Java Framework Algoritma Particle Swarm Optimization untuk Multi Depot Multi Traveling Salesman Problem

TIM DEVELOPER

SULFAYANTI
A. AMIRUL ASNAN CIRUA
SUGIARTO COKROWIBOWO
RAHMATIA

Universitas Sulawesi Barat

Java Framework Algoritma Particle Swarm Optimization untuk Multi Depot Multi Traveling Salesman Problem

Pustaka kerangka kerja yang dibangun menggunakan bahasa pemrograman java untuk mengimplementasikan algoritma Particle Swarm Optimization (PSO) dalam menyelesaikan masalah distribusi Mutli Depot Multi Traveling Salesman Problem

Traveling Salesman Problem (TSP) adalah masalah optimisasi kombinatorial dengan kompleksitas NP-Hard yang telah dikenal sangat luas. TSP merupakan bagian dari kelas hard optimization problems yang telah banyak digunakan sebagai tolok ukur pada berbagai metode atau algoritma optimisasi. Masalah optimisasi kombinatorial pada TSP adalah masalah Hamiltonian Cycle dengan biaya minimum. Pada siklus Hamiltonian setiap vertex (kecuali vertex awal) akan dikunjungi tepat satu kali. Siklus ini dikerjakan pada graf berbobot tak berarah. Ada banyak penerapan TSP di bidang penjadwalan, transportasi, logistik, dan manufaktur. Varian lanjutan model TSP ini adalah Multiple Traveling Salesman Problem (MTSP) dan Multiple Depot Multiple Traveling Salesman Problem (MDMTSP). Jika pada TSP hanya terdapat satu salesman maka pada model MTSP terdapat lebih dari satu salesman (multiagent model). Berikutnya MDMTSP merupakan generalisasi lebih lanjut dari MTSP yang memungkinkan untuk memodelkan TSP dengan lebih dari satu depot dan lebih dari satu salesman

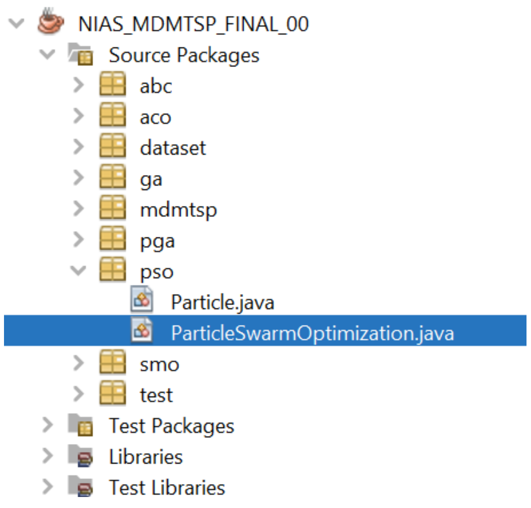
Nature Inspired Algorithms (NIAs) atau yang kita sebut dengan algoritma terispirasi alam merupakan kelompok algoritma yang terdiri dari serangkaian teknik penyelesaian masalah baru yang terinspirasi oleh fenomena alam. Secara umum, masalah optimisasi di dunia nyata sangatlah kompleks dengan karakteristik multi-objektif dan multi-dimensi. Jika menggunakan metode deterministik maka akan melibatkan usaha yang besar dan kompleksitas waktu yang sangat tinggi. Sehingga untuk mengatasi masalah yang sangat kompleks ini kemudian banyak algoritma terinspirasi alam yang diusulkan. Algoritma terinspirasi alam bekerja berdasarkan teknik stokastik untuk menemukan solusi optimal pada ruang pencarian yang lebih luas

Framework (Kerangka Kerja) berbasis Java ini dikembangkan menggunakan Algoritma **Ant Colony Optimization** untuk menyelesaikan masalah distribusi **Multi Depot Multi Traveling Salesman Problem**.

Souce Code

Java Framework for **Algoritma Particle Swarm Optimization** to solve **Multi Depot Multi Traveling Salesman Problem**

Struktur Directory



```
NIAS_MDMTSP_FINAL_00
  Source Packages
    abc
    aco
    dataset
    ga
    mdmtsp
    pga
    pso
      Particle.java
      ParticleSwarmOptimization.java
    smo
    test
  Test Packages
  Libraries
  Test Libraries
```

