CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

```
1. m = folium.Map(location=[-6.9842650934006, 110.3832977495],
2.                zoom_start=15,
3.                min_zoom=14,
4.                max_zoom=18)
5.
6. for index, location in data.iterrows():
7.   loc=[location["x"], location["y"]]
8.   no=([location["no"]],[location["x"], location["y"]])
9.   rumah=folium.Marker(
10.        location=loc,
11.        popup=no
12.   )
13.   rumah.add_to(m)
14. m
```

Line 1-4 has the function of declaring m which is the variable that calls folium, which is a library that will help to build a map. Location in line 1 has the function to pinpoint the starting preview when the output of folium is displayed; zoom_starts is the default initial zoom level of the map, while min_zoom and max_zoom is the minimum and maximum value on how the map is allowed to zoom. Line 6 is the start of the loop, loop here has the function to display each node, and will stop when the maximum number of node has been reached. Line 7 is declaring loc equals to the coordinates x and y. Line 8, declare no, which contains the number of nodes, and coordinates x and y. rumah in line 9 has the function to display the marker using folium marker, location is the node or equal to loc and popup is the information that will pop out when the marker is clicked, popup contains no that has been declared before. After that rumah will be added into m in line 13, and line 14 is to display the m.

```
15. hitung=0
16. for index, location in data.iterrows():
17.   if hitung!=40:
18.     loc = [(data["x"][index], data["y"][index]),
19.            (data["x"][index+1], data["y"][index+1])]
20.     folium.PolyLine(loc,
21.                     color='red',
22.                     weight=3,
23.                     opacity=1).add_to(m)
24.   else:
25.     break
26.   hitung=hitung+1
27. m
```

Line 15 is to declare a new variable called hitung. Line 16 is to loop the data. Line 17 is the function to execute the true or false part of a condition, here the condition is hitung, if hitung is not 40 then the condition is considered true, line 18-19 is defining loc, here loc contains [x1, y1, x2, y2] this is used to draw the path between nodes. Line 20 is drawing the line using polyline, with the data loc; color, weight, and opacity is the modification of the path. And in line 23 the polyline will be added into m. If the hitung is equal to 40 then the condition is considered false and went into the else condition, line 25 is to break to terminate the loop, in line 26, hitung is added by one each time the loop occurs, and lastly in line 27 the map is displayed again.

```
28. count=0
29. pipelength=0
30. for index, location in data.iterrows():
31.   radius = 6371.0
32.   if count != 40:
33.     x1 = radians(data["x"][index])
34.     y1 = radians(data["y"][index])
35.     x2 = radians(data["x"][index+1])
36.     y2 = radians(data["y"][index+1])
37.     count=count+1
38.   else:
39.     x1 = radians(data["x"][index])
40.     y1 = radians(data["y"][index])
41.     x2 = radians(data["x"][index-40])
42.     y2 = radians(data["y"][index-40])
43.   xdistance = x2-x1
44.   xdistance = y2 - y1
45.
46.   p = sin(dx / 2)**2 + cos(x1) * cos(x2) * sin(dy / 2)**2
47.   count = 2 * atan2(sqrt(p), sqrt(1 - p))
48.
49.   distance = radius * count
50.   pipelength = pipelength + distance
51.
52.   print("Result",[index],": ", distance)
53.   print("Pipelength: ",pipelength)
```

Line 28, 29 is used to declare new variables, 30 is to start another loop. Line 31 is declaring R, R here is the radius of the earth in kilometer. Line 32 is another if else conditions that has the same function at line 17 before. Line 33 and 34 is to declare x1 and y1, which contains the radians from the coordinates. While the line 35 and 36 is the same but the coordinates will be from the next node available. Line 37 is to add the count variable, if the conditions in ifelse is not met, then the line 39-42 will take actions, there the process is almost the same with line 33-36 the difference between them is the x2 and y2 will take the radians from the first coordinates or index[0]. Line 43-44 is to calculate the distance between y2 and y1 / x2

and x1 in a new variable called dy and dx. Line 46-49 is the practice of haversine formula, line 50 is to count the overall pipelength, and line 52 and 53 is to print the result.

```
54. num_node=484
55.
56. coordinates=[]
57. for index, location in data.iterrows():
58.   x=data["x"][index]
59.   y=data["y"][index]
60.   coordinates.append([x,y])
61. names = [i for i in range(num_homes)]
62. coordinates_dict = {name: coord for name,coord in zip(names,
    coordinates)}
63.
64. coordinates=np.array(coordinates)
65.
66. print(coordinates_dict)
```

Line 54 is to declare number of nodes, line 56 is to declare a new variable, and for line 57 is another loop, both the longitude and latitude will be inserted into variable x and y and will be merged together in the coordinates variable, line 61 is to declare names which contains the number of nodes, line 62 is declaring coordinates_dict, that contains names and also coordinate. Line 64 is to convert coordinates into array, and line 66 is to print the coordinates_dict.

```
67. distanceMatrix = distance.cdist(coordinates, coordinates, 'euclidean')
68. print(distanceMatrix)
69.
70. fig = plx.imshow(distanceMatrix)
71. fig.show()
```

