Java Framework for Spider Monkey Optimization to solve Multi Depot Multi Traveling Salesman Problem

TIM DEVELOPER

A. AMIRUL ASNAN CIRUA
WAWAN FIRGIAWAN
SUGIARTO COKROWIBOWO
RICH RONA SELAMAT MARPAUNG

Universitas Sulawesi Barat

Java Framework for Spider Monkey Optimization to Solve Multi Depot Multi Traveling Salesman Problem

Pustaka kerangka kerja yang dibangun menggunakan bahasa pemrograman java untuk mengimplementasikan algoritma Spider Monkey Optimization 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

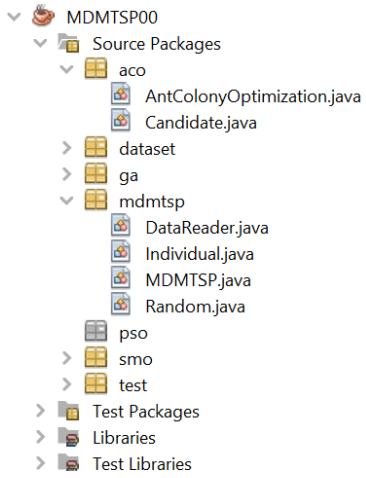
Nature Inspired Algorithms (NIAs) atau yang kita sebut dengan algoritma terispirasi alam merupakan kelompok algoritma yang terdiri dari serangkaian teknik penyelesaian masalah baru yang terinspirasi oleh fenomena alam. Secara umum, masalah optimisasi di dunia nyata sangatlah kompleks dengan karakteristik 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 **Spider Monkey Optimization** untuk menyelesaikan masalah distribusi **Multi Depot Multi Traveling Salesman Problem**.

Souce Code

Java Framework for **Spider Monkey Optimization** to solve **Multi Depot Multi Traveling Salesman Problem**

