Solving TSP with Genetic Algorithm (GA)

김 주 호, 신 의 진

(Joo-Ho Kim[1], Yi-Jin Sin[2])

[1]School of Mechanical & Control Engineering, Handong Global University

[2]School of Computer Science & Electrical Engineering, Handong Global University

Abstract: Genetic algorithm (GA) can be used as a effective solution-searching algorithm when solving NP class problems that are difficult to find solutions in certain ways. However, because Genetic algorithms focus on solving problems, they require higher computational costs compared to those used for P-class problems. Therefore, the goal of this project is to understand how to deal with the NP class problems by solving one of the NP hard problems, the Traveling Salesman Problem (TSP), through a Genetic algorithm. In addition, we compare the results of adjusting the parameter values of each process to slightly improve the inefficiency of the cost aspect and discuss how to determine the optimal parameters according to the case.

Keywords: Genetic Algorithm, Traveling Salesman Problem, Elitism, Crossover, Mutation

I. Introduction

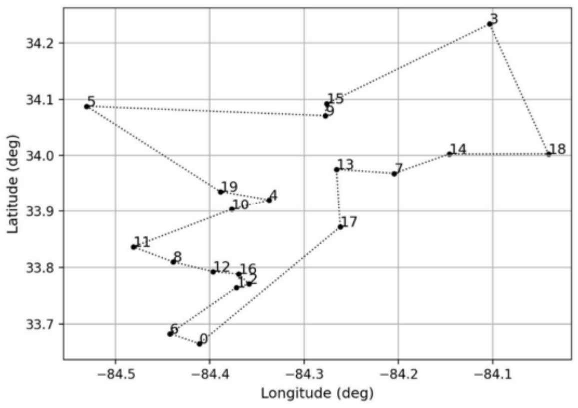


Fig 1. Visualization of coordinates

In this report, the optimal path of the shortest cycle is explored using the Genetic algorithm (GA) by studying how to find the shortest cycle path to visit each delivery point once and return to the origin through the location information of each delivery point. This study provides a path with the lowest fitness value by storing one cycle path calculated based on the Haversine formula as a fitness value. It is expected to improve delivery efficiency by exploring the optimal routes for a total of three different cities and providing them to drivers.

II. Program composition and theory

2.1 Program description

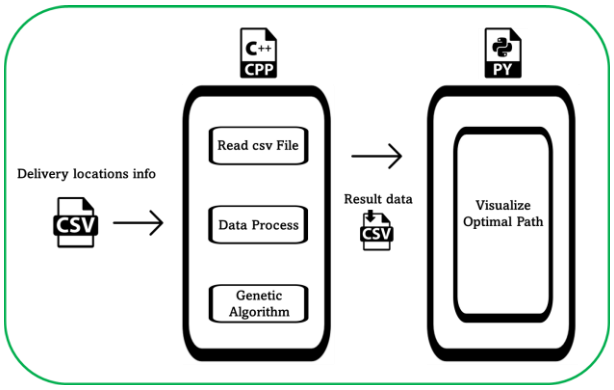
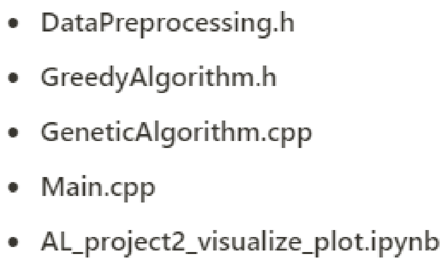


Fig 2. Visualization of program description

The composition of the program can be divided into four operations. Most of the operations are performed in C++, and Python is used for graph visualization. First, a csv file containing information of each airport is read and stored in an appropriate data type. Next, weather information and path information of each airport are inputted by the user and stored. In the third step, the GA is applied based on the stored data. The program outputs the optimal path. Finally, the resulting data is saved as a csv file and visualized through Python. Fig 3. represents the program consists of three code files.



- DataPreprocessing.h
- GreedyAlgorithm.h
- GeneticAlgorithm.cpp
- Main.cpp
- AL_project2_visualize_plot.ipynb

Fig 3. List of program codes

DataPreprocessing.h file contains delivery location information and functions related to data preprocessing that store the Haversine distance of each path in an adjacent matrix.

Greedy Algorithm.cpp file contains functions related to the Greedy algorithm, which is used to find a optimal path using the Greedy algorithm.

GeneticAlgorithm.cpp file contains functions related to the GA. This includes codes for initialization, fitness evaluation, selection, elitism, crossover, and mutation, which are included in the processes of the GA.

Main.cpp file is the main file of the program. When you run the program, it reads CSV data using the functions contained in **DataPreprocessing.h** and makes the data into adjacency matrices. The results of the Greedy algorithm are output first, and the results of the GA are output next. The user can set the size of the initial population and the parameter values for each production. The optimal path found through the program is converted into a **CSV** file and stored.

AL_project2_visualize_plot.ipynb file contains the code to visualize the graph of the initial condition or the graph of the result for the shortest path.

2.2 Flowchart of the program

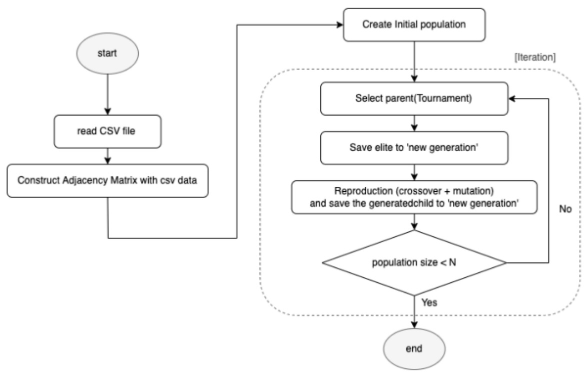
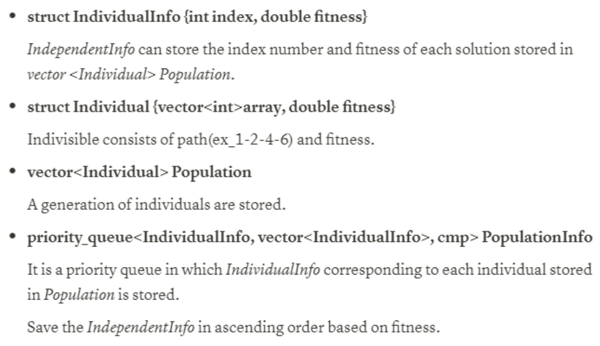

start
read CSV file
Construct Adjacency Matrix with csv data
Create Initial population
[Iteration]
Select parent(Tournament)
Save elite to 'new generation'
Reproduction (crossover + mutation)
and save the generatedchild to 'new generation'
population size < N
No
Yes
end

