Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)
DU012(A)

UNIVERSITI TUNKU ABDUL RAHMAN

LEE KONG CHIAN FACULTY OF ENGINEERING & SCIENCE (LKCFES)

Course Description	UECS2053 Artificial Intelligence
Title	Lab Report 2: Genetic Algorithm
Lecturer	Dr Shalini a/p Darmaraju
Practical Group	P3-G2
Date of submission	01/08/2024

Group Member:

No	Name	ID	Course	Year/Trimester
1	Lai Jien Weng	2104338	AM	Y2T3
2	Tan Wan Xuen	2207214	AM	Y2T3
3	Janice Ng Zhi Yan	2104783	AM	Y2T3
4				

Introduction

The Travelling Salesman Problem (TSP) is a classic optimization problem where the goal is to find the shortest possible route that visits each city once and returns to the starting city. It will be impossible to find the shortest travelling distance by exhaustive search as the time and space complexity of this problem increase exponentially. The genetic algorithm (GA) simulates the process of natural selection to iteratively improve the solution in a feasible period. This report covers various aspects of the GA, including population initialization, parent selection methods, survival selection, crossover, and mutation. Additionally, the performance of different parent selection methods (Random Selection, Tournament Selection, and Proportional Selection) is evaluated and compared.

TODO 1

Population Initialization

```
43      with open(filename, "r") as file:       # Read as corresponding file as "file"
44          reader = csv.reader(file)           # Read "file" in csv format
45
46          next(reader)                        # Skip the header row
47
48          for line in reader:                 # Iterate for each line in "csv"
49              _, x, y = map(int, line)        # Mapping x and y into int datatype
50              cityList.append(City(x, y))     # Append City object to the cityList with corresponding x, y value
```

This code reads a CSV file containing city coordinates, skips the header row, and populates a list with City objects created from the CSV data. This approach efficiently processes the input data for further use in the program.

TODO 2

Parent Selection

Tournament Selection is a popular method for choosing individuals from a population for reproduction, this is a selection operator widely used in GAs and is known for its simplicity and ease of controlling the selection pressure (Viescinski, 2024).

```
    if poolSize == None:
        poolSize = len(population)
    matingPool = []
    # Replacement starts here
    # Determine the tournament size as 10% of the population size, with a minimum of 2
    tournamentSize = max(2, math.ceil(len(population) / 10))

    for _ in range(poolSize):
        # Randomly select individuals for the tournament
        tournament = random.sample(population, tournamentSize)

        # Select the individual with the best (minimum) route distance from the tournament
        best = min(tournament, key=lambda route: Fitness(route).routeDistance())

        # Add the selected individual to the mating pool
        matingPool.append(best)
    # Replacement ends here
    return matingPool
```

The process begins by checking if the ‘poolSize’ parameter is provided; if not, it defaults to the size of the entire population, meaning the mating pool will be filled with as

many individuals as are currently in the population. The mating pool is then initialized as an empty list.

Next, the function sets the size of each tournament to be 10% of the population size and rounded up with a minimum size of 2. This ensures that at least two individuals are competing in each tournament.

For each selection round needed to fill the mating pool, the function randomly selects individuals from the population to participate in a tournament. The individual with the best fitness, which is the one with the shortest route distance calculated by a Fitness class among those in the tournament is chosen as a winner. This winner is then added to the mating pool.

This selection process is repeated until the mating pool is filled to the desired size. The mating pool, containing the selected individuals, is ultimately returned. This method favors individuals with higher fitness, thus steering the population towards better solutions over successive generations.

TODO 3

Parent Selection

Furthermore, one of the parent selections is proportional selection, also known as Roulette Wheel Selection. It aims to maintain diversity and produce better offspring (optimal solutions). Depending on the selection probability, this method assigns each individual a proportional slice of wheel. The probability of selecting a chromosome is directly proportional to its fitness (*Wheel Selection - an Overview | ScienceDirect Topics*, n.d.).

```
matingPool = []
fitness_results = []

# Fitness of each route in the population is calculated
for individual in population:
    fitness = Fitness(individual).routeFitness()
    fitness_results.append((individual, fitness))

# Routes based on fitness in descending order are sorted
# Higher fitness is better
ranked_routes = sorted(fitness_results, key=lambda x: x[1], reverse=True)

# Roulette wheel is set up
df = pd.DataFrame(np.array(ranked_routes, dtype=object), columns=["Route", "Fitness"])
df['cum_sum'] = df.Fitness.cumsum()
df['cum_perc'] = 100 * df.cum_sum / df.Fitness.sum()

# Individuals for the mating pool are selected
while len(matingPool) < poolSize:
    pick = 100 * random.random()  # "Spin the wheel" to generate values
    for i in range(len(df)):
        if pick <= df.iat[i, 3]:  # Survivor is selected based on value generated
            matingPool.append(df.iat[i, 0])
            break

return matingPool
```

