Genetic optimization

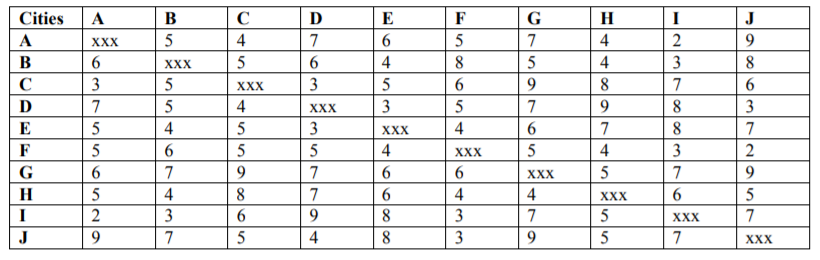
**Elements of Artificial Intelligence**

Student: Vanessa Maria Mercea

Teacher: Andrzej Bargiela

**Table of Contents**

1. Presenting the task ………………………..………………………………….....2
2. Design ……………………………………………………………………...…...3
3. Implementation …………………………………………………………...…….7
   1. Classes ……………………………………………………………………..9
4. City Class ………………………………………………………………9
5. Path Class ……………………………………………………………..10
6. Population Class ………………………………………………………12
7. GA Class ……………………………………………………………...13
8. Main Class …………………………………………………………….14
   1. Operators …………………………………………………………………15
9. Crossover Operators …………………………………………………..15
10. Partially-Mapped Crossover …………………………………..15
11. Order-Based Crossover ……………………………………….16
12. Cycle Crossover ……………………………………………… 17
13. Edge-Recombination Crossover ………………………………18
14. Mutation Operators …………………………………………………...20
15. Displacement Mutation ……………………………………….20
16. Swap Mutation ………………………………………….……..20
17. Scramble Mutation …………………………………………....21
    1. Selection of Parents ……………………………………………………....21
18. Tournament Selection ………………………………………………...21
19. Roulette-Wheel Selection ……………………………………………..22
20. **Presenting the task**

Travelling salesmen problem concerns minimizing the length of the journey between N cities without repeated visits to any city, and with return to the starting city. The table below provides distances between 10 cities. The connection between the cities are not symmetrical. The task is to design and implement a genetic optimization program that solves the travelling salesman problem for the case specified below.

1. **Design**

The first matter that must be considered when designing a Genetic Algorithm is the problem representation. In the situation of the Travelling Salesman Problem, the genotype is path-defined, which means possible permutations containing all the cities. The phenotype is represented by the order in which the cities are visited, once.

In order to see whether the implementation will be working, the optimal solution to this problem was computed using backtracking, and this solution has a distance of 36. There are many paths that lead to this solution. For this problem, the number of different paths that can be obtained is 1 814 400, which means 1 814 400 \* 10 iterations (10 due to the length of each path) for the straight-forward, backtracking solution. Computing the solution for this problem in this manner to find the minimum path is both time and resource-consuming. By using a good Genetic Algorithm design, the solution can be obtained in less than 1% of the previously mentioned number of iterations.

The path representation of the problem was used because opposed to the binary representation, using different crossover operators and mutations on the list paths only leads to valid solutions. For the binary representation, the binary operators sometimes lead to invalid tours and some repair functions would be necessary.

Initially, the path population size was set to 100, and contained randomly generated paths. Even though 100 is a small part of the total possible paths, this value could be decreased by a considerably high amount during implementation and experimentation.

The populations were evolved for 100 generations, at the beginning, until proper modelling of the problem had been achieved.

The fitness function for this problem was chosen based on the willingness to obtain short distance paths as offspring in order to achieve the expected solution. Therefore, the fitness of a path is defined by 1 / path distance.

Initially, to test how the optimization is working, partially-mapped crossover was implemented, along with displacement mutation. For the crossover and for the mutation, 2 parameters that control their occurrence rate were introduced.