Fig 4. Flow chart of the Program

2.3 Data structure description



- **struct IndividualInfo {int index, double fitness}**

  IndependentInfo can store the index number and fitness of each solution stored in *vector <Individual> Population*.
- **struct Individual {vector<int>array, double fitness}**

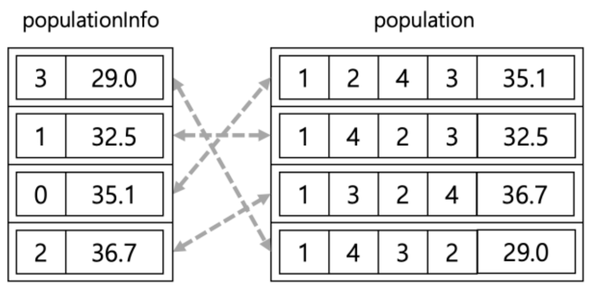
  Indivisible consists of path(ex_1-2-4-6) and fitness.
- **vector<Individual> Population**

  A generation of individuals are stored.
- **priority_queue<IndividualInfo, vector<IndividualInfo>, cmp> PopulationInfo**

  It is a priority queue in which *IndividualInfo* corresponding to each individual stored in *Population* is stored.

  Save the *IndependentInfo* in ascending order based on fitness.

Fig 5. Description of data structure



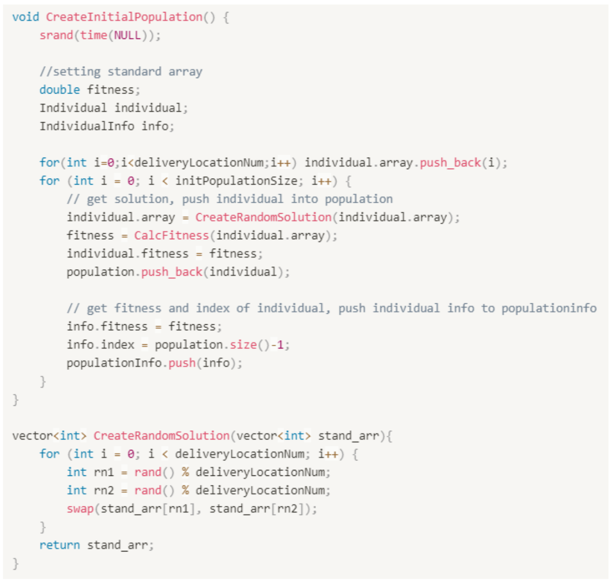
3	29.0
1	32.5
0	35.1
2	36.7

1	2	4	3	35.1
1	4	2	3	32.5
1	3	2	4	36.7
1	4	3	2	29.0

Fig 6. Example of using **population** and **populationInfo**

2.4 Initialization

In the initialization step, the size of the population is determined, and random population is generated. The size of the population tends to depend on the size of the city provided in the problem. Therefore, the larger the size of the city, the more cases exist, so the larger the size of the population should be set to provide sufficient diversity of genetic information so that sufficient search can be performed.



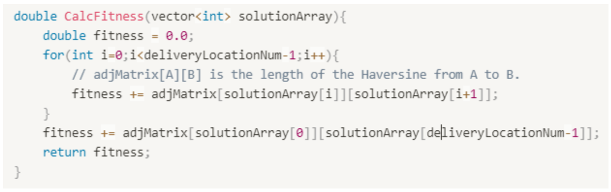
```
void CreateInitialPopulation() {
    srand(time(NULL));

    //setting standard array
    double fitness;
    Individual individual;
    IndividualInfo info;

    for(int i=0;i<deliveryLocationNum;i++) individual.array.push_back(i);
    for (int i = 0; i < initPopulationSize; i++) {
        // get solution, push individual into population
        individual.array = CreateRandomSolution(individual.array);
        fitness = CalcFitness(individual.array);
        individual.fitness = fitness;
        population.push_back(individual);

        // get fitness and index of individual, push individual info to populationinfo
        info.fitness = fitness;
        info.index = population.size()-1;
        populationInfo.push(info);
    }
}

vector<int> CreateRandomSolution(vector<int> stand_arr){
    for (int i = 0; i < deliveryLocationNum; i++) {
        int rn1 = rand() % deliveryLocationNum;
        int rn2 = rand() % deliveryLocationNum;
        swap(stand_arr[rn1], stand_arr[rn2]);
    }
    return stand_arr;
}
```

Fig 7. Pseudo code of Initialization

First, we create an array of numbers from 0 to (**deliveryLocationNum**-1) representing the total number of destinations. Initial random population was constructed by mixing the positions of the internal values of the array by the **deliveryLocationNum** value representing the number of delivery locations through the **CreateRandomSolution** function.

2.5 Fitness evaluation

In TSP, the problem of this project, the shortest distance is to return to the first point via destinations. Therefore, the criterion for determining the optimal solution is the distance of each cycle path. The distance is obtained using the Haversine Formula.



```
double CalcFitness(vector<int> solutionArray){
    double fitness = 0.0;
    for(int i=0;i<deliveryLocationNum-1;i++){
        // adjMatrix[A][B] is the length of the Haversine from A to B.
        fitness += adjMatrix[solutionArray[i]][solutionArray[i+1]];
    }
    fitness += adjMatrix[solutionArray[0]][solutionArray[deliveryLocationNum-1]];
    return fitness;
}
```

Fig 8. Pseudo code of calculating fitness value of path

The **solutionArray** containing the cycle paths visiting all delivery points is input and the fitness value of the cycle paths is combined and output from the adjacent matrix containing the fitness value of each path. In the TSP problem, the shortest path means the optimal path, so a low fitness value becomes the better fitness value.

$$\Theta = \frac{d}{r}$$

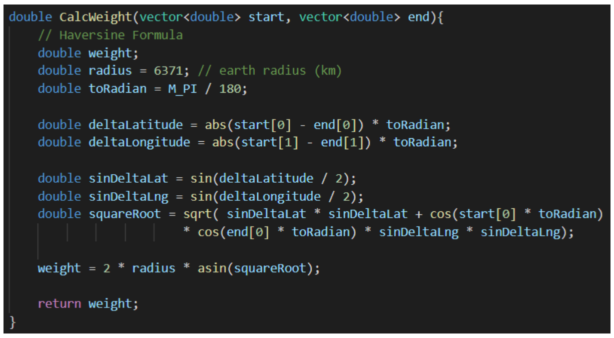
$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

$$\operatorname{hav}(\Theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

$$d = r\operatorname{archav}(h) = 2r\arcsin\left(\sqrt{h}\right)$$

$$\begin{aligned} d &= 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2-\varphi_1)+\cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2-\lambda_1)}\right) \\ &= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right)+\cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right) \end{aligned}$$

