AI Project Report

CS21B1061 Ch Siva Kashyap

CS21B1065 Venkat Sai Tarra

Topic: Travelling Salesman Problem using Genetic Algorithm

Genetic Algorithm:

Genetic algorithm (GA) consists of two steps. The initial step is the selection of individuals for the purpose of generating the new generation, while the second step is manipulating the chosen individuals to create the following generation using procedures like as crossover and mutation.

The individuals that are chosen for mating are determined by the selection mechanism. The primary aim of selection strategy is that "the better an individual, the higher the possibility that it will be a parent". Generally, crossover and mutation explore the search space, whereas selection reduces the search area within the population by discarding poor solution.

Different selection strategies used in the GA process will have a significant impact on the way the algorithm performs.

The Objective we are looking for is to examine the performance of GA when using different selection strategy methods especially in solving the travelling salesman problem (TSP), a classic example for NP-Hard problem.

- Fitness value/score is a measure of how good a particular solution to a problem fits the constraints of the problem. And Fitness function is used to Check the Quality of individual solution. The goal of GA is to go with individual with Better Fitness value.

The algorithm evolves the generation using after the initial generation is produced using the operators:

- Selection Operator
- Crossover Operator
- Mutation Operator

Selection Operator:

The Selection Operator selects members with fitness value high should have the high probability of mutate, but that poor individuals of the population still have a small probability of being selected, and this is important to ensure that the search process is global and does not simply converge to the nearest local optimum. Selection probability will be based on the “The Survival of the Fittest.”

There are two methods of selection strategies:

- Roulette Wheel Selection
- Tournament Selection

1.Roulette Wheel Selection:

In a proportional roulette wheel, each individual's selection corresponds to a particular region of the wheel, with a probability that is directly proportional to their fitness scores. The probability of choosing a parent can be equivalent to spinning a roulette wheel, with the size of each segment corresponding to the parent's fitness.

Those with the highest fitness are more likely to be selected. The total fitness values of the people make up the roulette wheel's circumference.

All segments have a chance, with a probability proportional to their width. By repeating this each time, the better individuals will be chosen more often than less width individuals.

Let f1, f2…, fn be fitness values of individual 1, 2…, n.

Then, selection probability pi is

$pi = fi/\Sigma (fi)$

- Advantage of proportional roulette wheel selection is that it gives a chance to all of them to be selected. And preserve the diversity among the population. Although individual with large fitness value gets more chance.
- Disadvantage of proportional roulette wheel is if the individual have the same fitness value then it will become very difficult for the population to select a better one. Since selection probabilities for fit and unfit individuals are very similar. The fitness function for minimization must be changed to a maximization function, as in the case of TSP, making it challenging to apply this selection technique to minimization problems. Although this somewhat resolves the selection problem.

2.Tournament Based Selection:

Tournament selection is the most popular selection method in genetic algorithm due to its efficiency and simple implementation.

In tournament selection, n people are chosen at random from the greater population, and the individuals selected then compete with others in the group/tournament. The individual with highest fitness goes to the next generation population.

The idea of "tournament size" means the number of competitors in each tournament, which usually equals two individuals.

Also, the selection process used in tournaments allows all candidates to be chosen, maintaining diversity, even though it decreases convergence.

- Advantage of tournament Selection is its efficient time complexity, especially if implemented in parallel, low chance domination of dominant individuals, and no requirement for fitness scaling or sorting.
- Disadvantage of tournament Selection Weak individuals have a smaller chance to be selected if tournament size is large.

Code Work:

Firstly we created the class which has x,y co-ordinates and distances between the cities as attributes

```
class City:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def distance(self, city):
    x1 = abs(self.x - city.x)
    y1 = abs(self.y - city.y)
    dist = np.sqrt(x1 ** 2 + y1 ** 2)
    return dist

  def __repr__(self):
    return "(" + str(self.x) + "," + str(self.y) + ")"
```

Next, we defined the fitness value which is defined as the distance between cities because the main constraint is to find min distance between cities. Fitness function returns the value 1/fitness_value. And generate_path function generates the path between the list of cities.