Above codes initiate with two empty list, '*matingPool*' and '*fitness_result*'. Fitness is then calculated by calling the function '*routeFitness*' and iteratively looping over every individual in the population. The calculation results are being stored in the '*fitness_result*' list. To facilitate the selection process, the fitness values are sorted in descending order. The higher the fitness values, the higher the probability of being selected. Furthermore, the Roulette wheel

is set up by creating a data frame with two columns, namely ‘Route’ and ‘Fitness’. To clarify the cumulative sum of fitness values and cumulative percentage of fitness values, two more columns, ‘*cum_sum’* and ‘*cum_prec’*, are generated. After setting up the Roulette wheel, the wheel is “spun” by picking a random number between 0 and 100. If the random number is less than or equal to the cumulative percentage of an individual, the individual is chosen and added to the mating pool. Lastly, the individuals selected will be shown when the function ‘*parentSelection’* is being called, as ‘*matingPool’* is the returned.

TODO 4

Survival Selection

The genetic algorithm simulates the natural selection process by selecting individuals with the highest fitness value among the existing population for each future generation (Bibek Rawat, 2022). The function aims to retain the top-performing individuals (elites) in the population, ensuring that their genetic material is carried forward into subsequent generations.

```
    # Replacement starts here

    # Merge chromosome with their fitness score into new list
    population_with_fitness = [(chromosome, Fitness(chromosome).routeFitness()) for chromosome in population]


    # Sort the chromosome based on their fitness score in descending order
    population_with_fitness.sort(key=lambda x: x[1], reverse=True)

    # Select the top eliteSize number of chromosomes
    elites = [chromosome for chromosome, _ in population_with_fitness[:eliteSize]]

    # Replacement ends here
```

Each individual in the population is paired with its fitness score using the ‘*Fitness’* class. The population is then sorted in descending order based on fitness scores. Higher fitness scores indicate better-performing individuals. The top ‘*eliteSize’* individuals are selected based on their fitness scores. These elites are extracted from the sorted list and returned.

The ‘*survivorSelection*’ function ensures that the genetic algorithm retains the best-performing individuals in each generation. By merging the population with their fitness scores, sorting them, and truncating the list to retain only the top elites. This approach effectively promotes the preservation of high-quality genetic material, which is essential for the convergence and optimization of the algorithm.

TODO 5

Crossover

Crossover is a method that mates two parents and produces one or more children and to be specific, it involves exchanging genetic material to produce improved products (Baeldung 2024).

```python
    # Replacement starts here
    size = len(parent1)
    point1, point2 = sorted(random.sample(range(size), 2))

    # Initialize offspring with None
    child1 = [None] * size
    child2 = [None] * size

    # Copy the segment from parents to children
    child1[point1:point2+1] = parent1[point1:point2+1]
    child2[point1:point2+1] = parent2[point1:point2+1]

    # Function to fill the remaining positions in child
    def fill_child(child, parent, point1, point2):
        # Copy the non-segment part of parent to the child
        current_index = (point2 + 1) % size
        for i in range(size):
            if i < point1 or i > point2:
                while parent[current_index] in child:
                    current_index = (current_index + 1) % size
                child[i] = parent[current_index]
                current_index = (current_index + 1) % size

    fill_child(child1, parent2, point1, point2)
    fill_child(child2, parent1, point1, point2)
    # Replacement ends here

    return child1, child2
```

The function starts by defining the size of the parents and randomly selecting two crossover points within this range. These points are used to determine a segment of each parent that will be directly copied to the corresponding position in the offspring, ensuring that each child inherits a specific part of the genetic material directly from one of the parents.

In order to complete the offspring solutions, the function initializes two child arrays filled with *'None'* to indicate unset positions. After copying segments from the parents to the children, the remaining positions in each child are filled using a helper function named *'fill_child'*. This function iteratively selects elements from the opposite parent and inserts them into the child, ensuring no duplicates and maintaining the integrity of a valid permutation. The process of selecting elements from the opposite parent is determined by checking if an element has already been placed in the child. If an element has been placed, the search for the next available element continues.

This mechanism helps in preserving the relative order and positions of the elements from the parents while ensuring that the offspring are viable solutions that could potentially combine the strengths of both parents.