Line 67 is to declare distanceMatrix, here the distance matrix is calculated using a library, the distanceMatrix will compute a distance between each pair of the two collection, here the collection is the variable coordinates and also coordinates. ‘euclidean’ is to declare which distance metric to use. Line 68 is to show the distanceMatrix and line 70 71, is the visualization of the distance matrix using heatmap.

```
72. def create_chromosome(_names):
73.     chromosome = copy.deepcopy(_names)
74.     np.random.shuffle(chromosome)
75.     return chromosome
```

Line 72 is to define the function create chromosome, with the parameter names. Line 73 has the function to create a new variable called chromosome, that contains the copy of names, and at line 74 the content of chromosome will be shuffled. Line 75 is to return the value chromosome.

```
76. def evalChromosome(_distanceMatrix, _chromosome):
```

```
77.     distance = 0
78.
79.     for i in range(len(_chromosome) - 1):
80.         _i = _chromosome[i]
81.         _j = _chromosome[i+1]
82.         distance += _distanceMatrix[_i][_j]
83.
84.     distance += distanceMatrix[_chromosome[-1], _chromosome[0]]
85.     return distance,
```

Here, line 76 is to define another function which is the chromosome evaluation, or in here called evalChromosome, the parameter that will be used is the distance matrix and also the chromosome, line 77 is to declare a new variable called distance, while 79 is to start a loop in the range of chromosome -1 length. In line 82, distance will be added with the distance matrix that contains the value of _i and _j. Line 84 is adding another distance with the distance matrix but with chromosome[-1] and also chromosome[0]. Line 85 is to return the distance value.

```
86. populationNumber = 500
87. maxGeneration= 15000
88. crossoverprob = .6
89. mutationprob = .4
```

Line 86 has the function to determine how many populations will be generated, while line 87 will determine the maximum number of generations. Crossover probability and also mutation probability will also be declared in line 88 and 89.

```
90. tool.register('attribute', random.random)
91. tool.register('index',create_chromosome,names)
92. tool.register('indi',      tools.initIterate,      creator.Individual,
    tool.index)
93. tool.register('popul', tools.initRepeat, list, tool.indi)
94. tool.register('eval', evalChromosome, distanceMatrix)
95. tool.register('select', tools.selTournament)
96. tool.register('mate', tools.cxPartialyMatched)
97. tool.register('mutate', tools.mutShuffleIndexes)
98.
99.
100.popul = tool.popul(n=populationNumber)
101.fitSet = list(tool.map(tool.eval, popul))
102.print(min(fitSet))
103.for ind, fit in zip(popul, fitSet):
104.     ind.fitness.values = fit
```

Line 90-97 is the process of registering all of the information into the toolbox, line 100 has the function to declare variable population, here population will use tool.popul with the parameter population number that has been declared before. Line 101, is where the population is evaluated, line 101 is the process of mapping the evaluation function to every population, and the next step is to assign the respective fitness.

```
105.for gen in range(0, maxGeneration):
106.
107.    if (gen % 10 == 0):
108.        print(f'Generation: {gen}' )
109.        print(f'Fitness: {fitness:}' )
110.
111.    child = tool.select(popul, len(popul), tournsize=3)
112.    child = list(map(tool.clone, offspring))
```

Line 105 is to start another loop that stop when the maximum number of generation has been reached. Line 107 will conduct another ifelse situation, when the generation is modulo with 50 equals to 0 then the conditions is true, if true then in line 108 the number of generation and the fitness score will be printed. Line 110-111 is the process of selecting the next generations, offspring will be generated, this value contains list of the exact copy from the population.

```
113.for o1, o2 in zip(child[0::2], child[1::2]):
114.        if np.random.random() < crossoverprob:
115.            tool.crossover(o1, o2)
116.            del o1.fitness.values
117.            del o2.fitness.values
118.for chromosome in child:
119.    if np.random.random() < mutationprob:
120.       tool.mutate(chromosome, indpb=0.01)
121.       del chromosome.fitness.values
```

This code is the process of crossover, line 112 is the looping where child1 and child2 in the range of merged offspring. Line 113 will be another ifelse condition where if the random value is lower than the crossover probability then the o1 and o2 will be deleted (line 115-116). Line 114 is where the child is crossovered, crossover method that will be used is the partially matched, as written in line 96. Line 117 is another loop for each chromosome that's available in the offspring and continue in the if else situation, if the random value is higher than the mutation probability, there will be a tool.mutate from line 97 the mutation process that will be used in this is the shuffle index with independent probability for each attribute to be exchanged is 0.01 or 1%. Line 120 is to delete the chromosome fitness value.

```
122.invalid = [ind for ind in child if not ind.fitness.valid]
123.    fitSet = map(tool.eval, invalid)
124.    for ind, fit in zip(invalid, fitnSet):
125.        ind.fitness.values = fit
```

Line 121-124 has the function to map the offspring from the fitness that's marked invalid in the variable invalid_ind.

```
126.    population[:] = offspring
127.
128.    currentSolution= tools.selBest(population, 2)[0]
```

```
129.     currentFitness = currentSolution.fitness.values[0]
130.
131.     if currentFitness<best_fit:
132.         solution = currentSolution
133.         fitness = currentFitness
134.
135.     fitness_list.append(fitness)
136.     solution_list.append(solution)
```

Line 125 is to replace the old population with the offspring. Then line 127 is to select the best population, and line 128 is to find the best fitness available. Line 130 explains if the current best fitness is lower than fitness then the best solution and best fitness will be replaced. Line 134-135 is where best fitness and best solution combined with the rest of best fitness/solution list.