Fig 9. Haversine formula



```
double CalcWeight(vector<double> start, vector<double> end){
    // Haversine Formula
    double weight;
    double radius = 6371; // earth radius (km)
    double toRadian = M_PI / 180;

    double deltaLatitude = abs(start[0] - end[0]) * toRadian;
    double deltaLongitude = abs(start[1] - end[1]) * toRadian;

    double sinDeltaLat = sin(deltaLatitude / 2);
    double sinDeltaLng = sin(deltaLongitude / 2);
    double squareRoot = sqrt( sinDeltaLat * sinDeltaLat + cos(start[0] * toRadian)
                        * cos(end[0] * toRadian) * sinDeltaLng * sinDeltaLng);

    weight = 2 * radius * asin(squareRoot);

    return weight;
}
```

Fig 10. Definition of function to apply Haversine formula

θ is the central angle of the arc connecting two points, γ is the radius of the earth (6371km), d is the distance between the two points, φ is latitude (rad), and λ is longitude (rad). The code below implements the Haversine Formula.

2.6 Selection

2.6.1 Tournament Selection

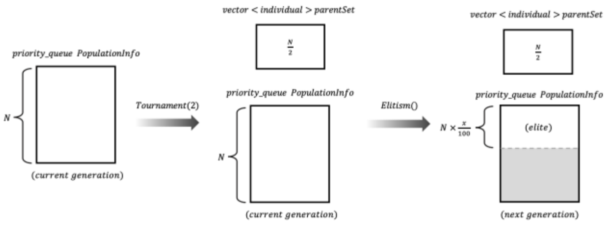

priority_queue PopulationInfo
N
(current generation)
Tournament(2)
vector < individual > parentSet
N/2
priority_queue PopulationInfo
N
(current generation)
Elitism()
vector < individual > parentSet
N/2
priority_queue PopulationInfo
N × x/100
(elite)
(next generation)

Fig 11. Flow of Selection function

Tournament selection selects a parent solution to generate the next generation and store it in the **vector<individual> parentSet**. The tournament randomly selects two individuals from the current generation and stores the individual with greater fitness in the **parentSet**. Individuals which participate in the tournament once cannot participate in the tournament again.

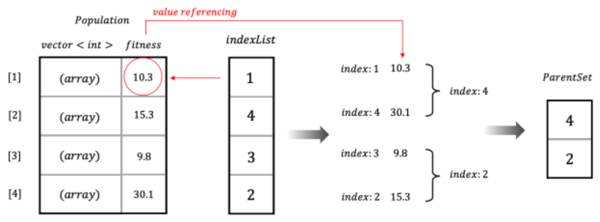


Fig 12. Process of Tournament selection

The implementation is as follows. First, indexes of individuals of current generation are randomly stored in **indexList**. Then, select two from the front of the **indexList** and proceed to the tournament. Repeat this process until all individuals participate in the tournament.

2.6.2 Rank-based Roulette wheel

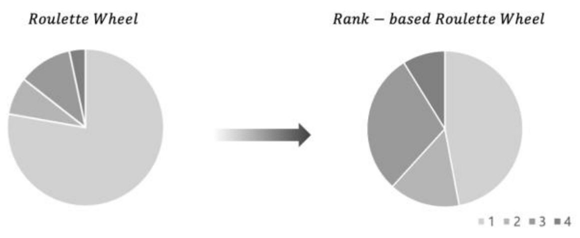
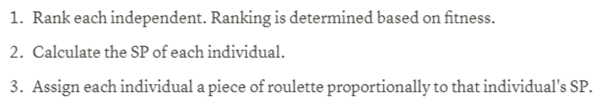


Fig 13. Proportional roulette wheel and Rank-based roulette wheel

There are various strategies in selection. We tried two strategies, tournament, and rank-based roulette wheel, and compared the two strategies. As a result, we decided to choose the tournament.

The rank-based roulette wheel is a method of assigning a piece of roulette to each individual and selecting the individual of the interval in which the corresponding number is included by drawing a random number. The roulette pieces size are proportional to the selection probabilities (SP)of individual.



1. Rank each independent. Ranking is determined based on fitness.
2. Calculate the SP of each individual.
3. Assign each individual a piece of roulette proportionally to that individual's SP.

Fig 14. The process of generating the roulette wheel

When the constructing the roulette wheel is completed, now spin the roulette as much as the number of parents wanted. Proportional roulette wheel selection allocates pieces of roulette in proportion to fitness. Therefore, if there is a very dominant indivisible, diversity can be reduced. However, since the rank-based roulette wheel allocates roulette pieces proportionally to the SP rather than fitness, certain independents can't extremely take up roulette pieces.

Thus, rank-based roulette wheel can select the individual with good genetics as a high probability while preserving diversity rather then proportional roulette wheel. Therefore, the quality of the results is improved. However, there are several disadvantages in rank-based roulette wheel.

Rank-based roulette wheel requires calculation of SP of all individuals of the population, and the process of selecting one individual by spinning the roulette also requires calculation. In addition, the roulette wheel can be select the same individual several times. If the same individual is included multiple times in the parent set, diversity decreases, and the convergence value is likely to be trapped locally optimum.

To prevent this, the selected individual should not be selected again and to confirm whether the individual is selected or not requires additional operations. As a result, rank-based roulette wheel takes much longer time than the tournament.

Table1. The result of each selection method about Atlanta

Selection Method	Optimum value(km)
Tournament	28.6228
Rank-base roulette wheel	30.3759

On the other hand, tournament is simple to implement. there is no need to sort individual, calculating SP, and the operation of assigning the roulette piece to the individual is also not required. It is very efficient in terms of time complexity. Also, it has low susceptibility to takeover by dominant individuals.

In this study, we confirmed that GA convergence value is successfully similar to global optimum even if the tournament is used. Therefore, we selected tournament which has more efficient time complexity then rank-based roulette wheel as selection strategy.

