Artificial intelligence

documentation for assignment 2

Contents

Assignment

The number of task is 2c.

The task was to solve Travelling Salesman Problem with these algorithms: genetic algorithm, tabu search and simulated annealing.

The Traveling Salesman Problem (TSP) is a classic optimization problem in which a salesman needs to visit a set of cities and return to his starting point while minimizing the total travel distance or cost. The salesman's goal is to find the most efficient route that allows him to visit each city exactly once and return to the starting city. The transportation cost between any two cities is directly proportional to the distance or length of the travel. In essence, the TSP seeks to determine the shortest possible closed route that connects all cities, ensuring that no city is visited more than once.

To solve this problem I choose genetic algorithm and tabu search.

Program was done in Java language.

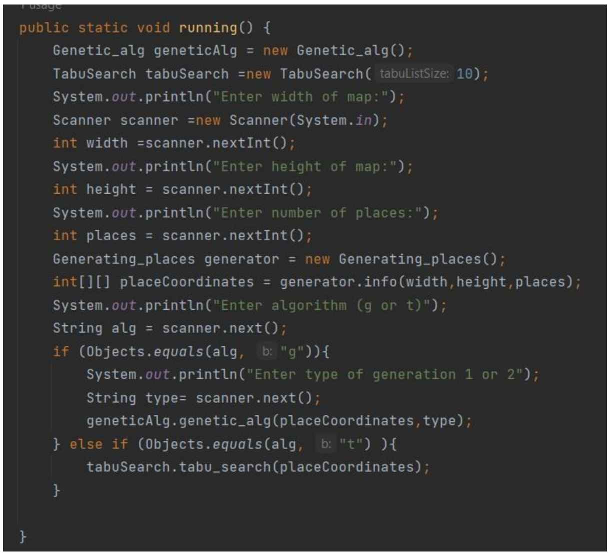
Alexandra Vetrov ID: 123102

Genetic algorithm

A Genetic Algorithm (GA) in machine learning is a search and optimization technique inspired by the process of natural selection. GAs are used to find the best combination of model parameters or features in order to optimize a machine learning model's performance. The Genetic Algorithm iteratively refines the population of solutions, aiming to find the best set of parameters or features that optimize the machine learning model's performance for a given task. GAs are particularly useful when searching through large solution spaces or when dealing with complex optimization problems in machine learning.

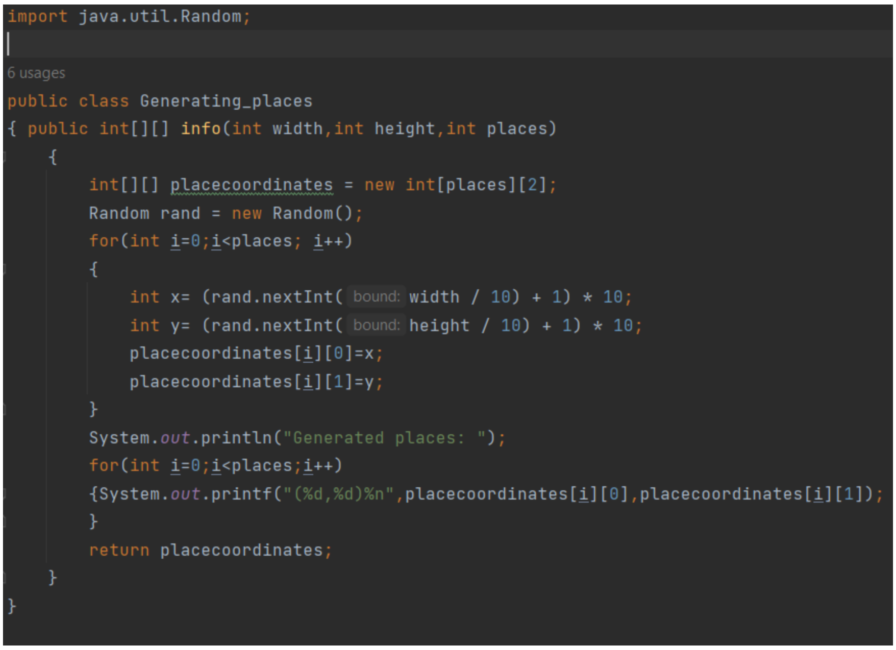
In my program I implement it like this:

In the beginning of program I define size of map, number of places and I choose which algorithm I want to run.



```
public static void running() {
    Genetic_alg geneticAlg = new Genetic_alg();
    TabuSearch tabuSearch =new TabuSearch( tabuListSize: 10);
    System.out.println("Enter width of map:");
    Scanner scanner =new Scanner(System.in);
    int width =scanner.nextInt();
    System.out.println("Enter height of map:");
    int height = scanner.nextInt();
    System.out.println("Enter number of places:");
    int places = scanner.nextInt();
    Generating_places generator = new Generating_places();
    int[][] placeCoordinates = generator.info(width,height,places);
    System.out.println("Enter algorithm (g or t)");
    String alg = scanner.next();
    if (Objects.equals(alg,  b: "g")){
        System.out.println("Enter type of generation 1 or 2");
        String type= scanner.next();
        geneticAlg.genetic_alg(placeCoordinates,type);
    } else if (Objects.equals(alg,  b: "t") ){
        tabuSearch.tabu_search(placeCoordinates);
    }

}
```

I have class “Generating_places” which generate coordinates of places.

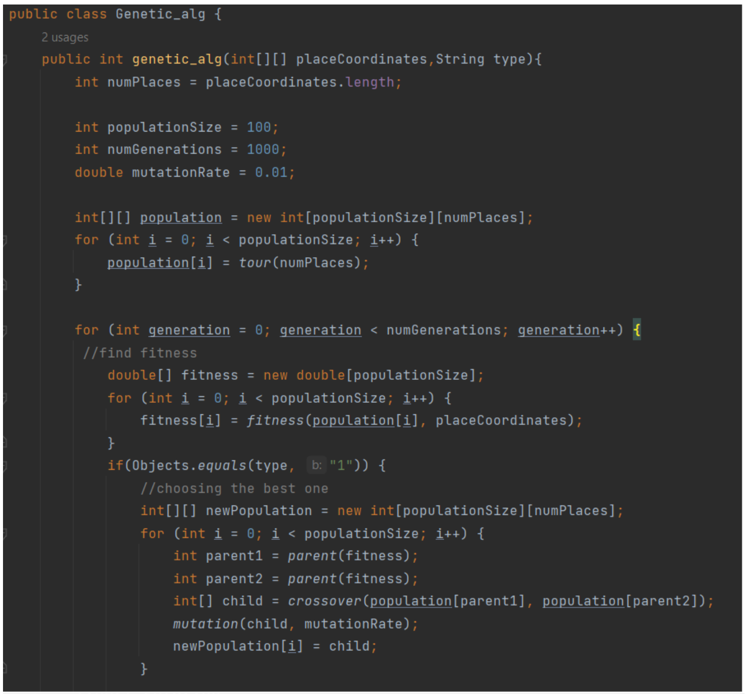


```
import java.util.Random;

6 usages
public class Generating_places
{ public int[][] info(int width,int height,int places)
    {
        int[][] placecoordinates = new int[places][2];
        Random rand = new Random();
        for(int i=0;i<places; i++)
        {
            int x= (rand.nextInt( bound: width / 10) + 1) * 10;
            int y= (rand.nextInt( bound: height / 10) + 1) * 10;
            placecoordinates[i][0]=x;
            placecoordinates[i][1]=y;
        }
        System.out.println("Generated places: ");
        for(int i=0;i<places;i++)
        {System.out.printf("(%d,%d)%n",placecoordinates[i][0],placecoordinates[i][1]);
        }
        return placecoordinates;
    }
}
```