Class ParticleSwarmOptimization

```
package pso;

import java.util.ArrayList;
import mdmtsp.Algorithm;
import mdmtsp.Individual;
import mdmtsp.MDMTSP;
import mdmtsp.Operation;
import mdmtsp.SearchOption;

public class ParticleSwarmOptimization implements Algorithm {

    // variables
    private MDMTSP mdmtsp = null;
    private Individual bestSolution = null;
    Operation operation = new Operation();

    // List Of Operations used
    // Search Options. Only SearchOption.NONE or SearchOption.PARTIAL
allowed
    // SearchOption.DEFAULT = SearchOption.PARTIAL
    private SearchOption swapOperation = SearchOption.NONE;
    private SearchOption slideOperation = SearchOption.NONE;
    private SearchOption flipOperation = SearchOption.NONE;
    private SearchOption breakpointOperation = SearchOption.NONE;
    private SearchOption startDepotOperation = SearchOption.NONE;

    // PSO PARAMETERS
    private int I;      //Total Number of Iterations
    private int N;      //Total Number of Particles
    private double pr;  //Perturbation Rate

    //VARIABLES
    private int t = 0;//iteration counter
    private Particle[] particle = null;//particles = population of
particles
    private Individual globalBest = null;

    public ParticleSwarmOptimization(MDMTSP mdmtsp, int populationSize, int
MAX_ITERATION, double perturbationRate) {
        this.mdmtsp = mdmtsp;
        this.N = populationSize;
        this.I = MAX_ITERATION;
        this.pr = perturbationRate;
    }

    public void setSearchOption(SearchOption swapOperation, SearchOption
slideOperation, SearchOption flipOperation, SearchOption
breakpointOperation, SearchOption startDepotOperation) {
        this.swapOperation = swapOperation;
        this.slideOperation = slideOperation;
        this.flipOperation = flipOperation;
        this.breakpointOperation = breakpointOperation;
        this.startDepotOperation = startDepotOperation;
    }

    @Override
    public boolean init() {
        boolean status = false;
        if (this.mdmtsp != null
                && I > 0
                && N > 0) {
            // initialize particles
            particle = new Particle[N];
            // generate random particle
            double globalFitness = 0;
            int indexOfGLobalSolution = -1;
            for (int i = 0; i < N; i++) {
                Individual previousBest = null;
                int[][] velocity = null;
                Individual solution = new Individual(this.mdmtsp);
                solution.generateRandomChromosome();
                solution.calculateFitness();
                previousBest = solution.clone();
                particle[i] = new Particle(previousBest, velocity,
solution);
                // Elitism
                if (globalFitness < solution.getFitness()) {
                    globalFitness = solution.getFitness();
                    indexOfGLobalSolution = i;
                }
            }
            globalBest =
particle[indexOfGLobalSolution].getSolution().clone();
            status = true;
        }
        return status;
    }

    @Override
    public void process() {
        if (init()) {
            int customerSize = mdmtsp.getCustomers().size();
            while (t <= I) {
                for (int i = 0; i < N; i++) {
                    Individual previousParticle =
particle[i].getSolution();// xi(t-1)
                    Individual previousBest =
particle[i].getPreviousBest();
                    int[][] previousVelocity = particle[i].getVelocity();

                    // CALCULATE NEW VELOCITY
                    double alpha = operation.randomUniform();//U(0,1)
                    double beta = operation.randomUniform();//U(0,1)

                    ArrayList<int[]> velocity = new ArrayList<>();
                    if (previousVelocity != null) {
                        for (int[] v : previousVelocity) {
                            velocity.add(v.clone());
                        }
                    }

                    // interaction with previousBest
                    if (alpha >= pr) {
                        int[][] velocityToPreviousBest =
operation.subtraction(previousParticle, previousBest);
                        if (velocityToPreviousBest != null) {
                            for (int[] v : velocityToPreviousBest) {
                                velocity.add(v.clone());
                            }
                        }
                    }

                    // interaction with global solution
                    if (beta >= pr) {
                        int[][] velocityToGlobalBest =
operation.subtraction(previousParticle, globalBest);
                        if (velocityToGlobalBest != null) {
                            for (int[] v : velocityToGlobalBest) {
                                velocity.add(v.clone());
                            }
                        }
                    }

                    int[][] VSS = null;//velocity swap sequence
                    if (!velocity.isEmpty()) {
                        VSS = new int[velocity.size()][2];
                        for (int j = 0; j < velocity.size(); j++) {
                            VSS[j] = velocity.get(j).clone();
                        }
                    }

                    // transform to BSS (basic swap sequence