Struktur Directory



Class SpiderMonkeyOptimization

```
package smo;

import mdmtsp.Individual;
import mdmtsp.MDMTSP;
import mdmtsp.Random;

public class SpiderMonkeyOptimization {

    // SMO Parameters
    //INPUT
    private int I;       //Total Number of Iterations
    private int N;       //Total Number of Spider Monkey
    private int MG;      //Allowed Maximum Group
    private double pr;   //Perturbation Rate
    private int LLL;     //Local Leader Limit
    private int GLL;     //Global Leader Limit

    double mutationRate = 0.9;

    // variables
    private MDMTSP mdmtsp = null;
```

```
    private Individual bestSolution = null;

    //VARIABLES
    private int t = 0;//iteration counter
    private int g = 0;//Current Number of Group
    private int groupSize = 1;//banyaknya spider monkey di setiap group
    private Individual[] spiderMonkey = null;//SM = Population of Spider
Monkey
    private Individual globalLeader = null;
    private int globalLeaderLimitCounter = 0;//GLLc = Global Leader Limit
Counter
    private Individual[] localLeader = null;//LL = List of Local Leader
    private int[] localLeaderLimitCounter = new int[g];//LLLc = Local
Leader Limit Counter of kth Group

    public SpiderMonkeyOptimization(MDMTSP mdmtsp, int
totalNumberOfIterations, int totalNumberOfSpiderMonkey, int
allowedMaximumGroup, double perturbationRate, int localLeaderLimit, int
globalLeaderLimit) {
        this.mdmtsp = mdmtsp;
        this.I = totalNumberOfIterations;
        this.N = totalNumberOfSpiderMonkey;
        this.MG = allowedMaximumGroup;
        this.pr = perturbationRate;
        this.LLL = localLeaderLimit;
        this.GLL = globalLeaderLimit;
    }

    public boolean init() {
        boolean status = false;
        if (this.mdmtsp != null
                && I > 0
                && N > 0) {
            // Initialization---------------------------------------------
-----
            if (this.MG < (N / 2)) {
                this.MG = N / 2;
            }
            if (this.MG <= 0) {
                this.MG = 1;
            }
            this.t = 1;//(1) t ← 1
            this.spiderMonkey = new Individual[N];//(2) create N spider
moneys and append them to SM

            //(3)Assign each SMi in SM with a random solution
```

```
            int indexGlobalLeader = -1;
            double globalFitness = 0;
            for (int i = 0; i < N; i++) {
                spiderMonkey[i] = new Individual(this.mdmtsp);
                spiderMonkey[i].generateRandomChromosome();
                spiderMonkey[i].calculateFitness();
                //System.out.println(spiderMonkey[i]);
                if (globalFitness < spiderMonkey[i].getFitness()) {
                    globalFitness = spiderMonkey[i].getFitness();
                    indexGlobalLeader = i;
                }
            }

            //(4) g = 1 initially consider all spiderMonkey into one group
            this.g = 1;
            this.groupSize = (int) Math.floor((double) N / (double) g);

            //(5) Select Local Leader and Global Leader // Both leaders are
same due to single group
            //GLOBAL LEADER
            this.globalLeader = spiderMonkey[indexGlobalLeader].clone();
            this.globalLeaderLimitCounter = 0;//GLLc

            //LOCAL LEADER
            this.localLeader = new Individual[g];
            this.localLeader[0] = this.globalLeader.clone();
            this.localLeaderLimitCounter = new int[g];//LLLc
            status = true;
        }
        return status;
    }

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

    public void process() {
        if (init()) {
            Operation op = new Operation();
            while (t <= I) {

//================================================================
                //[1] Update of Spider Monkeys

//================================================================
                //[1.1] UPDATE Spider Monkeys base on local Leader
```

```
                for (int k = 0; k < g; k++) {
                    //set lower and upper bound
                    int lowerBound = k * groupSize;
                    int upperBound = lowerBound + groupSize - 1;
                    if (k == g - 1) {
                        upperBound = N - 1;
                    }

                    //update spider monkey
                    for (int i = lowerBound; i <= upperBound; i++) {
                        double u = Random.getRandomUniform();//U(0,1)
                        if (u >= pr) {
                            int[] chromosomeLLk =
localLeader[k].getCustomerChromosome();
                            int[] chromosomeSMi =
spiderMonkey[i].getCustomerChromosome();
                            int r = Random.getRandomBetween(lowerBound,
upperBound);
                            int[] chromosomeRSMr =
spiderMonkey[r].getCustomerChromosome();

                            int[][] ss1 = op.subtraction(chromosomeSMi,
chromosomeLLk);
                            int[][] ss2 = op.subtraction(chromosomeSMi,
chromosomeRSMr);
                            int[][] ss = op.merge(ss1, ss2);

                            // Apply SS into SMi to calculate newSM
                            // Tentative SO
                            Individual tentative = spiderMonkey[i].clone();
                            tentative.swapSequence(ss);
                            tentative.calculateFitness();

                            if (tentative.getFitness() >
spiderMonkey[i].getFitness()) {
                                spiderMonkey[i] = tentative;
                            } else {
                                //Mutate
                                double rm = Random.getRandomUniform();
                                if (rm > mutationRate) {
                                    Individual mutant =
spiderMonkey[i].clone();
                                    // customerMutation
                                    int index = Random.getRandomBetween(0,
mutant.getChromosome()[0].length - 1);
                                    mutant.customerMutation(index);
```

```
                                    mutant.calculateFitness();
                                    if (mutant.getFitness() >=
spiderMonkey[i].getFitness()) {
                                        spiderMonkey[i] = mutant.clone();
                                    }

                                    // salesmanMutation
                                    index = Random.getRandomBetween(0,
mutant.getOriginDepot().length - 1);
                                    mutant.salesmanMutation(index);
                                    mutant.calculateFitness();
                                    if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                                        spiderMonkey[i] = mutant.clone();