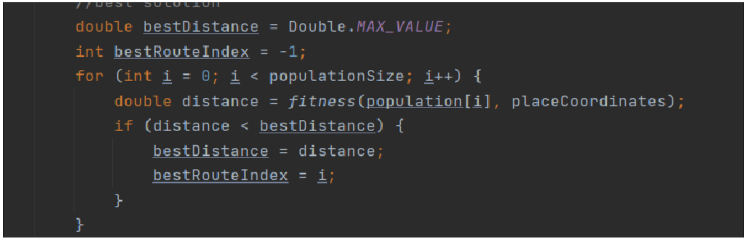
Alexandra Vetrov ID: 123102

In class Genetic_alg I define the parameters, such as population size, number of generations, and mutation rate.

genetic_alg: This is the main function. It takes the coordinates of places and the type of parent selection method as input. It initializes the population, evaluates fitness, selects parents, performs crossover, mutation, and iterates through generations to find the best route for the Traveling Salesman Problem.

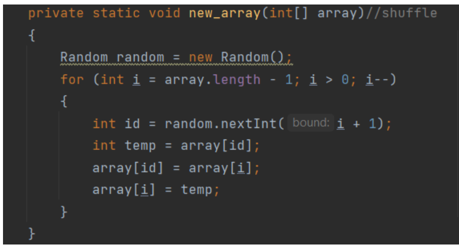


```
public class Genetic_alg {
    2 usages
    public int genetic_alg(int[][] placeCoordinates,String type){
        int numPlaces = placeCoordinates.length;

        int populationSize = 100;
        int numGenerations = 1000;
        double mutationRate = 0.01;

        int[][] population = new int[populationSize][numPlaces];
        for (int i = 0; i < populationSize; i++) {
            population[i] = tour(numPlaces);
        }

        for (int generation = 0; generation < numGenerations; generation++) {
         //find fitness
            double[] fitness = new double[populationSize];
            for (int i = 0; i < populationSize; i++) {
                fitness[i] = fitness(population[i], placeCoordinates);
            }
            if(Objects.equals(type,  b: "1")) {
                //choosing the best one
                int[][] newPopulation = new int[populationSize][numPlaces];
                for (int i = 0; i < populationSize; i++) {
                    int parent1 = parent(fitness);
                    int parent2 = parent(fitness);
                    int[] child = crossover(population[parent1], population[parent2]);
                    mutation(child, mutationRate);
                    newPopulation[i] = child;
                }
```

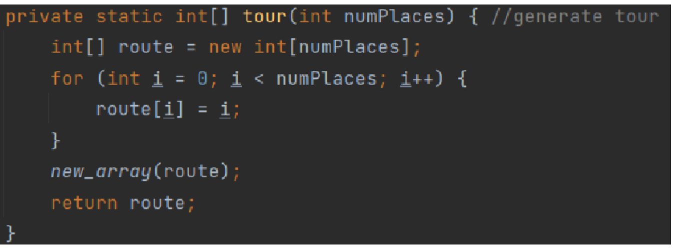


```
        //best solution
        double bestDistance = Double.MAX_VALUE;
        int bestRouteIndex = -1;
        for (int i = 0; i < populationSize; i++) {
            double distance = fitness(population[i], placeCoordinates);
            if (distance < bestDistance) {
                bestDistance = distance;
                bestRouteIndex = i;
            }
        }
```

tour: This function generates a random initial route by creating a permutation of cities to visit. It ensures that each city is visited exactly once.



```
private static void new_array(int[] array)//shuffle
{
    Random random = new Random();
    for (int i = array.length - 1; i > 0; i--)
    {
        int id = random.nextInt( bound: i + 1);
        int temp = array[id];
        array[id] = array[i];
        array[i] = temp;
    }
}
```

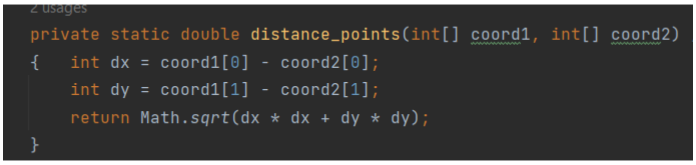


```
private static int[] tour(int numPlaces) { //generate tour
    int[] route = new int[numPlaces];
    for (int i = 0; i < numPlaces; i++) {
        route[i] = i;
    }
    new_array(route);
    return route;
}
```

new_array: This function shuffles an array randomly. It's used to shuffle the initial route
fitness: Given a route and place coordinates, this function calculates the total distance traveled in the route by summing the Euclidean distances between consecutive cities.

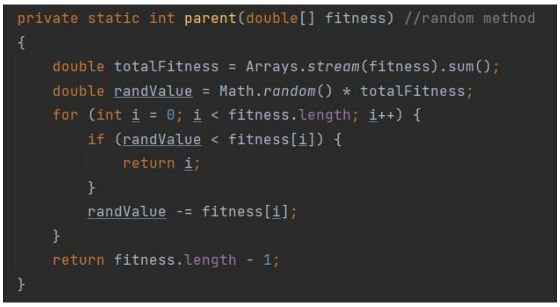
Alexandra Vetrov ID: 123102

distance_points: Calculates the Euclidean distance between two points, represented as coordinate pairs.

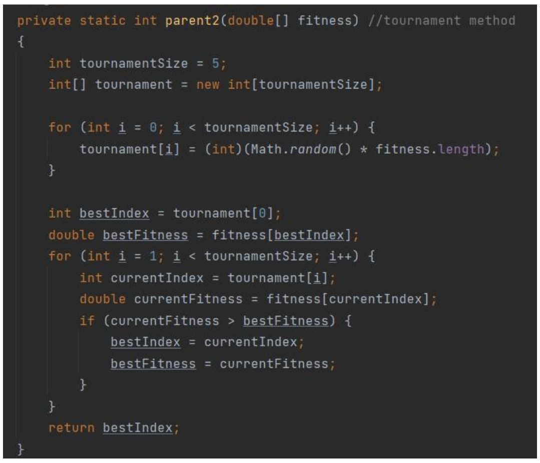


```
2 usages
private static double distance_points(int[] coord1, int[] coord2)
{   int dx = coord1[0] - coord2[0];
    int dy = coord1[1] - coord2[1];
    return Math.sqrt(dx * dx + dy * dy);
}
```

parent: Selects parents for crossover using a random selection method. The probability of selection is proportional to the fitness of each solution.



