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

UEMH3073 Artificial Intelligence

Practical 2 Report

Genetic Algorithm

2024

Student ID	Student Name	Practical Group	Program
2202489	Ng Yen Keat	P1	MH
2200574	Ong Hong Qing	P1	MH
2102457	Kam York Yong	P1	MH
2202291	Goh Guang Yi	P1	MH

Table of Contents

TODO 1 : Population Initialization

```
    # Replacement starts here
    try:
        with open(filename, 'r') as file:
            lines = file.readlines()
            for line in lines[1:]:
                part = line.strip().split(',')
                x,y = int(part[1]),int(part[2])
                cityList.append(City(x, y))
    except Exception as e:
        print(f"Error reading file: {e}")

    return cityList

    # Replacement ends here
```

```
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
```

The third function is initialPopulation() which calls the secc

```
def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population
```

```
# Test code for TODO1
cityList = genCityList('cities50.txt')
population = initialPopulation(100,cityList)
print(len(population))
population
```

```
100

[[(75,44),
  (92,75),
  (78,60),
  (70,32),
```

Figures 1.1, 1.2, 1.3: functional code and test code for TODO 1

Population initialization is the first step in a genetic algorithm to produce the same space dimensions of a certain number of unique populations from the given input data, and it is done by randomly arranging the routes between cities. First, the code reads all the data in the text file, and it strips and splits the three different types of data with a comma separator for each row except for the lines [0], which contain the title labels. Then, it converts part [1] and part[2] into integers because the previous type is a string. After all the data are processed, the function will return the ‘cityList,’ where the list contains all the x y coordinates of cities.

TODO 2 : Tournament Parent Selection

```python
def parentSelection_tournament(population, parent_size=2, tournament_size=3, poolSize=None):

    if poolSize == None:
        poolSize = len(population)

    # Replacement starts here
    iteration = 0
    matingPool = []

    while (iteration < parent_size):
       bestFitness = 0
       Pool = random.sample(population[0:poolSize],tournament_size)
       for i in range (len(Pool)):
           fitness = Fitness(Pool[i]).routeFitness()
           print(fitness)
           if fitness > bestFitness:
               bestFitness = fitness
               print("Best Fitness Score is:{}".format(bestFitness))
               bestPopulation = population[i]
       matingPool.append(bestPopulation)
       iteration += 1
       __builtins__.print()

    # Replacement ends here

    return matingPool
```

```python
# Test code for TODO 2
cityList = genCityList('cities50.txt')
population = initialPopulation(100,cityList)
parentSelection_tournament(population,2,3)
```

```
0.00039630851798782516
'Best Fitness Score is:0.00039630851798782516'
0.00039146058377927
0.00039920183811144414
'Best Fitness Score is:0.00039920183811144414'

0.00036769513473103 65
'Best Fitness Score is:0.00036769513473103 65'
0.00040020479416789137
'Best Fitness Score is:0.00040020479416789137'
0.00039766723609 78724


[[(61,21),
  (34,83),
  (78,60),
```

Figures 2.1, 2.2: functional code and test code for TODO 2

In tournament parent selection, a subset of individuals (‘tournament_size’) is chosen from the population; for this example, three individuals are randomly chosen among pool sizes. Furthermore, the fittest individual will be chosen from the subset of individuals to the mating pool to pass their good gene to the next generation. We can customize the ‘parent_size’ parameter to define the number of parents produced from this tournament selection as it iterates the selection process until the desired number of parents is produced.

TODO 3 : Proportional Parent Selection (Roulette Wheel method)

```python
def parentSelection_proportional(population, parent_size=2, poolSize=None):

    if poolSize == None:
        poolSize = len(population)

    matingPool = []

    # Replacement starts here
    totalFitnessScore = 0
    score = []
    relativeFitnessScore = []
    cumulativeScore = []
    currentScore = 0
    iteration = 0
    escape = False

    totalFitnessScore = sum([Fitness(p).routeFitness() for p in population[:poolSize]])
    score = ([Fitness(p).routeFitness() for p in population[:poolSize]])

    for i in range(len(score)):
       relativeFitnessScore.append(score[i]/totalFitnessScore)

    for i in range(len(score)):
       currentScore += relativeFitnessScore[i]
       cumulativeScore.append(currentScore)

    while(iteration < parent_size):
        randomNumber = random.random()   # produce a random number between 0 to 1
        __builtins__.print("random number: ",randomNumber)

        for i in range (len(cumulativeScore)):
            if randomNumber <= cumulativeScore[0]:
                __builtins__.print("population_location[i]: ",i)
                __builtins__.print("cumulative score[0]: ", cumulativeScore[i])
                __builtins__.print()
                matingPool.append(population[0])
                break

            if cumulativeScore[i] < randomNumber <= cumulativeScore[i+1]:
                __builtins__.print("population_location[i+1]: ",i+1)
                __builtins__.print("cumulative score[i]: ", cumulativeScore[i],",","cumulative score[i+1]: ",cumulativeScore[i+1])
                __builtins__.print()
                matingPool.append(population[i+1])
                break

        iteration += 1

    # Replacement ends here

    return matingPool,totalFitnessScore
```

```python
# Test code for TODO 3
cityList = genCityList('cities50.txt')
population = initialPopulation(100,cityList)
parent,score = parentSelection_proportional(population,2)
parent,score
```

```
random number:  0.3171587644067373
population_location[i+1]:  31
cumulative score[i]:  0.31195280082697713 , cumulative score[i+1]:  0.32217269153441197

random number:  0.34984953137675723
population_location[i+1]:  34
cumulative score[i]:  0.3419910690756855 , cumulative score[i+1]:  0.35194378375788865


([[(9,23),
   (80,68),
   (10,31),
   (34,83),
   (92,57),
   (6,11),
   (1,77),
   (33,60),
   (59,3),
```

Figures 3.1, 3.2: functional code and test code for TODO 3

The proportional parent selection method uses a roulette wheel to select the suitable individuals for the next generation. A roulette wheel is created from the relative fitness of every individual. It is represented in a pie chart where the area occupied by every individual is proportional to the relative fitness value. In this code, the parent size can be set to determine how many parents are produced in this function. The 'poolSize' is implemented to provide additional features where it needs to define the population size specifically, for example, when it needs to preserve the elite population size, so the pool size needs to exclude it. The code summarizes all the fitness to calculate the total fitness score and divides each individual's fitness score to get the relative fitness score. Then, the relative fitness score is added to calculate the cumulative score. A random number between 0 and 1 is generated for selection. When the generated random number is smaller than the first cumulative number, the selected parent will be the first index of the population. When the random number is allocated between two different cumulative scores, the parent will be selected based on the higher cumulative score. Therefore, the parents are selected based on cumulative probabilities; when the fitness value is higher, it has a higher chance of being selected to become a parent. Lastly, the selected parents are returned to 'matingPool'.

TODO 4 : Survival Selection

```
# Replacement starts here
# Process elite list
def form_correction(elites):
    return [chromosome for chromosome in elites[0]]

# Reduce population to elite size
def cut_down(eliteSize):
    elites.append(population[:eliteSize])

# Assign fitness to chromosome as its last element
def addFitness():
    for chromosome in population:
        score = Fitness(chromosome).routeFitness()
        chromosome.append(score)

# Remove fitness
def removeFitness():
    for chromosome in population:
        chromosome.pop()

# Sort population according to fitness
def mergeSort(population):
    if len(population) <= 1:
        return population

    mid_p = len(population) // 2
    left = mergeSort(population[:mid_p])
    right = mergeSort(population[mid_p:])
    sorted_population = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i][-1] > right[j][-1]:
            sorted_population.append(left[i])
            i += 1
        else:
            sorted_population.append(right[j])
            j += 1

    sorted_population.extend(left[i:])
    sorted_population.extend(right[j:])
    return sorted_population

def score(elites):
    elites_scores = ([Fitness(e).routeFitness() for e in elites])
    return elites_scores


# Main Program for survival selection
addFitness()
population = mergeSort(population)
removeFitness()
cut_down(eliteSize)
elites = form_correction(elites)
score = score(elites)
# Replacement ends here
```

Figure 4.1: New codes for TODO4

The codes replace the initial codes with merge, sort, and truncate. The "form_correction(elites)" takes a list of elites and returns a list of chromosomes from the first element of the list. In other words, the "elites" list has the actual elite chromosomes as its first element. The "cut_down(eliteSize)" reduces the population to keep the top elements and append the reduced list to the "elites" list. Moreover, the "addFitness()" calculates the fitness score and appends it to the end of each chromosome in the population. The "removeFitness()" function then pops the last element from the chromosome, which is the fitness score that is added initially. The "mergeSort(population)" aids in sorting the population with the merge sort algorithm depending on the fitness score. The population is divided iteratively into halves, sorted, and merged again in sorted order. Lastly, the "score(elites)" calculates the fitness scores for the list of elite chromosomes. To execute the survival selection, the main program first adds the fitness score to the chromosomes in the population and sorts them based on their fitness score before removing the scores. The elite list is then reduced to top elements only. The survival selection selects only the fittest chromosomes for the next generation by ensuring that chromosomes having higher fitness scores have a higher chance of contributing to the next generation.