For the selection of the parents out of the current population, at first, Roulette wheel selection was implemented, and the output was most of the time converging (but not reaching) to the expected value. However, it was not too satisfying, as it was time consuming because it was iterating through all the paths of the current population, during the crossover loop. So quite a lot to process. Using a more random-based selection seemed to be quicker. Therefore, tournament selection was also implemented. Having these both, experimentation could be done different types of crossing over or mutations. Tournament selection worked nicely, although it was leading to paths with a distance higher on average than roulette wheel selection, it was very quick, so adjustments could be done to the other parameters of the genetic algorithm in order to lead to a better solution.

Regarding the reproduction of the current generation, initially, the implementation involved keeping a quite large number of the best-fitness individuals from the current population for the next one, but that lead to the local minimum problem, as all the paths began converging to the best solution found until some point. Mutation had almost no impact on the population. This was happening because the population diversity was constantly decreasing which had a very negative impact since there was such a small coverage of the exploration space. Another attempt was to use the crossover rate to control the parents’ crossover. When this rate was lower than random, the parents were directly copied into the offspring. This option was better than the previous one, but the results would get better as the crossover rate was higher, meaning the less parents are copied into offspring, the better. These being considered, implementation was modified to discard all the current population and only keep the offspring.

At this point, more types of crossover operators were considered, implemented and tested. These other crossovers are: cycle crossover, order-based crossover (version 1) and edge recombination crossover. These were chosen based on research about other TSP versions and their solutions, and on the fact that a path representation of the individuals had been chosen.

For mutations, two other versions were implemented: swap and scramble. The reason for this choice was their synergy with some of the crossover operators mentioned above. These synergies were determined on average, for some TSP versions, by researchers. This synergy however, was affected by the other parameters of the Genetic Algorithm that were used/chosen, and by the problem particular data. Therefore, although it was expected of some crossover operators to behave in a certain way, the outcome was very dependent to this problem, and emphasized other synergies.

Normally for Genetic Algorithm problems, the optimal answer is not known, so the purpose is only to optimize as much ass possible. For this case, since the optimal path distance could be found through backtracking, all optimization was done until the outcome was optimal.

After having properly designed the algorithm and implemented all of the above, the correct answer would not be reached too often, with any combination of selection/operators.

By checking the population evolution, it was noticed that the population would converge to a value and generate new individuals within a small range around that value. This means that the crossovers were producing good offspring but there was still not enough diversity. Increasing the value of the mutation probability would only get too much diversity for the first few generations and make the solution jump from a value to another. Having a large parameter for the mutation rate was not leading to anything. Thus, a way of controlling or adapting this mutation rate had to be introduced. A simple approach was chosen, that would count for how many times the fittest of the generations has been the same. If this would happen for more than a number N of generations (N = 5 for instance) then the mutation rate would adapt. This adapt consists of multiplying the mutation rate by a predefined parameter. For a stable and slow increase, 1.1 was chosen. By using this, the problem with the diversity was solved, and the exploration of space was done smooth enough in order to explore new space without jumping from a value to another. This was happening because when the outcome of the current generation would be different, the mutation rate would not change, and by the time it would get stuck in the local minimum, the mutation rate increase would cause a jump away from that. If the evolution of generations was done for more than 1000 times, this would lead to too much diversity, but as the algorithm should be properly designed to solve the problem in as few generations as possible, this mutation solution was appropriate.

A crossover rate was also introduced to control crossover occurrences, and generate some more randomness in the parent choosing. It was observed that it is best to allow crossover of the first parents chose by the selection function. Trying to repeat the process and find other parents would only decrease the rate of success, so having such a parameter was no good to this particular problem.

At this time, the algorithm is performing quite nicely, but while observing the generations and their outputs, it was noticed that values very close to the correct one were found after very few generations, but were discarded. The offspring of those values would have worse fitness than their parents sometimes, due to the crossover mechanisms. Because losing some very good individuals would increase the chances of diverging from the correct answer, elitism was added as an option to the reproduction. In this way, the best individual of the current generation would always have a place in the population of the next generation. Thus, good solutions were not discarded, while the diversity would still be good enough due to the mutation adaption.