                    int[][] BSS = operation.callBasicSwapSequence(VSS,
customerSize);

                    // UPDATE Xi(t)
                    Individual newParticle = previousParticle.clone();
                    if (BSS != null && BSS.length > 0) {
                        if (swapOperation == SearchOption.DEFAULT) {
                            operation.swapSequence(newParticle, BSS);
                        } else if (swapOperation == SearchOption.PARTIAL) {
                            newParticle =
operation.swapSequenceWithPartialSearch(newParticle, BSS);
                        }
                    }

                    // UPDATE Xi using Reproduction Mechanism
                    double u = operation.randomUniform();//U(0,1)
                    if (u >= pr) {
                        newParticle = mutation(newParticle);
                    }

                    // UPDATE VELOCITY
                    int[][] newVelocity =
operation.subtraction(previousParticle, newParticle);
                    newParticle.calculateFitness();

                    // UPDATE Pi
                    if (newParticle.getFitness() >
previousBest.getFitness()) {
                        previousBest = newParticle.clone();
                    }

                    // UPDATE PARTICLE_i
                    particle[i] = new Particle(previousBest, newVelocity,
newParticle);

                    // UPDATE G
                    if (previousBest.getFitness() >
globalBest.getFitness()) {
                        globalBest = previousBest.clone();
                    }
                }

                //increment t = t+1
                t++;
            }
            bestSolution = globalBest;
        }// end of if (init())
    }

    public Individual mutation(Individual individual) {
        Individual newIndividual = individual.clone();
        int routeSize = newIndividual.getRoute().length;

        // SLIDE OPERATION --------------------------------------------------
-----
        if (slideOperation != SearchOption.NONE) {
            // random two crossover points
            int fromIndex = operation.randomBetween(0, routeSize - 1);
            int toIndex = fromIndex;
            while (toIndex == fromIndex) {
                toIndex = operation.randomBetween(0, routeSize - 1);
            }
            if (fromIndex > toIndex) {
                int temp = fromIndex;
                fromIndex = toIndex;
                toIndex = temp;
            }
            int slideUnit = operation.randomBetween(1, Math.abs(toIndex -
fromIndex));
            // set the result
            if (slideOperation == SearchOption.DEFAULT) {
                operation.slideRoute(newIndividual, fromIndex, toIndex,
slideUnit);
            } else if (slideOperation == SearchOption.PARTIAL) {
                newIndividual =
operation.slideRouteWithPartialSearch(newIndividual, fromIndex, toIndex,
slideUnit);
            }
        }// end of SLIDE OPERATION -----------------------------------------
-----

        // FLIP OPERATION ----------------------------------------------------
-----
        if (flipOperation != SearchOption.NONE) {
            // random two crossover points
            int fromIndex = operation.randomBetween(0, routeSize - 1);
            int toIndex = fromIndex;
            while (toIndex == fromIndex) {
                toIndex = operation.randomBetween(0, routeSize - 1);
            }
            if (fromIndex > toIndex) {
                int temp = fromIndex;
                fromIndex = toIndex;
                toIndex = temp;
            }
            // set the result
            if (flipOperation == SearchOption.DEFAULT) {
                operation.flipRoute(newIndividual, fromIndex, toIndex);
            } else if (flipOperation == SearchOption.PARTIAL) {
                newIndividual =
operation.flipRouteWithPartialSearch(newIndividual, fromIndex, toIndex);
            }
        }// end of FLIP OPERATION ------------------------------------------
-----

        // BREAKPOINT OPERATION ----------------------------------------------
-----
        if (breakpointOperation != SearchOption.NONE) {
            // set the result
            if (breakpointOperation == SearchOption.DEFAULT) {
                operation.breakpointMutation(newIndividual);
            } else if (breakpointOperation == SearchOption.PARTIAL) {
                newIndividual =
operation.breakpointMutationWithPartialSearch(newIndividual);
            }
        }// end of BREAKPOINT OPERATION -----------------------------------
-----

        // START DEPOT OPERATION ---------------------------------------------
-----
        if (startDepotOperation != SearchOption.NONE) {
            // set the result
            if (startDepotOperation == SearchOption.DEFAULT) {
                operation.startDepotMutation(newIndividual);
            } else if (startDepotOperation == SearchOption.PARTIAL) {
                newIndividual =
operation.startDepotMutationWithPartialSearch(newIndividual);
            }
        }// end of START DEPOT OPERATION ----------------------------------
-----

        // set new Spider Monkey - i
        //newIndividual.calculateFitness();
        return newIndividual;
    }

    @Override
    public Individual getBestSolution() {
        return this.bestSolution;
    }

}
```