'Initial population' 10 'Selected elites' 5 'elites_score: ' [0.00044509721884151336, 0.00042943938105517195, 0.0004105616306968323, 0.00040618898541441574, 0.0004060359037237635]	'Initial population' 100 'Selected elites' 5 'elites_score: ' [0.00047259471937242595, 0.0004482305320835305 4, 0.0004416385618445782, 0.0004381462620011673, 0.00043568139877690965]
Average elite score = 4.194646×10^{-4}	Average elite score = 4.472583×10^{-4}

Table 4.1: Comparison of initializing 10 and 100 populations

By setting different initial population size, the average elite score obtained is also different. An example of the results is tabulated in Table 4.1, where the average elite score for 10 initial population is lower than 100 initial population. This is owing to the fact that the results of the smaller population are stuck in local optima, where the global optimum solution is unable to be found due to limited genetic diversity. On the other hand, the genetic algorithm is able to explore the solution space more efficiently and effectively with the larger population to achieve overall higher fitness score. However, this requires more computational resources.

TODO 5 : Crossover

```python
def crossover(parent1, parent2):

    # Replacement starts here

    # Determine the size of the parents (number of cities)
    size = len(parent1)
    # Randomly select two crossover points
    cxpoint1, cxpoint2 = sorted(random.sample(range(size), 2))

    # Ensure valid crossover points (cxpoint2 should be at least one position greater than cxpoint1)
    while (cxpoint2 == cxpoint1 or cxpoint2 < cxpoint1+1):
        cxpoint1, cxpoint2 = sorted(random.sample(range(size), 2))

    # Initialize empty children with None values
    child1 = [None] * size
    child2 = [None] * size

    # Copy the crossover segment from parent2 to child1 and from parent1 to child2
    child1[cxpoint1:cxpoint2] = parent2[cxpoint1:cxpoint2]
    child2[cxpoint1:cxpoint2] = parent1[cxpoint1:cxpoint2]

    # Helper function to map values from parent to child ensuring no duplicates
    def map_values(child, parent, start, end):
        for i in range(start, end):
            value = parent[i]
            if value not in child:
                child[i] = value
            else:
                # Find a value to replace in the child if a duplicate is found
                while value in child:
                    value = parent[child.index(value)]
                child[i] = value

    # Map values from parent1 to child1 and parent2 to child2 to fill in the remaining positions
    map_values(child1, parent1, cxpoint2, size)
    map_values(child1, parent1, 0, cxpoint1)
    map_values(child2, parent2, cxpoint2, size)
    map_values(child2, parent2, 0, cxpoint1)

 # Replacement ends here

    return child1, child2 # Return the generated children
```

```python
# Test code for TODO 5
parent1 = [10,9,8,7,6,5,4,3,2,1]
parent2 = [15,14,13,12,11,10,9,8,7,6]

child1,child2 = crossover(parent1,parent2)
__builtins__.print("parent1: ",parent1)
__builtins__.print("parent2: ",parent2)
__builtins__.print()
__builtins__.print("child1: ",child1)
__builtins__.print("child2: ",child2)
```

```
parent1:  [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
parent2:  [15, 14, 13, 12, 11, 10, 9, 8, 7, 6]

child1:  [10, 9, 8, 7, 11, 5, 4, 3, 2, 1]
child2:  [15, 14, 13, 12, 6, 10, 9, 8, 7, 11]
```

Figures 5.1, 5.2: functional code and test code for TODO 5

The crossover function is required for genetic algorithms which handle the Travelling Salesman Problem (TSP), resulting in two viable children from two parent routes. The function starts by calculating the size of the parent routes, which indicate the number of cities, and then picks two crossover point at random, ensuring that the first is less than the second. It checks that these points are distinct and well-spaced. Next, the code creates two empty child routes with None values. It then transfers segments from the parents to the children using the crossover locations, moving a segment from the second parent to the first child and vice versa. To guarantee that the children are valid TSP routes, the function adds a helper called map_values, which checks for duplicates and replaces them with values from the parents. Finally, the function returns the two valid children, guaranteeing that each city is visited only once on each route. This procedure assures that the resultant children are valid permutations of the cities, keeping the TSP solution.

TODO 6 : Mutation

```
def insertion_mutation(route):
   if (random.random() < mutationProbability):
       if len(route) > 1:
           # Randomly pick index to remove
           remove_index = random.randint(0, len(route) - 1)
           # Remove the index
           city_reinsert = route.pop(remove_index)
           # Select a new position to insert the element
           insert_index = random.randint(0, len(route) - 1)
           # Reinsert into the list
           route.insert(insert_index, city_reinsert)
           return route

   else:
       return route
```

Figure 6.1: New codes for TODO6

The new codes replace the previous codes with the insertion mutation approach. The defined “insertion_mutation(route)” function performs insertion mutation on the given “route” list. Primarily, “random.random()” generates a random value between 0 and 1. The insertion mutation will only proceed if the generated value is less than a fixed value of “mutationProbability”. This is to ensure that the procedures will only take place by chance. If the insertion mutation occurs, the length of the route is checked and the procedure will only continue if the route contains more than one city. A random index within the route will then be selected, and the city at the selected index will be removed from the list but stored in “city_reinsert”. A new index within the route is then randomly selected to reinsert the removed city. The procedures are done and the results are returned. Nonetheless, if the mutation procedures do not take place as the generated value is larger than “mutationProbability”, the original route is returned unchanged. Based on the codes in Figure 6.2, the results are tabulated in Table 6.1.

```
# Test code for TODO 6
parent = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
mutation = mutate(parent,0.3)
__builtins__.print(mutation)
```

Figure 6.2: Code testing for TODO6

1st try	[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1, 13, 14, 15] Removed index = 1
2nd try	[0, 1, 2, 3, 4, 5, 10, 6, 7, 8, 9, 11, 12, 13, 14, 15] Removed index = 10
3rd try	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] No insertion mutation
4th try	[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 9, 15] Removed index = 9

Table 6.1: Multiple results for code testing for TODO6

TODO 7 : Running Many Generations

Default Settings:

```
filename = 'cities50.txt'
popSize = 200
eliteSize = 5
mutationProbability = 0.01
iteration_limit = 100
parent_size = 20
tournament_size = 3
```

Figure 7.1: Default Settings for producing generation after a certain number of iterations.

After we created different functions for the genetic algorithm such as parent selection for choosing the parent for crossover and mutation through multiple iterations, survival selection for choosing the elites group to preserve the good genes traits through the process, crossover process is to generate different variation between parents by crossing over some parts of the genes and the mutation process is to add some randomness to prevent predetermined convergence to local maximum or local minimum point.

Next, we set all the parameters to the default settings which shown as Figure 7.1. To investigate how each of the variable influenced the best distance output in the three parent selection methods, we will change only one variable at each experiment and recorded the data in tables for “cities50.txt” and “cities500.txt” datasets.

i) Random Parent Selection

```
def oneGeneration(population, eliteSize, mutationProbability,parent_size,tournament_size):

    # First we preserve the elites
    elites,scores = survivorSelection(population, eliteSize)

    # Then we calculate what our mating pool size should be and generate the mating pool
    poolSize = len(population) - eliteSize
    matingpool = parentSelection_random(population, poolSize)
    #matingpool = parentSelection_tournament(population, parent_size,tournament_size, poolSize)
    #matingpool,score = parentSelection_proportional(population, parent_size, poolSize)
```

a1) popSize (cities50.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	1998.80	2105.93	1962.73	1884.48	1929.70	1976.33	74.96
100 (Default)	1504.20	1681.97	1612.60	1650.45	1596.48	1609.14	60.32
200	1437.68	1738.09	1501.80	1537.65	1457.17	1534.48	107.59

a2) popSize (cities500.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	24221.76	24186.82	24120.31	24424.04	24226.80	24235.95	101.44
100 (Default)	23228.76	23149.00	22961.37	22934.97	23562.85	23167.39	226.70
200	23229.11	23294.15	23292.54	23504.28	23421.22	23348.26	99.89

Table 7.1, 7.2: comparison of pop size in random selection method for ‘cities50.txt’ and ‘cities500.txt’.

