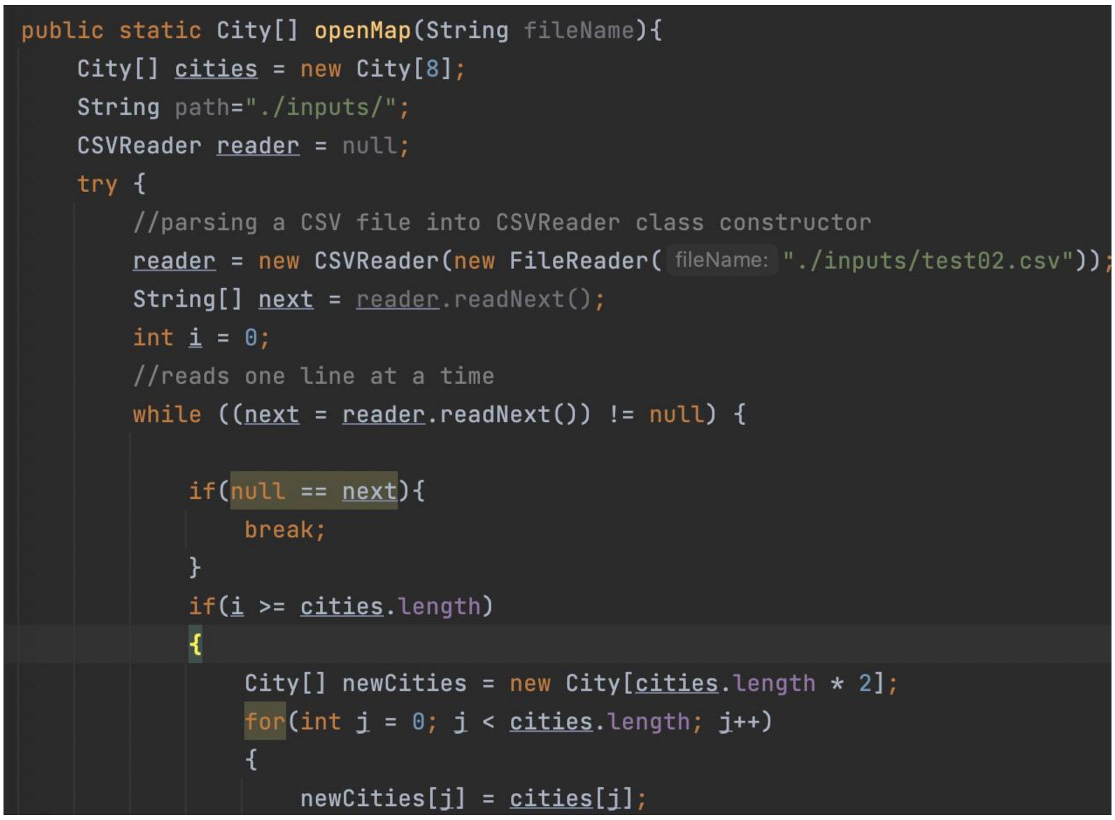
REPORT

#1.Introduction

THE TRAVELING SALESMAN PROBLEM
• The Traveling Salesman Problem (TSP) is an NP-hard algorithm for finding the shortest tour of n cities (or, in general, points in a two-dimensional space).
• NP-hard means that deterministic solutions have complexity O(K^n).
• Your solution will be based on the Christofides algorithm
• Optimization will utilize any/all of the following methods:
• Random swapping, 2-opt and/or 3-opt improvement, simulated annealing, ant colony optimization, genetic algorithms, etc.

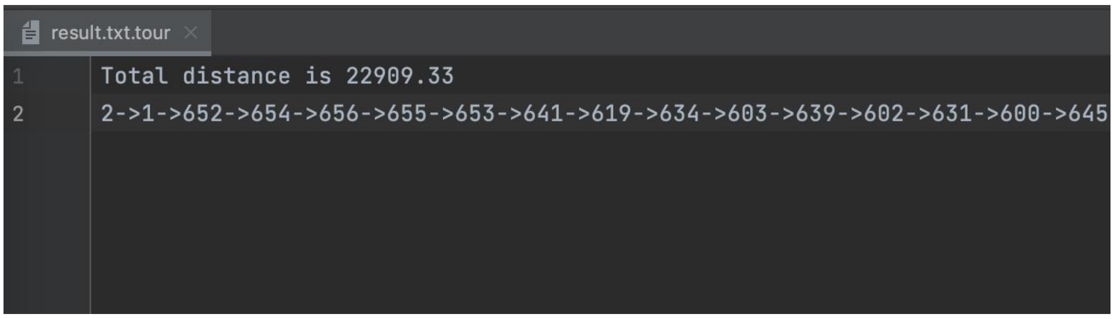
#2.Conclusion



```java
public static City[] openMap(String fileName){
    City[] cities = new City[8];
    String path="./inputs/";
    CSVReader reader = null;
    try {
        //parsing a CSV file into CSVReader class constructor
        reader = new CSVReader(new FileReader( fileName: "./inputs/test02.csv"));
        String[] next = reader.readNext();
        int i = 0;
        //reads one line at a time
        while ((next = reader.readNext()) != null) {

            if(null == next){
                break;
            }
            if(i >= cities.length)
            {
                City[] newCities = new City[cities.length * 2];
                for(int j = 0; j < cities.length; j++)
                {
                    newCities[j] = cities[j];
```

Pic 2.1

We use openCSV to deal with csv file and add data into City class. Only 674 data have coordinates, we will number all the data, the id of the first data is 0, and the id of the last data is 673. The final route result will show the id of the location.