After the parameters were tuned, most of the crossover and mutation operators’ combinations would lead to a very satisfying output. The roulette selection of parents would be too slow and would not have a good synergy with the new parameters of the problem, leading to values very close to the correct output but very rarely equal. Therefore, tournament selection was kept for all the other adjustments. From now on, the population and the number of generations could be adjusted to lower values. Best setting that could be achieved while still getting 100% correct answers consisted in a population of 50 individuals, evolved for 80 generations. A further decrease to any of these would lead to a decrease in the percentage of correct answers. The decrease would be small, but it is important to get a confident result. For this adjustment, while having elitism and mutation adaption allowed, the crossovers and mutation operators showed the results below. For each of these, the other parameters such as mutation probability or tournament size were tuned to get the best output of the current mixing of operators. The results were gathered from 100 different runs for each case. One case had an evolution of 80 generations for a population of 50, which at the beginning contains random individuals.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PMX | OX1 | CX | ER |
| Displacement M | 97%  (max repetitions = 3, tournament size = 7) | 100%  (max repetitions = 5, tournament size = 6) | 99%  (max repetitions = 5, tournament size = 6) | 41%  (max repetitions = 3, tournament size = 7) |
| Swap M | 86%  (max repetitions = 3, tournament size = 6) | 99%  (max repetitions = 5, tournament size = 6) | 92%  (max repetitions = 5, tournament size = 5) | 58%  (max repetitions = 3, tournament size = 7) |
| Scramble M | 93%  (max repetitions = 5, tournament size = 5) | 97%  (max repetitions = 5, tournament size = 7) | 88%  (max repetitions = 5, tournament size = 5) | 91%  (max repetitions = 5, tournament size = 25) |

After several runs and parameter adjustments, it was observed that disabling the mutation adaption only leads to poor results. Maximum ~ 65% correct answers. Disabling elitism does not cause big changes. It worsens the Scramble mutation row, but the other results are similar. Disabling both leads to almost no success rate, so these parameters are necessary for this design.

The best results were gained from order-based crossover combined with displacement mutation. Generally, the order-based crossover had good, stable outcome. This shows that if a certain parent path has a good fitness, trying to maintain its’ cities in the same order in the offspring will result in a high chance of getting an offspring with good fitness. Displacement mutation has a good synergy with this because it selects an area of the parent path and moves it to another position, thus keeping most of the order intact. The edge recombination crossover operator is less likely to produce stray edges over time, and is very complicated compared to the other operators, but unfortunately, it only worked well with the scramble mutation, and for a large tournament size (equal to half the number of parents). The cycle operator also had good synergy with the displacement, and if ran again, from time to time it also produces 100% correct answers. That’s because, in cycle operator the order is also kept if a cycle ends either after a very short period, or a very long one. All the other combinations result in not-so-stable outcome, and would not be quite a good solution for this problem. If the number of parents is increased, these might lead to better results, but it is best to keep the population size to a minimum.

Overall, the total number of iterations by using the best parameters would be:

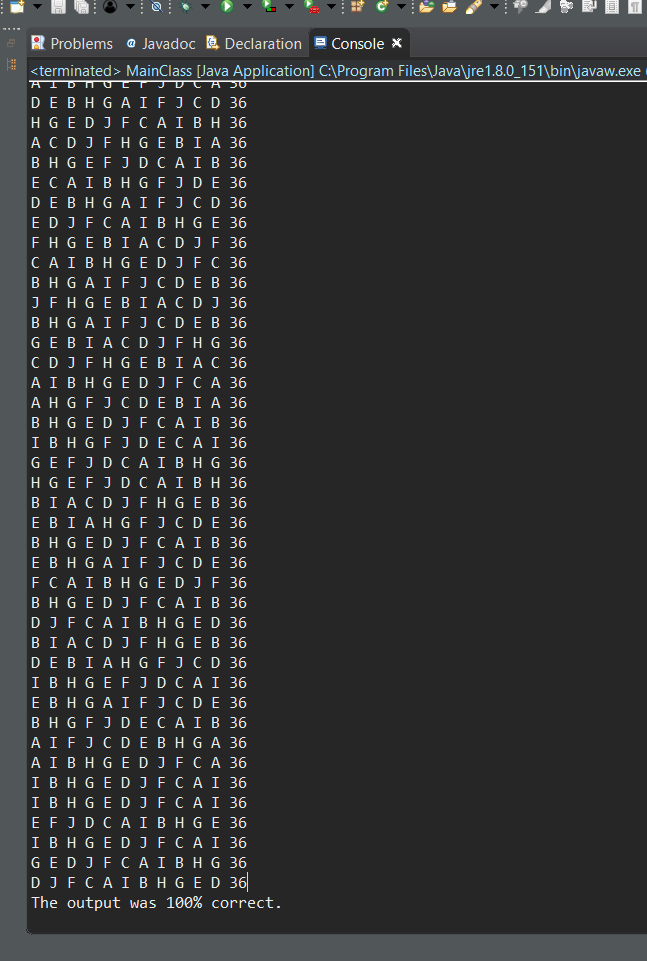
80 \* 50 \* (10 + 10), where 80 – number of generations, 50 – size of population, 10 – the maximum iterations required by one displacement, 10 – the number of iterations required by one ox crossover. This number represents 0.44% of the number of iterations backtracking uses, and the algorithm only starts with 0.002% of the total exploration space to find the correct answer. There are 70 different optimal paths for this particular input data, which might be the reason why the algorithm is not having great difficulties in finding such a path after a short period of time.

1. **Implementation**

Regarding implementation, the code for obtaining the backtracking straight-forward output was written in Python 2.7. After this, it was decided that a good approach of this problem would be from an object-oriented perspective, so the code for the Genetic Algorithm was written in Java.

The code is well commented, so going into detail is not necessary. Code snippets are attached.

This is the console showing the paths and their distances, and that the design presented above gives 100% correct answers.



* 1. **Classes**

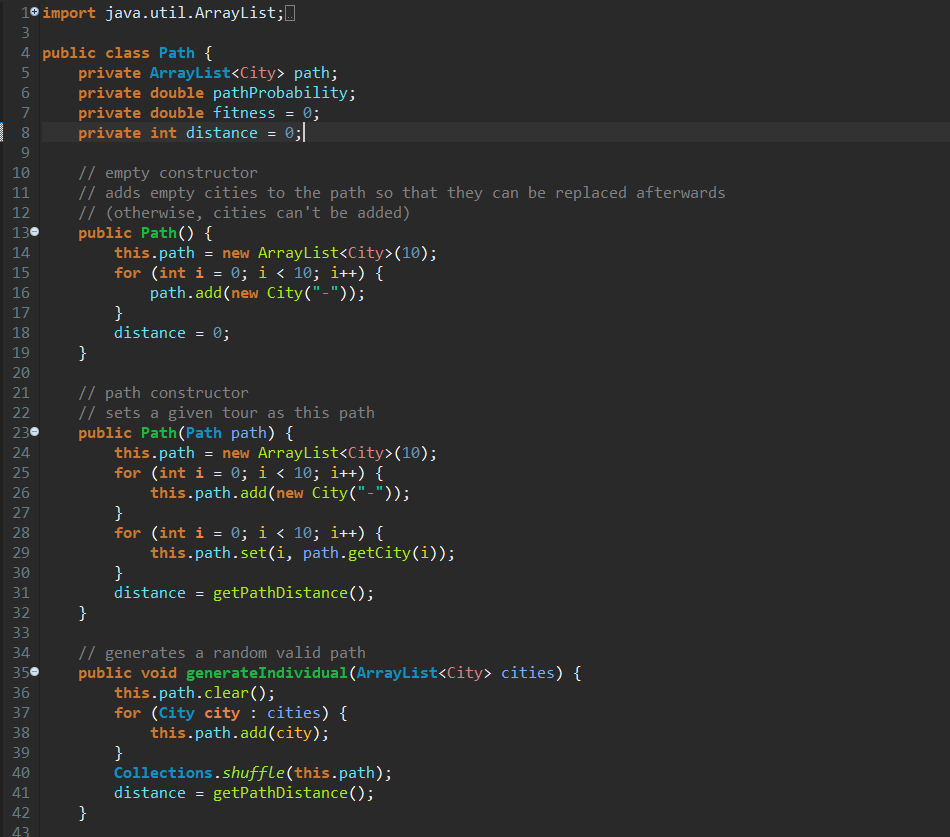
For this implementation, 5 classes were used to define the problem. These are: City, Path, Population, GA (Genetic Algorithm), and the MainClass.