Class Particle

```
package pso;

import mdmtsp.Individual;

public class Particle {

    private Individual previousBest;//P
    private int[][] velocity;
    private Individual X;// individual (solution)
    public Particle(Individual previousBest, int[][] velocity, Individual
solution) {
        this.previousBest = previousBest;
        this.velocity = velocity;
        this.X = solution;
    }


    public Particle clone() {
        Particle cloneParticle = null;
        if(X!=null&&previousBest!=null){
            cloneParticle = new Particle(previousBest.clone(),
velocity.clone(), X.clone());
        }
        return cloneParticle;
    }

    public Individual getPreviousBest() {
        return previousBest.clone();
    }

    public int[][] getVelocity() {
        return velocity;
    }

    public Individual getSolution() {
        return X.clone();
    }

}
```

Class Individual

```
package mdmtsp;

import java.util.ArrayList;

public class Individual {

    private MDMTSP mdmtsp = null;
    private int[][] chromosome = null;
    private int[][] originDepot = null;
    private double distance = Double.MAX_VALUE;
    private double fitness = 0;
    private final double DEPOT_CHANGE_THRESHOLD = 0.5;
    public Individual() {
    }

    public Individual(MDMTSP mdmtsp) {
        this.mdmtsp =mdmtsp;
    }

    public Individual clone() {
        Individual cloneIndividu = null;
        if (chromosome != null) {
            cloneIndividu = new Individual();
            cloneIndividu.setMDMTSP(mdmtsp);
            cloneIndividu.chromosome = null;
            cloneIndividu.chromosome = new int[chromosome.length][];
            for (int h = 0; h < chromosome.length; h++) {
                cloneIndividu.chromosome[h] = new
int[chromosome[h].length];
                for (int i = 0; i < chromosome[h].length; i++) {
                    cloneIndividu.chromosome[h][i] = chromosome[h][i];
                }
            }
            cloneIndividu.findOriginDepot();
            cloneIndividu.calculateFitness();
        }
        return cloneIndividu;
    }

    public void setMDMTSP(MDMTSP mdmtsp) {
        this.mdmtsp = mdmtsp;
    }

    public int[][] generateRandomChromosome() {
        chromosome = null;
        if (mdmtsp != null) {
            ArrayList<Integer> depots = mdmtsp.getDepots();
            ArrayList<Integer> customers = mdmtsp.getCustomers();
            if (!depots.isEmpty() && !customers.isEmpty()) {
                ArrayList<Integer> unsetcustomers = new ArrayList<>();
                for (int i = 0; i < customers.size(); i++) {
                    unsetcustomers.add(customers.get(i));
                }
                chromosome = new int[2][customers.size()];
                //random unique position for salesman
                ArrayList<Integer> salesmanPosition = new ArrayList<>();
                salesmanPosition.add(0);// the first salesman should be
placed in first index
```

```
                int m = 1;
                while (m < mdmtsp.getNumberOfSalesmans() && m <
customers.size()) {
                    int r = Random.getRandomBetween(1, customers.size() -
1);
                    while (salesmanPosition.contains(r)) {
                        r = Random.getRandomBetween(1, customers.size() -
1);
                    }
                    salesmanPosition.add(r);
                    m++;
                }
                //set gens for chromosome
                for (int i = 0; i < customers.size(); i++) {
                    //random customers
                    int randomIndex = Random.getRandomBetween(0,
unsetcustomers.size() - 1);
                    int alele = unsetcustomers.get(randomIndex);
                    chromosome[0][i] = alele;
                    unsetcustomers.remove(randomIndex);
                    //random depots as salesman origin depot
                    chromosome[1][i] = -1;
                    if (salesmanPosition.contains(i)) {
                        int randomDepot = Random.getRandomBetween(0,
depots.size() - 1);
                        int depot = depots.get(randomDepot);// set origin
depot for salesman
                        chromosome[1][i] = depot;
                    }
                }
            }
        }
        return chromosome;
    }

    public boolean resetCustomers() {
        boolean status = false;
        if (chromosome != null && mdmtsp != null) {
            ArrayList<Integer> customers = mdmtsp.getCustomers();
            if (!customers.isEmpty()) {
                ArrayList<Integer> unsetcustomers = new ArrayList<>();
                for (int i = 0; i < customers.size(); i++) {
                    unsetcustomers.add(customers.get(i));
                }
                // set new customers
                for (int i = 0; i < chromosome[0].length; i++) {
                    //random customers
                    int randomIndex = Random.getRandomBetween(0,
unsetcustomers.size() - 1);
                    int alele = unsetcustomers.get(randomIndex);
                    chromosome[0][i] = alele;
                    unsetcustomers.remove(randomIndex);
                }
                status = true;
            }
        }
        return status;
    }

    public boolean resetSalesmans() {
        boolean status = false;
        if (chromosome != null && mdmtsp != null) {
            ArrayList<Integer> depots = mdmtsp.getDepots();
            if (!depots.isEmpty()) {
                //random unique position for salesman
                ArrayList<Integer> salesmanPosition = new ArrayList<>();
                salesmanPosition.add(0);// the first salesman should be
placed in first index
                int m = 1;
                while (m < mdmtsp.getNumberOfSalesmans() && m <
chromosome[1].length) {
                    int r = Random.getRandomBetween(1, chromosome[1].length
- 1);
                    while (salesmanPosition.contains(r)) {
                        r = Random.getRandomBetween(1, chromosome[1].length
- 1);
                    }
                    salesmanPosition.add(r);
                    m++;
                }
                // set new salesmans
                for (int i = 0; i < chromosome[1].length; i++) {