5.2. Results

5.2.1. Result with 40 Nodes

Result of genetic algorithm when given 40 nodes, genetic algorithm is able to optimize the waste transport route.

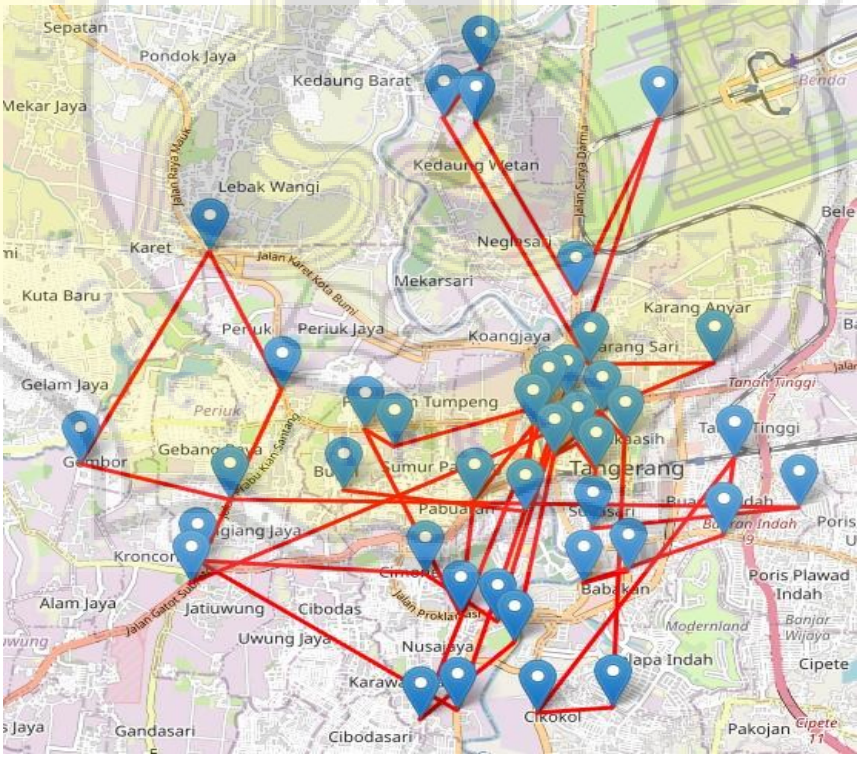

Figure 5.1 40 Nodes Initial Route

Figure 5.1 shows the original route from the data, where the route is determined by the number of index and nothing more.

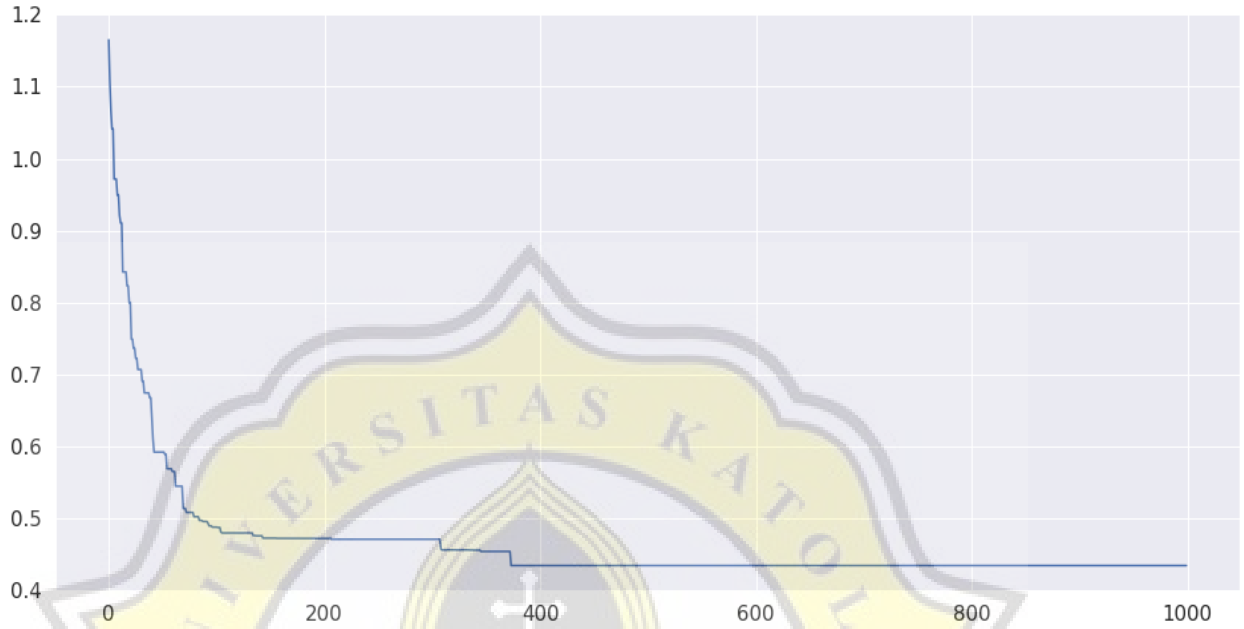

Figure 5.2 40 Nodes Fitness Graph

Figure 5.2 shows the process of computing the fitness of genetic algorithm, since the maximum number of generations is 1000, genetic algorithm will stop when generations has reached 1000, the chart shows that the fitness value is determined at about 300-400 generations, with the value less than 0.5.