1. **Class City**



The class City will be instantiated into 10 City objects: A-J. In this way, all the paths will represent valid tours.

1. **Class Path**



This class represents a tour of the salesman, and it holds information about the fitness, the distance, and the cities it contains.

All paths are of size 10, but when the distance is computed, the distance towards the initial city is considered too. There was no reason for the GA to see the paths as a closed cycle, as the actual number of solutions is the same as if the paths were an acyclic tour with one visit in each city. The only difference is that when computing the distances.

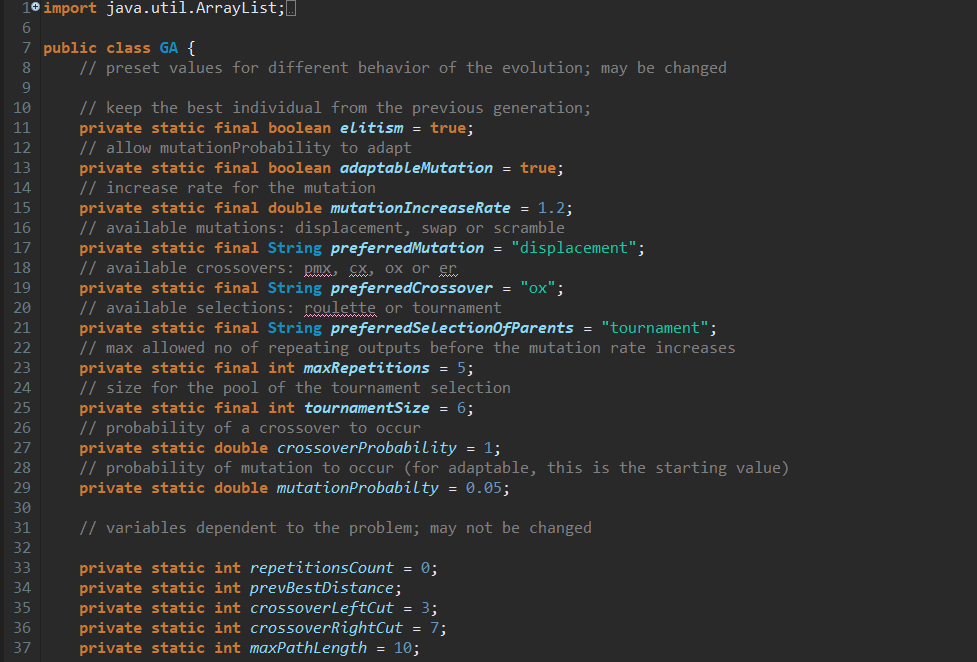


1. **Class Population**

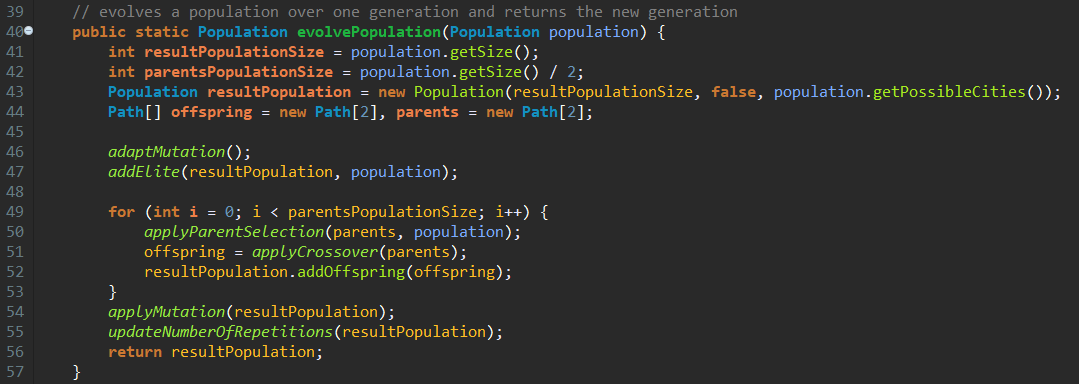


1. **Class GA**