                    //random depots as salesman origin depot
                    chromosome[1][i] = -1;
                    if (salesmanPosition.contains(i)) {
                        int randomDepot = Random.getRandomBetween(0,
depots.size() - 1);
                        int depot = depots.get(randomDepot);// set origin
depot for salesman
                        chromosome[1][i] = depot;
                    }
                }
```

```java
                status = true;
            }
        }
        return status;
    }

    public int[][] findOriginDepot() {
        originDepot = null;
        if (mdmtsp != null && chromosome != null) {
            int N = mdmtsp.getNumberOfSalesmans();
            originDepot = new int[N][2];//originDepot[i][0]=position;
originDepot[i][1]=origin depot;
            //initialize originDepot with -1
            for (int i = 0; i < originDepot.length; i++) {
                originDepot[i][0] = -1;//position
                originDepot[i][1] = -1;//origin depot
            }
            //find salesman position and salesman's origin depot
            int n = 0;
            for (int i = 0; i < chromosome[1].length; i++) {
                if (chromosome[1][i] >= 0 && n < originDepot.length) {
                    originDepot[n][0] = i;//position
                    originDepot[n][1] = chromosome[1][i];//origin depot
                    n++;
                }
            }
        }
        return originDepot;
    }

    public boolean swapCustomers(int index1, int index2) {
        boolean status = false;
        if (chromosome != null
                && index1 >= 0
                && index2 >= 0
                && index1 < chromosome[0].length
                && index2 < chromosome[0].length
                && index1 != index2) {
            int temp = chromosome[0][index1];
            chromosome[0][index1] = chromosome[0][index2];
            chromosome[0][index2] = temp;
            status = true;
        }
        return status;
    }
    public boolean swapSalesman(int salesmanIndex1, int salesmanIndex2) {
        boolean status = false;
        findOriginDepot();
        if (chromosome != null
                && originDepot != null
                && salesmanIndex1 >= 0
                && salesmanIndex2 >= 0
                && salesmanIndex1 < originDepot.length
                && salesmanIndex2 < originDepot.length) {
            int position1 = originDepot[salesmanIndex1][0];
            int depot1 = originDepot[salesmanIndex1][1];
            int position2 = originDepot[salesmanIndex2][0];
            int depot2 = originDepot[salesmanIndex2][1];
            boolean swap = true;
            //The first salesman cannot be empty
            //never swap with depot = -1
            if ((position1 == 0 && depot2 == -1)
                    || (position2 == 0 && depot1 == -1)
                    || position1 < 0
                    || position2 < 0
                    || depot1 < 0
                    || depot2 < 0
                    || position1 >= chromosome[1].length
                    || position2 >= chromosome[1].length) {
                swap = false;
            }
            if (swap) {
                chromosome[1][position1] = depot2;
                chromosome[1][position2] = depot1;
                findOriginDepot();
                status = true;
            }
        }
        return status;
    }

    public boolean swapOperation(int index1, int index2) {
        return swapCustomers(index1, index2);
    }

    public int[][] swapSequence(int[][] swapOperations) {
        int[][] newChromosome = null;
        if (swapOperations != null && chromosome != null) {
            calculateFitness();
            Individual clone = clone();
            for (int i = 0; i < swapOperations.length; i++) {
                int index1 = swapOperations[i][0];
                int index2 = swapOperations[i][1];
                clone.swapOperation(index1, index2);
                clone.calculateFitness();
                if (clone.getFitness() > this.getFitness()) {
                    setChromosome(clone.getChromosome());
                    calculateFitness();
                }else{
                    System.out.println("NOX");
                }
            }
            newChromosome = chromosome;
        }
        return newChromosome;
    }

    public boolean shiftChange(int salesmanIndex, int destinationIndex) {
        // slesmanIndex: 0 to (number of salesman - 1)
        // destinationIndex; 0 to (chromosome length - 1)
        boolean status = false;
        findOriginDepot();
        if (chromosome != null
                && originDepot != null
                && salesmanIndex >= 0
                && salesmanIndex < originDepot.length
                && destinationIndex >= 0
                && destinationIndex < chromosome[1].length) {
            int position1 = originDepot[salesmanIndex][0];
            int depot1 = originDepot[salesmanIndex][1];
            int position2 = destinationIndex;
            int depot2 = chromosome[1][destinationIndex];
            boolean swap = true;
            //The first salesman cannot be empty
            //never swap with depot = -1
            if ((position1 == 0 && depot2 == -1)
                    || (position2 == 0 && depot1 == -1)
                    || position1 < 0
                    || position2 < 0
                    || position1 >= chromosome[1].length
                    || position2 >= chromosome[1].length) {
                swap = false;
            }
            if (swap) {
                chromosome[1][position1] = depot2;
                chromosome[1][position2] = depot1;
                findOriginDepot();
                status = true;
            }

        }
        return status;
    }

    public boolean customerMutation(int index) {
        boolean status = false;
        if (chromosome != null && index >= 0 && index <
chromosome[0].length) {
            int index1 = index;
            int index2 = index;
            while (index2 == index1) {
                index2 = Random.getRandomBetween(0, chromosome[0].length -
1);
            }
            status = swapCustomers(index1, index2);
        }