From the above comparison, the population size will affect the best distance result. Both ‘cities50.txt’ and ‘cities500.txt’ have a similar trend in which the best distance decreases as the population increases. The reason is that when fewer individuals are in a small population, the mating opportunity for better individuals decreases, resulting in a non-optimized result. The result is essential for the Traveling Salesman Problem as the shorter the distance, the better the result. The optimum distance is selected based on the shortest distance. Hence, we can conclude that the greater the population, the better the selection result.

b1) eliteSize (cities50.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	1658.62	1699.20	1728.09	1850.92	1687.00	1724.77	66.90
5 (Default)	1604.59	1702.57	1441.37	1566.19	1651.28	1593.20	88.60
10	1251.00	1422.03	1539.01	1359.20	1347.17	1383.68	95.01

b2) eliteSize (cities500.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	24968.18	24911.14	24896.09	24674.20	24385.16	24766.95	215.56
5 (Default)	24758.71	25066.06	24997.93	24750.66	24479.87	24810.65	207.86
10	24010.01	24423.90	24319.80	24792.58	25169.19	24543.10	400.62

Table 7.3, 7.4: comparison of elite size in random selection method for ‘cities50.txt’ and ‘cities500.txt’.

Elitism is a parameter that specifies the number of top-performing individuals remaining in the next generation. It ensures that the best solutions are preserved and not affected by crossover or mutation processes. From the above table, the average best distance decreases when the elite size increases. When the elite size increases, the number of preservations for the best solution rises, resulting in a better crossover opportunity for optimum results. In the table of cities500.txt, the abnormal increase in average best distance when the elite size is 5 may be due to a reduction in diversity, causing the algorithm to be unable to explore the solution space effectively. When the elite size becomes 10, more individuals are preserved, resulting in high selection pressure and better results.

c1) mutationProbability (cities50.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	1507.47	1588.86	1470.86	1474.97	1443.10	1497.05	50.24
1% (Default)	1521.04	1539.46	1562.33	1503.96	1422.58	1509.87	47.76
5%	1558.84	1553.85	1574.05	1617.92	1641.99	1589.33	34.68
10%	1458.25	1592.88	1550.01	1662.31	1672.04	1587.10	78.59

c2) mutationProbability (cities500.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	25336.64	24900.68	24752.63	24997.51	24930.87	24805.15	263.50
1% (Default)	25116.07	24509.41	24835.90	24390.65	25104.88	24791.38	298.59
5%	25113.25	24641.02	24763.97	25082.49	24519.04	24823.95	236.88
10%	24794.21	22764.17	24327.23	23565.31	24531.07	23996.40	739.74

Table 7.5, 7.6: comparison of mutation probability in random selection method for ‘cities50.txt’ and ‘cities500.txt’.

The mutation probability determines the likelihood that an individual will undergo the mutation process. The probability cannot be too large to prevent the selection process from becoming random selection. The genetic algorithm reacts differently in data from 50 cities and 500 cities. The results in 50 cities are not dependent on mutation much as the algorithm can thoroughly explore the individuals in the population even though the mutation probability is low. However, the 500 cities will depend on mutation as the search space is too ample for the algorithm to explore. In the table of 500 cities, the average best distance decreases when mutation probability increases as mutation increases diversity and prevents the algorithm from getting stuck in local optima to find a better solution. However, the increase in mutation probability disrupts the algorithm's convergence in 50 cities, causing the result to be abnormal.

d1) iteration (cities50.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	1962.45	2035.51	1890.84	1991.50	1896.93	1995.45	68.35
100 (Default)	1616.75	1484.08	1541.67	1514.65	1367.49	1504.93	81.58
200	1354.39	1405.07	1288.77	1270.57	1322.17	1328.19	47.97

d2) iteration (cities500.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	25420.67	25415.08	24936.13	25153.76	25318.08	25248.74	183.73
100 (Default)	25088.50	25259.86	24822.99	25139.72	25000.87	25062.39	146.12
200	25284.90	25018.17	25107.67	25229.02	25247.20	25177.39	99.31

Table 7.7, 7.8: comparison of iterations in random selection method for ‘cities50.txt’ and ‘cities500.txt’.

The iteration indicates the number of explorations the algorithm can go through in the search space. In 50 cities, iteration is significant as the algorithm can explore the search space thoroughly, resulting in a shorter distance. The increase in iterations only helps a little in 500 cities as the search space is too large, and finding an optimum solution is difficult in an large search space. Besides, the populations in 500 cities are complex and more challenging when dealing with the selection process. The increase in iterations will result in better solutions, but the improvement will decrease when the size of the search space increases.

ii) Tournament Parent Selection

```
def oneGeneration(population, eliteSize, mutationProbability,parent_size,tournament_size):

    # First we preserve the elites
    elites,scores = survivorSelection(population, eliteSize)

    # Then we calculate what our mating pool size should be and generate the mating pool
    poolSize = len(population) - eliteSize
    #matingpool = parentSelection_random(population, poolSize)
    matingpool = parentSelection_tournament(population, parent_size,tournament_size, poolSize)
    #matingpool,score = parentSelection_proportional(population, parent_size, poolSize)
```

a1) popSize (cities50.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	1710.52	1719.88	1385.01	1754.55	1583.67	1630.73	135.82
100 (Default)	1662.44	1671.51	1550.71	1624.03	1692.76	1640.29	50.01
200	1833.63	1740.12	1919.67	1551.54	1639.39	1736.87	131.63

a2) popSize (cities500.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	23242.28	23704.73	23499.75	23228.66	22951.83	23325.45	256.95
100 (Default)	22864.45	22960.99	22846.12	22714.40	24365.89	23150.37	612.82
200	22041.36	22851.69	23513.30	23402.40	22686.29	22899.01	531.75

Table 7.9, 7.10: comparison of population size in tournament selection method for 'cities50.txt' and 'cities500.txt'.

Based on the result, the number of pop size did not drastically affect much to the best distance in 'cities50.txt' and 'cities500.txt' datasets. However, an increase in initial population size did introduce more standard deviations to the best distance result and gives in more variety and randomness to the combinations of initial population for the tournament selection method thus the children produced in the following iteration will be much different across the trials.

b1) eliteSize (cities50.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	1717.36	1847.62	1795.33	1877.40	1793.42	1806.23	54.70
5 (Default)	1497.19	1678.30	1613.50	1622.55	1467.00	1575.71	80.16
10	1619.25	1417.47	1616.47	1622.77	1631.26	1581.44	82.14

b2) eliteSize (cities500.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	23646.22	24368.14	24186.20	23849.93	24225.09	24055.12	265.84
5 (Default)	24090.70	23752.24	23436.77	22918.09	23897.90	23619.14	410.49
10	22700.58	22263.40	22815.84	23658.78	23373.69	22962.46	496.54

Table 7.11, 7.12: comparison of elite size in tournament selection method for 'cities50.txt' and 'cities500.txt'.

The elite size parameters improve the best distance result when it increases. Five number of elite size selection is the best parameter in this case as it improves significantly when compared to only 1 elite population to be preserved. This can be explained as the algorithm retains certain amounts of good population and at the same time allowing enough of breath for the exploration space to be search. When the elite size is 10, there is every little of improvement as it only allows lesser of the parent size to be in part of mutation and crossing over thus leading lesser improvements.

c1) parentSize (cities50.txt)

parentSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
5	1994.49	1969.15	1913.10	1844.36	1816.33	1907.49	68.84
20 (Default)	1713.40	1683.32	1716.77	1633.54	1830.17	1715.44	64.67
50	1322.72	1432.69	1398.31	1640.97	1475.71	1454.08	106.04

c2) parentSize (cities500.txt)

parentSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
5	25018.18	24432.21	25102.15	25296.11	24705.62	24910.85	305.86
20 (Default)	22087.26	22218.76	23064.09	22694.73	23296.12	22672.19	467.15
50	21958.38	21604.95	21365.10	21231.77	20737.84	21379.61	404.89

Table 7.13, 7.14: comparison of parent size in tournament selection method for 'cities50.txt' and 'cities500.txt'.

The parent size in this tournament selection method impacts a lot to the average best distance result when the parent size increases. It is because when the number of parents being selected increases will increase the probability finding better routes combination thus it yields more varieties and produce better child population throughout each iteration.

d1) mutationProbability (cities50.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	2022.96	2000.26	1894.83	1874.39	1808.13	1920.11	80.34
1% (Default)	1731.74	1597.50	1599.97	1750.00	1714.35	1678.71	66.27
5%	1393.05	1291.74	1456.92	1425.75	1392.92	1392.08	55.51
10%	1351.61	1255.29	1142.04	1472.31	1331.62	1310.57	109.31

d2) mutationProbability (cities500.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	23785.89	23432.29	23158.27	23998.79	24164.06	23707.86	367.92
1% (Default)	23685.94	23062.70	22916.84	23359.61	23418.90	23288.80	271.71
5%	21715.16	21340.84	21996.40	21116.54	22392.17	21712.22	455.16
10%	21843.46	21453.19	21268.66	21199.60	21104.21	21373.82	261.14

Table 7.15, 7.16: comparison of mutation probability in tournament selection method for ‘cities50.txt’ and ‘cities500.txt’.