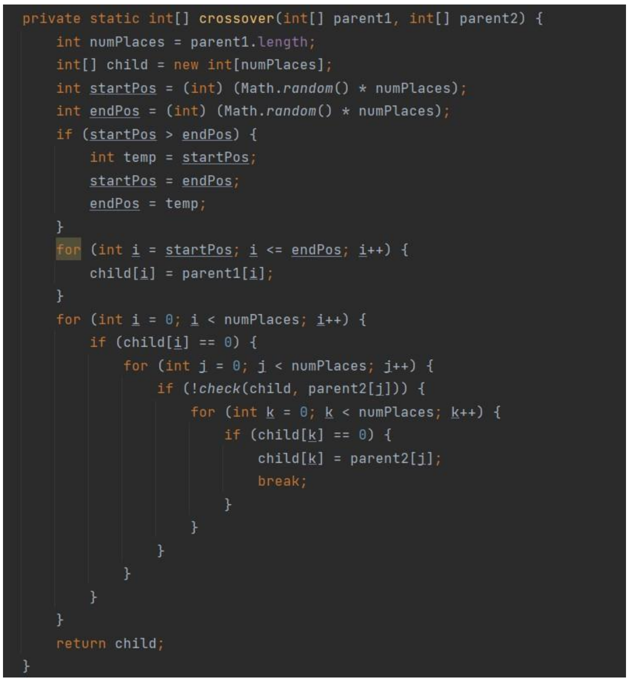
```
private static int parent(double[] fitness) //random method
{
    double totalFitness = Arrays.stream(fitness).sum();
    double randValue = Math.random() * totalFitness;
    for (int i = 0; i < fitness.length; i++) {
        if (randValue < fitness[i]) {
            return i;
        }
        randValue -= fitness[i];
    }
    return fitness.length - 1;
}
```



```
private static int parent2(double[] fitness) //tournament method
{
    int tournamentSize = 5;
    int[] tournament = new int[tournamentSize];

    for (int i = 0; i < tournamentSize; i++) {
        tournament[i] = (int)(Math.random() * fitness.length);
    }

    int bestIndex = tournament[0];
    double bestFitness = fitness[bestIndex];
    for (int i = 1; i < tournamentSize; i++) {
        int currentIndex = tournament[i];
        double currentFitness = fitness[currentIndex];
        if (currentFitness > bestFitness) {
            bestIndex = currentIndex;
            bestFitness = currentFitness;
        }
    }
    return bestIndex;
}
```

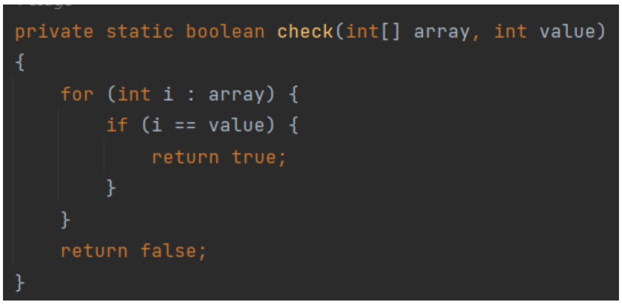
parent2: Selects parents for crossover using a tournament selection method. It randomly selects a group of individuals, evaluates their fitness, and chooses the one with the highest fitness.

crossover: Combines two parent routes to create a child route. A random subsequence is selected from each parent, and the child route is created by combining these subsequences.



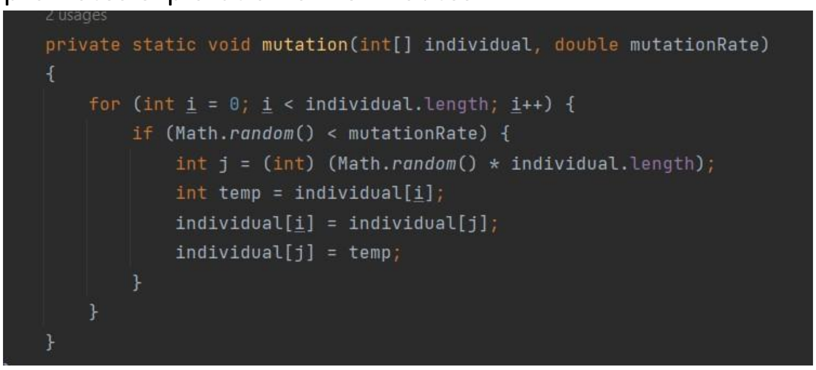
```
private static int[] crossover(int[] parent1, int[] parent2) {
    int numPlaces = parent1.length;
    int[] child = new int[numPlaces];
    int startPos = (int) (Math.random() * numPlaces);
    int endPos = (int) (Math.random() * numPlaces);
    if (startPos > endPos) {
        int temp = startPos;
        startPos = endPos;
        endPos = temp;
    }
    for (int i = startPos; i <= endPos; i++) {
        child[i] = parent1[i];
    }
    for (int i = 0; i < numPlaces; i++) {
        if (child[i] == 0) {
            for (int j = 0; j < numPlaces; j++) {
                if (!check(child, parent2[j])) {
                    for (int k = 0; k < numPlaces; k++) {
                        if (child[k] == 0) {
                            child[k] = parent2[j];
                            break;
                        }
                    }
                }
            }
        }
    }
    return child;
}
```

check: Checks if a specific element exists in an array. It's used to ensure that a city is not included multiple times in a route.



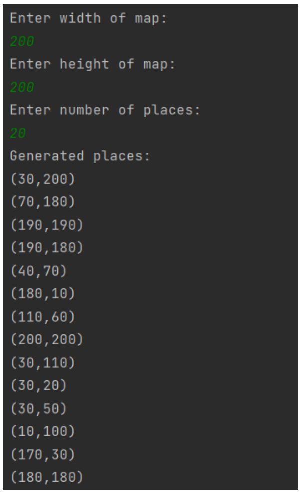
```java
private static boolean check(int[] array, int value)
{
    for (int i : array) {
        if (i == value) {
            return true;
        }
    }
    return false;
}
```

mutation: Introduces random changes to an individual route with a given mutation rate. This promotes exploration of new routes.

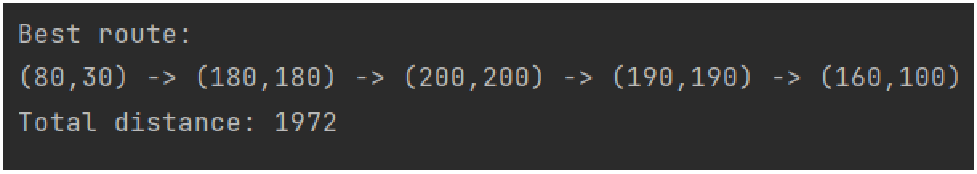


```java
2 usages
private static void mutation(int[] individual, double mutationRate)
{
    for (int i = 0; i < individual.length; i++) {
        if (Math.random() < mutationRate) {
            int j = (int) (Math.random() * individual.length);
            int temp = individual[i];
            individual[i] = individual[j];
            individual[j] = temp;
        }
    }
}
```