        return status;
    }

    public boolean salesmanMutation(int salesmanIndex) {
        boolean status = false;
        findOriginDepot();
        if (mdmtsp != null
                && chromosome != null
                && originDepot != null
                && salesmanIndex >= 0
                && salesmanIndex < originDepot.length) {
            ArrayList<Integer> depots = mdmtsp.getDepots();
            int position = originDepot[salesmanIndex][0];//position
            int depot1 = originDepot[salesmanIndex][1];//old depot
            if (position >= 0 && position < chromosome[1].length) {
                int depot2 = depot1;//new depot
                if (depots.size() == 1) {
                    double rd = Random.getRandomUniform();
                    if (rd > DEPOT_CHANGE_THRESHOLD && position > 0) {
                        depot2 = -1;
                    }
                } else {
                    while (depot2 == depot1) {
                        depot2 = -1;
                        int randomDepot = -1;
                        if (position == 0) {
                            randomDepot = Random.getRandomBetween(0,
depots.size() - 1);
                        } else {
                            double rd = Random.getRandomUniform();
                            if (rd > DEPOT_CHANGE_THRESHOLD) {
                                depot2 = -1;
                            } else {
                                randomDepot = Random.getRandomBetween(-1,
depots.size() - 1);
                            }
                        }
                        if (randomDepot >= 0 && randomDepot <
depots.size()) {
                            depot2 = depots.get(randomDepot);
                        }
                    }
                }
                //set new depot
                chromosome[1][position] = depot2;
                findOriginDepot();
                status = true;
            }
        }
        return status;
    }

    public double calculateDistance() {
        distance = Double.MAX_VALUE;
        double[][] adjacency = mdmtsp.getAdjacency();
        if (chromosome != null && adjacency != null) {
            distance = 0;
            int depot = chromosome[1][0];
            int nodeOrigin = depot;
            int nodeDestination = chromosome[0][0];
            distance += adjacency[nodeOrigin][nodeDestination];
            for (int i = 1; i < chromosome[0].length; i++) {
                int customer = chromosome[0][i];
                if (chromosome[1][i] != -1) {
                    //close the previous route
                    nodeOrigin = nodeDestination;
                    nodeDestination = depot;
                    distance += adjacency[nodeOrigin][nodeDestination];
                    //open new route
                    depot = chromosome[1][i];
                    nodeOrigin = depot;
                    nodeDestination = chromosome[0][i];
                    distance += adjacency[nodeOrigin][nodeDestination];
                } else {
                    nodeOrigin = nodeDestination;
                    nodeDestination = customer;
                    distance += adjacency[nodeOrigin][nodeDestination];
                }

                //check last customer
                if (i == chromosome[0].length - 1) {
                    // close the last route
                    nodeOrigin = nodeDestination;
                    nodeDestination = depot;
                    distance += adjacency[nodeOrigin][nodeDestination];
                    break;
                }
            }
        }
        return distance;
    }

    public double calculateFitness() {
        fitness = 0;
        calculateDistance();
        if (distance > 0) {
            fitness = 1.0 / distance;
        }
        return fitness;
    }

    public void setChromosome(int[][] chromosome) {
        this.chromosome = chromosome;
    }

    public int[][] getChromosome() {
        return this.chromosome;
    }

    public int[] getCustomerChromosome(){
        int[]customerChromosome = null;
        if(chromosome!=null){
            customerChromosome=new int[chromosome[0].length];
            for (int i = 0; i < customerChromosome.length; i++) {
                customerChromosome[i]=chromosome[0][i];
            }
        }
        return customerChromosome;
    }

    public void setCustomerChromosome(int[]customerChromosome){
        if(customerChromosome!=null){
            chromosome[0]=new int[customerChromosome.length];
            for (int i = 0; i < customerChromosome.length; i++) {
                chromosome[0][i]=customerChromosome[i];
            }
        }
    }

    public int[][] getOriginDepot() {
        return this.originDepot;
    }

    public double getDistance() {
        return this.distance;
    }

    public double getFitness() {
        return this.fitness;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        if (chromosome != null) {
            int depot = chromosome[1][0];
            int nodeOrigin = depot;
            int nodeDestination = chromosome[0][0];
            int n = 1;
            sb.append("route-" + n + ": " + nodeOrigin + " - " +
nodeDestination);
            for (int i = 1; i < chromosome[0].length; i++) {
                int customer = chromosome[0][i];
                if (chromosome[1][i] != -1) {
                    //close the previous route
                    sb.append(" - " + depot + "\n");
                    //open new route
                    depot = chromosome[1][i];
                    nodeOrigin = depot;
                    nodeDestination = chromosome[0][i];
                    n++;
                    sb.append("route-" + n + ": " + nodeOrigin + " - " +
nodeDestination);
                } else {
                    nodeOrigin = nodeDestination;
                    nodeDestination = customer;
                    sb.append(" - " + nodeDestination);
                }
                //check last customer
                if (i == chromosome[0].length - 1) {
                    //close the last route
                    sb.append(" - " + depot);
                }
            }
        }
        return sb.toString();
    }

}
```

