Solving Travelling Salesman Problem with Genetic Algorithm

Özyeğin University

CS 451 – Introduction to Artificial Intelligence

Assignment-2

Prepared by: Göksel Can ÖNAL

April 26, 2021

Introduction

Travelling Salesman Problem

Travelling salesman problem (TSP) tries to find shortest possible route among given cities. Salesman's aim is that visit the all given cities and return the origin city with shortest route. It is a well-known algorithmic problem in the computer science. Although it seems an easy problem, it is very difficult to solve with huge amount of cities. It is a combinatorial optimization problem. As the number of cities increases, number of possible routes increases exponentially. For example, if there are 4 cities, there are 4! possible routes and it can be solved easily but if there are 20 cities, there are 20! possible choices. So if you think there are thousands of cities, it may take years to solve. Even though the problem is very hard to solve, there are some heuristics and algorithms are known. In this assignment, travelling salesman problem is tried to solve with genetic algorithm.

Genetic Algorithm

Genetic algorithm is a heuristic that is inspired by Charles Darwin. This algorithm is the process of natural selection. Fittest individuals are selected in order to produce child of next generation. There are 5 steps of the algorithm in order to make an evolution. Steps are initial population, fitness function, selection, crossover and mutation. In this assignment a route can be thought as a chromosome. Each route has cities. A city can be thought as a gene of its chromosome. Also, population is like a package which can store chromosomes. Evolution process done by this steps. Selection from population according to fitness values. Crossing over for producing child with selected chromosomes and mutation for producing child. After make this process for population size times, new population can be produced.

Main goal of this assignment, implement the genetic algorithm to solve travelling salesman problem for many different cases and analyze the test results of that cases. In this report, you will be informed used classes and the genetic algorithm itself in the Algorithm section. In section Implementation Details, detailed implemented methods, its pseudo codes and how they work are discussed. In Result section, implemented test cases results are visualized with graphs and tables to see difference between different cases easily. In Conclusion part, there is a summary about that report and overcome about that traveling salesman problem with genetic algorithm.

Algorithms

First, initial population must be created. Initial population can be thought as a package which has many randomly ordered different routes. Second, fitness function is required to make elimination. Fitness function determines how fit an individual route is. In the population, there are many routes. These routes have its own length of route. According to fitness function, the shorter route length one is more fit. Third, selection is just a tournament that has a winner with fittest route. The selection is used to select parents in the process. Fourth step is crossover. Crossover produces a child from the selected parents. It just randomly selects some parts of parents and combines them to make a new child route. Fifth is mutation. After child is produced, mutation can be occurred on that child with a given probabilistic mutation rate. Using these steps many children will be produced and a new population will be created with these collection of children. So this is an evolutionary algorithm.

Every city has its own UID and x, y values to represent coordinates. City manager is used for creating new cities and add these cities to city_lst list. The list represents the country. In this assignment given cities (Figure 1.1) are used as a country.

City_0(60, 200)	City_10(180, 100)
City_1(180, 200)	City_11(60, 80)
City_2(80, 180)	City_12(120, 80)
City_3(140, 180)	City_13(180, 60)
City_4(20, 160)	City_14(20, 40)
City_5(100, 160)	City_15(100, 40)
City_6(200, 160)	City_16(200, 40)
City_7(140, 140)	City_17(20, 20)
City_8(40, 120)	City_18(60, 20)
City_9(100, 120)	City_19(160, 20)

Figure 1.1

Route class is used for creating a random ordered route. It uses the whole cities in figure 1.1 and generate a route in random way. Also routes distances and fitness values are calculated calc_route_distance and calc_fitness methods respectively in Route class. RouteManager class takes given cities and population size. It is used for creating population, it adds routes to list population size times. GeneticAlgorithmSolver class tries to solve traveling salesman problem with evolve method. Evolve method runs 100 times. According to elitism value, it can work differently. If the elitism is false, evolve method copies the given population then make tournament for selecting two parents, implement crossovers the selected parents for producing the child and according to mutation rate, mutation may be applied to the child. Then, child is added to new population. This process done by length of given population times and it adds all the evolved children to copied population. So at the end, there is a new evolved population. Since evolve method runs 100 times, it creates new population and implement evolve method on that new population again and again. If the elitism is true, first it takes the best route from the given population and adds to new population then it just makes the same process with the previous but it does not implement mutation on the child. The aim is here keeping the best route of given population in the next generation. This method is increase the probability of finding shortest path more. In this assignment, shortest possible routes tried to find by these processes. Given (Figure 1.1) population are evolved over and over again and the best fitting route is selected from the last evolved population.

Implementation Details

In main method, country is built by given cities and 50 size of population is created with RouteManager then shortest route of the population is printed in the terminal. Then, same population is exposured to evolution process. Evolution is done by 100 times.