This crossover approach is particularly effective in problems where the order of elements is crucial, such as routing or scheduling issues, as it tends to maintain a high degree of inherited quality from the parent solutions while introducing enough variation to explore new potential solutions in the solution space. The resulting offspring are then returned, ready to be used in subsequent generations of the algorithm.

TODO 6

Mutation

Moreover, mutation is a random small tweak made between the genes to produce new solutions (*Autoblocks: Full-Stack Monitoring, Debugging, and Testing*, n.d.). It promotes diversity and prevent premature convergence.

```
mutated_route = route[:]

# Perform insertion mutation
for i in range(len(route)):
    if random.random() < mutationProbability:
        # Choose a gene to move
        gene_to_move_index = random.randint(0, len(mutated_route) - 1)
        gene_to_move = mutated_route.pop(gene_to_move_index)

        # Choose a new position to insert the gene
        new_position = random.randint(0, len(mutated_route))

        # Insert the gene at the new position
        mutated_route.insert(new_position, gene_to_move)

return mutated_route
```

Above codes are used to perform the insertion mutation. Initially, a copy of the chromosomes is generated so that the original route remains unchanged. To carry out insertion mutation, for loop is used to iterates over each gene in the route. If the random number between 0 and 1 synthesised is less than the mutation probability, the gene chosen will be removed. A new position is then assigned by randomly choosing a position within the range of the length of *'mutated_route'*. Since the new position is chosen, then the gene is inserted into a new position.

```
route = genCityList('cities50.txt')
mutated = mutate(route, 1)
# Give a pretty high chance for mutation
print('Original route')
print(route)
print('Mutated route')
print(mutated)
```

The file 'cities50.txt' is read to generate a list of city coordinates. The function 'mutate' is then called, and the original route and mutated route are listed out. Upon comparison, the chromosomes successfully undergo mutation. However, not all chromosomes undergo mutation to balance the exploration and exploitation in the search space.

TODO 7

Performance evaluation

```python
def runGA(parentSelection, cityList, popSize=20, eliteSize=5, mutationProbability=0.01, iteration_limit=100):
    # Initialize the population
    population = initialPopulation(popSize, cityList)
    # List to store the best distance in each iteration
    best_distances = []

    # Iterate for a specified number of generations
    for i in range(iteration_limit):
        # Generate the next generation of the population
        population = oneGeneration(population, eliteSize, mutationProbability, parentSelection)
        # Calculate distances for each route in the population
        distances = [Fitness(p).routeDistance() for p in population]
        # Find the index of the best route (minimum distance)
        index = np.argmin(distances)
        # Retrieve the best route
        best_route = population[index]
        # Retrieve the best distance
        min_dist = min(distances)
        # Store the best distance
        best_distances.append(min_dist)
        # Print the best distance for the current iteration
        print(f"Iteration {i}, Best distance: {min_dist}")

    # Print the optimal path found
    print("Optimal path is " + str(best_route))

    return best_distances
```