In [4]:
```
def fitness_func(route):
  fitness_value = 0
  for i in range(len(route)-1):
    fitness_value += route[i].distance(route[i+1])
  return 1/fitness_value
```

In [5]:
```
def generate_path(citylist):
  return random.sample(citylist, len(citylist))
```

In [6]:
```
def plot_cities(citylist, title, line=False):
    X = np.array([c.x for c in citylist])
    y = np.array([c.y for c in citylist])
    if line:
        plt.plot(X, y, '-o')
    else:
        plt.scatter(X, y)
    plt.title(title)
```

Further, we initialized the genetic algorithm and the below code is to generate the population necessary. The ordered_routes function is to find fitness value for every individual in population and sort accordingly.

In [7]:
```
def starting_population(citylist, size_of_population):
  population = []
  while len(population) < size_of_population:
    new_route = generate_path(citylist)
    population.append(new_route)
  return population
```

In [8]:
```
def ordered_routes(population):
  fitness_values = {}
  for i in range(len(population)):
    fitness_values[i] = fitness_func(population[i])
  return sorted(fitness_values.items(), key = operator.itemgetter(1), reverse = True)
```

Then we implemented the Roulette Wheel Selection Algorithm in which we created an array with selection results where the selected population that is who wins according to their fitness value is appended. And we also calculating the cumulative percentage by dividing (its fitness value)/ (summation of all fitness values) *100. The more this percentage the more probability of selection. And we are selecting the required amount of individual by this and storing in the selection_results array.

In [9]:
```
def roulette_selection(ranked_population, elite_size):

  selection_results = []
  df = pd.DataFrame(np.array(ranked_population), columns = ['Index', 'Fitness'])
  df['Cum_Sum'] = df.Fitness.cumsum()
  df['Cum_Perc'] = 100*df.Cum_Sum/df.Fitness.sum()

  for i in range(0, elite_size):
      selection_results.append(ranked_population[i][0])
  for i in range(0, len(ranked_population) - elite_size):
      pick = 100*random.random()
      for i in range(0, len(ranked_population)):
          if pick <= df.iat[i,3]:
              selection_results.append(ranked_population[i][0])
              break

  return selection_results
```

Then we implemented Tournament Selection Algorithm in which we created a tournament size of our choice. We will then pick a random population of that size and there will be competition among them and the individual with highest fitness value is picked and likewise we will pick random again in the same method and winner are stored. we will do this until we get required number of individuals.

In [10]:
```
def tournament_selection(ranked_population, elite_size, tournament_size):

  selection_results = []

  for i in range(0, elite_size):
    selection_results.append(ranked_population[i][0])

  for i in range(0, len(ranked_population) - elite_size):
    competors = random.sample(ranked_population, tournament_size)
    winner = competors[0]
    for j in range(tournament_size):
      if winner[1] < competors[j][1]:
        winner = competors[j]
    selection_results.append(winner[0])

  return selection_results
```

This part of algorithm is where crossover between the population happens and, in this case, the code is using single point crossover.

In the one_point_breed function, we will generate the random number and at that point we will cut the parent and we will attach the 2nd part of 1st to 1st part of 2nd. And we will join them

The op_breed_population function is for applying the same type of single point crossover to all the population except the elite members we selected at starting. This will do the whole crossover of all the population.

In [12]:

```
def one_point_breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child

def op_breed_population(mating_pool, elite_size):

    children = []

    length = len(mating_pool) - elite_size
    pool = random.sample(mating_pool, len(mating_pool))

    for i in range(0,elite_size):
        children.append(mating_pool[i])

    for i in range(0, length):
        child = one_point_breed(pool[i], pool[len(mating_pool)-i-1])
        children.append(child)
    return children
```

This part of algorithm is where mutation happens and the below code is for mutation

In the mutate funtion part the mutation will take place according to the mutation rate value and if at a city we will generate a random value and if that random value is less than our mutation rate, we will swap that city with any random city in the whole population.

And in the ‘mutate_population’ function , mutation for all the population happens and will return the value of mutated_population.