Tournament

It takes the population and make a copy of that population to a RouteManager which is called selection. Selection has length of tournament size. At first the aim is normalizing the fitness values. It just sums of all routes' distances then it divides all routes' distance by the sum and append the

result a list which is called probability. So the list is store the probability of all routes in order to make the calculation easy. Shortest route has high probability. The method takes a route from the given population and adds the selected route to selection. Method does not select a route randomly; it selects most likely to the route of high probability. It is kind of random but not really much. These process is done by tournament size times. So at the end, there is a given size of population routes. In order to find winner, the shortest route (best route) is selected as winner and method returns the winner.

```
selection <- RouteManager (cities, tournament_size)
selection_idx <- 0
loop length of selection times
        sum_fitness <- 0
        for every route in routes
                fitness_val <- route.calc_distance ()
                fitness_val <- fitness_val * 100
                sum_fitness <- sum_fitness + fitness_val
        probability <- list ()
        for every route in routes
                normalized_fitness = (route.calc_distance() *100) /sum_fitness
                probability <- append (normalized_fitness)
        index < - 0
        random_num <- random number between 0 - 1
        while random_num is greater than 0
                random_num <- random_num – probability[index]
                index <- index + 1
        index <- index + 1
        selected_route <- routes.get_route (index)
        selection <- set_rouete (selection_idx, selected_route)
        selection_idx <- selection_idx + 1
winner <- selection.find_best_route ()
return winner
```

Crossover

It takes 2 route for producing a new one by combining the given routes. It takes a random value between 0 and length of route, if the random value is length of route 1, in other words, if random value is index of the last city of route 1, it just selects last city from route 1 then append whole part from route 2 except the selected city from route1. The combined route is a child gene. Let's assume route 1 is ACDFEB, route 2 is FEDCBA and the selected random value is 5. It can be seen in figure 2.1. Route 1 and route 2 can be thought as parents to produce child.

parent_1 => ACDFEB

parent_2 => FEDCBA

random_1 => 5

child => BFEDCA

Figure 2.1

As it can be seen, last city is selected from parent1, rest of the cities are selected from parent2 except the selected (underlined gene) city from parent1. if random value index is not the last city of route 1, it takes an another random value between random_1 + 1 and length of route. It selects the cities between random_1 and random_2 (random_2 is not included) from route_1 and then takes the whole cities from route_2 except the selected cities from route_1. Again assume parent 1 and parent 2 is same in the above example but there are two random values 2 and 5. In figure 2.2, it can be seen.

parent_1 => ACDFEB

parent_2 => FEDCBA

random_1 => 2

random_2 => 5

child => DFECBA

Figure 2.2

It is clear that random values are used for parent 1, selected cities from parent 1 is determining which cities will be selected from parent 2. So the method randomly produces a new route by using parents.

```
child <- Route(cities)
start <- a random integer between 0 – length of route_1
if start is less than length of route_1 -1
        end <- a random integer between start + 1 – length of route_2
        s <- end - start
        k <- 0
        while end – start is greater than 0
                random_city <- route_1.get_city (start)
                child <- assign_city (k, random_city)
                start <- start + 1
                k <- k + 1
        for every city in route_2
                random_city <- route_2.get_city(city)
                if random_city not in child.route
                        child <- assign_city (s, random_city)
                        s <- s + 1
else
        k = 1
        random_city = route_1.get_city(start)
        child <- assign_city (0, random_city)
        for every city in route_2
                random_city <- route_2.get_city (city)
                if random_city not in child.route:
                        child <- assign_city (k, random_city)
                        k <- k + 1
return child
```

Mutate

It takes a route and swap two random cities in the route. The method implements this process by random also. According to the given mutation rate, mutation can be applied the route or not. It takes two random values between 0 and length of route. Then, it takes two cities that have been placed to the selected random values' index of route. Then, swap these cities. In figure 2.3, it can be seen.

child => ACBFED

random_1 => 3

random_2 => 1

mutated_child => AFBCED

Figure 2.3

The method does not implement these process every time. In the beginning it takes a random value between 0 and 1. If the random value is less than the given mutation rate, it means route will be mutated, else mutation is not implemented. Also, there is a case of random 1 and random 2 is the same value. In this kind of situation, it does not swap cities (Figure 2.4).