2.7 Elitism

Appropriate elitism applications help GA converge to global optimum more quickly. In this program, before creating children through parent objects and changing generations, the most suitable individual (i.e., elite) of x% in the current generation is passed to the next generation without any crossover. All current generation is deleted except for the elites of x%.

```
void Elitism(){
    int eliteNum = population.size()*elitismPercent/100;
    vector<Individual> eliteSet;
    Individual elite;
    priority_queue<IndividualInfo, vector<IndividualInfo>, cmp> eliteInfo;
    int eliteIndex;
    IndividualInfo info;

    for(int i=0;i<eliteNum;i++){
        // save elite solution to elite vector
        eliteIndex = populationInfo.top().index;
        // save elite soluation
        elite.array = population[eliteIndex].array;
        elite.fitness = populationInfo.top().fitness;
        eliteSet.push_back(elite);
        // save elite soluation info
        info.index = eliteSet.size()-1;
        info.fitness = elite.fitness;
        eliteInfo.push(info);
        // delete elite solution from current population
        populationInfo.pop();
    }
    population.clear();
    // change current generation population and elite population
    eliteSet.swap(population); // swap delete previous vector.
    swap(populationInfo, eliteInfo);
}
```

Fig 14. Pseudo code of Elitism

2.8 Reproduction

The production step is to create new child objects by applying the production method to the parent object selected in the selection step. Representative methods of reproduction include Crossover, Mutation, and Elitism.

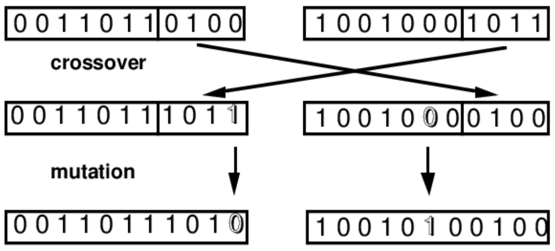


Fig 15. Crossover and mutation in GA (Reference: Padró, Francesc & Ozón, Javier & Aplicada, Departament. (1995). Graph Coloring Algorithms for Assignment Problems in Radio Networks.)

Crossover is a method of creating a new offspring by crossing randomly selected crossover sections from the genetic information of two randomly selected parents.

Mutation is a method of randomly twisting genetic information within an individual to create new descendants.

Elitism is a method of transferring dominant individuals with a high fitness value to the next generation, thereby increasing the chances of survival of dominant individuals so that they can approach the global optimum.

Through these reproduction operations, the diversity of genetic information can be increased, and the increase in diversity prevents being trapped in a local minimum and increases the possibility of convergence to a global optimum.

III. Reproduction strategy

3.1 Partially mapped crossover (PMX)

Although there are various methods of crossover, this program uses partially mapped crossover (PMX), one of several types of crossover. The process of PMX is as follows.

$$A_{parent} = (3\ 4\ 8\ |\ 2\ 7\ 1\ |6\ 5), \qquad A_{swapRange} = (2\ 7\ 1)$$

$$\Downarrow\Uparrow$$

$$B_{parent} = (4\ 2\ 5\ |\ 1\ 6\ 8\ |3\ 7), \qquad B_{swapRange} = (1\ 6\ 8)$$

Fig 16. Step1 of the PMX

$$\begin{aligned} A_{child} &= (3\ 4\ \textcircled{8}\,|\ 1\ 6\ 8\ |\textcircled{6}\,5) \\ B_{child} &= (4\,\textcircled{2}\,5|\ 2\ 7\ 1\ |3\,\textcircled{7}) \end{aligned} \Longrightarrow \begin{aligned} A_{child} &= (3\ 4\ *\ |\ 1\ 6\ 8\ |*\ 5) \\ B_{child} &= (4\ *\ 5\ |\ 2\ 7\ 1\ |3\ *) \end{aligned}$$

Fig 17. Step2 of the PMX

$$A_{child} = (3\ 4\ *\ |\ 1\ 6\ 8\ |*\ 5), \quad A_{needChange} = (8\ 6)$$

$$B_{child} = (4\ *\ 5\ |\ 2\ 7\ 1\ |3\ *), \quad B_{needChange} = (2\ 7)$$

$$\begin{aligned} A_{child} &= (3\ 4\ 2\ |\ 1\ 6\ 8\ |7\ 5) \\ B_{child} &= (4\ 8\ 5\ |\ 2\ 7\ 1\ |3\ 6) \end{aligned}$$

Fig 18. Step3 of the PMX

First, genetic information of a randomly set section [a, b] in two parent objects is exchanged with each other, as shown in Fig #. Second, mark the elements in the interval [a, b] after the exchange, as shown in Fig #, if they already exist in duplicate in their own elements. Finally, in the exchanged state as shown in Fig #, only the elements marked by each mark are kept in order and exchanged again.

```
void Crossover(vector<int> a, vector<int> b) {
    int mutationRandomnum;
    int start = random_value; // 1 ~ deliveryLocationNum
    int end = random_value;  // end > start

    //swaping the data in the range (start, end)
    for (int i = start; i <= end; i++) {
        a_swapRange.push(a_child[i]);
        b_swapRange.push(b_child[i]);
        swap(a_child[i], b_child[i]);
    }

    //check if array has overlapping elememts
    for(int i=0 ; i<=end-start ; i++){ // repeate rangeNum
        int a_swapElement = a_swapRange.front();
        int b_swapElement = b_swapRange.front();
        b_swapRange.pop();
        a_swapRange.pop();

        for (int j = 0; j < deliveryLocationNum; j++) {
            if (j >= start && j <= end) continue;
            if (a_swapElement == b_child[j]) {
                b_needChange.push(j);
            }
            if (b_swapElement == a_child[j]) {
                a_needChange.push(j);
            }
        }
    }

    //change overleapping elements
    int changeCnt = b_needChange.size();
    for(int i=0;i<changeCnt;i++){
        swap(a_child[a_needChange.top()], b_child[b_needChange.top()]);
        a_needChange.pop();
        b_needChange.pop();
    }
}
```

Fig 19. Pseudo code of PMX

3.2 Reverse Sequence Mutation (RSM)

Although there are various methods of mutation, this program uses Reverse Sequence Mutation (RSM), one of several types of crossover. RSM is a method of creating new descendants by exchanging the positions of elements in a randomly set interval [start, end] within a single parent entity. As the interval between the randomly set sections gradually narrows, genetic information located at start and end is exchanged.

$$A_{parent} = (1\ 2\ 3\ |\ 4\ 5\ 6\ |7\ 8)$$

$$If\ start = 2, end = 5$$

$$A_{parent} = (1\ 2\ \underset{start}{3}\ |\ 4\ 5\ \underset{end}{6}\ |7\ 8)$$

$$\Downarrow$$

$$A_{child} = (1\ 2\ 6\ |\ 5\ 4\ 3\ |7\ 8)$$

Fig 20. Steps of the RSM

```
vector<int> Mutation(vector<int> a){
    vector<int> a_mutation = a;

    //Reverse Sequence Mutation(RSM)
    int start = random_value; // 1 ~ deliveryLocationNum
    int end = random_value;  // end > start

    //Swapping genetic information located at start and end
    while (start < end) {
        swap(a_mutation[start], a_mutation[end]);
        start++;
        end--;
    }
    return a_mutation;
}
```

Fig 21. Pseudo code of RSM

3.3 Selection of reproduction policy

A key point to consider in the policy selection of reproduction is the rate of convergence with increasing initiation and the preservation of diversity in the next generation population for various exploration.