Class MDMTSP

```
package mdmtsp;

import java.util.ArrayList;

public class MDMTSP {

    private double[][] adjacency = null;
    private int M = 0;//NUMBER_OF_SALESMANS
    private ArrayList<Integer> depots = null;
    private ArrayList<Integer> customers = null;

    public MDMTSP() {
    }

    public MDMTSP(String filename, int[] depots, int numberOfSalesmans) {
        readDataset(filename);
        setDepots(depots);
        setCustomers();
        setNumberOfSalesmans(numberOfSalesmans);
    }

    public void readDataset(String filename) {
        this.adjacency = new DataReader().read(filename);
    }

    public void setDepots(int[] depots) {
        if (depots != null && depots.length > 0 && adjacency != null &&
adjacency.length > 0) {
            this.depots = new ArrayList<>();
            for (int i = 0; i < depots.length; i++) {
                if (depots[i] >= 0 && depots[i] < adjacency.length) {
                    this.depots.add(depots[i]);
                }
            }
        }
    }

    public void setCustomers() {
        if (adjacency != null && adjacency.length > 0 && this.depots !=
null) {
            customers = new ArrayList<>();
            for (int i = 0; i < adjacency.length; i++) {
                if (!depots.contains(i)) {
                    customers.add(i);
                }
            }
        }
    }

    public void setNumberOfSalesmans(int M) {
        if (M > 0) {
            this.M = M;
        }
    }

    public double[][] getAdjacency(){
        return this.adjacency;
    }

    public ArrayList<Integer> getDepots() {
        return this.depots;
    }

    public ArrayList<Integer> getCustomers() {
        return this.customers;
    }

    public int getNumberOfSalesmans() {
        return this.M;
    }

}
```