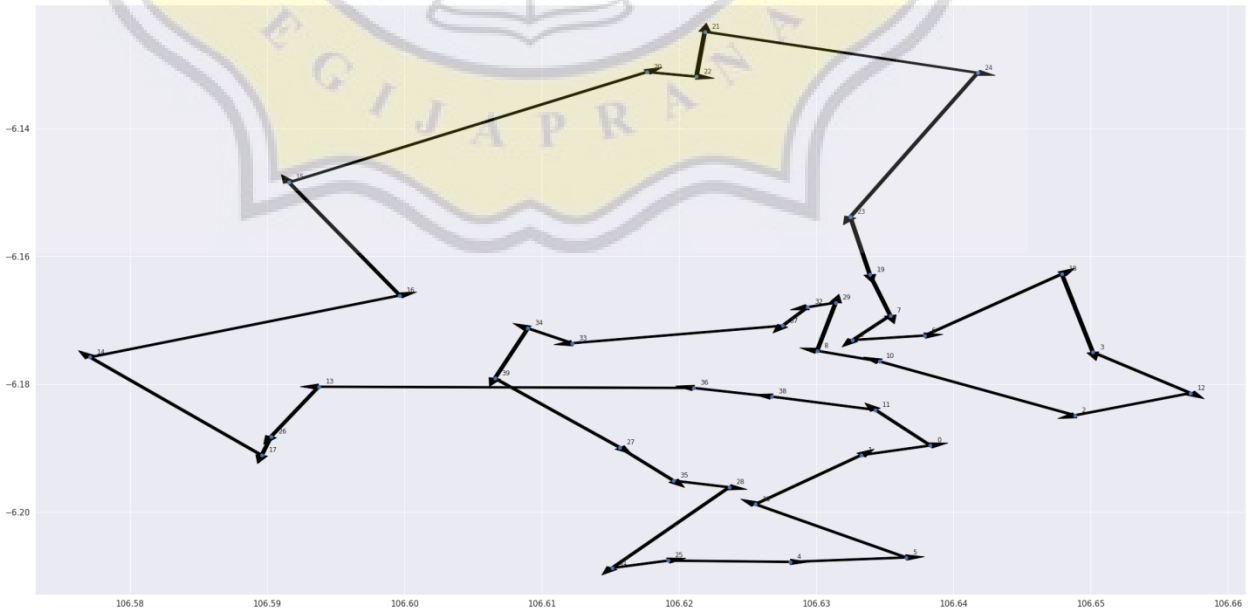

Figure 5.3 40 Nodes Genetic Algorithm Route

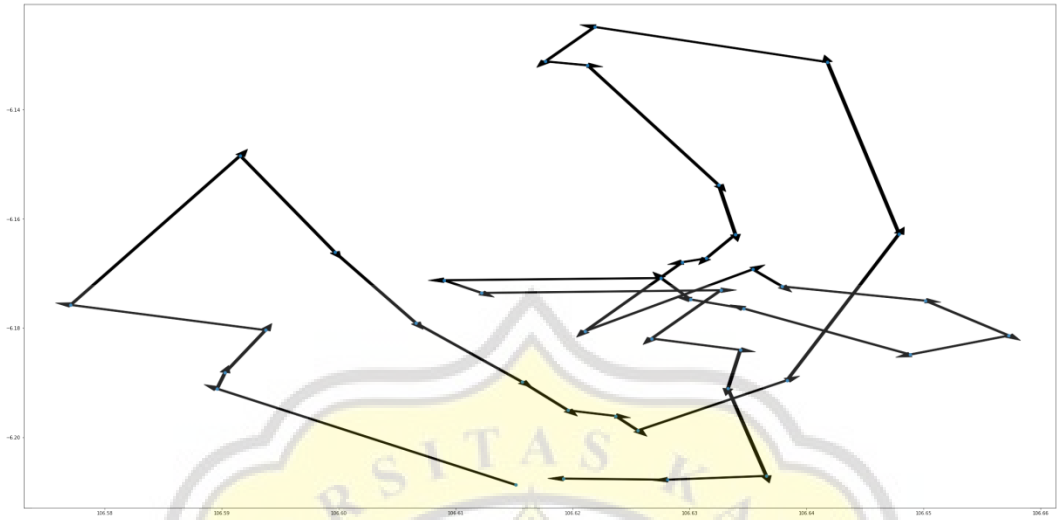

Figure 5.4 40 Nodes Brute Force Route

Figure 5.3 and 5.4 shows the route generated by genetic algorithm and also brute force, by the graph the route produced by genetic algorithm shown to be more neat.