These are the modifiable and non-modifiable parameters:

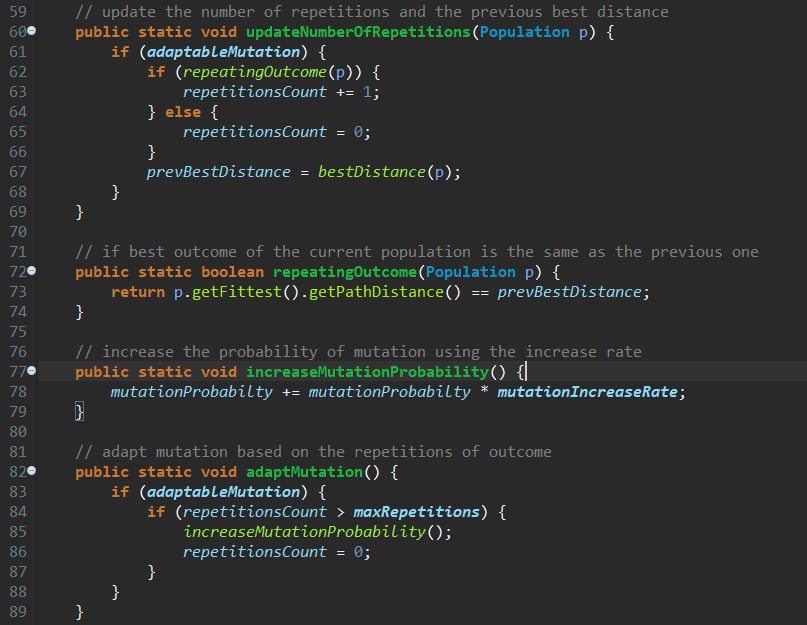


The function that evolves the current population once:



The apply functions are used to call the appropriate preferred selection and crossover/mutation operators.

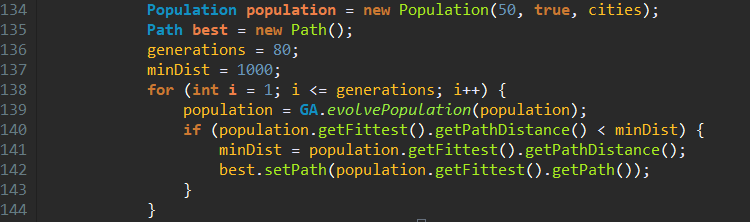
Helping functions for the mutation update:



The rest of the GA class is presented in the next section, describing Operators, Selections, and mutation adapting.

1. **Main Class**

The main class creates the cities and sets the distances between them. The evolution of the population is done in this part of the code. For 80 generations, it evolves the population of size 50, and keeps the best distance that was computed and its’ path.



* 1. **Operators**

1. **Crossover Operators**
2. Partially-Mapped Crossover (PMX)