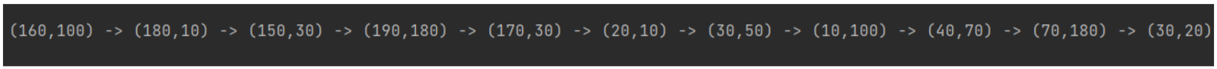
Output for size map 200*200 and 20 places



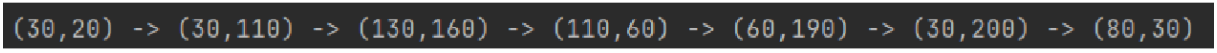
```
Enter width of map:
200
Enter height of map:
200
Enter number of places:
20
Generated places:
(30,200)
(70,180)
(190,190)
(190,180)
(40,70)
(180,10)
(110,60)
(200,200)
(30,110)
(30,20)
(30,50)
(10,100)
(170,30)
(180,180)
```



```
Best route:
(80,30) -> (180,180) -> (200,200) -> (190,190) -> (160,100)
Total distance: 1972
```



```
(160,100) -> (180,10) -> (150,30) -> (190,180) -> (170,30) -> (20,10) -> (30,50) -> (10,100) -> (40,70) -> (70,180) -> (30,20)
```



```
(30,20) -> (30,110) -> (130,160) -> (110,60) -> (60,190) -> (30,200) -> (80,30)
```

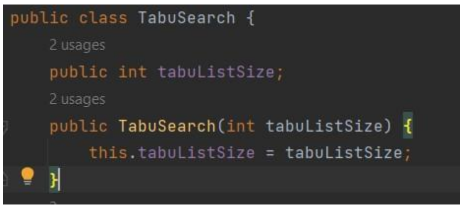
Alexandra Vetrov ID: 123102

Tabu Search

Tabu Search is a metaheuristic and combinatorial optimization technique used to solve complex problems, especially those involving discrete decision variables and large solution spaces. It is particularly effective for tackling problems like the Traveling Salesman Problem (TSP) and various other optimization challenges. Tabu Search is known for its ability to efficiently navigate through complex and rugged solution spaces, and it is widely applied in various fields, including operations research, logistics, engineering, and artificial intelligence.

The tabu list helps avoid revisiting previously explored routes, allowing the algorithm to explore a broader solution space.
Tabu search in my program:

Constructor: The TabuSearch class is initialized with the size of the tabu list. The tabu list keeps track of recently visited routes to avoid revisiting them.



```
public class TabuSearch {
    public int tabuListSize;
    public TabuSearch(int tabuListSize) {
        this.tabuListSize = tabuListSize;
    }
```

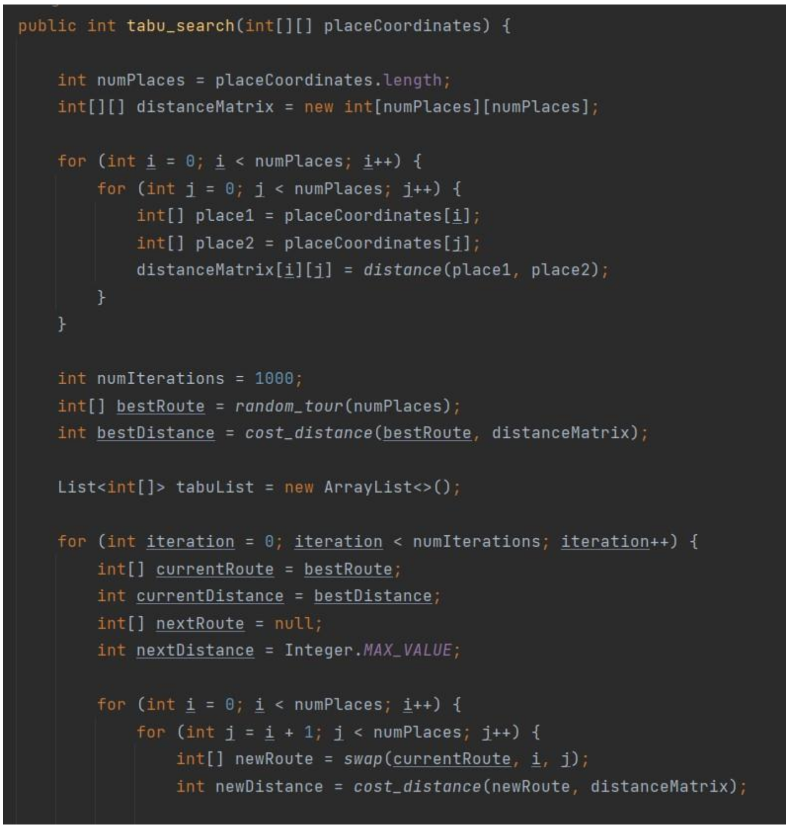
tabu_search function: This is the main function that performs the Tabu Search algorithm. It takes the place coordinates as input and returns the best distance found.

Distance Matrix Initialization: The code computes a distance matrix that stores the distances between all pairs of places based on their coordinates. This matrix is used to calculate the distance between places more efficiently.

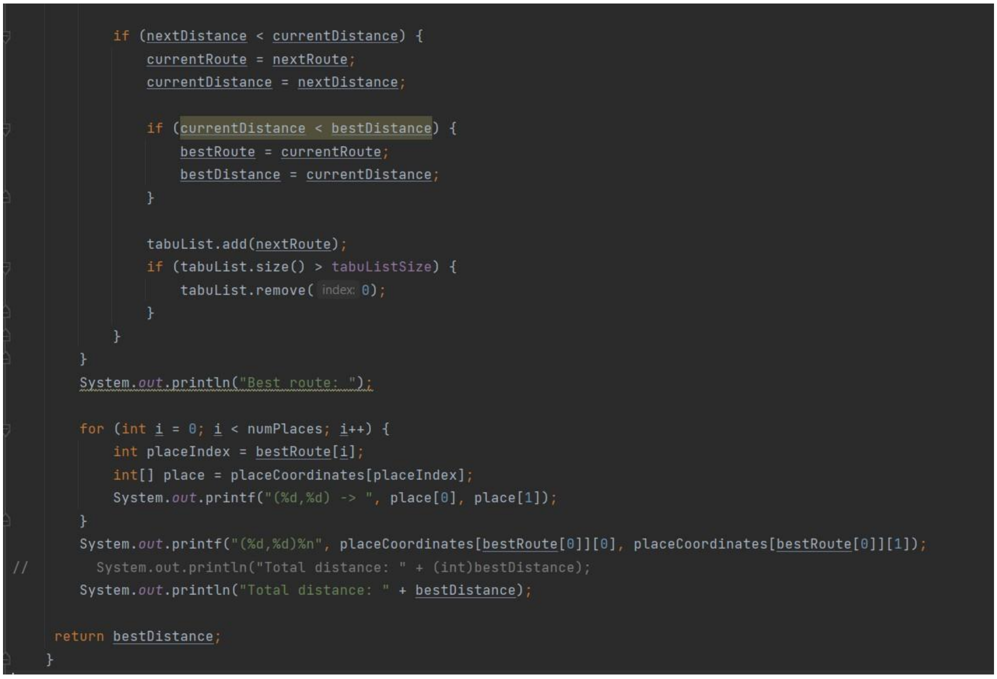
Tabu Search Iteration: The main loop of the Tabu Search runs for a specified number of iterations (numIterations). In each iteration, it attempts to improve the current route.