Table 5.1. Before and After Genetic Algorithm Comparison

	Original	**Genetic Algorithm**	**Brute Force**
Result [0]	0.5631612106884862	1.7083730491896456	2.8932641702388286
Route Length	0.5631612106884862	1.7083730491896456	2.8932641702388286
Result [1]	1.7437025959569896	0.3658067586384525	0.12045321306959018
Route Length	2.3068638066454756	2.074179807828098	3.013717383308419
…	…	…	…
Result [38]	2.243642201142117	0.22837772829356223	0.9901011031766647
Route Length	62.16260620824876	35.740669887802426	38.31329870232137
Result [39]	3.4251133284295787	0.22227086178143585	0.45538121382352215
Route Length	65.58771953667834	35.962940749583865	38.76867991614489

5.2.2. Result with 80 Nodes

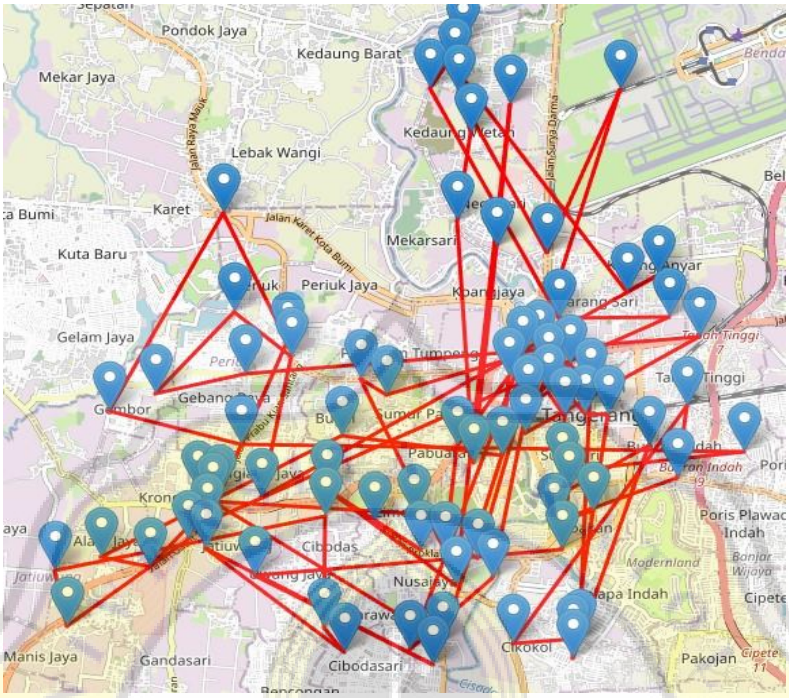

Figure 5.1 80 Nodes Initial Route

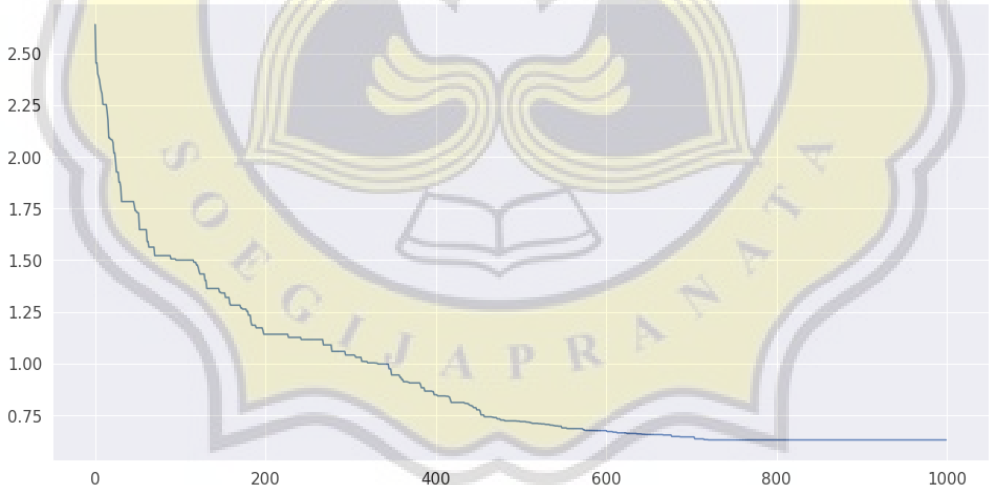

Figure 5.2 80 Nodes Fitness G