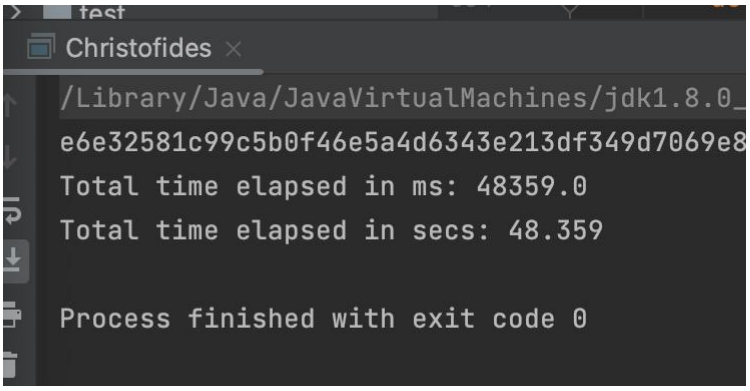


```
result.txt.tour
Total distance is 22909.33
2->1->652->654->656->655->653->641->619->634->603->639->602->631->600->645
```

Pic 2.2

As we can see in result.txt.tour, the total distance is 22909.33 meters
And the tour is below:
2->1->652->654->656->655->653->641->619->634->603->639->602->631->600->645->657->658->573->636->646->643->630->609->622->597->624->581->599->644->7->23->18->21->22->587->606->14->11->9->17->10->13->15->12->20->16->19->435->405->282->477->171->185->449->443->441->439->378->372->304->363->431->427->424->418->417->416->499->476->472->471->409->392->470->455->321->452->146->459->450->388->436->434->430->422->411->396->132->538->458->460->180->243->465->283->286->199->277->301->278->325->224->292->269->433->444->158->287->265->320->253->565->508->497->297->163->246->556->408->423->167->550->379->285->165->549->344->284->254->413->445->382->244->375->149->49->39->46->55->25->53->54->48->38->42->34->24->27->43->198->432->467->311->240->316->337->451->312->534->389->429->160->544->247->336->210->279->448->488->206->211->494->195->189->257->421->466->271->659->41->36->28->52->44->57->26->29->47->30->50->35->51->37->56->33->32->45->62->31->40->63->60->59->61->8->58->662->660->661->648->665->666->203->651->650->649->514->509->362->349->272->535->425->464->551->326->208->177->315->360->331->155->250->366->517->521->290->569->475->130->560->182->561->193->563->209->564->216->503->568->222->238->335->377->383->394->402->136->288->137->554->356->358->139->567->323->374->376->510->523->384->387->395->398->140->118->572->317->175->186->319->428->361->352->350->347->346->332->263->496->461->506->524->446->457->527->504->397->245->454->437->373->232->129->559->190->226->116->546->256->403->463->469->322->138->420->268->197->217->380->135->391->176->134->241->273->142->536->119->480->540->123->124->481->483->489->555->162->490->227->512->307->117->147->169->183->381->385->386->390->393->242->280->502->89->99->70->115->75->73->92->74->98->77->85->65->100->105->76->103->109->667->668->88->83->111->107->79->64->87->97->66->110->108->96->91->95->86->94->81->80->78->93->82->113->114->72->67->90->69->106->102->104->101->84->68->112->71->671->670->669->296->299->145->187->518->294->128->511->516->235->196->219->338->541->127->252->345->234->261->664->557->295->213->179->533->314->249->259->482->125->260->205->168->309->513->414->515->528-

>298->479->478->485->505->236->419->293->215->484->473->120->369->351->300->221->406->225->270->303->166->327->266->404->412->537->157->415->495->407->371->202->400->212->410->310->359->313->258->291->289->571->354->526->474->531->126->220->370->353->529->233->248->172->532->306->181->440->267->184->318->188->218->522->276->148->201->558->342->230->343->426->305->438->462->525->121->255->275->329->133->340->231->156->302->520->229->334->173->131->141->545->543->364->368->170->308->507->456->191->194->663->468->348->264->333->492->493->487->486->491->214->151->399->159->367->161->530->192->174->207->539->365->178->328->324->200->547->204->548->339->447->498->453->500->501->154->164->330->552->144->553->223->150->153->281->237->615->614->593->592->584->586->642->623->608->601->613->638->262->542->239->576->519->251->228->357->143->341->401->562->122->570->355->566->274->152->442->578->598->596->579->590->588->617->610->604->594->618->627->582->637->0->3->632->647->633->595->583->612->577->591->580->575->629->621->628->673->672->607->620->574->585->616->640->626->625->589->605->635->611->4->5->6



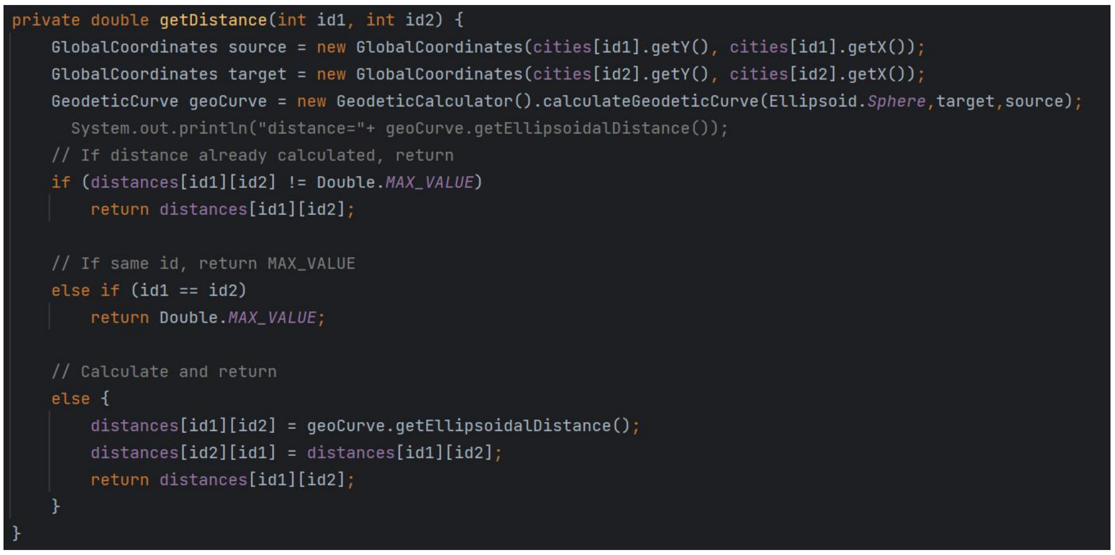
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_
e6e32581c99c5b0f46e5a4d6343e213df349d7069e8
Total time elapsed in ms: 48359.0
Total time elapsed in secs: 48.359

Process finished with exit code 0
```

Pic 2.3

The elapsed time is 48 seconds.

#3.Methods

3.1 Calculate the distance between points



```java
private double getDistance(int id1, int id2) {
    GlobalCoordinates source = new GlobalCoordinates(cities[id1].getY(), cities[id1].getX());
    GlobalCoordinates target = new GlobalCoordinates(cities[id2].getY(), cities[id2].getX());
    GeodeticCurve geoCurve = new GeodeticCalculator().calculateGeodeticCurve(Ellipsoid.Sphere,target,source);
      System.out.println("distance="+ geoCurve.getEllipsoidalDistance());
    // If distance already calculated, return
    if (distances[id1][id2] != Double.MAX_VALUE)
        return distances[id1][id2];