Current and Next Routes: It maintains the current best route and its distance (currentRoute and currentDistance). It also keeps track of the next potential route (nextRoute) and its distance (nextDistance).

Neighborhood Search: The code explores the neighborhood of the current route by swapping two places and computes the distance of the new route. It identifies the best neighbor (route with the lowest distance) that is not in the tabu list.



```
public int tabu_search(int[][] placeCoordinates) {

    int numPlaces = placeCoordinates.length;
    int[][] distanceMatrix = new int[numPlaces][numPlaces];

    for (int i = 0; i < numPlaces; i++) {
        for (int j = 0; j < numPlaces; j++) {
            int[] place1 = placeCoordinates[i];
            int[] place2 = placeCoordinates[j];
            distanceMatrix[i][j] = distance(place1, place2);
        }
    }

    int numIterations = 1000;
    int[] bestRoute = random_tour(numPlaces);
    int bestDistance = cost_distance(bestRoute, distanceMatrix);

    List<int[]> tabuList = new ArrayList<>();

    for (int iteration = 0; iteration < numIterations; iteration++) {
        int[] currentRoute = bestRoute;
        int currentDistance = bestDistance;
        int[] nextRoute = null;
        int nextDistance = Integer.MAX_VALUE;

        for (int i = 0; i < numPlaces; i++) {
            for (int j = i + 1; j < numPlaces; j++) {
                int[] newRoute = swap(currentRoute, i, j);
                int newDistance = cost_distance(newRoute, distanceMatrix);
```



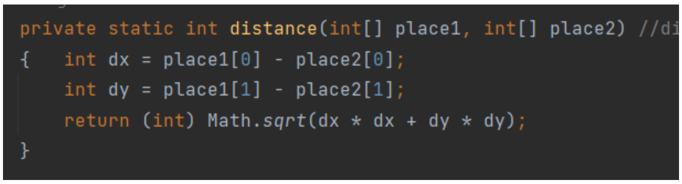
```
        if (nextDistance < currentDistance) {
            currentRoute = nextRoute;
            currentDistance = nextDistance;

            if (currentDistance < bestDistance) {
                bestRoute = currentRoute;
                bestDistance = currentDistance;
            }

            tabuList.add(nextRoute);
            if (tabuList.size() > tabuListSize) {
                tabuList.remove( index: 0);
            }
        }
    }
    System.out.println("Best route: ");

    for (int i = 0; i < numPlaces; i++) {
        int placeIndex = bestRoute[i];
        int[] place = placeCoordinates[placeIndex];
        System.out.printf("(%d,%d) -> ", place[0], place[1]);
    }
    System.out.printf("(%d,%d)%n", placeCoordinates[bestRoute[0]][0], placeCoordinates[bestRoute[0]][1]);
//      System.out.println("Total distance: " + (int)bestDistance);
    System.out.println("Total distance: " + bestDistance);

 return bestDistance;
}
```

Tabu List Management: The code adds the best neighbor to the tabu list and removes the oldest entry if the tabu list size exceeds the specified limit (tabuListSize).

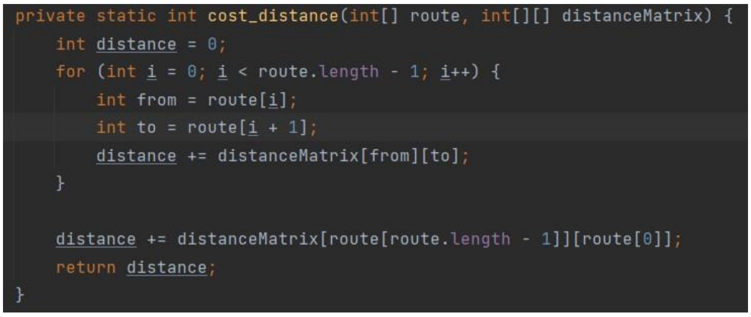
Updating the Best Route: If the distance of the best neighbor is less than the current best distance, the current route is updated, and the best route and best distance are updated accordingly.

Alexandra Vetrov ID: 123102

Distance Calculation: The code defines a distance function to calculate the Euclidean distance between two places based on their coordinates. The cost_distance function calculates the total distance of a route using the distance matrix.



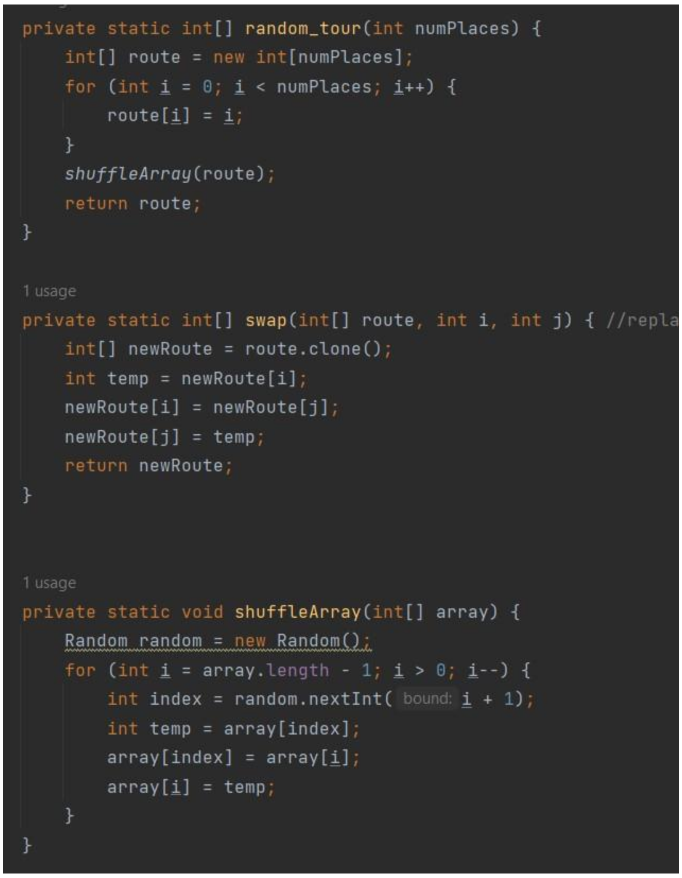
```
private static int distance(int[] place1, int[] place2) //di
{   int dx = place1[0] - place2[0];
    int dy = place1[1] - place2[1];
    return (int) Math.sqrt(dx * dx + dy * dy);
}
```



```
private static int cost_distance(int[] route, int[][] distanceMatrix) {
    int distance = 0;
    for (int i = 0; i < route.length - 1; i++) {
        int from = route[i];
        int to = route[i + 1];
        distance += distanceMatrix[from][to];
    }

    distance += distanceMatrix[route[route.length - 1]][route[0]];
    return distance;
}
```

Random Initial Route: The random_tour function generates a random initial route by shuffling an array of place indices. This route is used as the starting point for the Tabu Search. wap Function: The swap function is used to swap two places in a route.