child => ACBFED

random_1 => 4

random_2 => 4

mutated_child => ACBFED

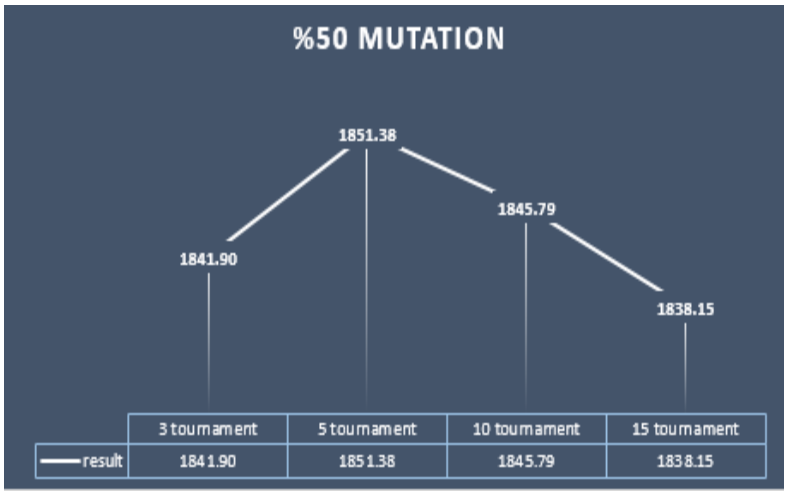
Figure 2.4

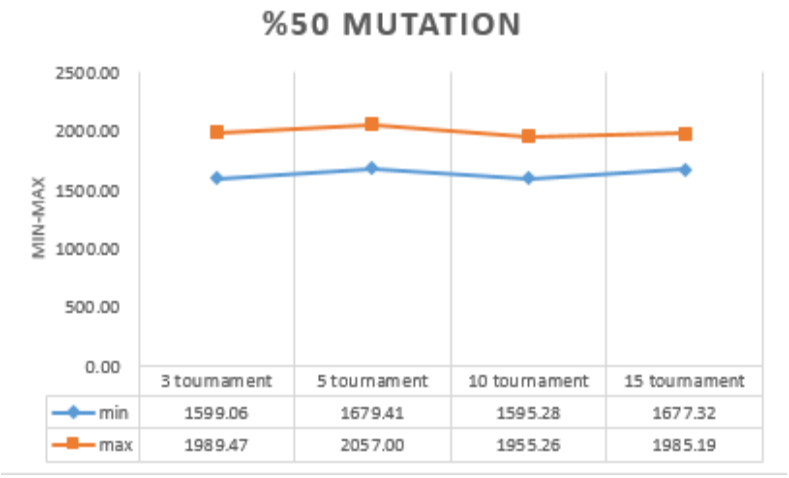
```
for every city in route
        if a random value between 0 - 1 is less than mutation_rate
                rand_1 <- a random integer between 0 – length of route
                rand_2 <- a random integer between 0 – length of route
                dna_1 <-route.get_city (rand_1)
                dna_2 <-route.get_city (rand_2)
                temp <- dna_1
                route <- assign_city (rand_1, dna_2)
                route <- assign_city (rand_2, temp)
return route
```

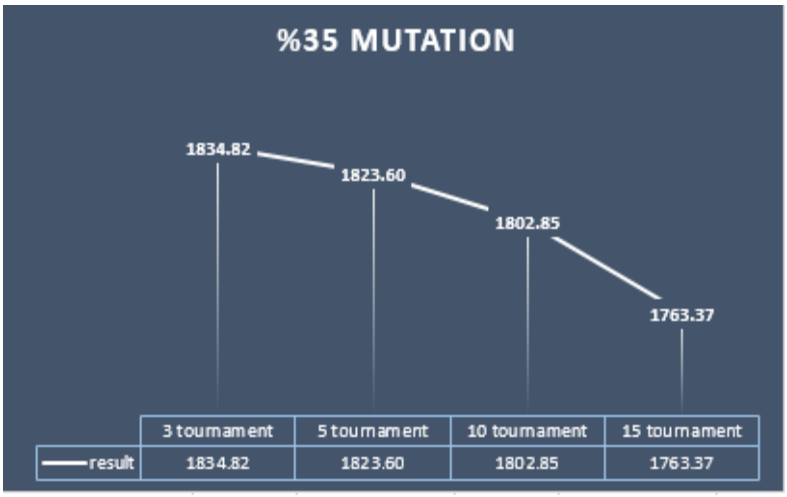
Evaluation process uses these methods, tournament, crossover and mutate. All these processes occur in random way. So the aim is there providing variety. Since this algorithm uses stochastics process, it cannot guarantee the optimum solution. Also the results can change according to given mutation rate, population size and tournament size even the if you run the program again. The aim is that assignment is finding the shortest route by testing different values of mutation rates and sizes.