    // If same id, return MAX_VALUE
    else if (id1 == id2)
        return Double.MAX_VALUE;

    // Calculate and return
    else {
        distances[id1][id2] = geoCurve.getEllipsoidalDistance();
        distances[id2][id1] = distances[id1][id2];
        return distances[id1][id2];
    }
}
```

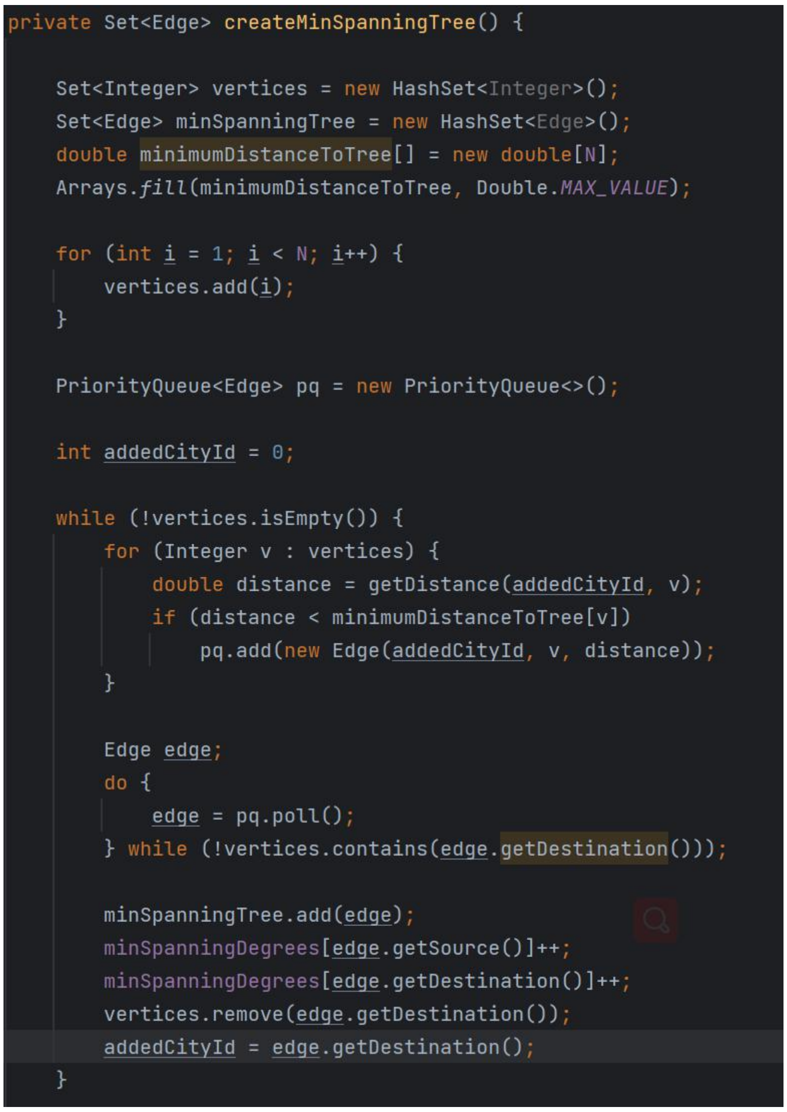
Pic 3.1

As shown in Pic 3.1, Our first step is to calculate the distance between the points. The method getDistance(int id1, int id2) takes two integer parameters , calculates and returns the great-circle distance (in kilometers) between the two cities using their geographic coordinates (latitude and longitude).

The method first creates two GlobalCoordinates objects source and target for the two cities using their latitude and longitude coordinates retrieved from the cities array. It then creates a GeodeticCurve object geoCurve using the GeodeticCalculator class from the geotools library. The geoCurve object is used to calculate the great-circle distance between the two cities using the calculateGeodeticCurve method.

The method then checks if the distance between the two cities has already been calculated and stored in the distances array. If so, it returns the precalculated distance. If the two cities have the same index, it returns the maximum possible value of Double. Otherwise, it calculates and returns the distance between the two cities using the geoCurve object and stores it in the distances array for future use.

3.2 Create the graph and the minimum spanning tree



```java
private Set<Edge> createMinSpanningTree() {

    Set<Integer> vertices = new HashSet<Integer>();
    Set<Edge> minSpanningTree = new HashSet<Edge>();
    double minimumDistanceToTree[] = new double[N];
    Arrays.fill(minimumDistanceToTree, Double.MAX_VALUE);

    for (int i = 1; i < N; i++) {
        vertices.add(i);
    }

    PriorityQueue<Edge> pq = new PriorityQueue<>();

    int addedCityId = 0;

    while (!vertices.isEmpty()) {
        for (Integer v : vertices) {
            double distance = getDistance(addedCityId, v);
            if (distance < minimumDistanceToTree[v])
                pq.add(new Edge(addedCityId, v, distance));
        }

        Edge edge;
        do {
            edge = pq.poll();
        } while (!vertices.contains(edge.getDestination()));