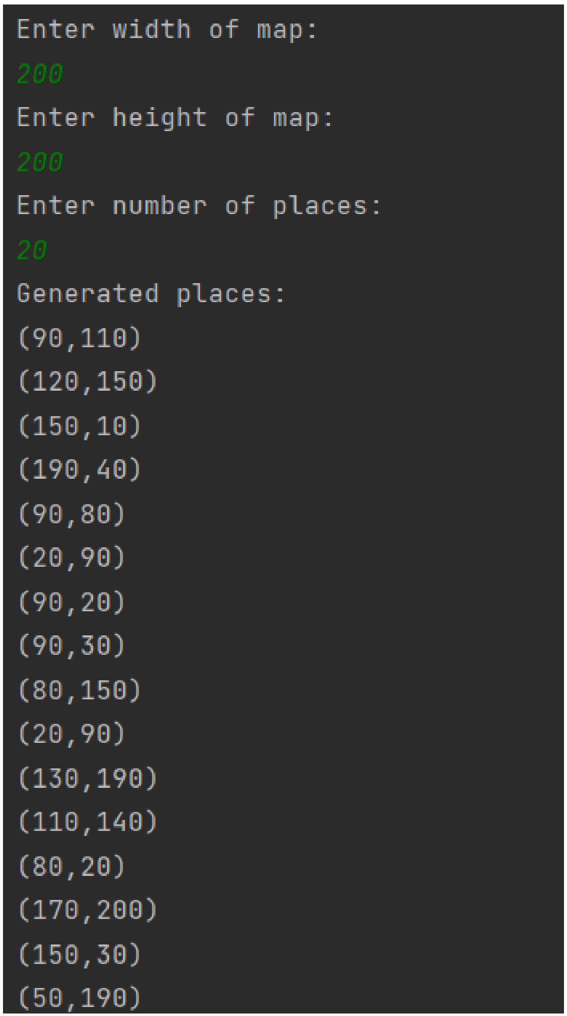
Shuffle Array: The shuffleArray function shuffles an array using the Fisher-Yates shuffle algorithm. It's used to randomize the initial route.



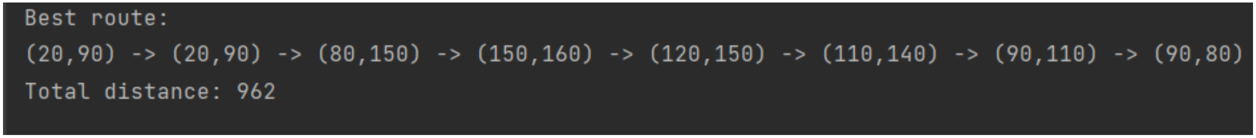
```
private static int[] random_tour(int numPlaces) {
    int[] route = new int[numPlaces];
    for (int i = 0; i < numPlaces; i++) {
        route[i] = i;
    }
    shuffleArray(route);
    return route;
}

1 usage
private static int[] swap(int[] route, int i, int j) { //repla
    int[] newRoute = route.clone();
    int temp = newRoute[i];
    newRoute[i] = newRoute[j];
    newRoute[j] = temp;
    return newRoute;
}


1 usage
private static void shuffleArray(int[] array) {
    Random random = new Random();
    for (int i = array.length - 1; i > 0; i--) {
        int index = random.nextInt( bound: i + 1);
        int temp = array[index];
        array[index] = array[i];
        array[i] = temp;
    }
}
```

Alexandra Vetrov ID: 123102

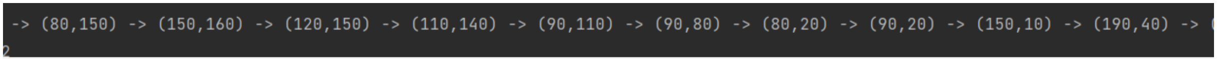
Output:



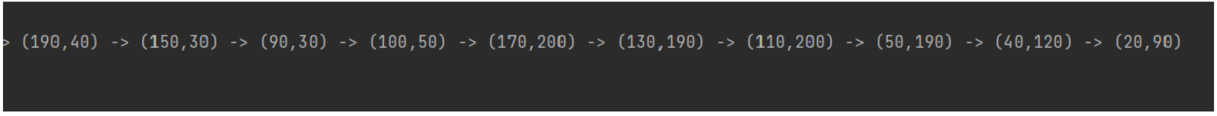
```
Enter width of map:
200
Enter height of map:
200
Enter number of places:
20
Generated places:
(90,110)
(120,150)
(150,10)
(190,40)
(90,80)
(20,90)
(90,20)
(90,30)
(80,150)
(20,90)
(130,190)
(110,140)
(80,20)
(170,200)
(150,30)
(50,190)
```



```
Best route:
(20,90) -> (20,90) -> (80,150) -> (150,160) -> (120,150) -> (110,140) -> (90,110) -> (90,80)
Total distance: 962
```



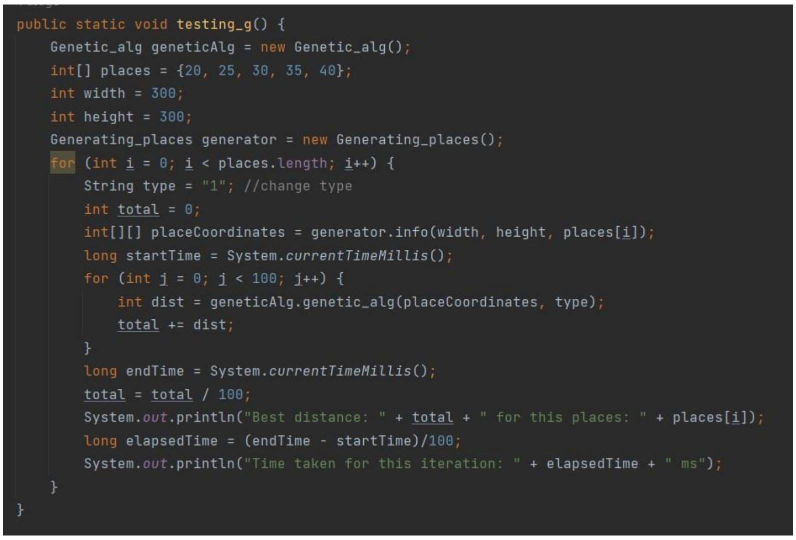
```
 -> (80,150) -> (150,160) -> (120,150) -> (110,140) -> (90,110) -> (90,80) -> (80,20) -> (90,20) -> (150,10) -> (190,40) -> (
2
```



```
> (190,40) -> (150,30) -> (90,30) -> (100,50) -> (170,200) -> (130,190) -> (110,200) -> (50,190) -> (40,120) -> (20,90)
```