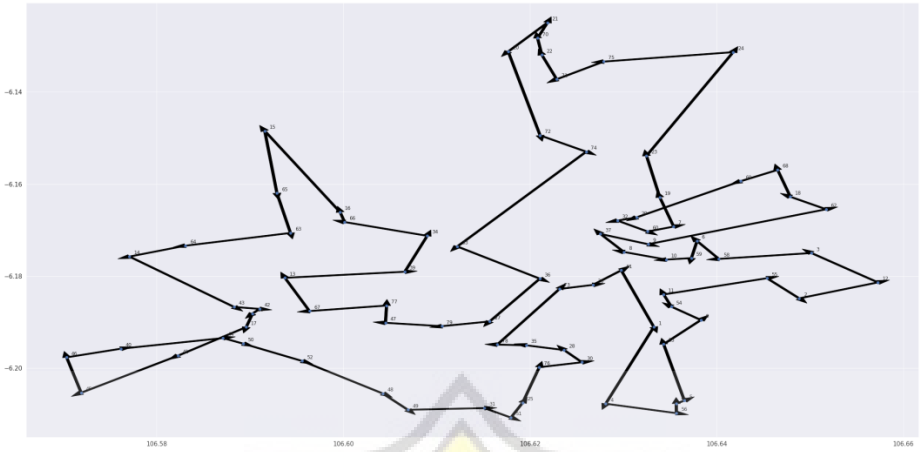

Figure 5.3 80 Nodes Genetic Algorithm Route

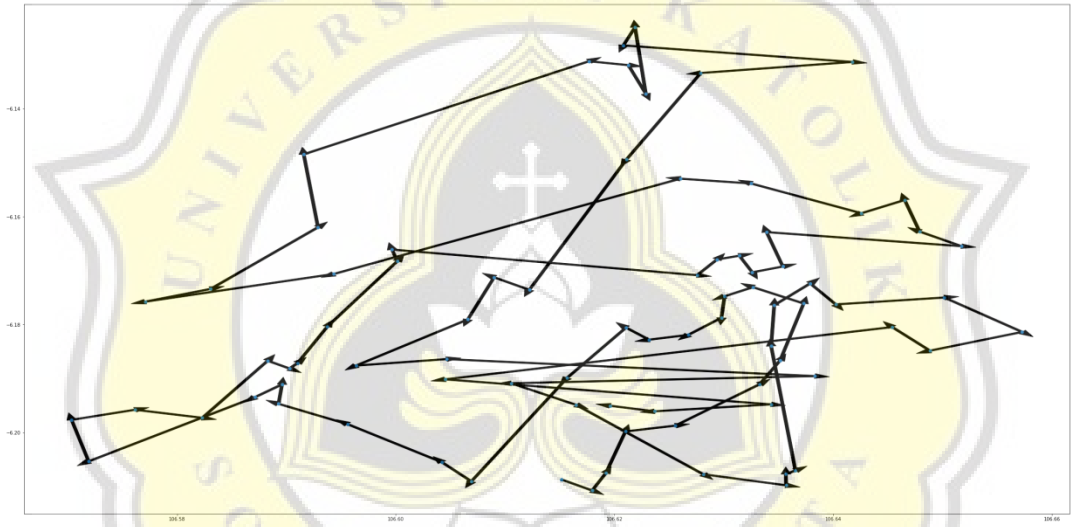

Figure 5.4 80 Nodes Brute Force Route

Table 5.2. Before and After Genetic Algorithm Comparison

	Original	**Genetic Algorithm**	**Brute Force**
Result [0]	0.5631612106884862	0.24009883037246396	0.3211068157342795
Route Length	0.5631612106884862	0.24009883037246396	0.3211068157342795
Result [1]	1.7437025959569896	0.44624071854333186	0.17301471897737694
Route Length	2.3068638066454756	0.6863395489157958	0.49412153471165643
…	…	…	…
Result [78]	0.6722039905553483	2.133363596132943	0.45230817911995236
Route Length	123.14582049540456	47.36012963637782	69.13986255574876
Result [79]	2.540565999548715	0.4341426294850353	0.6602499972833791
Route Length	125.68638649495328	47.79427226586286	69.80011255303214