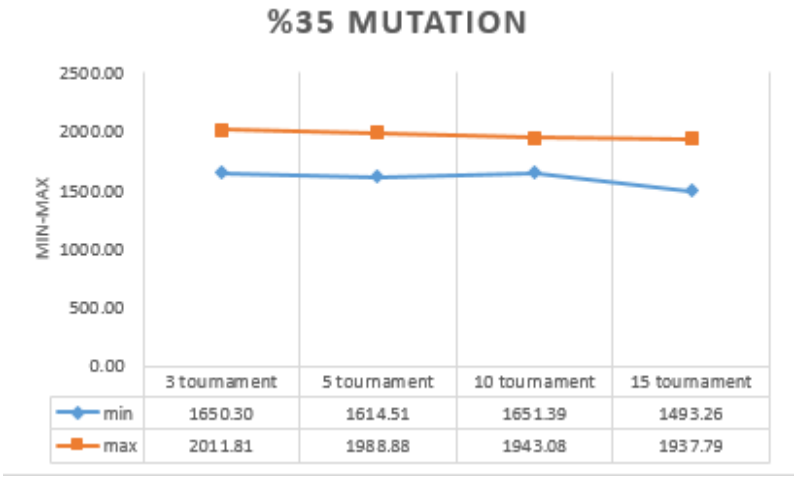
Results

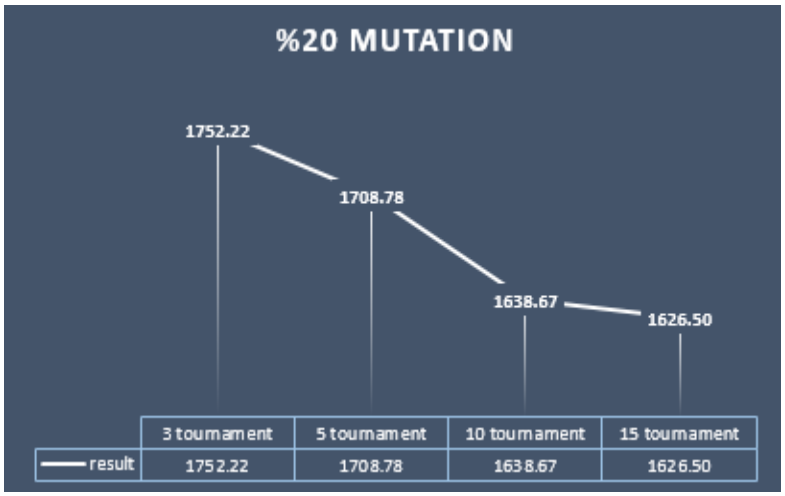
In order to analyze the results, many test case are applied. First phase, elitism is used as false. According to the different mutation rates, algorithm is tested with different tournament sizes. For example, mutation rate is selected as 0.5 and tournament size is 3. Algorithm is implemented 40 times and the average of 40 results is taken. This method is implemented with tournament sizes 3, 5, 10, 15 and mutation rate %50, %35, %20, %10, %5. For every case, algorithm is implemented 40 times. Average route distance, min route distance and max route distance are indicated below with graphs. In the next page, it can be seen that average route path (left) and min-max (right) graphs with using different mutation rate and different tournament size cases.

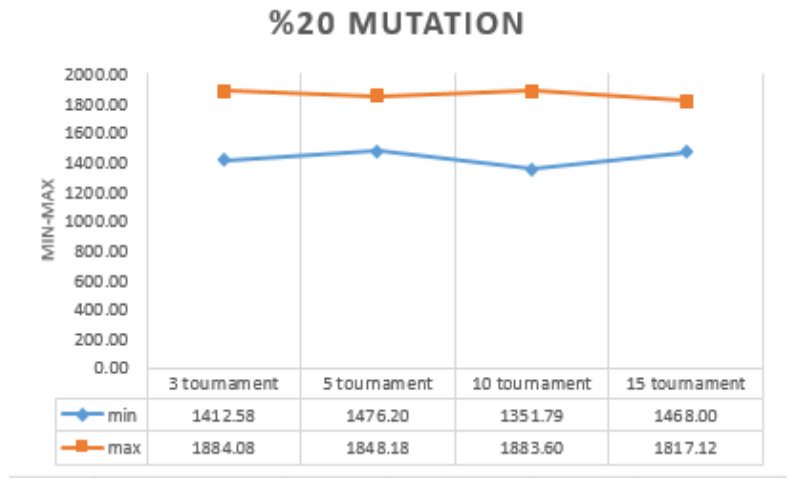

%50 MUTATION
1851.38
1845.79
1841.90
1838.15
3 tournament
5 tournament
10 tournament
15 tournament
result
1841.90
1851.38
1845.79
1838.15

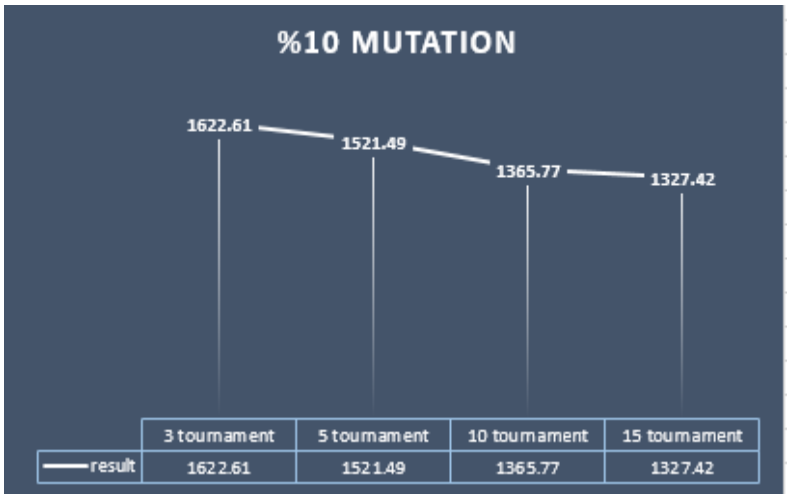

%50 MUTATION
2500.00
2000.00
1500.00
1000.00
500.00
0.00
MIN-MAX
3 tournament
5 tournament
10 tournament
15 tournament
min
1599.06
1679.41
1595.28
1677.32
max
1989.47
2057.00
1955.26
1985.19