Class Random

```
package mdmtsp;

public class Random {
    private static java.util.Random random = new java.util.Random();

    public static int getRandomBetween(int min, int max) {
        if(min>max){
            int temp = min;
            min = max;
            max = temp;
        }
        return random.nextInt(1 + max - min) + min;
    }

    public static double getRandomUniform(){
        return random.nextDouble();
    }
}
```

Class DataReader

```
package mdmtsp;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class DataReader {

    public double[][] adjacency = null;
    public String EDGE_WEIGHT_TYPE = null;

    public double[][] read(String filename) {
        try {
            File file = new File(filename);
            Scanner sc = new Scanner(file);
            int dimension = 0;
            has_next:
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] values = line.split(":");
                if (values[0].trim().equalsIgnoreCase("DIMENSION")) {
                    dimension = Integer.parseInt(values[1].trim());
                    adjacency = new double[dimension][dimension];
                } else if
(values[0].trim().equalsIgnoreCase("EDGE_WEIGHT_TYPE")) {
                    String type = values[1].trim().toUpperCase();
                    EDGE_WEIGHT_TYPE = type;
                } else if
(values[0].trim().equalsIgnoreCase("EDGE_WEIGHT_SECTION") &&
EDGE_WEIGHT_TYPE.equalsIgnoreCase("EXPLICIT")) {
                    int r = 0;
                    int c = 0;
                    while (sc.hasNextLine() && r < dimension) {
                        line = sc.nextLine();
                        if (line.equalsIgnoreCase("EOF")) {
                            break has_next;
                        } else {
                            values = line.split("\s+");
                            for (String s : values) {
                                s = s.trim();
                                if (s.length() > 0) {
                                    double d = Double.parseDouble(s);
                                    adjacency[r][c] = d;
                                    c++;
                                    if (c >= dimension) {
                                        c = 0;
                                        r++;
                                    }
                                }
                            }
                        }
                    }
                } else if
(values[0].trim().equalsIgnoreCase("NODE_COORD_SECTION") &&
EDGE_WEIGHT_TYPE.equalsIgnoreCase("GEO")) {
                    int i = 0;
                    double[][] nodeCoordinate = new double[dimension][2];
                    while (sc.hasNextLine() && i < dimension) {
                        line = sc.nextLine();
                        if (line.equalsIgnoreCase("EOF")) {
                            break has_next;
                        } else {
                            values = line.split("\s+");
                            //int number =
Integer.parseInt(values[0].trim());
                            double x =
Double.parseDouble(values[1].trim());
                            double y =
Double.parseDouble(values[2].trim());
```

```
                            nodeCoordinate[i][0] = x;
                            nodeCoordinate[i][1] = y;
                            i++;
                        }
                    }
                    //calculate distance;
                    adjacency = distancesInGEO(nodeCoordinate);
                    //calculate distance;
//                    for (int j = 0; j < dimension; j++) {
//                        double x1 = nodeCoordinate[j][0];
//                        double y1 = nodeCoordinate[j][1];
//                        for (int k = j; k < dimension; k++) {
//                            double x2 = nodeCoordinate[k][0];
//                            double y2 = nodeCoordinate[k][1];
//                            //double distance = Math.sqrt(Math.pow((x1-
x2), 2)+Math.pow((y1-y2), 2));
//
//                            adjacency[j][k] = distance;
//                            adjacency[k][j] = distance;
//                        }
//                    }
                }


            }
        } catch (FileNotFoundException ex) {

//Logger.getLogger(DataReader.class.getName()).log(Level.SEVERE, null, ex);
        }
        return adjacency;
    }

    static double[][] distancesInGEO(double[][] nodes) {
        int dim = nodes.length;
        double[] latitude = new double[dim];
        double[] longitude = new double[dim];

        final double PI = Math.PI;//3.141592;
        for (int i = 0; i < dim; i++) {
            int deg = (int)(nodes[i][0]);
            double min = nodes[i][0] - deg;
            latitude[i] = PI * (deg + 5 * min / 3.0) / 180;
            deg =  (int)(nodes[i][1]);
            min = nodes[i][1] - deg;
            longitude[i] = PI * (deg + 5 * min / 3.0) / 180;
        }
        double[][] d = new double[dim][dim];

        final double RRR = 6378.388;
        for (int i = 0; i < dim; i++) {
            for (int j = i + 1; j < dim; j++) {
                double q1 = Math.cos(longitude[i] - longitude[j]);
                double q2 = Math.cos(latitude[i] - latitude[j]);
                double q3 = Math.cos(latitude[i] + latitude[j]);
                //d[i][j] = (int) (RRR * Math.acos(0.5 * ((1.0 + q1) * q2 -
(1.0 - q1) * q3)) + 1.0);
                d[i][j] = (int)(RRR * Math.acos(0.5 * ((1.0 + q1) * q2 -
(1.0 - q1) * q3)) + 1.0);
                d[j][i] = d[i][j];
            }
        }
        return d;
    }
}
```