3.3.1 First Method

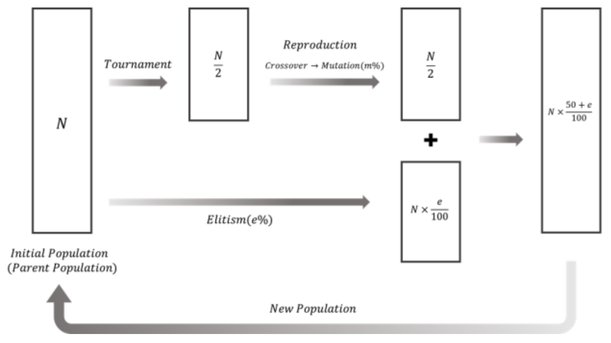


Fig 22. Flowchart of 1st method

The first attempt to policy selection consists of two computational paths for the population. The first operation is a method of applying elitism to the current population to the next generation. The second operation is a method of applying reproduction to elements selected as tournament in the current population and transferring them to the next generation. The computational results of the two methods are combined again to become the next generation of populations.

Reproduction operations applied to elements selected through the tournament include crossover and mutation. In the following method, the crossover operation was applied first, and the mutation operation was applied to the result with a certain probability.

For crossover, two offspring are created for two randomly selected parents in the current population. The mutation operation is applied to each of the two offspring at a certain probability, and it is passed on to the next generation.

The reason for the design as follows is that crossover tends to converge to the optimal solution that appears in the current population as iteration increases. At this time, if mutation is applied to an individual to which probabilistic crossover is applied, offspring with random genetic information can be obtained, so diversity of genetic information can be secured.

3.3.2 Second Method

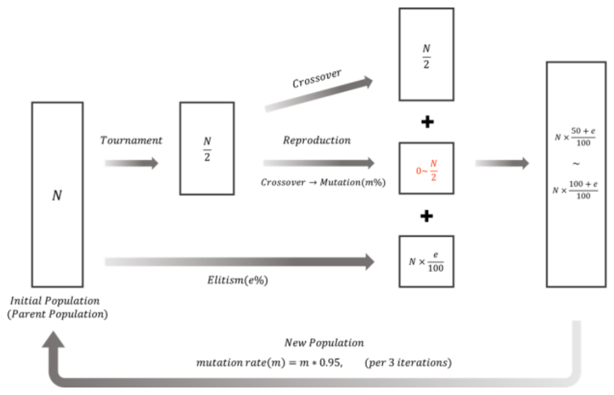


Fig 23. Flowchart of 2nd method

The second attempt to make policy selection is a method to improve the problems of the first method. For the first method, mutation was applied to the crossover result, and the probability of mutation was constant for each iteration. In this case, the occurrence of the mutation operation reduces the proportion of the offspring population to which only the crossover operation is applied in the next generation, which reduces the rate of convergence to the optimal solution as the iteration increases.

In addition, fixing the probability of mutation occurrence in all iterations is inefficient because it continues to provide diversity even when it is considered unnecessary to add diversity in genetic information to the optimal solution. As iteration increases, most of the population converges to the genetic information of the optimal solution. Therefore, as iteration increases, the probability of occurrence of mutation needs to decrease compared to the initial probability. In other words, in the modified method, the offspring generated as a result of the crossover operation are passed to the next generation first, and the offspring generated by the crossover are further passed to the next generation by applying probabilistic mutation to the offspring generated by the crossover.

Through this method, the rate of convergence for the optimal solution was increased by increasing the proportion of the offspring population to which only the crossover operation was applied in the previous method. In addition, the probability of occurrence of mutation was set to gradually decrease with the increase of iteration, so that the amount of newly added genetic information was reduced as iteration increased, which could converge faster to the optimum value.

3.3.3 Comparison of policy performance

To compare the performance of the two methods, the parameter values are set equally to **initPopulationSize**=10000, **crossover**=45, and **mutation**=40 determined as the optimal parameters in the subsequent process, and the results of each method are compared. The performance comparison is performed 20 times for each method to compare the average of the converged Iteration, the average of the converged values, and the optimal value found. For the comparison of performance, each method is performed 20 times, and the average of the Iterations whose values converge, the average of the converged values, and the optimal value found are compared with each other.

Table2. Results for each method

Method	The number of iteration (mean)	Value of convergence (mean)	Optimum value
1st Method	101.10	397.90	367.187
2nd Method	247.1	196.613	170.132

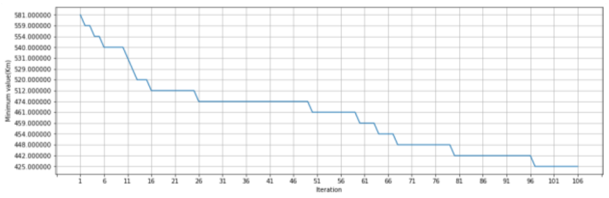


Fig 24. Graph of 1st method

In the case of the first method, the rate at which the values converge is faster than the second method, but the average of the convergence values and the fitness value of the optimization are higher than that of the second method. The solution found by the first method differs greatly from the optimal solution found by the second method. The rapid convergence of the results of the first method to the local minimum can be interpreted as not sufficiently searching for new genetic information and rapidly reducing the diversity of genetic information.

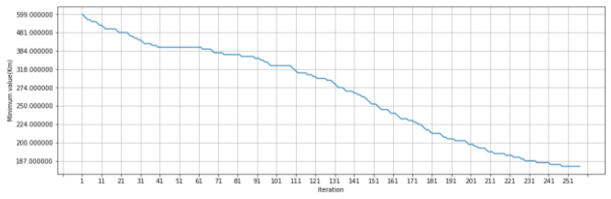


Fig 25. Graph of 2nd method

The 2nd Method prevents the diversity of genetic information from rapidly decreasing and sufficiently increases the search for new genetic information, which slows the convergence rate compared to the first method, but because the fitness value of the optimal solution is better, the second method is selected as a reproduction policy.