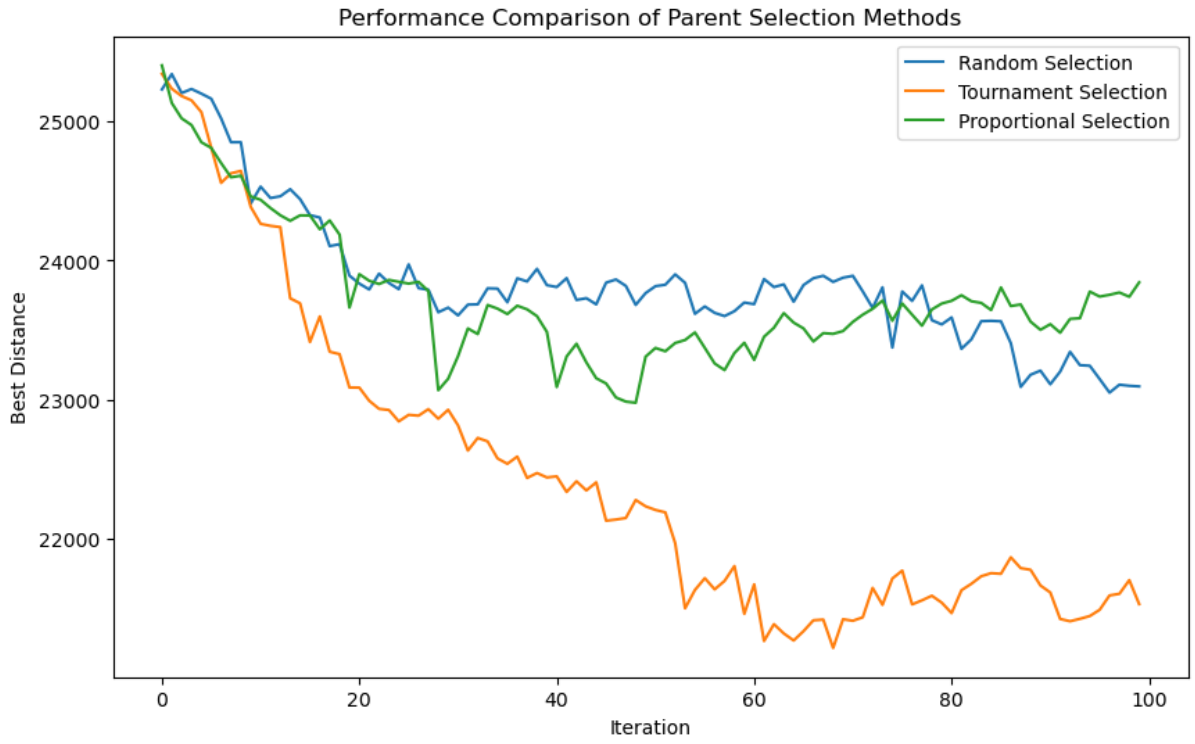
We initialize the population using *initialPopulation*. Then, we iteratively evolve the population through the *oneGeneration* function. In each iteration, calculates the route distances for the current population, identifies the best route, and records the best distance. Then, it prints the best distance for each iteration.

```python
filename = 'cities500.txt'
popSize = 20
eliteSize = 5
mutationProbability = 0.01
iteration_limit = 100

# Generate the list of cities from the "cities500.txt"
cityList = genCityList(filename)

# Run the GA with Random Selection
random_results = runGA(randomSelection, cityList, popSize, eliteSize, mutationProbability, iteration_limit)
# Run the GA with Tournament Selection
tournament_results = runGA(tournamentSelection, cityList, popSize, eliteSize, mutationProbability, iteration_limit)
# Run the GA with Proportional Selection
proportional_results = runGA(proportionalSelection, cityList, popSize, eliteSize, mutationProbability, iteration_limit)
```

The genetic algorithm was run with a population size of 20, elite size of 5, mutation probability of 0.01, and iteration limit of 100. The *genCityList* function generated the initial list of cities from the "*cities500.txt*" file. We run the genetic algorithm with 3 different parent selections to compare their performances.

```python
# Plotting the results
plt.figure(figsize=(10, 6))
# Plot the results for Random Selection
plt.plot(random_results, label="Random Selection")
# Plot the results for Tournament Selection
plt.plot(tournament_results, label="Tournament Selection")
# Plot the results for Proportional Selection
plt.plot(proportional_results, label="Proportional Selection")
plt.xlabel('Iteration')
plt.ylabel('Best Distance')
plt.title('Performance Comparison of Parent Selection Methods')
plt.legend()
plt.show()
```

Then, we visualize the performance of each parent selections method by line graph using *matplotlib.*



We see that tournament selection outperformed the other 2 parent selections methods in this case. In the first 100 iterations, tournament selection could find the shortest, best distance, giving the shortest route possible to address the travelling salesman problem (TSP).

Conclusion

In this report, we successfully implemented a Genetic Algorithm to address the Traveling Salesman Problem. The algorithm involved several key steps: population initialization, parent selection, survival selection, crossover, and mutation. Through performance evaluation, we compared three parent selection methods: Random Selection, Tournament Selection, and Proportional Selection. Among these, Tournament Selection demonstrated the best performance, achieving the shortest, best distance within the first 100 iterations.

References

Autoblocks: Full-stack monitoring, debugging, and testing. (n.d.). Www.autoblocks.ai. https://www.autoblocks.ai/glossary/mutation

Baeldung (2024).*Partially mapped crossover in genetic algorithms.* Retrieved from https://www.baeldung.com/cs/ga-pmx-operator

Bibek Rawat, D. D. (2022). A Comparative Review Between Various Selection Techniques In. *International Journal of Computer Sciences and Engineering*, (p. 15). India.

Viescinski, A. (2024). *Tournament Selection in Genetic Algorithms.* Retrieved from https://www.baeldung.com/cs/ga-tournament-selection

Wheel Selection - an overview | ScienceDirect Topics. (n.d.). Www.sciencedirect.com. https://www.sciencedirect.com/topics/computer-science/wheel-selection