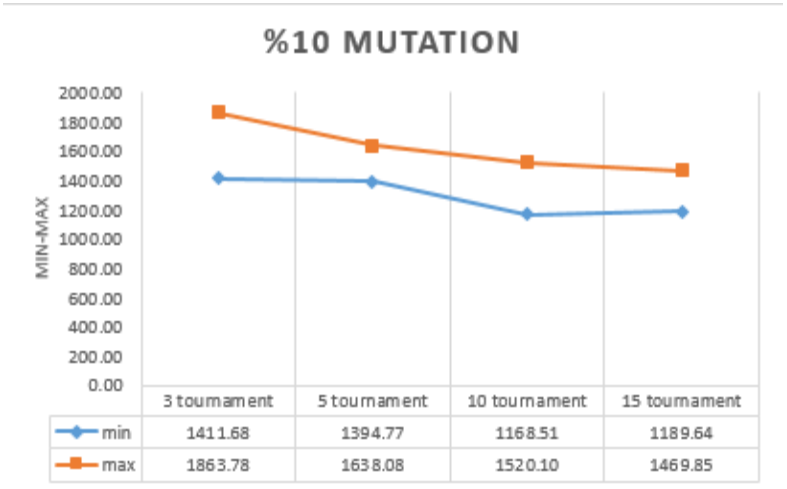

%35 MUTATION
1834.82
1823.60
1802.85
1763.37
3 tournament
5 tournament
10 tournament
15 tournament
result
1834.82
1823.60
1802.85
1763.37

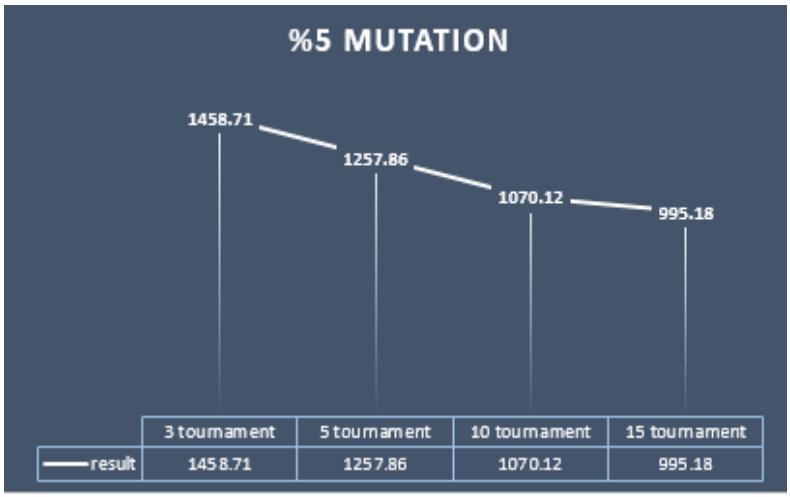

%35 MUTATION
2500.00
2000.00
1500.00
1000.00
500.00
0.00
MIN-MAX
3 tournament
5 tournament
10 tournament
15 tournament
min
1650.30
1614.51
1651.39
1493.26
max
2011.81
1988.88
1943.08
1937.79

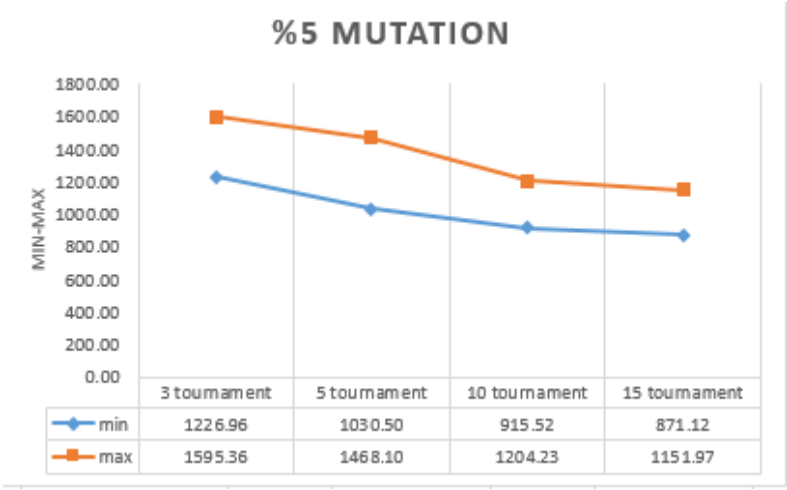

%20 MUTATION
1752.22
1708.78
1638.67
1626.50
3 tournament
5 tournament
10 tournament
15 tournament
result
1752.22
1708.78
1638.67
1626.50


%20 MUTATION
2000.00
1800.00
1600.00
1400.00
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00
MIN-MAX
3 tournament
5 tournament
10 tournament
15 tournament
min
1412.58
1476.20
1351.79
1468.00
max
1884.08
1848.18
1883.60
1817.12