        minSpanningTree.add(edge);
        minSpanningDegrees[edge.getSource()]++;
        minSpanningDegrees[edge.getDestination()]++;
        vertices.remove(edge.getDestination());
        addedCityId = edge.getDestination();
    }
```

Pic 3.2

As shown in the Pic 3.2, we are trying to create the min_spanning_tree. This code defines a private method called createMinSpanningTree that calculates the minimum spanning tree (MST) of a graph represented by a set of vertices and edges. The method returns a Set of edges that make up the minimum spanning tree.

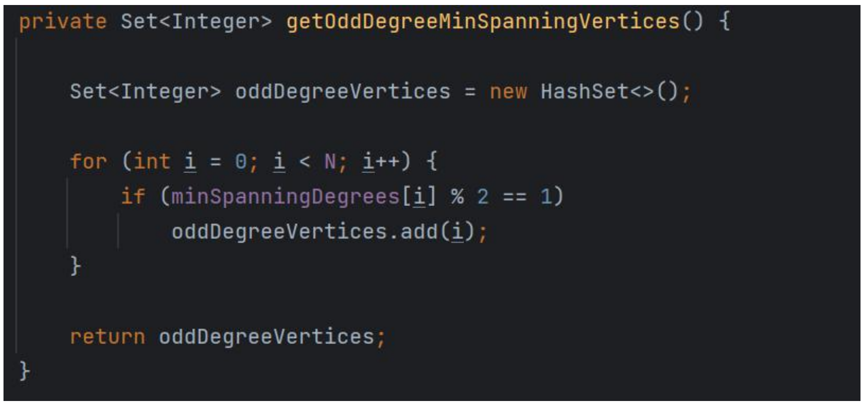
We are going to make a graph first. The method starts by creating a set of vertices vertices and a set of edges minSpanningTree, both initially empty. It also creates an array minimumDistanceToTree of double values of length N, where N is the number of vertices in the graph, and sets all elements to Double.MAX_VALUE.

The method then adds all vertex indices except the first one to the vertices set. It creates a PriorityQueue pq to hold edges and sets the ID of the first city to 0.

According to the reference website(https://en.wikipedia.org/wiki/Christofides_algorithm), the second step is to create the minimum spanning tree from the graph.

The method then enters a loop that iterates until all vertices have been added to the minimum spanning tree. Within the loop, the method calculates the distance between the recently added vertex and each vertex in the vertices set using the getDistance method, and adds the edge connecting the two vertices to the priority queue pq if the distance is less than the current minimum distance to that vertex. It then removes edges from pq until it finds the minimum-weight edge that reaches a vertex outside of the minimum spanning tree. This edge is then added to minSpanningTree, and the degree of both vertices in the edge is incremented in the minSpanningDegrees array. The destination vertex of the edge is then removed from the vertices set, and the ID of the most recently added city is updated to the destination of the newly added edge.

3.3 Get Odd Degree



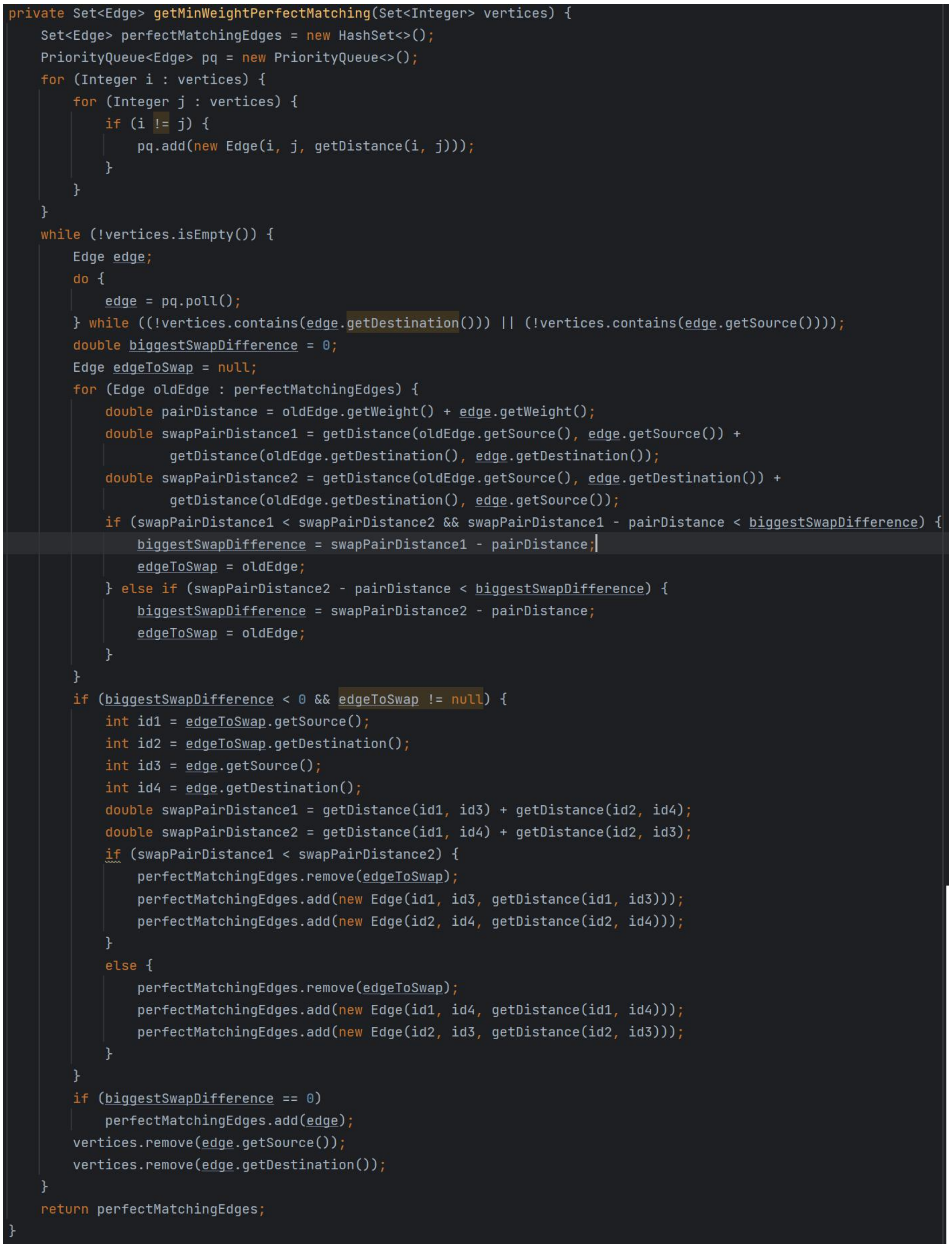
```java
private Set<Integer> getOddDegreeMinSpanningVertices() {

    Set<Integer> oddDegreeVertices = new HashSet<>();

    for (int i = 0; i < N; i++) {
        if (minSpanningDegrees[i] % 2 == 1)
            oddDegreeVertices.add(i);
    }

    return oddDegreeVertices;
}
```

Pic 3.3

This code defines a private method called getOddDegreeMinSpanningVertices that returns a Set of Integer values representing the vertices in a minimum spanning tree that have odd degrees.

The method starts by creating an empty HashSet called oddDegreeVertices to store the odd-degree vertices. It then enters a loop that iterates through all the vertices in the graph, which are numbered from 0 to N-1. For each vertex, the method checks if its degree in the minimum spanning tree is odd by calculating minSpanningDegrees[i] % 2. If the result is 1, the vertex ID is added to the oddDegreeVertices set. Finally, the method returns the set of odd-degree vertices.

3.4 Construct a minimum-weight perfect matching