                                    }
                                }
                            }//end of else

                        }//end of if (u >= pr)
                    }//end of for (int i = lowerBound; i <= upperBound;
i++)
                }//end of for (int k = 0; k < g; k++)

                //[1.2] UPDATE Spider Monkeys base on global Leader
                for (int k = 0; k < g; k++) {
                    //set lower and upper bound
                    int lowerBound = k * groupSize;
                    int upperBound = lowerBound + groupSize - 1;
                    if (k == g - 1) {
                        upperBound = N - 1;
                    }

                    //update spider monkey
                    for (int i = lowerBound; i <= upperBound; i++) {
                        double u = Random.getRandomUniform();//U(0,1)
                        double prob = 0.9 * ((double)
globalLeader.getDistance() / (double) spiderMonkey[i].getDistance()) +
0.1;//prob(i)
                        if (u <= prob) {
                            int[] chromosomeGl =
globalLeader.getCustomerChromosome();
                            int[] chromosomeSMi =
spiderMonkey[i].getCustomerChromosome();
                            int r = Random.getRandomBetween(0, N - 1);
                            int[] chromosomeRSMr =
spiderMonkey[r].getCustomerChromosome();
```

```
                            int[][] ss1 = op.subtraction(chromosomeSMi,
chromosomeGl);
                            int[][] ss2 = op.subtraction(chromosomeSMi,
chromosomeRSMr);
                            int[][] ss = op.merge(ss1, ss2);

                            //Apply SS into SMi to calculate newSM
                            // Tentative SO
                            Individual tentative = spiderMonkey[i].clone();
                            tentative.swapSequence(ss);
                            if (tentative.getFitness() >
spiderMonkey[i].getFitness()) {
                                spiderMonkey[i] = tentative;
                            } else {
                                //Mutate
                                double rm = Random.getRandomUniform();
                                if (rm > mutationRate) {
                                    Individual mutant =
spiderMonkey[i].clone();
                                    // customerMutation
                                    int index = Random.getRandomBetween(0,
mutant.getChromosome()[0].length - 1);
                                    mutant.customerMutation(index);
                                    mutant.calculateFitness();
                                    if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                                        spiderMonkey[i] = mutant.clone();
                                    }

                                    // salesmanMutation
                                    index = Random.getRandomBetween(0,
mutant.getOriginDepot().length - 1);
                                    mutant.salesmanMutation(index);
                                    mutant.calculateFitness();
                                    if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                                        spiderMonkey[i] = mutant.clone();
                                    }
                                }
                            }//end of else

                        }//end of if (u >= pr)
                    }//end of for (int i = lowerBound; i <= upperBound;
i++)
                }//end of for (int k = 0; k < g; k++)
```

```
                //////MUTATION
                // MUTATION-----------------------------------------------
-----
                /*
                for (int i = 0; i < spiderMonkey.length; i++) {
                    double rm = Random.getRandomUniform();
                    double mutationRate = 0.2;
                    if (rm > mutationRate) {
                        Individual mutant = spiderMonkey[i].clone();
                        // customerMutation

                        int index = Random.getRandomBetween(0,
mutant.getChromosome()[0].length - 1);

                        mutant.customerMutation(index);
                        mutant.calculateFitness();
                        if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                            spiderMonkey[i] = mutant.clone();
                        }

                        // salesmanMutation
                        index = Random.getRandomBetween(0,
mutant.getOriginDepot().length - 1);
                        mutant.salesmanMutation(index);
                        mutant.calculateFitness();
                        if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                            spiderMonkey[i] = mutant.clone();
                        }
                    }
                }
                 */

//=================================================================
                //[2] Update of Local Leaders and Global Leader

//=================================================================
                //[2.1] check new local leader
                Individual newGlobalLeader = globalLeader.clone();
                for (int k = 0; k < g; k++) {
                    //set lower and upper bound
                    int lowerBound = k * groupSize;
                    int upperBound = lowerBound + groupSize - 1;
                    if (k == g - 1) {
```

```
                        upperBound = N - 1;
                    }

                    Individual newLocalLeader = localLeader[k].clone();
                    for (int i = lowerBound; i <= upperBound; i++) {
                        if (spiderMonkey[i].getFitness() >
newLocalLeader.getFitness()) {
                            newLocalLeader = spiderMonkey[i];
                        }
                    }

                    if (newLocalLeader.getFitness() >
localLeader[k].getFitness()) {
                        localLeader[k] = newLocalLeader.clone();
                        localLeaderLimitCounter[k] = 0;
                    } else {

localLeaderLimitCounter[k]++;//localLeaderLimitCounter[k] =
localLeaderLimitCounter[k] + 1;
                    }

                    if (localLeader[k].getFitness() >
newGlobalLeader.getFitness()) {
                        newGlobalLeader = localLeader[k];
                    }

                }//end of for (int k = 0; k < g; k++)

                //check new global leader
                if (newGlobalLeader.getFitness() >
globalLeader.getFitness()) {
                    globalLeader = newGlobalLeader.clone();
                    globalLeaderLimitCounter = 0;
                } else {