Higher mutation probability in tournament selection will introduce some randomness and probability to escape from the local minima point where some of the excellent route combination result has higher probability able to be produced and being preserved in the elite group compared to low mutation probability case. Thus, after throughout many iterations of crossing over, only better traits will be preserved so the best distance improved at higher mutation probability scenarios.

e1) iteration (cities50.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	2039.65	1853.14	1772.14	1891.79	1983.79	1908.10	94.66
100 (Default)	1635.62	1712.74	1716.97	1499.91	1758.56	1664.76	91.49
200	1383.79	1545.73	1466.31	1606.99	1516.18	1503.80	75.32

e2) iteration (cities500.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	23989.69	24439.03	24322.47	24425.79	24113.40	24258.08	177.75
100 (Default)	23605.77	23158.24	22800.70	22683.54	23345.21	23118.69	340.76
200	21640.24	21429.86	21836.56	22996.25	23555.14	22291.61	832.71

Table 7.17, 7.18: comparison of number of iterations in tournament selection method for ‘cities50.txt’ and ‘cities500.txt’.

Increasing the number of iterations in the tournament selection method improves solution quality for both 'cities50.txt' and 'cities500.txt', as seen by decreased average best distances. For 'cities50.txt', the average best distance drops from 1908.10 to 1503.80 as the number of iterations increases from 10 to 200, as does the standard deviation, showing more consistent results. The average best distance for 'cities500.txt' improves with more iterations, from 24258.08 to 22291.61, however the standard deviation first increases at 100 iterations before falling at 200. The increased variety in 'cities500.txt' outcomes is due to the bigger issue size and complexity, which necessitates more iterations for the algorithm to efficiently explore and utilize the solution space. Overall, more iterations result in better, more consistent solutions, especially for smaller problem sizes.

Higher iterations will lead to more crossover and mutation; therefore, the best distance result will be further reduced as the iteration goes. Additionally, the reason that the standard deviation goes up so high in ‘cities500.txt’ when the iteration is 200, is that leading the algorithm be able to explore in more solution space and the inherent randomness accumulates in iteration and also it might lead to overfitting to local optima or diverge therefore the result of large varieties among trials existed.

f1) tournamentSize (cities50.txt)

tournament size	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
3 (Default)	1968.15	1859.74	1752.45	1439.05	1416.87	1687.25	222.54
10	1520.75	1537.79	1667.18	1599.50	1507.85	1566.61	59.30
20	1481.76	1726.16	1392.67	1570.04	1623.69	1558.86	114.79

e2) tournamentSize (cities500.txt)

tournament size	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
3 (Default)	22986.39	22450.77	21895.02	22936.06	23188.61	22691.37	466.09
10	21862.64	23373.45	23103.20	22327.74	22197.21	22572.85	570.58
20	22438.78	22766.48	22805.10	22796.75	23537.26	22868.87	360.96

Table 7.19, 7.20: comparison of tournament size in tournament selection method for ‘cities50.txt’ and ‘cities500.txt’.

When the tournament size is lower (for example,k=3), the selection pressure is lower therefore there is a higher chance of selecting individual with below average fitness. Hence, it will maintain the diversity and explores a broader range of solutions, but it will have slower convergence, so it has higher standard deviation value. Higher tournament size parameters does not mean to improve the best distance result, it will increase the selection pressure so that there is a higher chance of selecting individual with above average fitness and this will lead to acceleration of convergence to exploit for higher fitness individual and in the same time it lost its diversity and may stuck into premature convergence and as result its standard deviation normally lower when compared the tournament size equals to 3 and 20 in cities50.txt and cities500.txt.

iii) Proportional Parent Selection

```
def oneGeneration(population, eliteSize, mutationProbability,parent_size,tournament_size):

    # First we preserve the elites
    elites,scores = survivorSelection(population, eliteSize)

    # Then we calculate what our mating pool size should be and generate the mating pool
    poolSize = len(population) - eliteSize
    #matingpool = parentSelection_random(population, poolSize)
    #matingpool = parentSelection_tournament(population, parent_size,tournament_size, poolSize)
    matingpool,score = parentSelection_proportional(population, parent_size, poolSize)
```

a1) popSize (cities50.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	1413.98	1502.75	1556.17	1312.14	1506.52	1458.31	86.26
100 (Default)	1516.31	1409.33	1640.41	1580.58	1533.43	1536.01	76.59
200	1516.65	1441.68	1569.21	1560.84	1663.43	1550.36	72.38

a2) popSize (cities500.txt)

popSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	22066.74	22808.71	22991.35	22177.27	21875.22	22383.86	436.23
100 (Default)	21497.23	22299.43	22825.67	22120.63	23187.52	22386.10	583.71
200	22964.01	22014.65	22847.25	22209.11	22412.52	22489.51	364.19

Table 7.21, 7.22: comparison of pop size in Proportional selection method for ‘cities50.txt’ and ‘cities500.txt’.

Smaller population size resulted in better average best distances for both datasets (cities50.txt and cities500.txt), but with increased variability. This most likely occurs because smaller populations allow the genetic algorithm to explore the solution space more quickly, perhaps discovering better solutions faster but with less consistency. Larger population numbers produced more consistent results , but they did not always provide the best average distances. This is because bigger populations have a more diversified gene pool, which reduces the likelihood of premature convergence and promotes exploration while simultaneously lowering the convergence rate. As a result, the appropriate population size is determined by the combination of solution quality and consistency required for the given issue.

b1) eliteSize (cities50.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	1683.86	1655.51	1791.59	1730.98	1808.18	1734.02	59.17
5 (Default)	1510.31	1449.93	1832.63	1606.39	1596.09	1599.07	130.18
10	1632.24	1646.18	1752.80	1596.30	1835.36	1692.58	88.40

b2) eliteSize (cities500.txt)

eliteSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
1	23286.87	23430.06	23300.48	24294.33	23028.88	23468.12	433.12
5 (Default)	22700.67	21791.07	22681.68	21734.45	22347.79	22251.13	418.43
10	21793.48	21266.98	22358.24	22266.84	21902.94	21917.70	388.60

Table 7.23, 7.24: comparison of elite size in Proportional selection method for ‘cities50.txt’ and ‘cities500.txt’.

Increasing the elite size improved the average best distances but had different effects on consistency between the two datasets. For cities50.txt, a moderate elite size (5) achieved the best average distance (1599.07), but with high variability (σ = 130.18), whereas the smallest elite size (1) was more consistent but produced poorer results. This shows that a modest elite size provides for better solutions by keeping more high-quality individuals, but at the cost of higher variability. In cities500.txt, the largest elite size (10) obtained the best average distance (21917.70) with the lowest variability (σ = 388.60), showing that a larger elite size effectively maintains a diverse gene pool, preventing premature convergence and improving both performance and consistency. As a result, an optimal elite size creates a deal between keeping high-quality individuals and maintaining genetic diversity while tailoring the solution to the problem's complexity.

c1) parentSize (cities50.txt)

parentSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
5	2048.30	1983.73	1847.23	1828.85	1739.29	1889.48	111.47
20 (Default)	1457.73	1551.21	1388.06	1556.57	1486.19	1487.95	62.62
50	1470.08	1415.15	1379.38	1325.89	1366.71	1391.44	48.58

c2) parentSize (cities500.txt)

parentSize	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
5	24177.31	24212.52	24655.15	24558.80	24948.74	24510.50	288.14
20 (Default)	22023.61	22438.37	21645.06	22299.87	22432.64	22167.91	301.61
50	21749.85	22196.70	21965.04	22441.74	21743.35	22019.34	268.93

Table 7.25, 7.26: comparison of parent size in Proportional selection method for ‘cities50.txt’ and ‘cities500.txt’.

The results show that increasing the parentSize enhances the performance and consistency of the proportional selection method. For cities50.txt, the average best distance falls from 1889.48 (parentSize = 5) to 1391.44 (parentSize = 50), with a matching drop in standard deviation from 111.47 to 48.58. Similarly, for cities500.txt, the average best distance drops from 24510.50 (parentSize = 5) to 22019.34 (parentSize = 50), while the standard deviation falls from 288.1 to 268.9. This improvement is due to the fact that larger parentSize provides for a more diverse and robust pool of potential solutions, improving the algorithm's ability to explore the solution space more efficiently. As a result, the chance of finding better solutions improves, but the variability between trials reduces. As a result, a bigger parentSize produces better and more stable results in this proportional selection method.

d1) mutationProbability (cities50.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	1612.33	1730.28	1654.60	1621.89	1909.98	1705.82	110.17
1% (Default)	1625.20	1737.74	1470.88	1491.23	1456.42	1556.29	108.79
5%	1350.94	1332.96	1200.82	1342.05	1336.03	1312.56	56.21
10%	1194.93	1333.57	1380.07	1304.93	1406.01	1323.90	73.46

d2) mutationProbability (cities500.txt)

P (m)	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
0%	22733.09	22589.04	22302.22	23564.44	23072.16	22852.19	433.96
1% (Default)	22371.38	22027.21	22628.49	22204.12	23006.51	22447.54	342.61
5%	22651.61	22056.93	22253.13	21489.51	22084.27	22107.10	374.77

10%	21536.45	21380.08	21816.88	21187.84	22006.72	21585.60	294.52

Table 7.27, 7.28: comparison of mutation probability in Proportional selection method for 'cities50.txt' and 'cities500.txt'.