```java
private Set<Edge> getMinWeightPerfectMatching(Set<Integer> vertices) {
    Set<Edge> perfectMatchingEdges = new HashSet<>();
    PriorityQueue<Edge> pq = new PriorityQueue<>();
    for (Integer i : vertices) {
        for (Integer j : vertices) {
            if (i != j) {
                pq.add(new Edge(i, j, getDistance(i, j)));
            }
        }
    }
    while (!vertices.isEmpty()) {
        Edge edge;
        do {
            edge = pq.poll();
        } while ((!vertices.contains(edge.getDestination())) || (!vertices.contains(edge.getSource())));
        double biggestSwapDifference = 0;
        Edge edgeToSwap = null;
        for (Edge oldEdge : perfectMatchingEdges) {
            double pairDistance = oldEdge.getWeight() + edge.getWeight();
            double swapPairDistance1 = getDistance(oldEdge.getSource(), edge.getSource()) +
                    getDistance(oldEdge.getDestination(), edge.getDestination());
            double swapPairDistance2 = getDistance(oldEdge.getSource(), edge.getDestination()) +
                    getDistance(oldEdge.getDestination(), edge.getSource());
            if (swapPairDistance1 < swapPairDistance2 && swapPairDistance1 - pairDistance < biggestSwapDifference) {
                biggestSwapDifference = swapPairDistance1 - pairDistance;
                edgeToSwap = oldEdge;
            } else if (swapPairDistance2 - pairDistance < biggestSwapDifference) {
                biggestSwapDifference = swapPairDistance2 - pairDistance;
                edgeToSwap = oldEdge;
            }
        }
        if (biggestSwapDifference < 0 && edgeToSwap != null) {
            int id1 = edgeToSwap.getSource();
            int id2 = edgeToSwap.getDestination();
            int id3 = edge.getSource();
            int id4 = edge.getDestination();
            double swapPairDistance1 = getDistance(id1, id3) + getDistance(id2, id4);
            double swapPairDistance2 = getDistance(id1, id4) + getDistance(id2, id3);
            if (swapPairDistance1 < swapPairDistance2) {
                perfectMatchingEdges.remove(edgeToSwap);
                perfectMatchingEdges.add(new Edge(id1, id3, getDistance(id1, id3)));
                perfectMatchingEdges.add(new Edge(id2, id4, getDistance(id2, id4)));
            }
            else {
                perfectMatchingEdges.remove(edgeToSwap);
                perfectMatchingEdges.add(new Edge(id1, id4, getDistance(id1, id4)));
                perfectMatchingEdges.add(new Edge(id2, id3, getDistance(id2, id3)));
            }
        }
        if (biggestSwapDifference == 0)
            perfectMatchingEdges.add(edge);
        vertices.remove(edge.getSource());
        vertices.remove(edge.getDestination());
    }
    return perfectMatchingEdges;
}
```

Pic 3.4

As shown in Pic 3.4, this code defines a method getMinWeightPerfectMatching that takes a set of integers representing vertices in a graph and returns a set of edges representing a minimum-weight perfect matching for those vertices.

The method uses a modified version of the Hungarian algorithm to find the matching. It first creates a priority queue of all possible edges between the vertices, sorted by weight in ascending order. It then iterates over the priority queue, selecting the lowest-weight edge that connects two vertices that have not yet been matched.

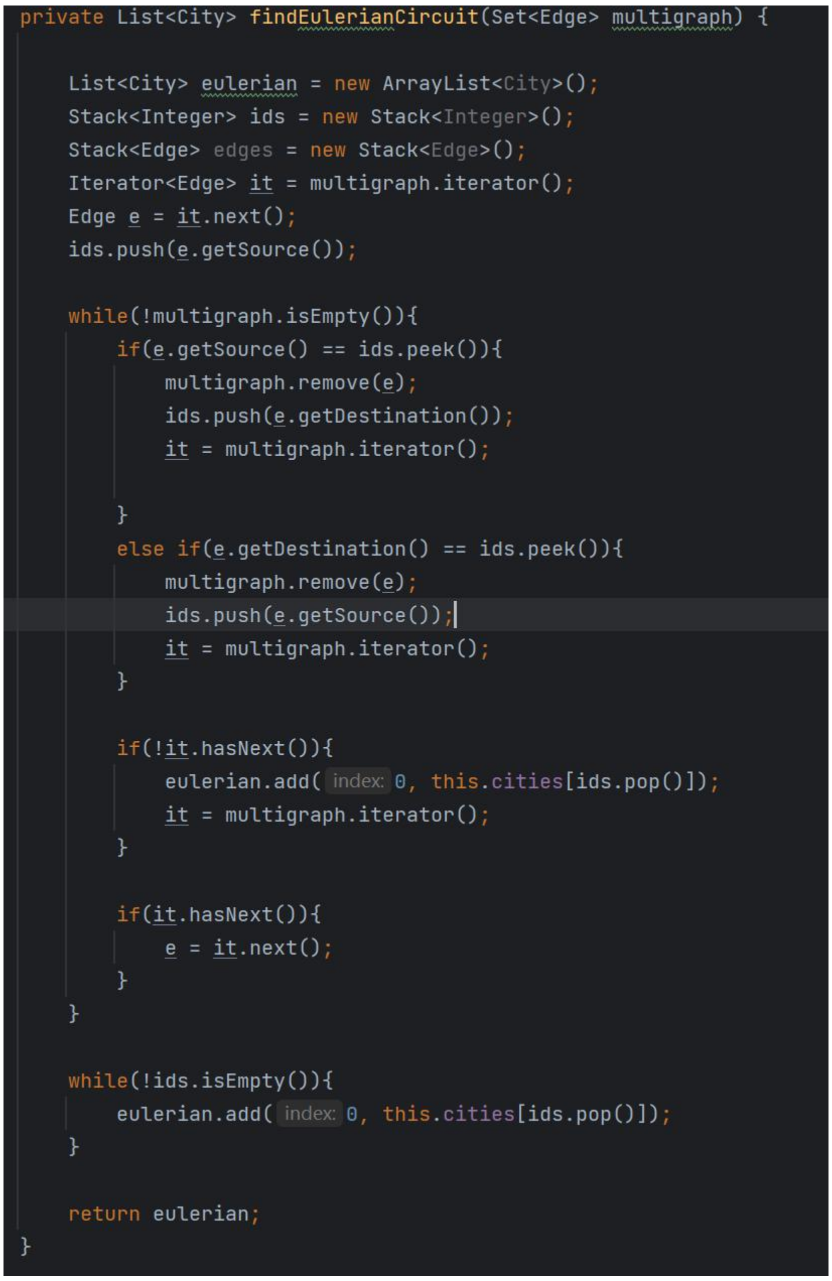
For each selected edge, the method checks if there is a previously selected edge that can be swapped with it to create a better matching. If so, it swaps the two edges in the matching set.

If there is no swap that results in a better matching, the method adds the selected edge to the matching set and removes the two vertices from the set of unmatched vertices.

Once all vertices have been matched, the method returns the set of edges representing the minimum-weight perfect matching.

We used an Edge class to represent edges in the graph, with each edge having a source and destination vertex and a weight. The getDistance method is called to calculate the distance (i.e., weight) between two vertices.

3.5 Construct a minimum-weight perfect matching