5.2.3. Result with 120 Nodes

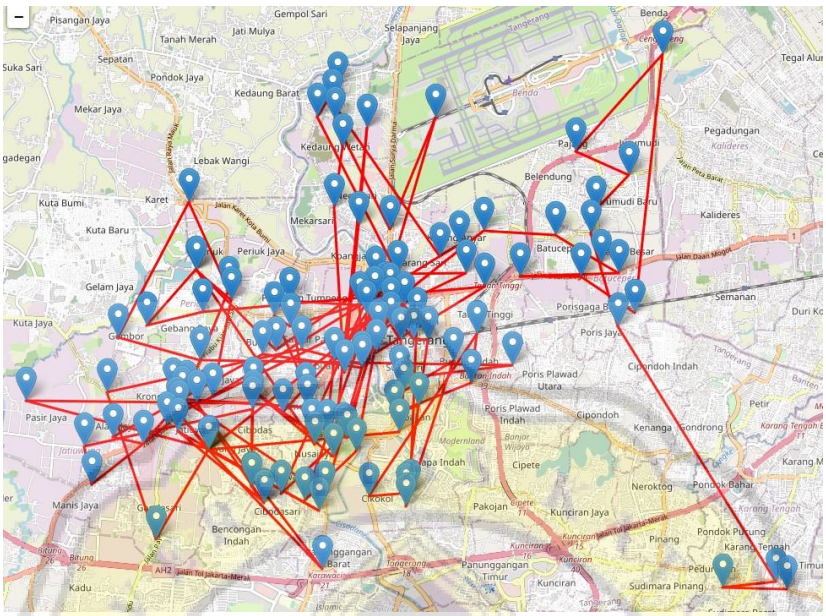

Figure 5.1 120 Nodes Initial Route

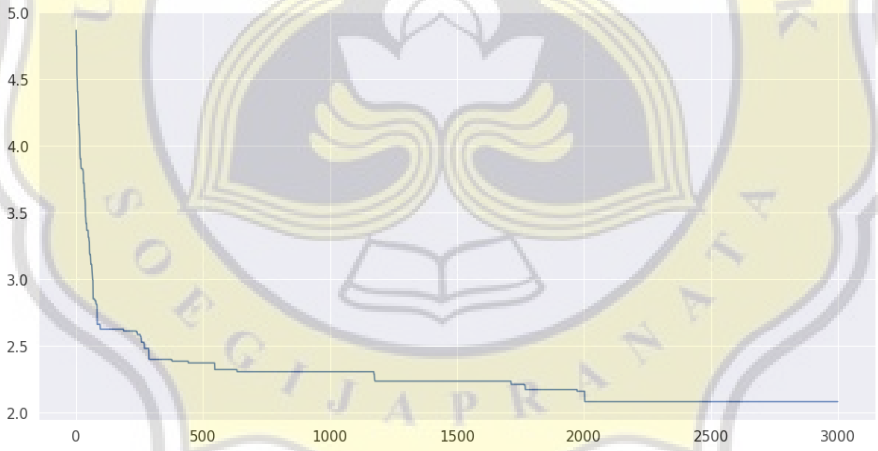

Figure 5.2 120 Nodes Fitness Graph

Figure 5.3 120 Nodes Genetic Algorithm Route

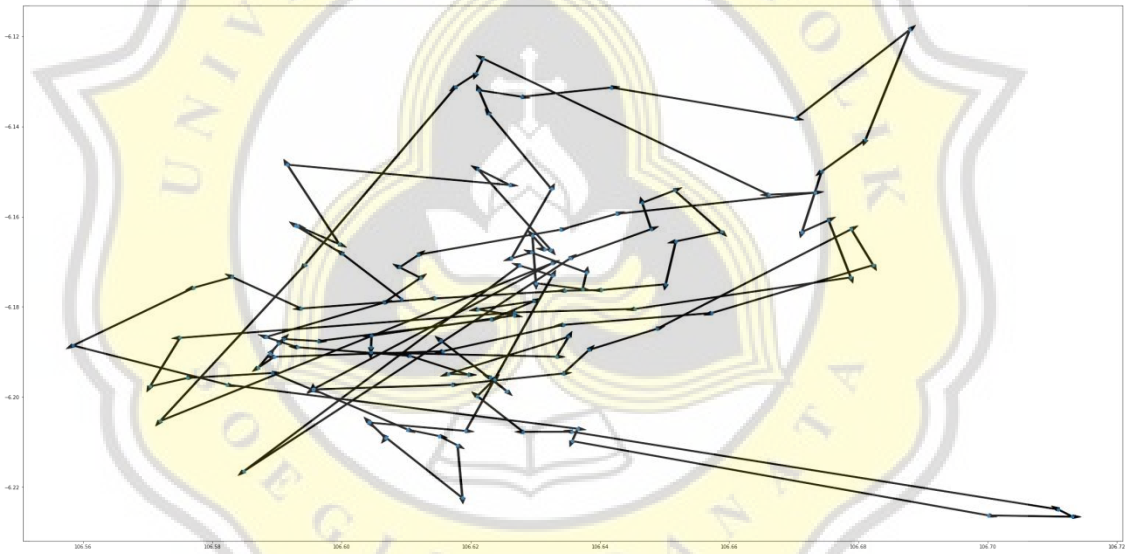

Figure 5.4 120 Nodes Brute Force Route

Table 5.3. Before and After Genetic Algorithm Comparison

	Original	**Genetic Algorithm**	**Brute Force**
Result [0]	0.5631612106884862	1.5923794282150716	1.0664342437622145
Route Length	0.5631612106884862	1.5923794282150716	1.0664342437622145
Result [1]	1.7437025959569896	0.40893632177290185	0.8308951431785069
Route Length	2.3068638066454756	2.0013157499879735	1.8973293869407213
…	…	…	…
Result [118]	1.4462307207016623	0.6088026306313461	1.1920458970030459
Route Length	196.5610145810444	149.07580418904732	176.35224570184624
Result [119]	1.0268353078008376	0.40140381468802305	0.7334800138807783