3.3.4 Optimal parameter value

Table 3. Results for parameter settings

Elitism	Mutation	The number of iteration (Mean)	Value of convergence (Mean)	Optimum value	Standard deviation
30	30	45	449.322	404.444	20.706
	35	51.35	435.917	386.551	22.586
	40	60.45	412.518	378.554	17.612
	45	62.85	417.133	362.035	18.422
35	30	64.95	418.286	384.248	22.255
	35	74.2	396.049	363.402	16.568
	40	85.25	383.392	353.899	16.768
	45	88.95	370.336	327.856	25.201
40	30	104	355.183	316.241	27.162
	35	112.45	339.578	278.009	29.942
	40	129.55	313.274	286.262	20.956
	45	136.05	302.387	247.451	22.846
45	30	199.45	236.286	200.845	21.329
	35	217.4	214.009	179.198	23.309
	40	**247.1**	**196.613**	**170.132**	**15.657**
	45	252.4	192.236	176.928	10.436

To find the optimal parameter value, fix the size of the initial population to 10000 (**initPopulationSize**=10000), change the value of the parameter value of Elitism and mutation, and compare the output results for each case. To better analyze the effect of changing parameter values on the results, we analyze them based on the results of a ‘**New_York.csv**’ file with a large data size. For each case, the results executed 20 times are recorded and analyzed based on the average value of the convergence, the average value of the convergence result, the optimal value during execution, and the standard deviation of the result value.

First, if only the value of elitism was increased under the same conditions, the number of populations going to the next generation increased, so the iteration required for convergence increased proportionally. As the increase in the ratio of elitism can preserve better genetic information, it can be confirmed that the difference between the average of the optimal value and the converging value decreases. In addition, as more dominant genetic information is preserved, the characteristics of good genetic information are quickly searched, and as a result, the average of the optimal value and the converging value is lowered.

Similarly, under the same conditions, the increase in mutation enables the search of various genetic information and increases the probability that the optimal value reaches the global optimum. Therefore, the difference and standard deviation between the optimal value and the average of the converged value decrease, and the optimal solution closer to the global optimum can be reached.

However, if the probability of mutation occurrence becomes too large, the diversity within the population becomes too large, which adversely affects convergence to one characteristic, resulting in an increase in the standard deviation value. Through the above analysis, the value of the parameter value was finally selected with **Elitism** = 45 and **Mutation** = 40.

IV. Program execution results

4.1 Cincinnati

We first selected the Cincinnati problem, which has the smallest data size, to test whether each step works well and derives the appropriate output in the process of implementing GA.

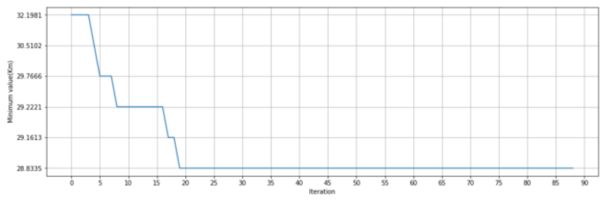
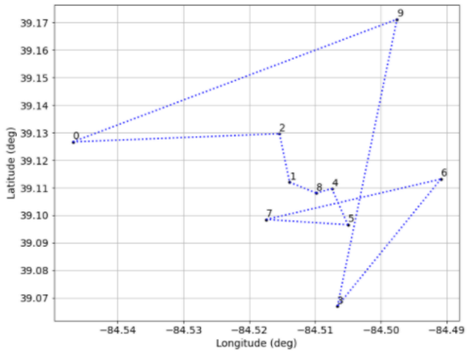


Fig 26. Graph of the convergence history

Table4. Result of the Cincinnati

Initial Population size	Iteration	Convergence value
20	113	28.6228
30	139	28.6228
40	152	28.6228
50	163	28.6228

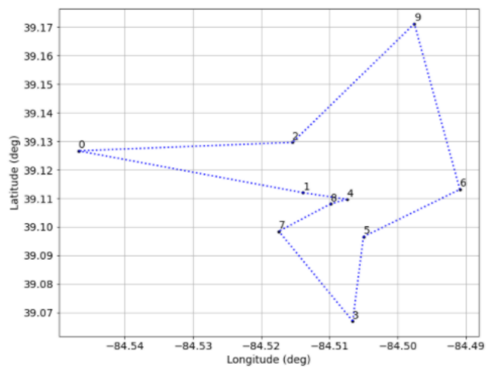
Cincinnati has only 10 delivery points. Therefore, when the percentage of elitism and mutation were fixed in 45,40 that we reason from parameter experiment, the convergence value was sufficiently derived even if the initial population size was 20.



Path: 0 - 2 -1 - 8 - 4 - 5 - 7 - 6 - 3- 9 - 0

Distance: 34.353km

Fig 27. Optimal path of Cincinnati found using Greedy algorithm



path: 4 - 8 - 7 - 3 - 5 - 6 - 9 - 2 - 0 - 1 - 4

Distance: 28.6228km

Fig 28. Optimal path of Cincinnati found using Genetic algorithm

4.2 Atlanta

Cincinnati had a small data size, so it was easy to derive convergence values. However, Atlanta has 20 delivery points so to find optimal percentage of elitism, mutation, and initial population size, it needed testing using various test cases. The percentage of elitism and mutation were tested based on 45 and 40, which were drawn by the parameter experiment.

Table5. Result of the elitism test

Initial Population size	elitism(%)	mutation(%)	Iteration	Convergence value
1000	30	40	48	246.15
1000	35	40	73	210.811
1000	40	40	111	210.811
1000	45	40	207	204.972
1000	50	40	-	-

First, to find the appropriate percentage of elitism, we fixed the initial size of the population and percentage of mutation, and beginning from 30% of elitism, the increasing values were tested. The results of the test show that the output value(TABLE2) is trapped in the local minimum and the output value is not sufficiently converged in less than 45% of the elitism. In more than 50% of the population size of the next generation often become larger than the previous generation population size, and the algorithm can be trapped into an infinite loop. Thus, 45% is appropriate.

Table6. Result of the mutation test

Initial Population size	elitism(%)	mutation(%)	Iteration	Convergence value
1000	45	20	122	208.054
1000	45	25	138	206.522
1000	45	30	160	204.972
1000	45	35	181	206.075
1000	45	40	211	205.887
1000	45	45	227	204.972
1000	45	50	250	204.972

Next, to find the appropriate percent of mutation, we fixed the initial size with arbitrary value and elitism with 45%, beginning from 20% of mutation, the increasing values were tested. The results of the test show that the output value (Table3.) is trapped in the local minimum and the output value is not sufficiently converged in less than 45% of the elitism. If it exceeds 45%, it is not efficient because it increases only useless iteration. Therefore, the 45% is appropriate.