```java
private List<City> findEulerianCircuit(Set<Edge> multigraph) {

    List<City> eulerian = new ArrayList<City>();
    Stack<Integer> ids = new Stack<Integer>();
    Stack<Edge> edges = new Stack<Edge>();
    Iterator<Edge> it = multigraph.iterator();
    Edge e = it.next();
    ids.push(e.getSource());

    while(!multigraph.isEmpty()){
        if(e.getSource() == ids.peek()){
            multigraph.remove(e);
            ids.push(e.getDestination());
            it = multigraph.iterator();

        }
        else if(e.getDestination() == ids.peek()){
            multigraph.remove(e);
            ids.push(e.getSource());
            it = multigraph.iterator();
        }

        if(!it.hasNext()){
            eulerian.add( index: 0, this.cities[ids.pop()]);
            it = multigraph.iterator();
        }

        if(it.hasNext()){
            e = it.next();
        }
    }

    while(!ids.isEmpty()){
        eulerian.add( index: 0, this.cities[ids.pop()]);
    }

    return eulerian;
}
```

Pic 3.5

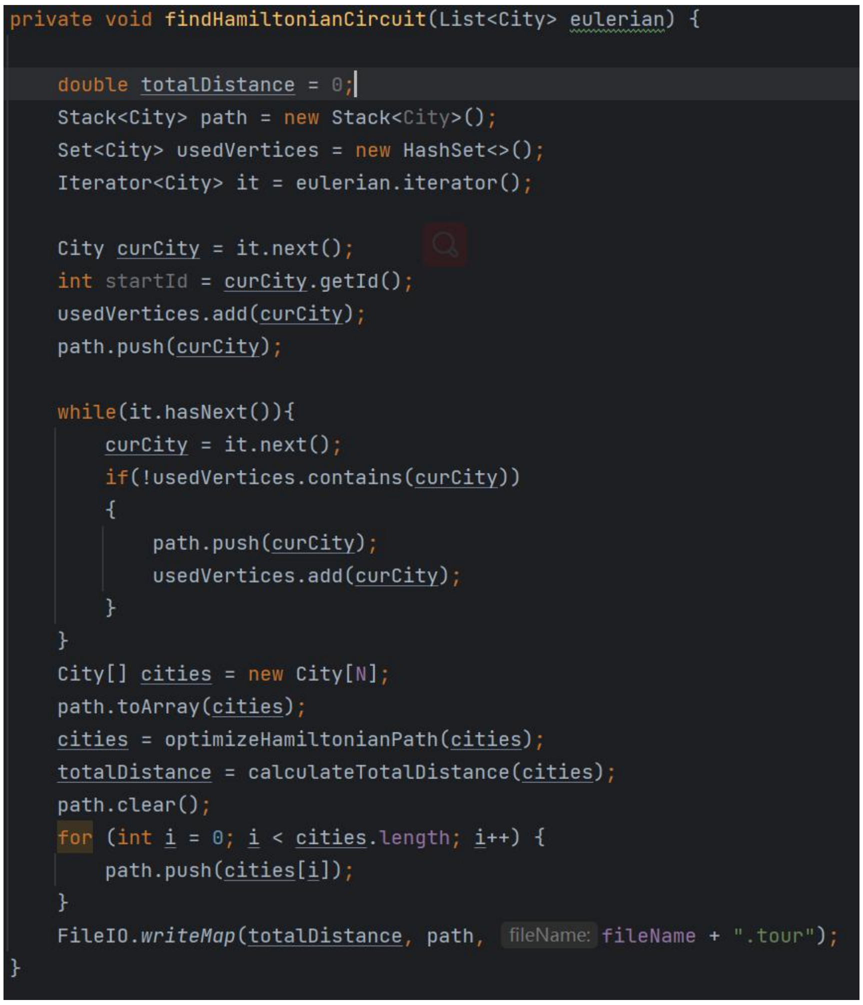
A shown in Pic 2.5, this code is a method that takes a set of edges (multigraph) as input and returns a list of cities (eulerian) that form an Eulerian circuit using the given edges.

Here's a step-by-step explain of what are we doing:

1. The method initializes an empty list eulerian, and two stacks ids and edges.
2. It gets an iterator for the set of edges and retrieves the first edge e.
3. The method pushes the source city of e onto the ids stack.
4. The method enters a loop that runs while the set of edges is not empty.

5. Within the loop, the method checks if the top of the ids stack matches the source or destination city of e.
6. If the top of the stack matches the source or destination city of e, the method removes e from the set of edges, pushes the opposite endpoint onto the ids stack, and resets the iterator it to ensure no null reference.
7. If the top of the stack does not match the source or destination city of e, the method advances to the next edge in the set using the it iterator.
8. If there are no more edges left in the set, the method removes the top city ID from the ids stack and adds the corresponding city to the beginning of the eulerian list.
9. The method updates the current edge e to the next edge in the set, if there is one.
10. The loop continues until there are no more edges left in the set.
11. After the loop, the method pops the remaining city IDs from the ids stack and adds the corresponding cities to the beginning of the eulerian list.
12. Finally, the method returns the eulerian list containing the ordered list of cities forming an Eulerian circuit.

3.6 Find a Hamiltonian Circuit



```java
private void findHamiltonianCircuit(List<City> eulerian) {

    double totalDistance = 0;
    Stack<City> path = new Stack<City>();
    Set<City> usedVertices = new HashSet<>();
    Iterator<City> it = eulerian.iterator();