1. Order-Based Crossover (OX1)



1. Cycle Crossover (CX)



1. Edge-Recombination Crossover (ER)

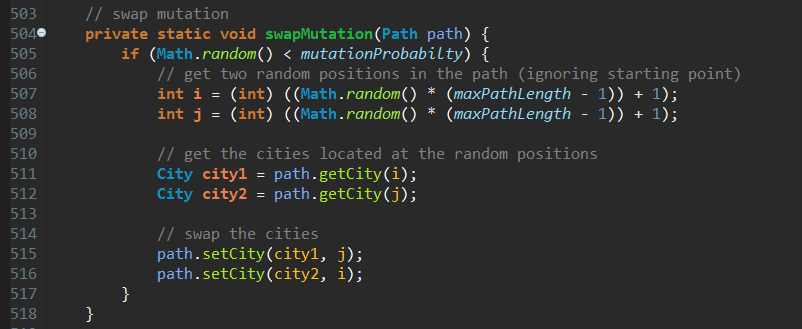




1. **Mutation Operators**
2. Displacement Mutation



1. Swap Mutation

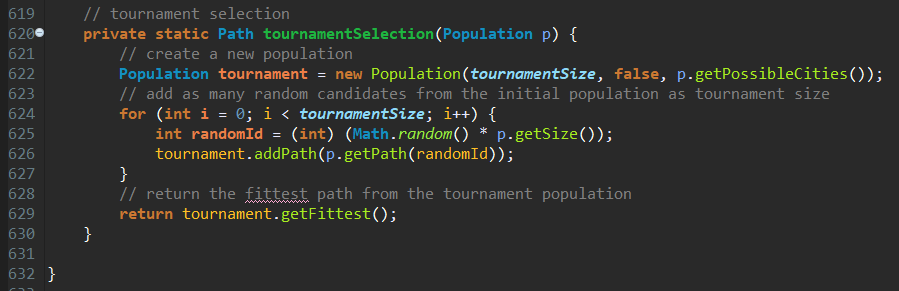


1. Scramble Mutation

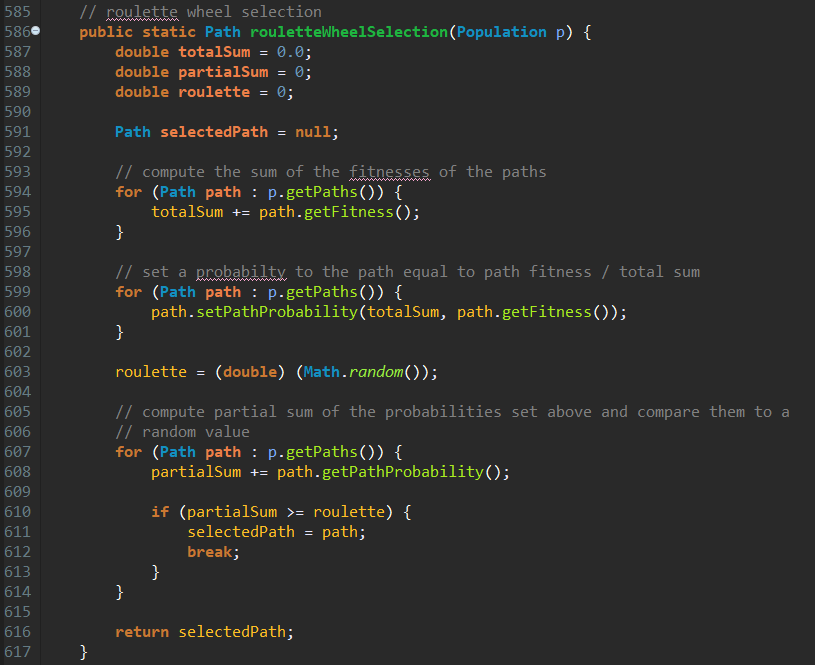


* 1. **Selection of Parents**

1. Tournament Selection



1. Roulette-Wheel Selection



**Bibliography**

1. Elements of Artificial Intelligence Courses
2. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators, P. Larranaga, C.M.H. Kuijpers, R.H. Murga I. Inza and S. Dizdarevic, Department of Computer Science, Spain
3. Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation, Gokturk Ucoluk, Department of Computer Engineering, Turkey
4. Genetic Algorithms for the Traveling Salesman Problem, Jean-Yves Potvin, Centre of Transport Research, University of Montreal
5. <http://www.rubicite.com/Tutorials/GeneticAlgorithms.aspx>
6. <https://www.tutorialspoint.com/genetic_algorithms/>
7. https://www. stackoverflow.com/