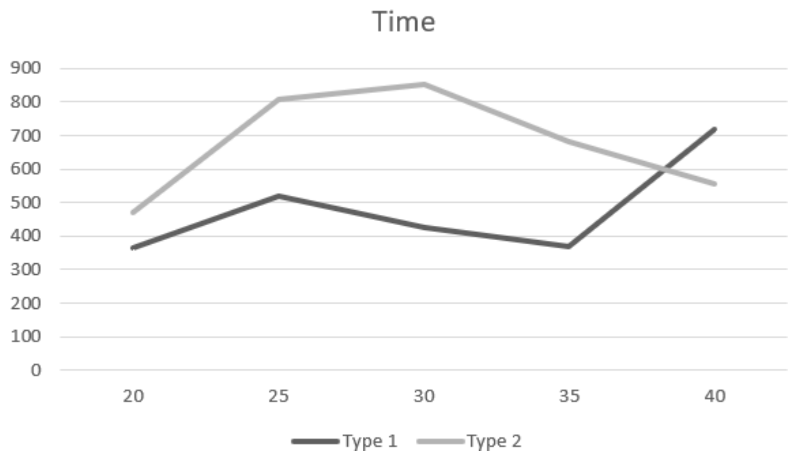
Testing

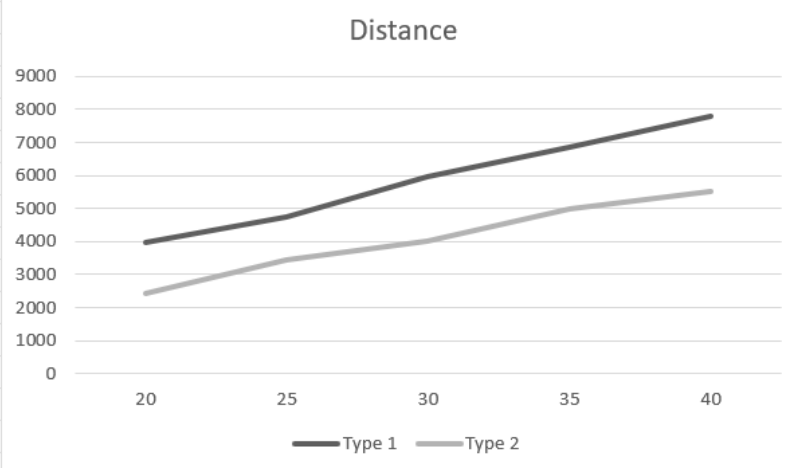
Genetic algorithm



```
public static void testing_g() {
    Genetic_alg geneticAlg = new Genetic_alg();
    int[] places = {20, 25, 30, 35, 40};
    int width = 300;
    int height = 300;
    Generating_places generator = new Generating_places();
    for (int i = 0; i < places.length; i++) {
        String type = "1"; //change type
        int total = 0;
        int[][] placeCoordinates = generator.info(width, height, places[i]);
        long startTime = System.currentTimeMillis();
        for (int j = 0; j < 100; j++) {
            int dist = geneticAlg.genetic_alg(placeCoordinates, type);
            total += dist;
        }
        long endTime = System.currentTimeMillis();
        total = total / 100;
        System.out.println("Best distance: " + total + " for this places: " + places[i]);
        long elapsedTime = (endTime - startTime)/100;
        System.out.println("Time taken for this iteration: " + elapsedTime + " ms");
    }
}
```

To test the algorithm, I compare two methods of population generation: first is the radnom method of selecting the father, the other is the tournament method, the selection of the dominant element

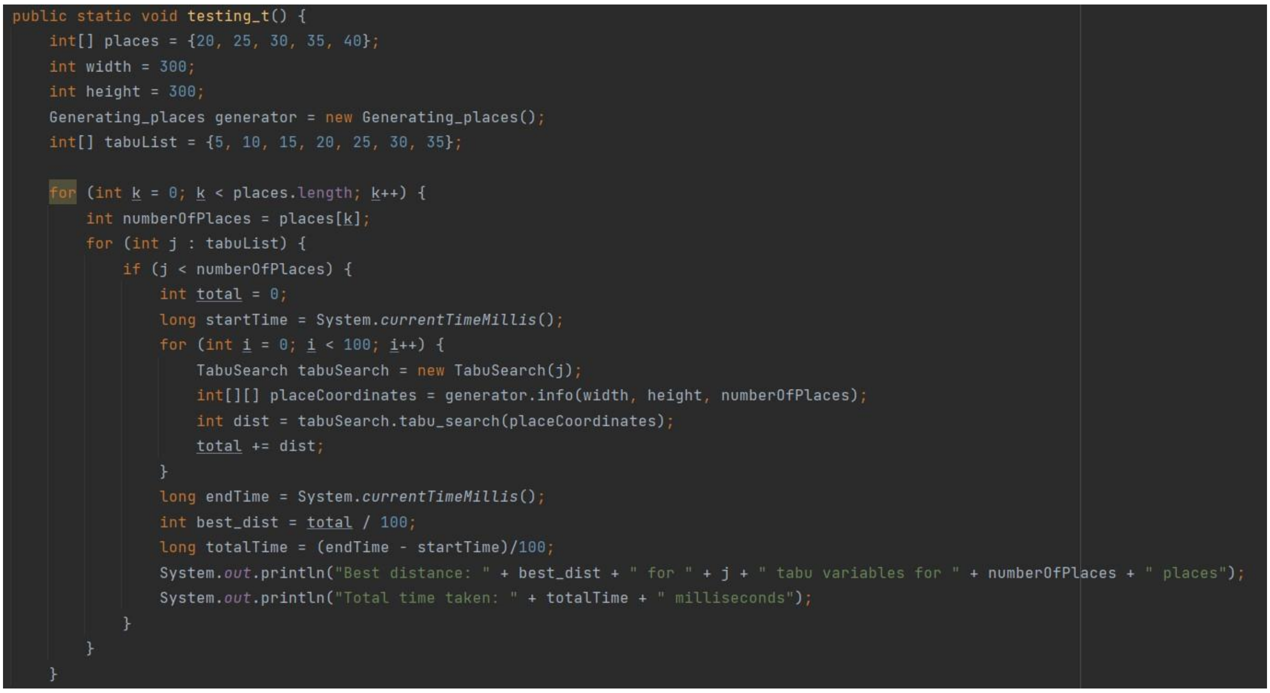




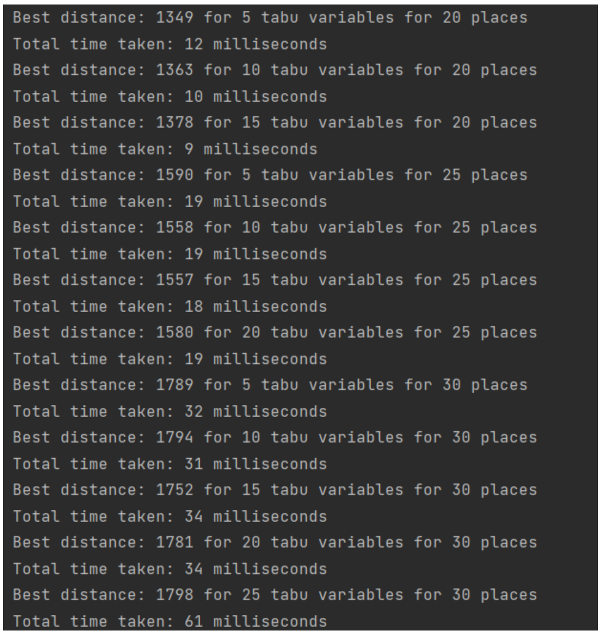
As a result, we see that the first type is executed faster, but from the second graph we see that the average length of the path is greater than type 2, which is executed longer