Route Length	197.58784988884523	149.47720800373534	177.08572571572702

5.2.4. Result with 160 Nodes

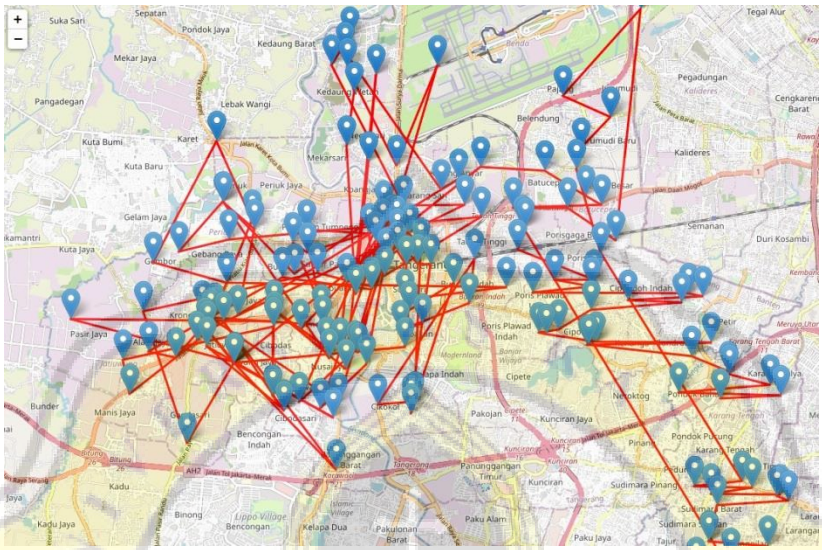

Figure 5.1 160 Nodes Initial Route

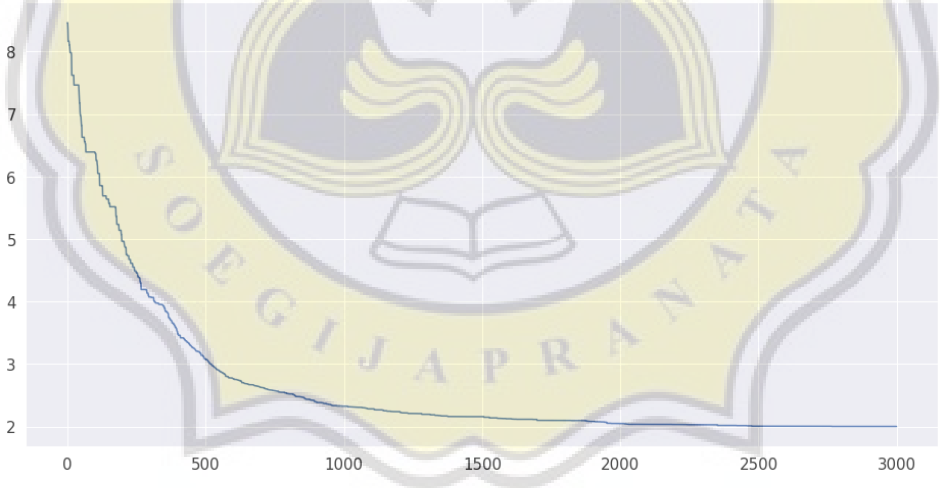

Figure 5.2 160 Nodes Fitness Graph

Figure 5.3 160 Nodes Genetic Algorithm Route

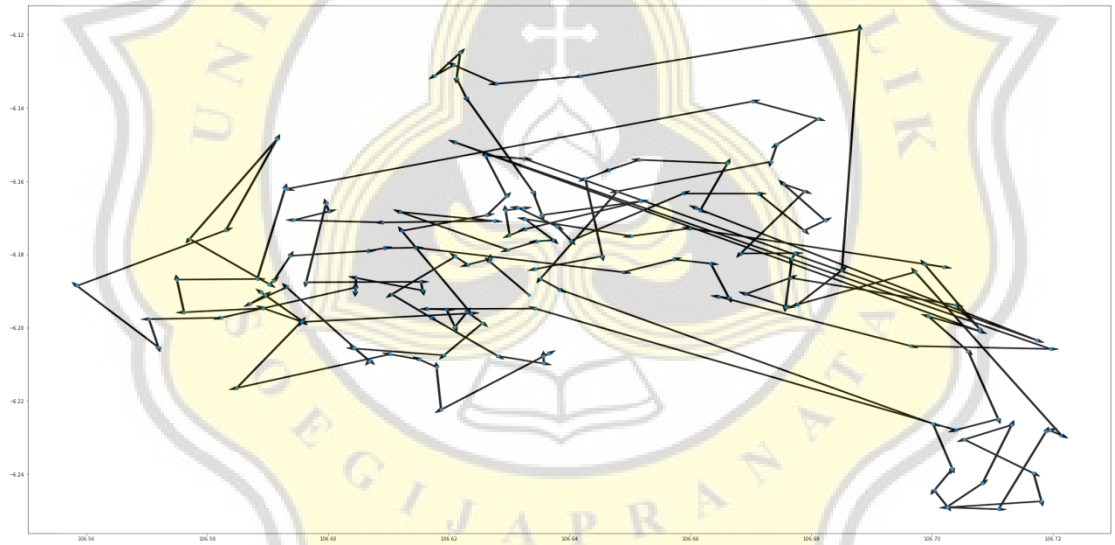

Figure 5.4 160 Nodes Brute Force Route

Table 5.4. Before and After Genetic Algorithm Comparison

	Original	**Genetic Algorithm**	**Brute Force**
Result [0]	0.5631612106884862	0.02903115341224859	0.7813824906675501
Route Length	0.5631612106884862	0.02903115341224859	0.7813824906675501
Result [1]	1.7437025959569896	0.19775520216635067	0.0225142204910293
Route Length	2.3068638066454756	0.22678635557859927	0.8038967111585793
…	…	…	…
Result [158]	1.5214820293907358	0.5023232733131163	1.2360518380234788
Route Length	251.36570578287692	161.49328537426783	210.25904693025487

Result [159]	7.536517853946805	0.7367009841523663	0.2537507246858879
Route Length	258.9022236368237	162.2299863584202	210.51279765494075