By increasing the mutation probability of the proportional parent selection, the evolution of the population may experience several effects including increased genetic diversity. Genetic variation that is introduced to the population will be higher when the mutation probability is higher, which aids in preventing premature convergence to local optima. This is due to the fact that more potential solutions will be explored. However, a higher mutation probability may lead to the disruption of good solutions. While it introduces beneficial diversity to the population, strong chromosomes are potentially lost due to excessive mutation, resulting in a longer period needed for the convergence process. Therefore, the optimal mutation rate can be difficult to determine. In this experiment, it can be seen that mutation probability of 5% is suitable for "cities50.txt" whereas 10% for "cities500.txt". In this experiment, the highest mutation probability used is only 10% as the behavior of the algorithm will become more stochastic with higher mutation probability. This leads to less predictable outcomes, increasing the difficulty of anticipating the performance and convergence characteristics of the genetic algorithm.

e1) iteration (cities50.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	1999.25	1969.91	1993.69	1956.74	1975.76	1979.07	15.59
100 (Default)	1406.65	1657.38	1601.44	1578.11	1514.81	1551.68	85.72
200	1336.02	1335.81	1491.83	1312.16	1382.87	1371.74	64.28

e2) iteration (cities500.txt)

iteration	Best distance					Average best distance	σ
	Trial1	Trial2	Trial3	Trial4	Trial5		
10	24679.48	24850.28	24509.60	24426.05	24489.88	24591.06	154.35
100 (Default)	22335.18	21814.79	22608.29	21976.64	21918.04	22130.59	296.20
200	21603.16	21157.45	21860.42	21416.23	21595.12	21526.48	232.57

Table 7.29, 7.30: comparison of number of iterations in Proportional selection method for 'cities50.txt' and 'cities500.txt'.

As shown in Tables 7.29 and 7.30, it can be clearly seen that the average best distances for both cases decrease when the iteration increases and achieve the lowest value for iteration 200. This is due to the algorithm being allowed for extensive exploration of the solution space with higher value of iterations. More iterations also make more improvements to the population and reduce the risk of premature convergence to suboptimal solutions. Moreover, a higher value of iterations provides more opportunities to refine the solutions before obtaining the most optimal solution. As in the experiment, the maximum number of iterations used is 200. This is to limit the computational time and resources required in completing the process.

Plotted best distance graph for ‘cities500.txt’ (Elite size case)

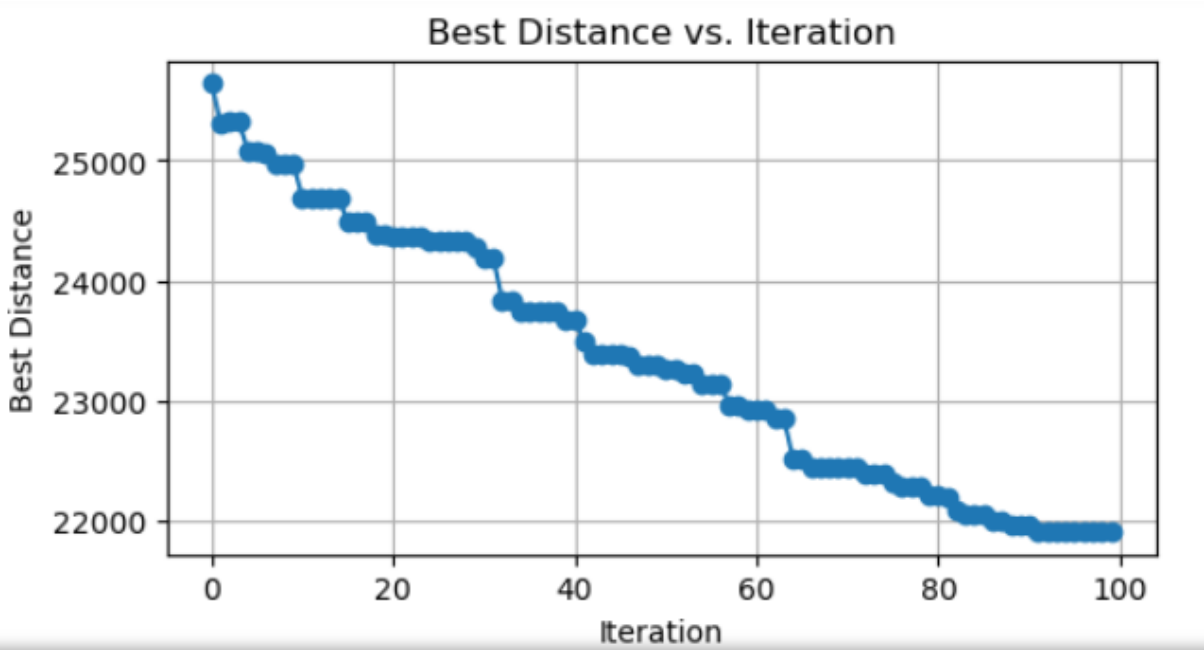
The plotted graphs are referred to the default case for three parent selection method

```
filename = 'cities500.txt'
popSize = 100
eliteSize = 5
mutationProbability = 0.01
iteration_limit = 100
parent_size = 20
tournament_size = 3

best_distances = []

cityList = genCityList(filename)

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 i in range(iteration_limit):
    population = oneGeneration(population, eliteSize, mutationProbability,parent_size,tournament_size)
    distances = [Fitness(p).routeDistance() for p in population]
    index = np.argmin(distances)
    best_route = population[index]
    min_dist = min(distances)
    best_distances.append(min_dist)
    print("Best distance for population in iteration " + str(i) +
          ": " + str(min_dist))

print("Optimal path is " + str(best_route))
```

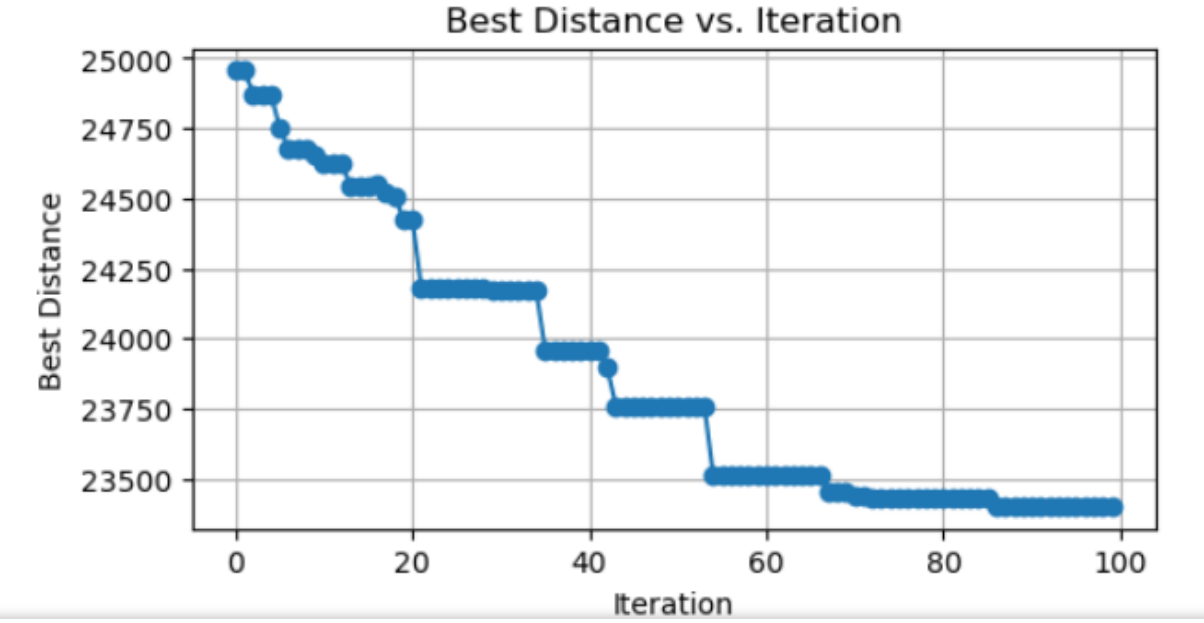
```
plt.figure(figsize=(6, 3))
plt.plot(range(iteration_limit), best_distances, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Best Distance')
plt.title('Best Distance vs. Iteration')
plt.grid(True)
plt.show()
```