According to these graphs, all cases are indicated below. As it can be seen in tables, there are 3 columns which are average, min and max. For all column, most efficient one is colored with yellow and most inefficient is colored with orange. According to yellow and orange cells, best and worst cases are indicated for every table with green and red color respectively.

mutation rate=0.5	average	min	max
3 tournament	1841.90	1599.06	1989.47
5 tournament	1851.38	1679.41	2057.00
10 tournament	1845.79	1595.28	1955.26
15 tournament	1838.15	1677.32	1985.19

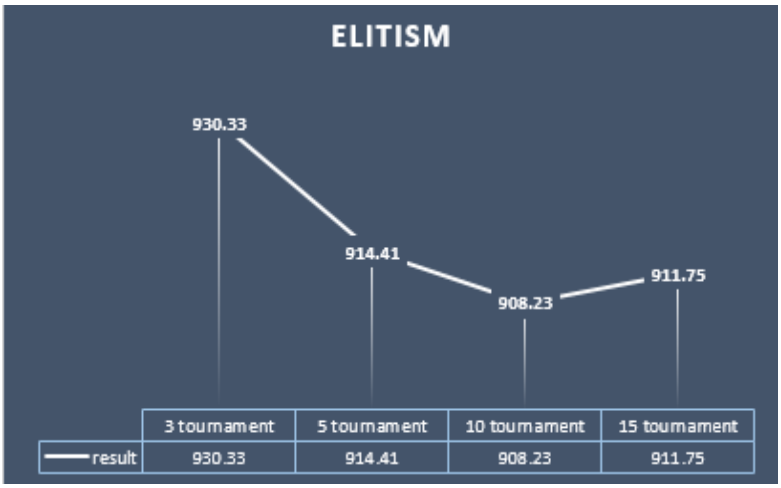
mutation rate=0.35	average	min	max
3 tournament	1834.82	1650.30	2011.81
5 tournament	1823.60	1614.51	1988.88
10 tournament	1802.85	1651.39	1943.08
15 tournament	1763.37	1493.26	1937.79

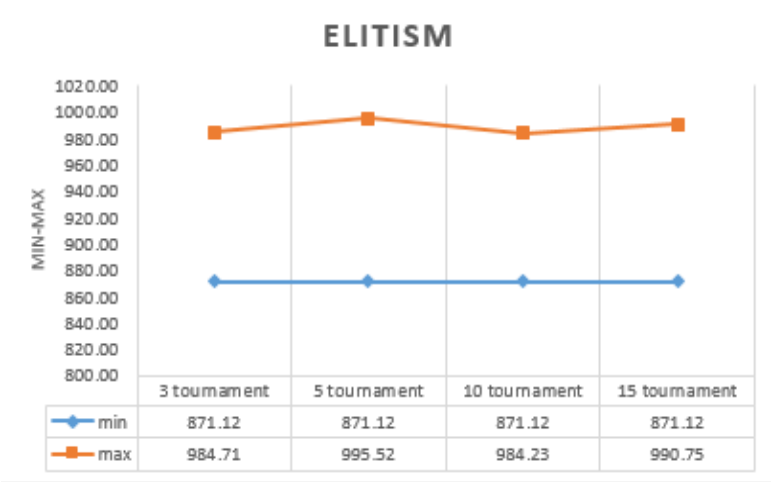
mutation rate=0.2	average	min	max
3 tournament	1752.22	1412.58	1884.08
5 tournament	1708.78	1476.20	1848.18
10 tournament	1638.67	1351.79	1883.60
15 tournament	1626.50	1468.00	1817.12

mutation rate=0.1	average	min	max
3 tournament	1622.61	1411.68	1863.78
5 tournament	1521.49	1394.77	1638.08
10 tournament	1365.77	1168.51	1520.10
15 tournament	1327.42	1189.64	1469.85

mutation rate=0.05	average	min	max
3 tournament	1458.71	1226.96	1595.36
5 tournament	1257.86	1030.50	1468.10
10 tournament	1070.12	915.52	1204.23
15 tournament	995.18	871.12	1151.97

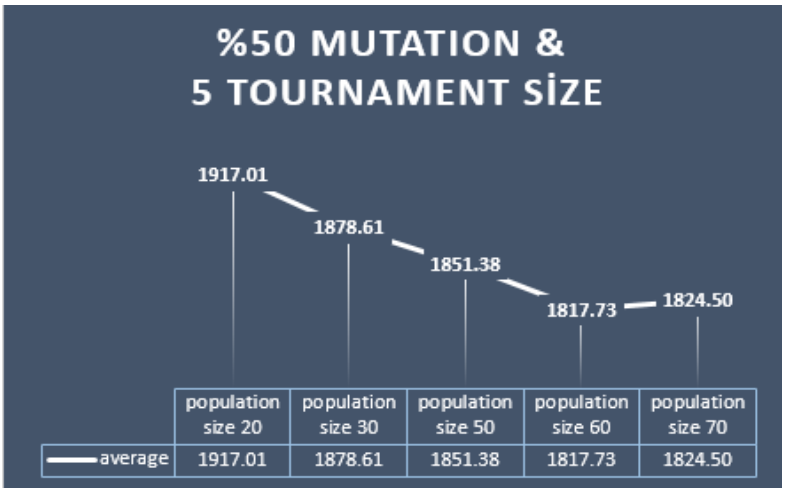
Second phase, elitism is implemented. Since elitist approach cause not undergo a mutation, mutation rate is not important here. Below it can be seen average graph and min-max graph also detailed table results of elitist cases.