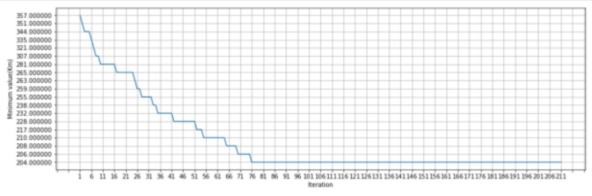


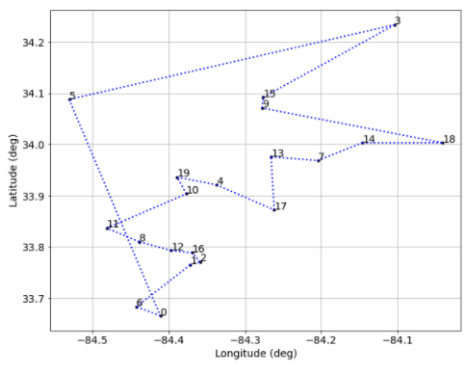
Fig 29. Graph of the convergence history

Table7. Result of the initial population size test

Initial Population size	elitism(%)	mutation(%)	Iteration	Convergence value
100	45	45	181	205.839
200	45	45	193	205.887
300	45	45	203	205.887
400	45	45	209	204.972
500	45	45	213	204.972
600	45	45	216	204.972
700	45	45	220	204.972

Last, to find the appropriate initial population size, we fixed the elitiem and mutation with 45% both, beginning from size 100, the increasing values were tested.

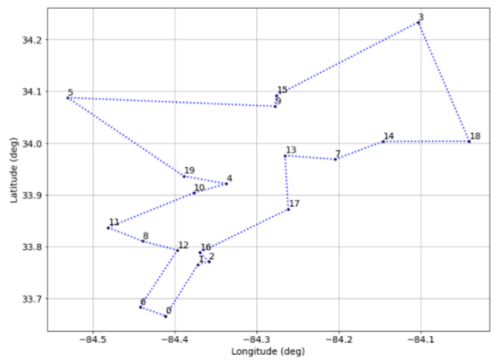
As a result of the test, the output value converges to the global minimum from size 400 to 500. Less than size 400 is not enough for iteration to find a convergence value, and the output value is tends to stuck in the local minimum. Therefore, 500 were chosen as the appropriate size.



Path: 0 - 6 -1 - 2 - 16 - 12 - 8 - 11 - 10 - 19 - 4 - 17 - 13 - 7 - 14 - 18 - 9 - 15 - 3 - 5 - 0

Distance: 231.98km

Fig 30. Optimal path of Atlanta found using Greedy algorithm



Path: 17 - 31 - 7 - 14 - 18 - 3 - 15 - 9 - 5 - 19 - 4 - 10 - 11 - 8 - 12 - 6 - 0 - 1 - 2 - 16 - 17

Distance: 204.972km

Fig 31. Optimal path of Atlanta found using Genetic algorithm

4.3 New York

Cincinnati and Atlanta had small data sizes, so we could specify a case where value converged successfully. However, New York has 68 delivery points, so it was not possible to confirm the perfect convergence point by checking all cases like previous cases. If the initial size exceeds 10,000 for the percentage of elitism and mutations of 45,45 or 45,50, it takes a very long time to run the program once. Therefore, we designed the experiment to draw the conclusion by selecting the case with the smallest standard deviation of output values and the smallest output by testing with the testable cases 10 times each. To find the appropriate initial population size, we fixed the percent of elitism and mutation with 45 and 40, beginning from size 10,000, the increasing values were tested.

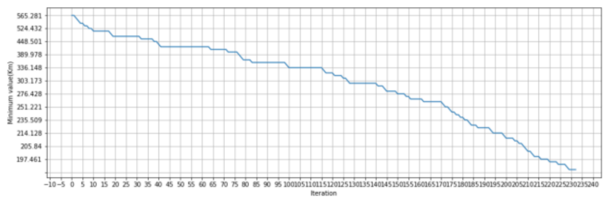


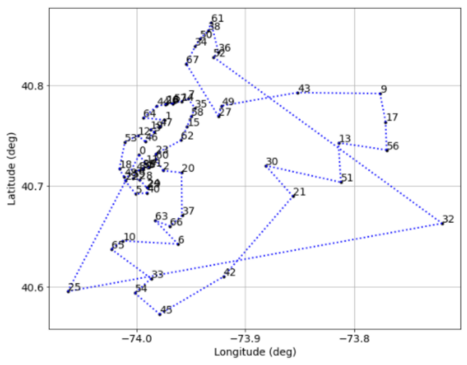
Fig 32. Graph of the convergence history

Table8. Result of the initial population size test

Initial Population size	elitism(%)	mutation(%)	Standard Deviation	Iteration	Minimum value
10,000	45	40	15.8176899	256	162.306
20,000	45	40	13.7332397	277	173.153
30,000	45	40	13.2587197	277	170.014

The results of the test showed that the standard deviation tended to decrease as the initial size increased, but he case that drew the minimum value was size 10,000.

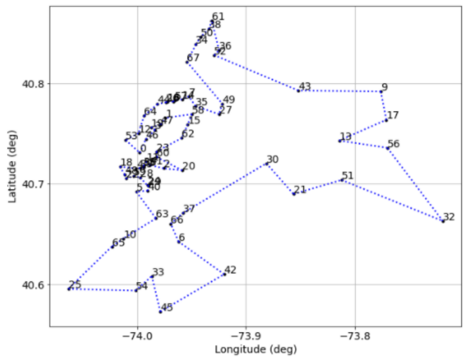
We tried to execute the testable case as much as possible, but we could not find a case where the output value converges successfully. Since the data size is large, it is estimated that we should run a case with a much larger initial population size than the testable case. However, if the initial population size exceeds 30,000 when the percent of elitism and mutation is 45 or more, it takes a very long time to execute, so it is impossible to test the case at present. Therefore, we decided to run the case with input size 10000 several times that drew good output despite the short execution time to find the path which has minimum distance possibly comes out.