1) Random parent selection method

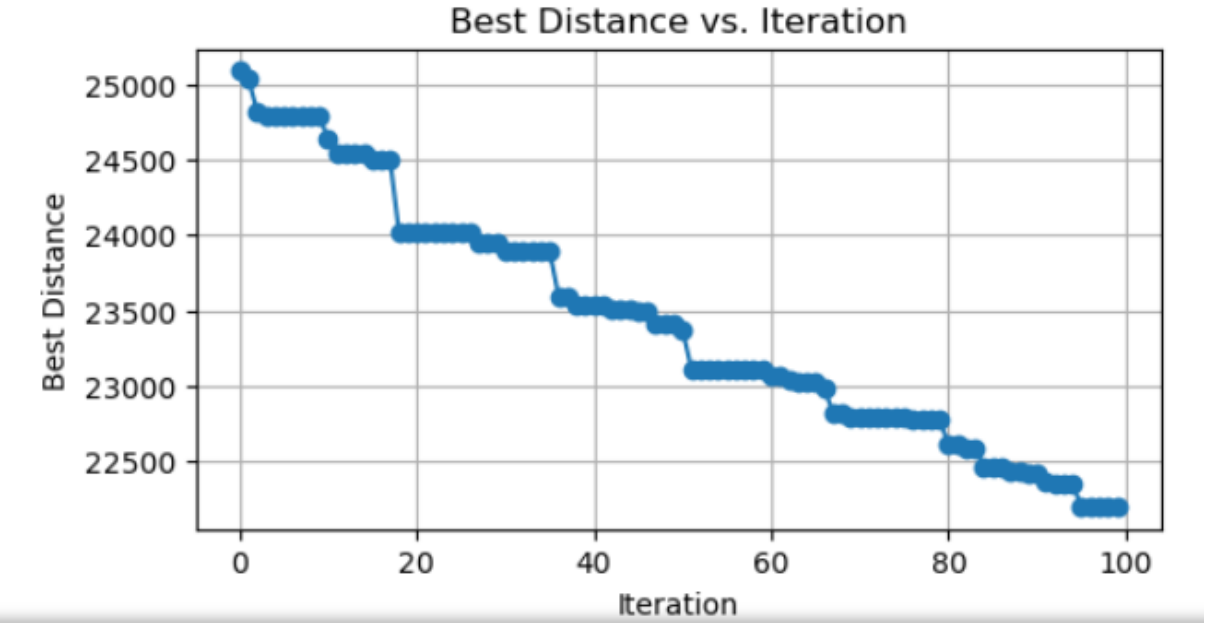
elite size = 1



elite size = 5

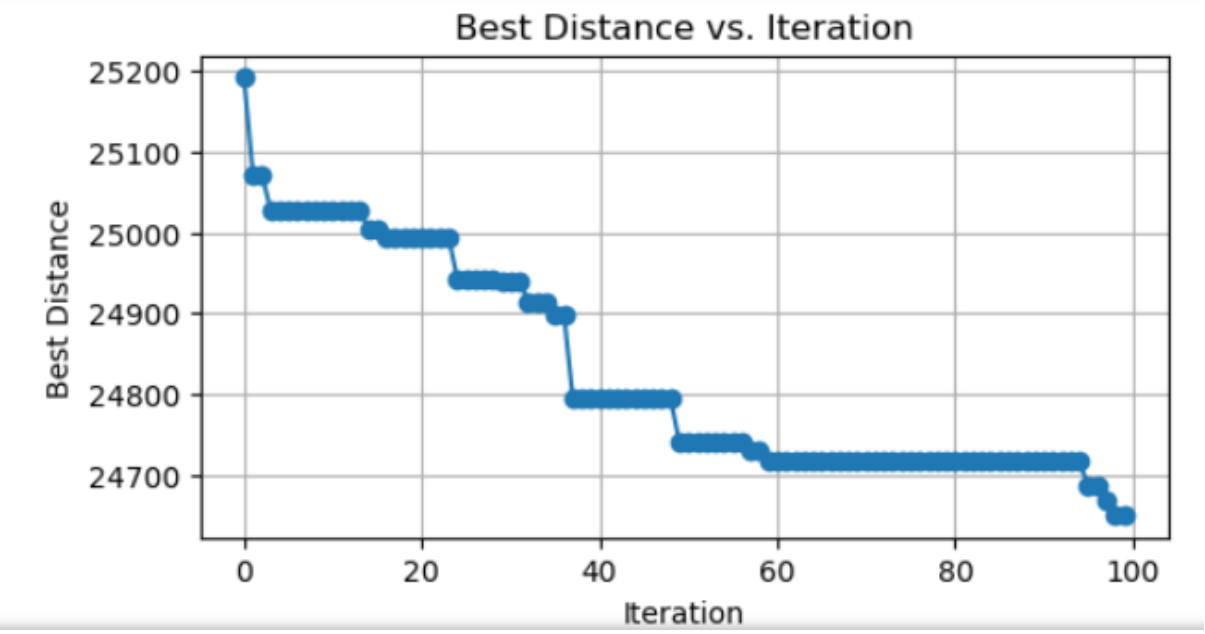


elite size = 10

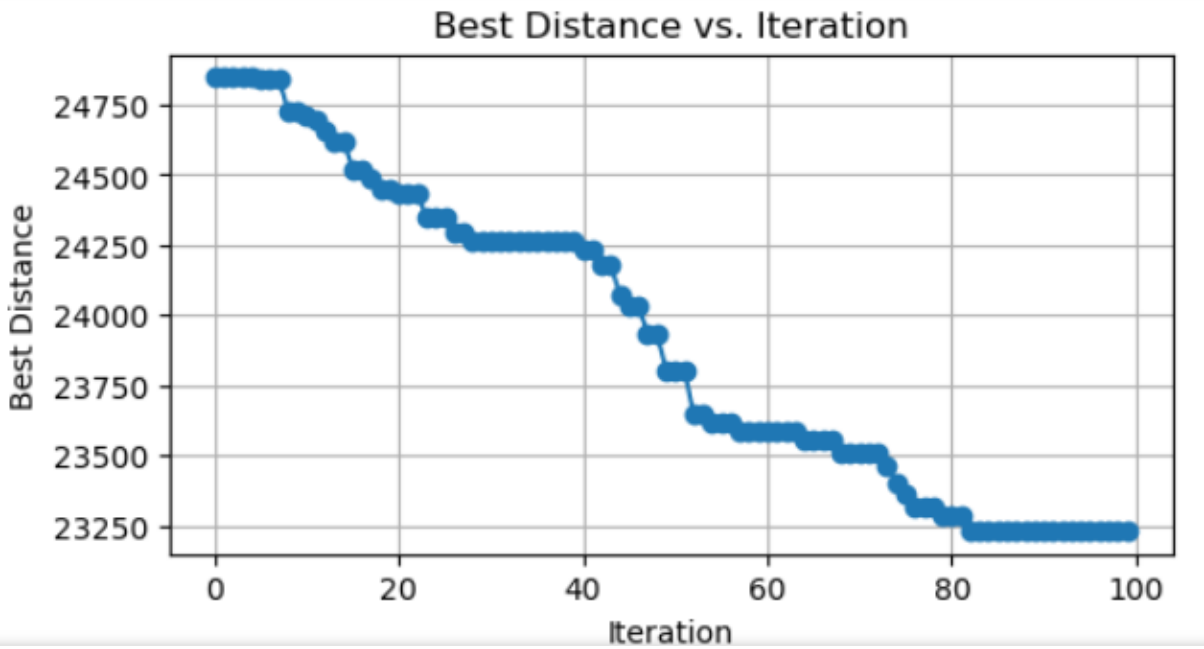


2) Tournament parent selection method

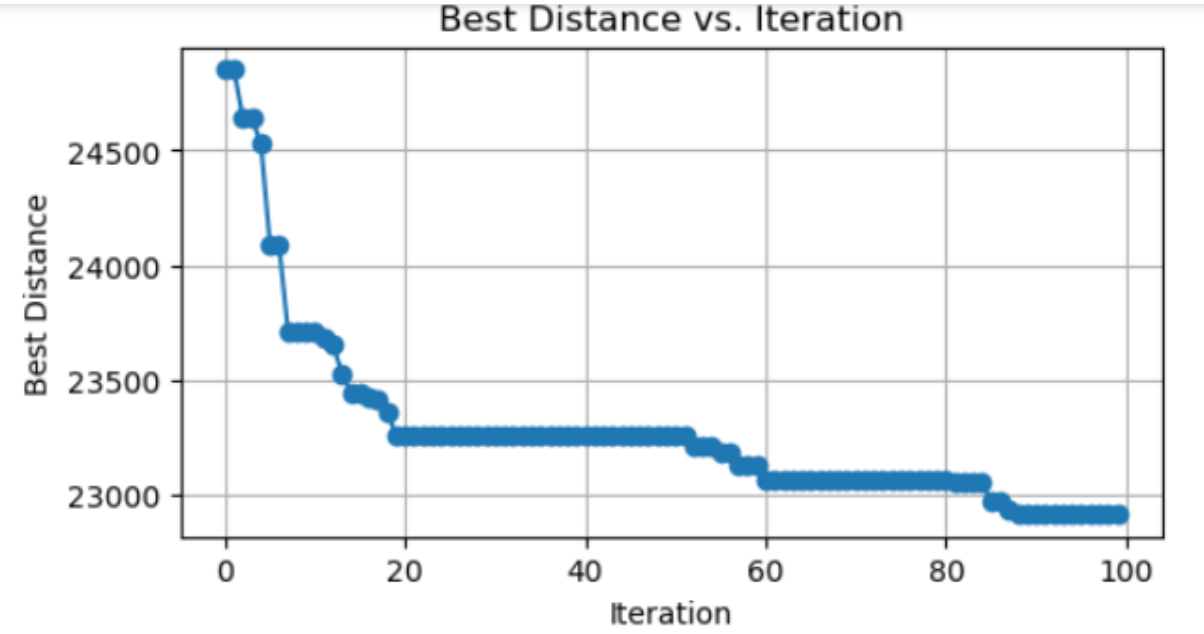
elite size = 1



elite size = 5

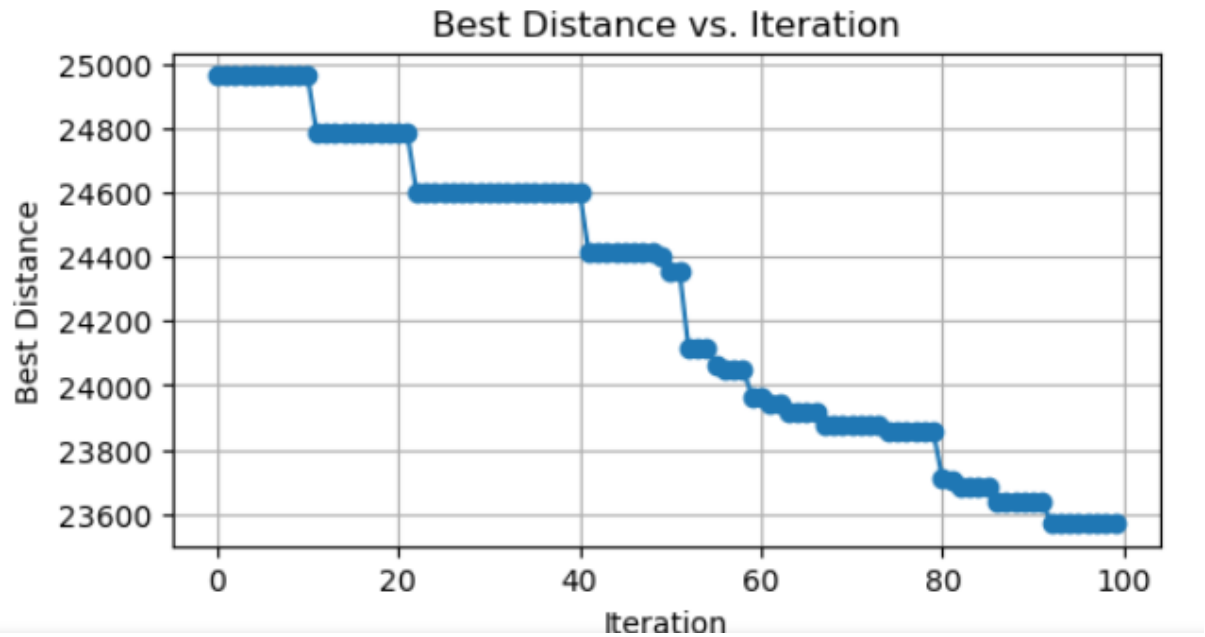


elite size = 10

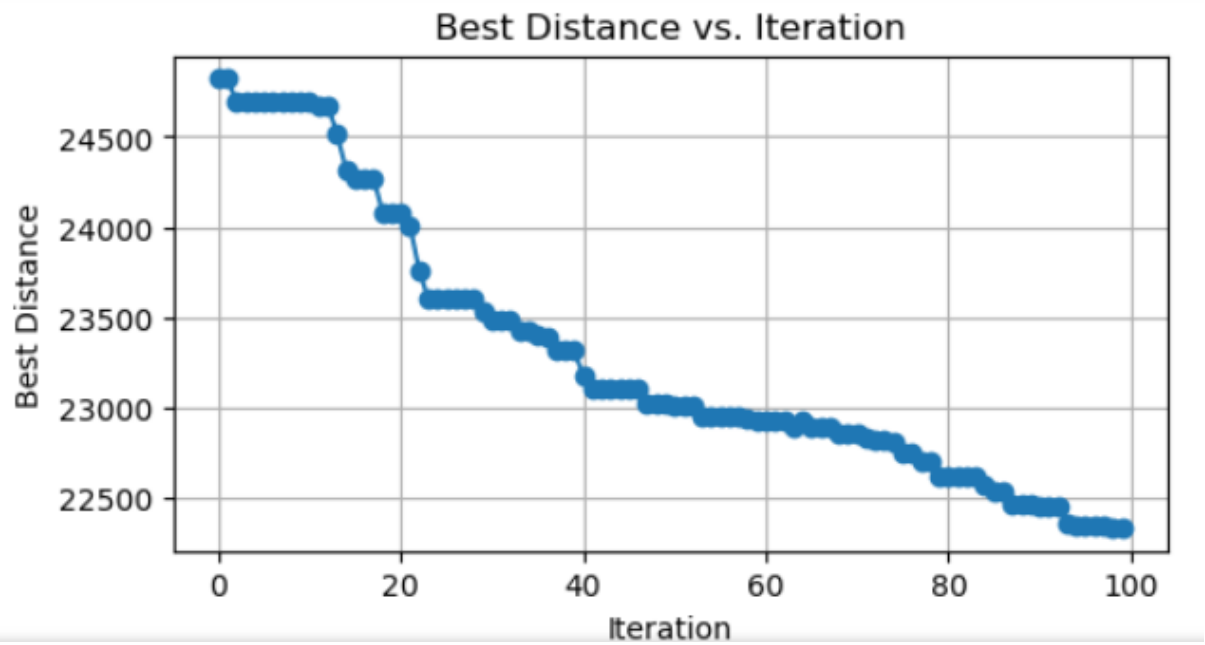


3) Proportional parent selection method

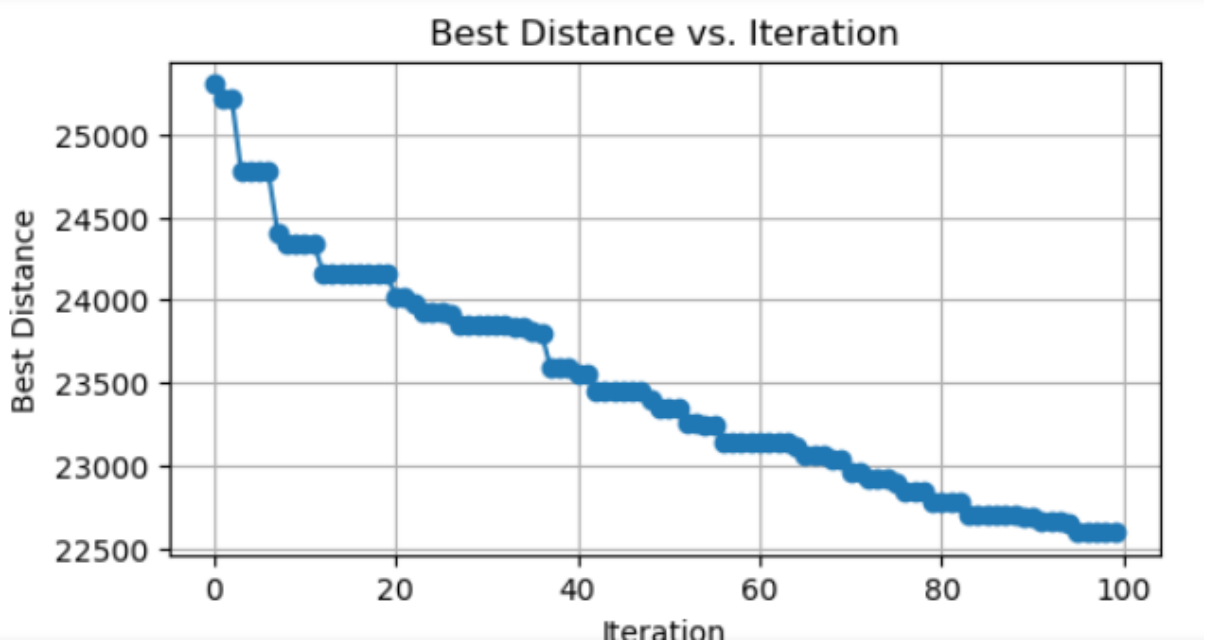
elite size = 1



elite size = 5



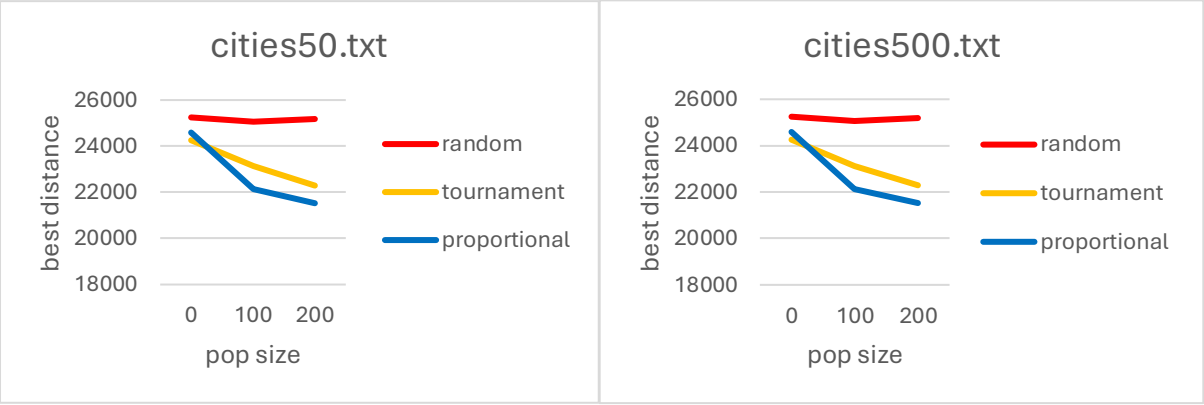
elite size = 10



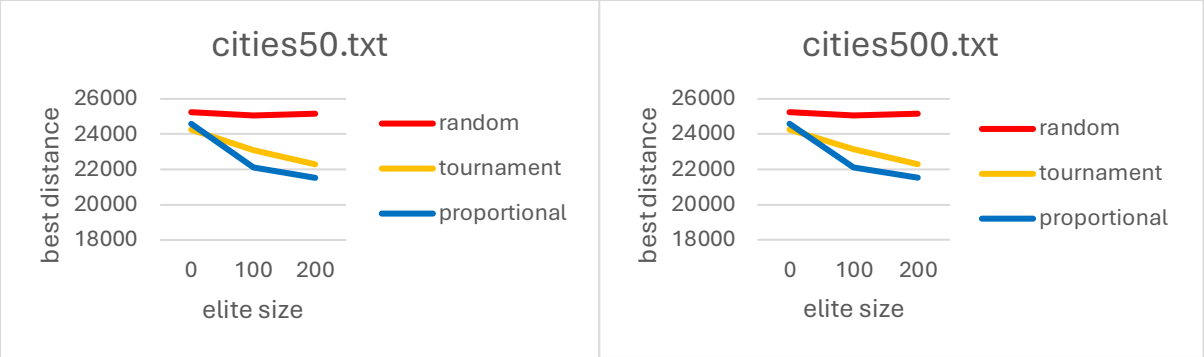
In conclusion, the gradient observed in the random selection method seems almost the same when the elite size increases. Besides, the gradient of reducing the best distance seems er in the tournament and proportional selection parent method as the more elite population