Tabu Search

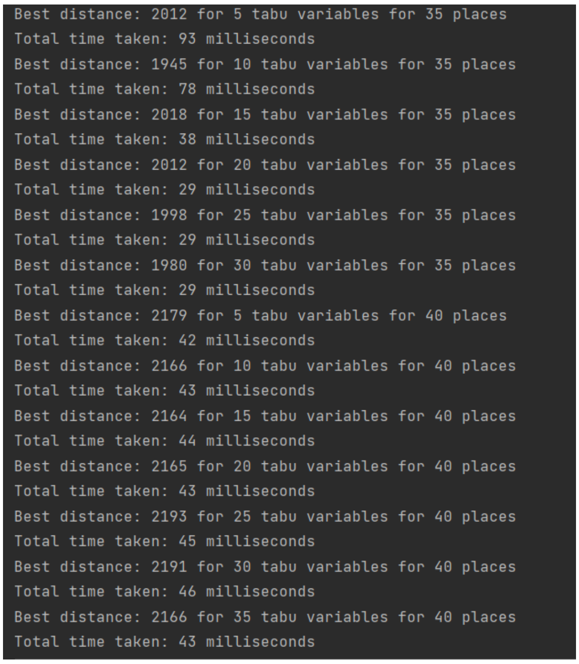
Testing for this algorithm is done in such a way that I have an array of the number of places from 20 to 40, as well as an array with the sizes of the tabu table, and I look at what is the optimal size of the tabu table for this problem according to the optimal path and execution time.



```
public static void testing_t() {
    int[] places = {20, 25, 30, 35, 40};
    int width = 300;
    int height = 300;
    Generating_places generator = new Generating_places();
    int[] tabuList = {5, 10, 15, 20, 25, 30, 35};

    for (int k = 0; k < places.length; k++) {
        int numberOfPlaces = places[k];
        for (int j : tabuList) {
            if (j < numberOfPlaces) {
                int total = 0;
                long startTime = System.currentTimeMillis();
                for (int i = 0; i < 100; i++) {
                    TabuSearch tabuSearch = new TabuSearch(j);
                    int[][] placeCoordinates = generator.info(width, height, numberOfPlaces);
                    int dist = tabuSearch.tabu_search(placeCoordinates);
                    total += dist;
                }
                long endTime = System.currentTimeMillis();
                int best_dist = total / 100;
                long totalTime = (endTime - startTime)/100;
                System.out.println("Best distance: " + best_dist + " for " + j + " tabu variables for " + numberOfPlaces + " places");
                System.out.println("Total time taken: " + totalTime + " milliseconds");
            }
        }
    }
```

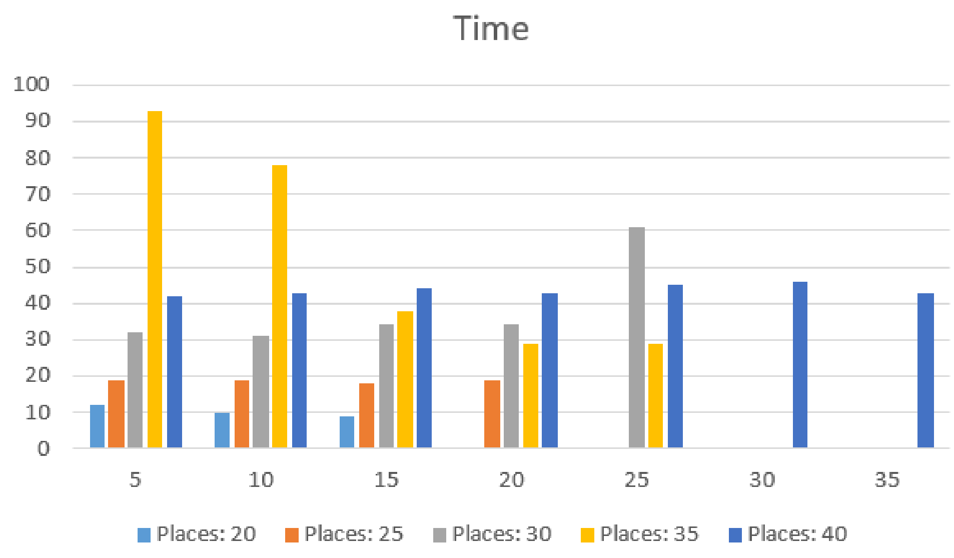


```
Best distance: 1349 for 5 tabu variables for 20 places
Total time taken: 12 milliseconds
Best distance: 1363 for 10 tabu variables for 20 places
Total time taken: 10 milliseconds
Best distance: 1378 for 15 tabu variables for 20 places
Total time taken: 9 milliseconds
Best distance: 1590 for 5 tabu variables for 25 places
Total time taken: 19 milliseconds
Best distance: 1558 for 10 tabu variables for 25 places
Total time taken: 19 milliseconds
Best distance: 1557 for 15 tabu variables for 25 places
Total time taken: 18 milliseconds
Best distance: 1580 for 20 tabu variables for 25 places
Total time taken: 19 milliseconds
Best distance: 1789 for 5 tabu variables for 30 places
Total time taken: 32 milliseconds
Best distance: 1794 for 10 tabu variables for 30 places
Total time taken: 31 milliseconds
Best distance: 1752 for 15 tabu variables for 30 places
Total time taken: 34 milliseconds
Best distance: 1781 for 20 tabu variables for 30 places
Total time taken: 34 milliseconds
Best distance: 1798 for 25 tabu variables for 30 places
Total time taken: 61 milliseconds
```



```
Best distance: 2012 for 5 tabu variables for 35 places
Total time taken: 93 milliseconds
Best distance: 1945 for 10 tabu variables for 35 places
Total time taken: 78 milliseconds
Best distance: 2018 for 15 tabu variables for 35 places
Total time taken: 38 milliseconds
Best distance: 2012 for 20 tabu variables for 35 places
Total time taken: 29 milliseconds
Best distance: 1998 for 25 tabu variables for 35 places
Total time taken: 29 milliseconds
Best distance: 1980 for 30 tabu variables for 35 places
Total time taken: 29 milliseconds
Best distance: 2179 for 5 tabu variables for 40 places
Total time taken: 42 milliseconds
Best distance: 2166 for 10 tabu variables for 40 places
Total time taken: 43 milliseconds
Best distance: 2164 for 15 tabu variables for 40 places
Total time taken: 44 milliseconds
Best distance: 2165 for 20 tabu variables for 40 places
Total time taken: 43 milliseconds
Best distance: 2193 for 25 tabu variables for 40 places
Total time taken: 45 milliseconds
Best distance: 2191 for 30 tabu variables for 40 places
Total time taken: 46 milliseconds
Best distance: 2166 for 35 tabu variables for 40 places
Total time taken: 43 milliseconds
```

Alexandra Vetrov ID: 123102



Best size of tabu table for solving problem with 20 places is 15, for 25 is 20, for 30 places is 10,for 35 is 20 and for 40 is 20.