Path: 0 - 11 - 41 - 31 - 55 - 39 - 4 - 59 - 28 - 24 - 29 - 40 - 5 - 22 - 48 - 18 - 53 - 12 - 46 - 3 - 19 - 47 - 1 - 64 - 44 - 26 - 16 - 8 - 57 - 14 - 7 - 35 - 58 - 15 - 62 - 23 - 60 - 2 - 20 - 37 - 66 - 63 - 6 - 10 - 65 - 33 - 54 - 45 - 42 - 21 - 30 - 51 - 13 - 56 - 17 - 9 - 43 - 49 - 27 - 67 - 34 - 50 - 38 - 61 - 36 - 52 - 32 - 25 - 0

Distance: 202.551km

Fig 33. Optimal path of New York found using Greedy algorithm



Path: 0- 46- 3- 19- 47- 1- 58- 15- 62- 23- 60- 20- 2- 41- 11- 31- 55- 39- 4- 18- 48- 22- 59- 28- 24- 29- 40- 5- 63- 10- 65- 25- 54- 33- 45- 42- 6- 66- 37- 30- 21- 51- 32- 56- 13- 17- 9- 43- 52- 36- 61- 38- 50- 34- 67- 49- 27- 35- 7- 14- 57- 8- 16- 26- 44- 64- 12- 53- 0

Distance: 154.283km

Fig 34. Optimal path of New York found using Genetic algorithm

V. Discussion

5.1 Why are the results of the Genetic and Greedy algorithms different?

In the case of the Greedy algorithm, when searching for a single path, it does not consider the destination to be chosen thereafter, and searches for the path in priority for the closest distance from the current point. Therefore, the speed of determining the path is very fast, and many cases are not considered in the search of the path.

On the other hand, the GA generates a random population and evaluates the fitness value to continuously reflect the characteristics of good paths in the population. It continues to explore better paths as the search progresses and selects the path as the optimal path when the optimal path of the population converges. Because we consider the number of cases, we can find a path that is slow but close to the optimal path.

Thus, the path found by the GA with a more complex process is more likely to be closer to the global optimum than the path found by the Greedy algorithm.

5.2 Why is the solution found by the Genetic algorithm global optimum?

Table9. Results of each algorithm for Atlanta

Algorithm	Optimum value(km)
Ours	204.972
Reference	207.394

Comparing the distance of Atlanta's optimal path found using the GA and the distance provided by Reference, the two values are almost similar. This can be a basis for the fact that the optimal path of Atlanta found using GA is the global optimum value. Therefore, the optimal distance for Atlanta found in the project is the global optimum.

However, this cannot guarantee that GA will always reach the global optimum in all cases. In the case of Cincinnati and Atlanta, the input size of data is small, so the number of paths to consider is small. Therefore, if the appropriate parameter value is set, the global optimum can always be reached using the GA. However, as the input size of data increases, such as New York, the number of paths to be explored increases significantly, and although the results of GA may appear close to the global optimum, the global optimum will not always be found.

5.3 Can the Genetic algorithm always get the same solution?

The answers to the following questions may vary depending on the conditions in question. For example, cities suggested in the TSP problem, such as Cincinnati, where the number of destinations is small, the same solution can always be obtained by increasing the size of the initial population and securing sufficient diversity of genetic information.

However, in cities with many destinations, such as New York, the number of routes increases rapidly, increasing the probability that the route of the global optimum will not be searched through GA. Therefore, the probability that the optimal optimum found through the GA will be decreases and it becomes difficult to obtain the same solution all the time.

5.4 How can the problem of the algorithms that are always trapped in the local optimum be solved?

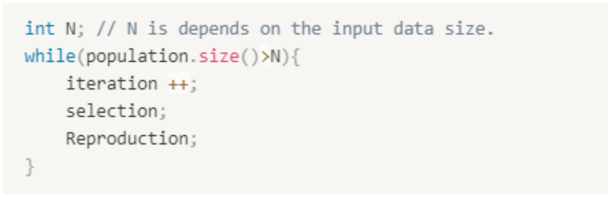
First, the fact that the solution is trapped in the local optimum means that the genetic information of the population lacks diversity, making it less likely to find a new optimal solution. Therefore, it is necessary to further add diversity of genetic information so that search for various solutions is made.

Factors that determine the diversity of genetic information include the size of the initial population and the reproduction operation. Since the initial population is a set of solutions composed of random genetic information, the larger the size of the population, the more diversity of genetic information exists.

The diversity of genetic information can also be increased through production operations such as crossover and mutation. crossover generates new genetic information by exchanging genetic information between two individuals in the population. In the case of crossover, new offspring are generated through the exchange of genetic information within the population, so the effect decreases as the genetic information of the population converges.

However, in the case of mutation, genetic information is twisted within one object to generate new genetic information. Therefore, there is a possibility that an unprecedented sequence of genetic information will be generated, and new genetic information will allow you to escape being trapped in the local optimum and increase the possibility of finding a global optimum.

5.5 How should the stopping criterion that terminates Genetic algorithms be defined?



```
int N; // N is depends on the input data size.
while(population.size()>N){
    iteration ++;
    selection;
    Reproduction;
}
```

Fig 35. Results for parameter settings

If the population size becomes smaller than N, the stopping criterion determines the best indivisible value by stopping the algorithm. In this case, N is set in consideration of the input data size. The reason for the existence of such a stopping criterion is to reduce the speed of the program execution and the overall computational cost by omitting the subsequent unnecessary annotation if the optimal values of the population have already converged.

VI. Conclusion

This project proposes how to find the optimal path in TSP, one of the NP class problems, through the GA. In each case, a single cycle path through all delivery points is calculated through the Haversine formula and expressed as a fitness value. The fitness value in each case is used as an indicator to determine that it is the optimal path, and it becomes a criterion for classifying it as good genetic information in the selection process of the GA. Compared with the Greedy algorithm, it can be confirmed that the path found by the Greedy algorithm is more likely to be close to the global optimum. Although the GA does not always look for global optimum, appropriately changing the parameter values within the GA considering the number of cases provided in the problem helps to bring the program's result closer to the global solution. Through these processes, as a result, the user could access the path assumed to be the optimal path without searching for all cases through the implemented program.

VI. Reference

[1] "최단거리 구하기, 하버사인 공식(Haversine Formula)", github. Io, last modified Sep 19, 2019, accessed Aprill 2, 2022, https://kayuse88.github.io/haversine/

[2] Razali, N. M., & Geraghty, J. (2011, July). Genetic algorithm performance with different selection strategies in solving TSP. In Proceedings of the world congress on engineering (Vol. 2, No. 1, pp. 1-6). Hong Kong, China: International Association of Engineers. https://setu677.medium.com/how-to-perform-roulette-wheel-and-rank-based-selection-in-a-genetic-algorithm-d0829a37a189

[3] Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, Showkat Gani, "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator", Computational Intelligence and Neuroscience, vol. 2017, Article ID 7430125, 7 pages, 2017. https://doi.org/10.1155/2017/7430125

[4] Otman, Abdoun & Abouchabaka, Jaafar & Tajani, Chakir. (2012). Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem. Int. J. Emerg. Sci.. 2. https://arxiv.org/ftp/arxiv/papers/1203/1203.3099.pdf