size was selected to retain its traits as it went through the iteration. For that reason, it will speed up the best distance to be reduced. However, it may lead to convergence at the local optima point.

Plotted different variables graph for ‘cities50.txt’ and ‘cities500.txt’

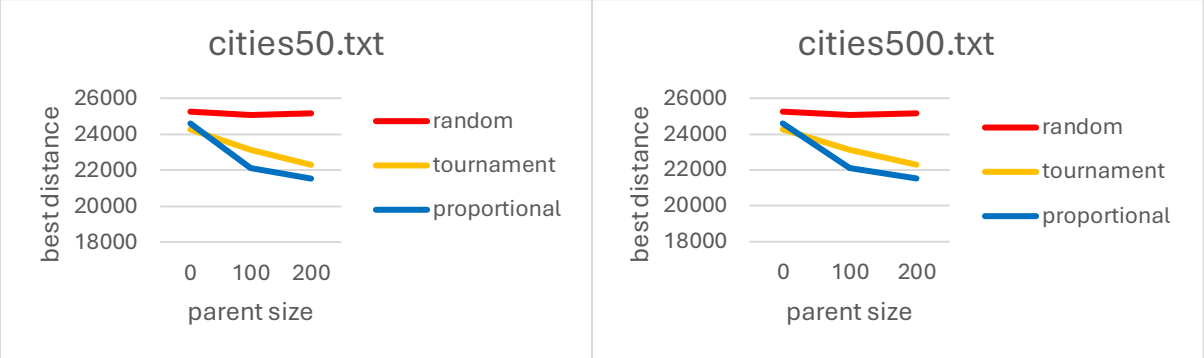
a) Pop size



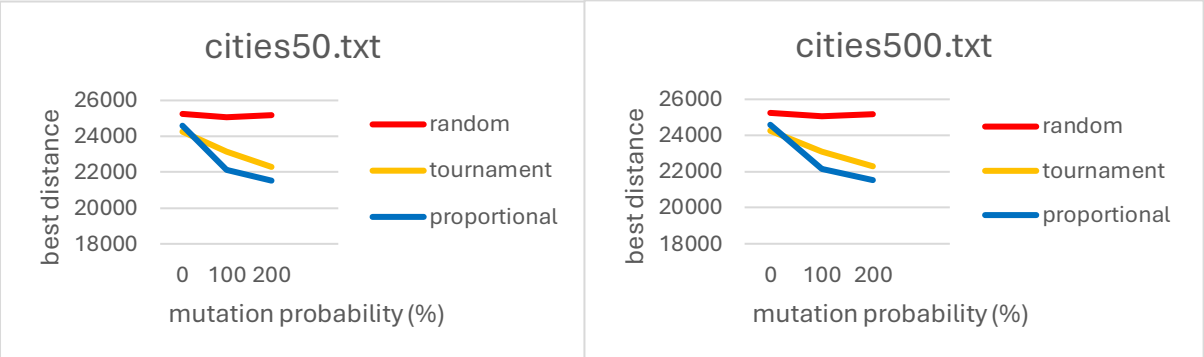
b) Elite size



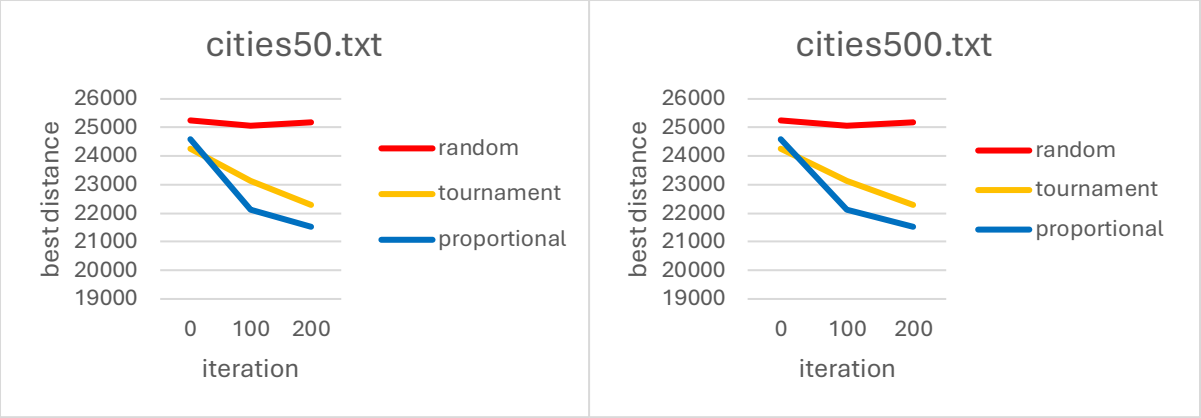
c) Parent size



d) Mutation probability



e) Iteration


cities50.txt
best distance
26000
25000
24000
23000
22000
21000
20000
19000
0
100
200
iteration
random
tournament
proportional
cities500.txt
best distance
26000
25000
24000
23000
22000
21000
20000
19000
0
100
200
iteration
random
tournament
proportional

References

1) Cratecode. (n.d.). *Roulette Wheel Selection*. [online] Available at: https://cratecode.com/info/roulette-wheel-selection. [Accessed on 25/7/2024]

2) Datta, W. by: S. (2024) *Genetic algorithms: Crossover probability and mutation probability*, *Baeldung on Computer Science*. Available at: https://www.baeldung.com/cs/genetic-algorithms-crossover-probability-and-mutation-probability [Accessed on 25/7/2024]

3) Garcia, V.F. (2022). *Elitism in Evolutionary Algorithms | Baeldung on Computer Science*. [online] www.baeldung.com. Available at: https://www.baeldung.com/cs/elitism-in-evolutionary-algorithms. [Accessed on 25/7/2024]

4) Viescinski (2024). Tournament Selection in Genetic Algorithms. | *Baeldung on Computer Science*. [online] Available at: https://www.baeldung.com/cs/ga-tournament-selection [Accessed on 25/7/2024]

5) Rivera, M.M., et al. (2022). Embedded system for model characterization developing intelligent controllers in industry 4.0, *Artificial Intelligence and Industry 4.0,* 57-91, doi: 10.1016/b978-0-323-88468-6.00004-8. [Accessed on 26/7/2024]