PERFORMANCE EVALUATION**

This section will explain the code used for performance evaluation and explain the data based on the results.

|  |
| --- |
| filename = 'cities400.txt'  popSize = 100  eliteSize = 5  mutationProbability = 0.01  iteration\_limit = 250  iteration\_limit\_map = []  for i in range(100, 500, 100):      iteration\_limit\_map.append(i) |

There will be several variable defined but the most important metric is the popSize which indicates the population size, eliteSize indicates the elites that will be able to breed in survivalSelection, the mutationProbability, and iteration\_limit\_map as this will run 4 times with 4 different iteration limit which are 100, 200 until 400.

|  |
| --- |
| mapOfAlgos = {      "random": parentRandomSelection,      "proportional": parentProportionalSelection,      "tournament": parentTournamentSelection,  }  results = []  params = []  for it in iteration\_limit\_map:      for key in mapOfAlgos:          params.append({              "algo": key,              "iterationLimit": it,              "popSize": popSize,          }) |

mapOfAlgos will hold the three different selection algorithms, results is to store the iteration and its corresponding distance for data visualization, and params is to handle the mapping of 4 different iterations limits with the algorithms.

|  |
| --- |
| def evaluatePerformance(cityList: list[City], params):      print("RUNNING PARAMS", params)      parentSelection = mapOfAlgos[params["algo"]]      population = initialPopulation(params["popSize"], cityList)      distances = [Fitness(p).routeDistance() for p in population]      min\_dist = min(distances)      for i in range(params["iterationLimit"]):          population = oneGeneration(population, eliteSize, mutationProbability, parentSelection)          distances = [Fitness(p).routeDistance() for p in population]          index = np.argmin(distances)          best\_route = population[index]          min\_dist = min(distances)          results.append({              "iteration": i,              "distances": distances,              "algo": params["algo"],          })      # print("Optimal path is " + str(best\_route))      print(f"BEST DISTANCE for {params} : ", min\_dist, "\n" )      return min\_dist |

A function evaluatePerformance will be define and lets break it down.

|  |
| --- |
| print("RUNNING PARAMS", params)      parentSelection = mapOfAlgos[params["algo"]]      population = initialPopulation(params["popSize"], cityList)      distances = [Fitness(p).routeDistance() for p in population]      min\_dist = min(distances) |

Some variables will defined, parentSelection is to chose the selection algorithm based on the mapOfAlgos, population is to initialize the population define in TODO 1, distances is to calculate the distance that was generated for the gene. And min\_dist is to find the shortest route in the current population.

|  |
| --- |
| for i in range(params["iterationLimit"]):          population = oneGeneration(population, eliteSize, mutationProbability, parentSelection)          distances = [Fitness(p).routeDistance() for p in population]          index = np.argmin(distances)          best\_route = population[index]          min\_dist = min(distances)          results.append({              "iteration": i,              "distances": distances,              "algo": params["algo"],          }) |

The function will then loop through the algo based on the iterationLimit. Population will then evolve, breed, and or mutate the new population based on the selected algorithm. Then the index, best\_route and min\_dist is to store the best route / the shortest route in the population. Lastly, the results for each iteration will be appended to the results list.

|  |
| --- |
| def generateScatterChart(result, algo = mapOfAlgos):      iteration\_data = [gen["iteration"] for gen in result]      distances\_data = [min(gen["distances"]) for gen in result]      algo\_names = list(algo.keys())      colors = ['red', 'blue', 'green']      plt.figure(figsize=(12, 6))        for i, algo\_name in enumerate(algo\_names[:3]):  # Limit to 3 algorithms          # Filter results for this specific algorithm          algo\_results = [r for r in result if r.get("algo") == algo\_name]            if algo\_results:  # Only plot if we have data for this algorithm              iteration\_data = [gen["iteration"] for gen in algo\_results]              distances\_data = [min(gen["distances"]) for gen in algo\_results]                plt.plot(iteration\_data, distances\_data,                      color=colors[i],                      alpha=0.8,                      label=f'{algo\_name}',                      linewidth=2)        plt.xlabel('Generation', fontsize=12, fontweight='bold')      plt.ylabel('Distance', fontsize=12, fontweight='bold')      plt.title('Algorithm Performance Comparison', fontsize=14, fontweight='bold')      plt.legend(loc='best', frameon=True, fancybox=True, shadow=True)      plt.grid(True, alpha=0.3)      plt.show() |

This function named generateScatterChart will generate a chart with the three algorithm based on the iteration limit.

A graph showing a number of data

AI-generated content may be incorrect.

|  |
| --- |
| cityList = genCityList(filename)  iteration = []  for i, param in enumerate(params):      min\_dist = evaluatePerformance(cityList, param)      iteration.append({"param": param, "dist": min\_dist})      if (i+1) % 3 == 0:          generateScatterChart(results, mapOfAlgos)          results.clear()  print(iteration) |

Lastly, this is to print the result of the algorithms, its minimal distance and the iteration limit

A screen shot of a computer

AI-generated content may be incorrect.