    City curCity = it.next();
    int startId = curCity.getId();
    usedVertices.add(curCity);
    path.push(curCity);

    while(it.hasNext()){
        curCity = it.next();
        if(!usedVertices.contains(curCity))
        {
            path.push(curCity);
            usedVertices.add(curCity);
        }
    }
    City[] cities = new City[N];
    path.toArray(cities);
    cities = optimizeHamiltonianPath(cities);
    totalDistance = calculateTotalDistance(cities);
    path.clear();
    for (int i = 0; i < cities.length; i++) {
        path.push(cities[i]);
    }
    FileIO.writeMap(totalDistance, path,  fileName: fileName + ".tour");
}
```

Pic 3.6

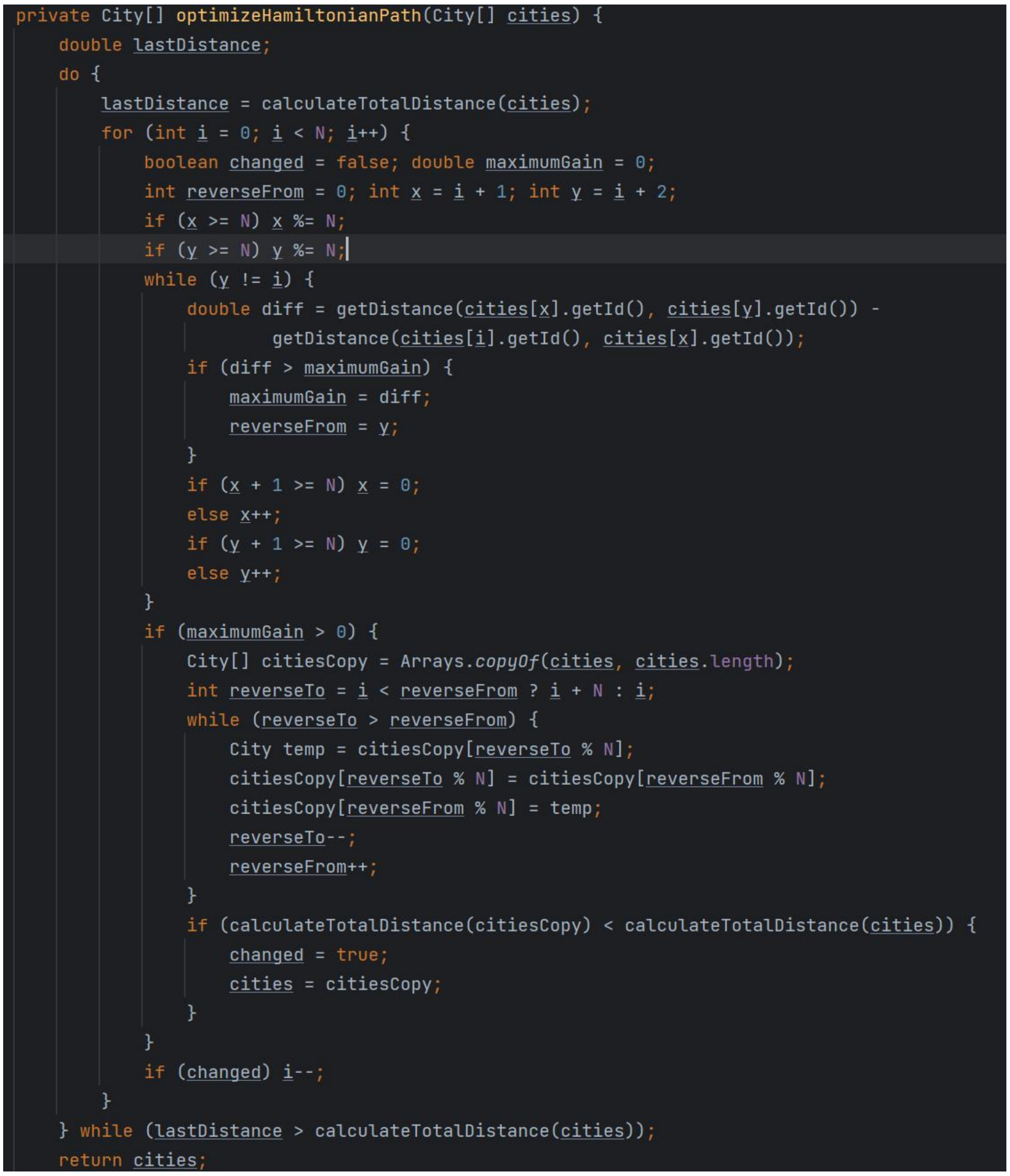
As shown in Pic 2.6, this is a method to find a Hamiltonian circuit given an Eulerian circuit. A Hamiltonian circuit is a path that visits every vertex of a graph exactly once.

The method first initializes some variables to keep track of the total distance, the path taken, and the vertices that have been used. It then iterates through the Eulerian circuit, adding cities to the path and the set of used vertices if they have not already been visited.

After the loop, the path is converted from a stack to an array of cities, and then the optimizeHamiltonianPath method is called to optimize the order of the cities in the path. The total distance is then calculated using the calculateTotalDistance method.

The optimized path is then returned to the stack and written to a file using the writeMap method from the FileIO class.

3.7 Optimizes the Hamiltonian path