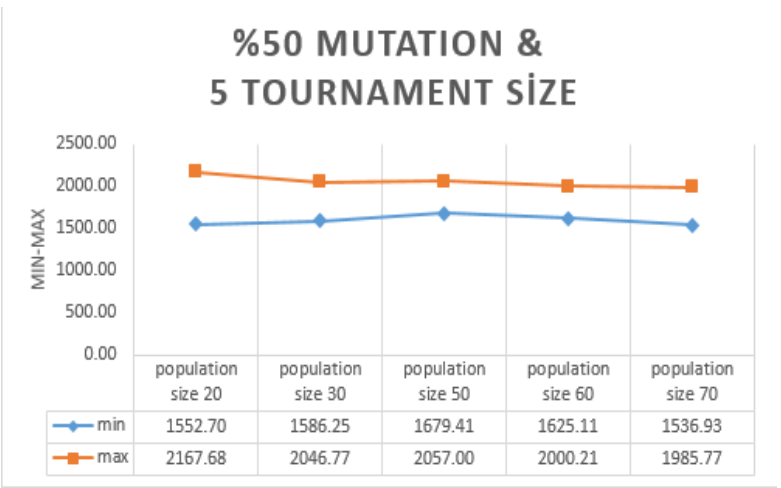




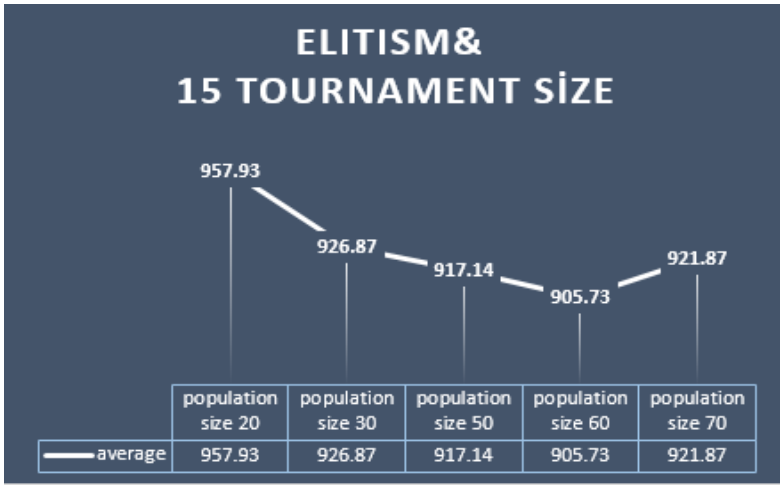
elitism	average	min	max
3 tournament	930.33	871.12	984.71
5 tournament	914.41	871.12	995.52
10 tournament	908.23	871.12	984.23
15 tournament	911.75	871.12	990.75

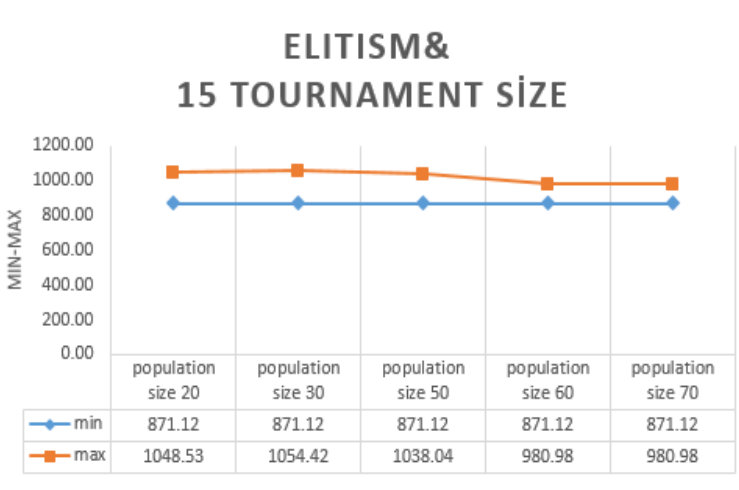
Third phase, after deciding most efficient case (elitism) and most inefficient case (mutation rate is 50% and tournament size is 5), new tests are applied in terms of different population sizes. For the determined cases, population size is used as different values and results are observed.