In [13]:

```
def mutate(population_instance, mutation_rate):
    for swapped in range(len(population_instance)):
        if(random.random() < mutation_rate):
            swapWith = int(random.random() * len(population_instance))

            city1 = population_instance[swapped]
            city2 = population_instance[swapWith]

            population_instance[swapped] = city2
            population_instance[swapWith] = city1
    return population_instance


def mutate_population(population, mutation_rate):
    mutated_population = []

    for ind in range(len(population)):
        mutated_individual = mutate(population[ind], mutation_rate)
        mutated_population.append(mutated_individual)
    return mutated_population
```

The Main function contains the city co-ordinates which here are being generated by random function and after populating the population with the city list then we start the selection process in both roulette-wheel method and Tournament method.

It displays the city coordinates it generated and initial distance and final distance based on the strategy. And it also displays the routes coordinates it is traversing while the algorithm running whether it is roulette wheel route coordinates or Tournament based route coordinates.

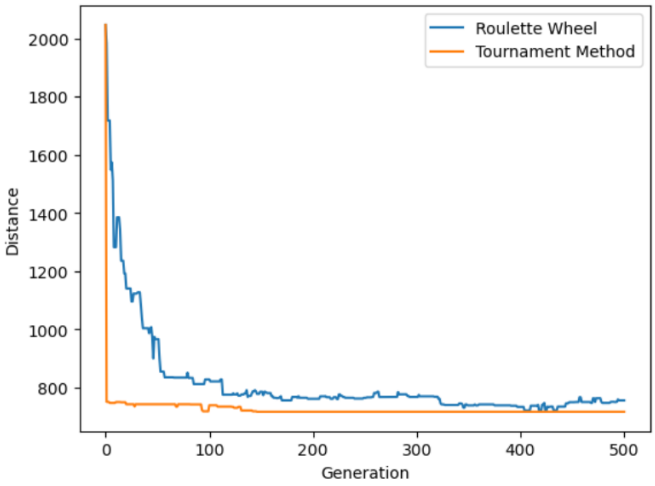
In [25]:

```
citylist = []

for i in range(0,25):
    citylist.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

print("Cities : ", citylist)

population = starting_population(citylist, size_of_population=100)

roulette_best_route = roulette_genetic_algo(population, elite_size=20, mutation_rate=0.01, no_of_generation=500)
tournament_best_route = tournament_genetic_algo(population, elite_size=20, mutation_rate=0.01, no_of_generation=500, tourname

print("Roulette Wheel best route : ",roulette_best_route)
print("Tournament based best route : ",tournament_best_route)
```

```
Cities :  [(111,41), (167,42), (56,115), (169,47), (124,64), (164,134), (52,168), (108,157), (158,147), (88,156), (139,185), (7
7,158), (112,135), (26,11), (149,33), (52,109), (113,129), (194,194), (60,114), (185,193), (186,160), (77,41), (109,0), (43,8),
(41,127)]
Initial distance: 1916.3496364733076
Final distance Roulette based: 711.318820677008
Initial distance: 1916.3496364733076
Final distance Tournament based: 766.5547290803415
Roulette Wheel best route :  [(109,0), (149,33), (167,42), (169,47), (124,64), (111,41), (77,41), (43,8), (26,11), (52,109), (6
0,114), (56,115), (41,127), (52,168), (77,158), (88,156), (108,157), (112,135), (113,129), (139,185), (185,193), (194,194), (18
6,160), (158,147), (164,134)]
Tournament based best route :  [(194,194), (185,193), (186,160), (164,134), (158,147), (139,185), (108,157), (88,156), (77,15
8), (112,135), (113,129), (124,64), (169,47), (167,42), (149,33), (111,41), (109,0), (77,41), (43,8), (26,11), (52,109), (60,11
4), (56,115), (41,127), (52,168)]
```

The below code is the plotting of Roulette wheel method and Tournament method for the 500 iterations with mutation rate of 0.01 and tournament size is 3 with population of 100. And the graph is plotted between the Distance vs Generation.

```
In [28]: 1 geneticAlgorithmPlot(citylist, size_of_population=100, elite_size=20, mutation_rate=0.01, no_of_generation=500, tournament_s
```



Conclusion:

In the scope of our research, we concluded that tournament selection method is better in the case of TSP as compared to the Roulette wheel method.

Contributions:

We both explored different methods of selection in genetic algorithm and worked on it together and discussed the way of writing the algorithm for different methods and documented the report.