```java
private City[] optimizeHamiltonianPath(City[] cities) {
    double lastDistance;
    do {
        lastDistance = calculateTotalDistance(cities);
        for (int i = 0; i < N; i++) {
            boolean changed = false; double maximumGain = 0;
            int reverseFrom = 0; int x = i + 1; int y = i + 2;
            if (x >= N) x %= N;
            if (y >= N) y %= N;
            while (y != i) {
                double diff = getDistance(cities[x].getId(), cities[y].getId()) -
                        getDistance(cities[i].getId(), cities[x].getId());
                if (diff > maximumGain) {
                    maximumGain = diff;
                    reverseFrom = y;
                }
                if (x + 1 >= N) x = 0;
                else x++;
                if (y + 1 >= N) y = 0;
                else y++;
            }
            if (maximumGain > 0) {
                City[] citiesCopy = Arrays.copyOf(cities, cities.length);
                int reverseTo = i < reverseFrom ? i + N : i;
                while (reverseTo > reverseFrom) {
                    City temp = citiesCopy[reverseTo % N];
                    citiesCopy[reverseTo % N] = citiesCopy[reverseFrom % N];
                    citiesCopy[reverseFrom % N] = temp;
                    reverseTo--;
                    reverseFrom++;
                }
                if (calculateTotalDistance(citiesCopy) < calculateTotalDistance(cities)) {
                    changed = true;
                    cities = citiesCopy;
                }
            }
            if (changed) i--;
        }
    } while (lastDistance > calculateTotalDistance(cities));
    return cities;
```

Pic 3.7

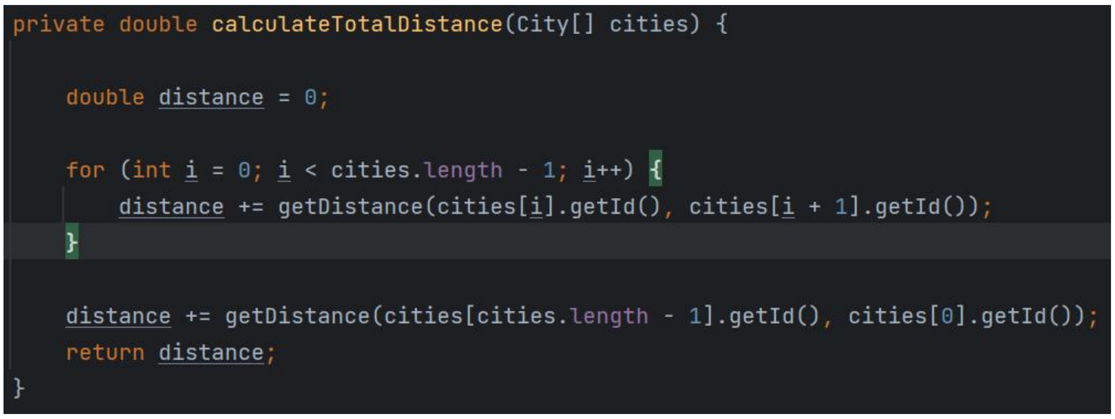
As shown in Pic 3.7, this method optimizes the Hamiltonian path (the path that visits every city exactly once) obtained from the previous method findHamiltonianCircuit(). It uses a variation of the 2-opt heuristic algorithm to iteratively swap edges in the path to try and improve its length.

The method starts by creating a copy of the original array of cities and then iterates over every city in the path. For each city, it checks every subsequent edge in the path and calculates the gain in distance if the two edges were swapped. If there is a positive gain, it marks the edge as the one that should be swapped.

Once the optimal edge is identified, the method creates a copy of the original array and reverses the path from the starting city to the ending city of the selected edge. It then checks whether the new path is shorter than the old path. If the new path is shorter, it updates the original array of cities and continues with the next iteration. If the new path is not shorter, it continues with the next iteration without changing the original array of cities.

This process continues until no further improvements can be made to the length of the path. Finally, the method returns the optimized array of cities, which is the order in which the cities should be visited to minimize the total distance traveled.

3.8 Calculates the Total Distance

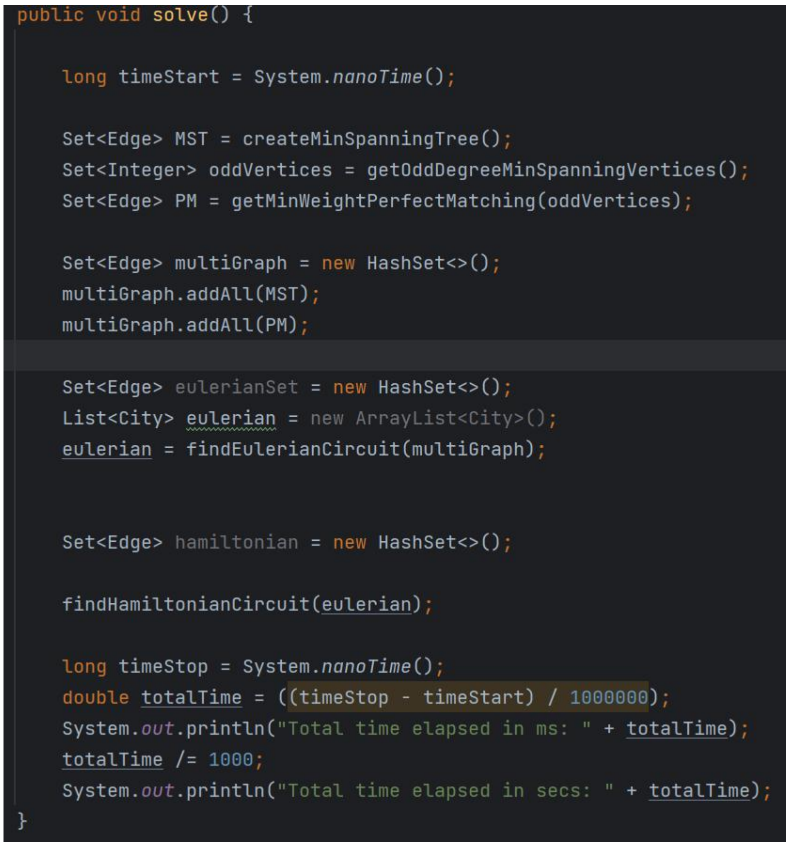


```java
private double calculateTotalDistance(City[] cities) {

    double distance = 0;

    for (int i = 0; i < cities.length - 1; i++) {
        distance += getDistance(cities[i].getId(), cities[i + 1].getId());
    }

    distance += getDistance(cities[cities.length - 1].getId(), cities[0].getId());
    return distance;
}
```

Pic 3.8

As shown in Pic 3.8, Add up the cities’ distances if they are in the path.

3.9 Calling the methods



```java
public void solve() {

    long timeStart = System.nanoTime();

    Set<Edge> MST = createMinSpanningTree();
    Set<Integer> oddVertices = getOddDegreeMinSpanningVertices();
    Set<Edge> PM = getMinWeightPerfectMatching(oddVertices);

    Set<Edge> multiGraph = new HashSet<>();
    multiGraph.addAll(MST);
    multiGraph.addAll(PM);

    Set<Edge> eulerianSet = new HashSet<>();
    List<City> eulerian = new ArrayList<City>();
    eulerian = findEulerianCircuit(multiGraph);


    Set<Edge> hamiltonian = new HashSet<>();

    findHamiltonianCircuit(eulerian);

    long timeStop = System.nanoTime();
    double totalTime = ((timeStop - timeStart) / 1000000);
    System.out.println("Total time elapsed in ms: " + totalTime);
    totalTime /= 1000;
    System.out.println("Total time elapsed in secs: " + totalTime);
}
```

Pic 3.9

This method looks like the main method which is the main entry point for the TSP solver. It solve the TSP problem by calling the previous method we’ve talked about. We solve the problem according to the reference in Wikipedia:

1. Given: complete graph whose edge weights obey the triangle inequality.
2. Calculate minimum spanning tree T.
3. Calculate the set of vertices O with odd degree in T.
4. Form the subgraph of G using only the vertices of O.
5. Construct a minimum-weight perfect matching M in this subgraph.
6. Unite matching and spanning tree T ∪ M to form an Eulerian multigraph.
7. Calculate Euler tour.
8. Remove repeated vertices, giving the algorithm's output.