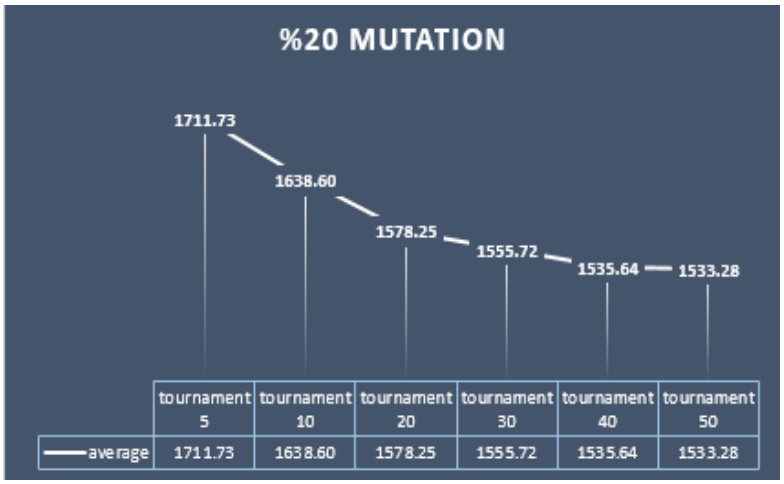
mutation rate= 0.5,tournament size=5	average	min	max
population size 20	1917.01	1552.70	2167.68
population size 30	1878.61	1586.25	2046.77
population size 50	1851.38	1679.41	2057.00
population size 60	1817.73	1625.11	2000.21
population size 70	1824.50	1536.93	1985.77

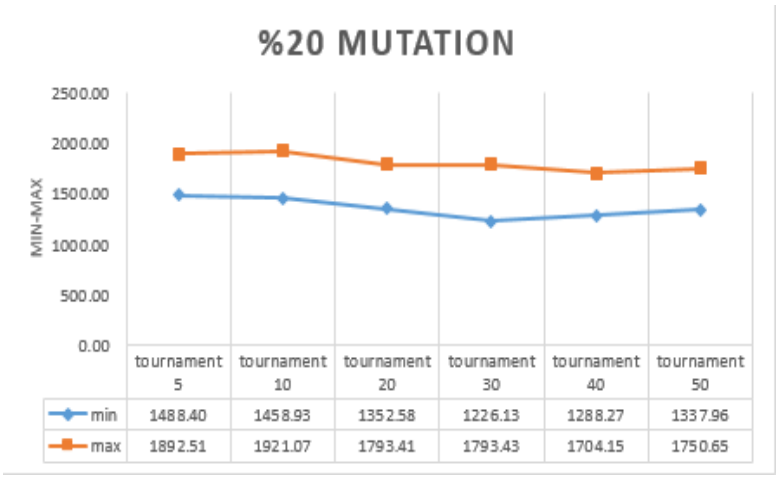




elitism, tournament size=15	average	min	max
population size 20	957.93	871.12	1048.53
population size 30	926.87	871.12	1054.42
population size 50	917.14	871.12	1038.04
population size 60	905.73	871.12	980.98
population size 70	921.87	871.12	980.98

Last phase, after deciding high population size or high tournament size will cause more optimal results, in order to strength that theory, one last test is applied. A random case is determined (%20 mutation rate and 50 population size) and for different tournament sizes, algorithm is used and as it can be seen, choosing higher size is resulted with finding more optimal routes.





mut_rate 0.2,pop_size = 50	average	min	max
tournament 5	1711.73	1488.40	1892.51
tournament 10	1638.60	1458.93	1921.07
tournament 20	1578.25	1352.58	1793.41
tournament 30	1555.72	1226.13	1793.43
tournament 40	1535.64	1288.27	1704.15
tournament 50	1533.28	1337.96	1750.65

Conclusion

Genetic algorithm is not guarantee the optimal solution and it works stochastically. So, the tests are not representing a fully meaningful results but it can give some general idea. According to first phase of test, tournament size 3 is bad case for travelling salesman problem and most of the table 15 size tournament is greatest case. So, when the tournament size is increased, it will increase the probability of finding most satisfied route. If the results are analyzed in terms of mutation rates, when the mutation rate is low, it finds more optimal routes. In general, 3 tournament size is loser and 15 tournament size is winner but as it can be seen, when mutation rate is 50%, general idea is not valid. So, that means when mutation rate is very high, expected result may not be obtained. In terms of the obtained results when mutation rate decreases or tournament size increases, algorithm finds more short distanced route. According to second phase of test which is used elitism, optimum case is 10 tournament size. If the results are analyzed, it can be seen that results are very optimal when it compared to results of first phase. Since elitist approach is not use mutation, results become optimal. The selection method, selects parents in terms of their fitness value, if the route is more fit, then it has high probability to choice. So, when parents are selected wisely and there is no mutation, results are affected positively. According to third phase, tests show that when the population size increases, it makes the results more optimal. In last phase, like third phase, increasing tournament size makes the results more optimal. Many different cases are implemented 40 times and among all route distance, obtained minimum result is 871.1173538. All elitist tests find minimum value even case tournament size 3. Since cases tested with 40 times, at least one of the test found that value. If the elitism is false, mutation rate 5% and tournament size 15 case found the same 871.1173538 value. As it discussed, low mutation rate and high tournament size shows its optimality.