                    globalLeaderLimitCounter++;
                }


//=================================================================
                //[3] Decision Phase of Local Leader and Global Leader

//=================================================================
                //[3.1] Decision Phase of Local Leader
                for (int k = 0; k < g; k++) {
                    if (localLeaderLimitCounter[k] > LLL) {
                        localLeaderLimitCounter[k] = 0;//LLLk ← 0
```

```
                        //set lower and upper bound
                        int lowerBound = k * groupSize;
                        int upperBound = lowerBound + groupSize - 1;
                        if (k == g - 1) {
                            upperBound = N - 1;
                        }

                        for (int i = lowerBound; i <= upperBound; i++) {
                            double u = Random.getRandomUniform();//U(0,1)
                            if (u >= pr) {
                                //Initialize SMi randomly
                                Individual newSM = new
Individual(this.mdmtsp);
                                newSM.generateRandomChromosome();
                                newSM.calculateFitness();
                                spiderMonkey[i] = newSM;
                            } else {
                                //Initialize SMi by interacting with the GL
and LL
                                int[] chromosomeGL =
globalLeader.getCustomerChromosome();
                                int[] chromosomeLLk =
localLeader[k].getCustomerChromosome();
                                int[] chromosomeSMi =
spiderMonkey[i].getCustomerChromosome();

                                int[][] ss1 = op.subtraction(chromosomeSMi,
chromosomeGL);
                                int[][] ss2 = op.subtraction(chromosomeSMi,
chromosomeLLk);
                                int[][] ss = op.merge(ss1, ss2);

                                //Apply SS into SMi to calculate newSM
                                // Tentative SO
                                Individual tentative =
spiderMonkey[i].clone();
                                tentative.swapSequence(ss);
                                if (tentative.getFitness() >
spiderMonkey[i].getFitness()) {
                                    spiderMonkey[i] = tentative;
                                } else {
                                    //Mutate
                                    double rm = Random.getRandomUniform();
                                    if (rm > mutationRate) {
```

```
                                        Individual mutant =
spiderMonkey[i].clone();
                                        // customerMutation
                                        int index =
Random.getRandomBetween(0, mutant.getChromosome()[0].length - 1);
                                        mutant.customerMutation(index);
                                        mutant.calculateFitness();
                                        if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                                            spiderMonkey[i] =
mutant.clone();
                                        }

                                        // salesmanMutation
                                        index = Random.getRandomBetween(0,
mutant.getOriginDepot().length - 1);
                                        mutant.salesmanMutation(index);
                                        mutant.calculateFitness();
                                        if (mutant.getFitness() >
spiderMonkey[i].getFitness()) {
                                            spiderMonkey[i] =
mutant.clone();
                                        }
                                    }
                                }//end of else

                            }
                        }
                    }
                }//end of for (int k = 0; k < g; k++)

                //[3.1] Decision Phase of Global Leader
                if (globalLeaderLimitCounter > GLL) {
                    globalLeaderLimitCounter = 0;
                    if (g < MG) {
                        //Divide the spider monkeys into g + 1 number of
groups
                        g++;// g = g + 1
                        groupSize = (int) Math.floor((double) N / (double)
g);
                        this.localLeader = new Individual[g];
                        this.localLeaderLimitCounter = new int[g];//LLLc

                        int indexGlobalLeader = -1;
                        double globalFitness = 0;
```

```
                        //check new local leader and global leader
                        for (int k = 0; k < g; k++) {
                            //set lower and upper bound
                            int lowerBound = k * groupSize;
                            int upperBound = lowerBound + groupSize - 1;
                            if (k == g - 1) {
                                upperBound = N - 1;
                            }

                            //find new local leader
                            Individual newLocalLeader =
spiderMonkey[lowerBound];
                            for (int i = lowerBound + 1; i <= upperBound;
i++) {
                                if (spiderMonkey[i].getFitness() >
newLocalLeader.getFitness()) {
                                    newLocalLeader = spiderMonkey[i];
                                }

                            }
                            this.localLeader[k] = newLocalLeader.clone();

                            //find new global leader
                            if (globalFitness <
this.localLeader[k].getFitness()) {
                                globalFitness =
this.localLeader[k].getFitness();
                                indexGlobalLeader = k;
                            }
                        }//end of for (int k = 0; k < g; k++)

                        //update GLOBAL LEADER
                        if
(this.localLeader[indexGlobalLeader].getFitness() >
this.globalLeader.getFitness()) {
                            this.globalLeader =
this.localLeader[indexGlobalLeader].clone();
                        }

                    } else {
                        //Disband all the groups and Form a single group.
                        System.out.println("DISBAND");
                        g = 1;
                        groupSize = (int) Math.floor((double) N / (double)
g);
                        this.localLeader = new Individual[g];
```

```
                        this.localLeader[0] = this.globalLeader.clone();
                        this.localLeaderLimitCounter = new int[g];//LLLc
                    }
                }

                //INCREMENT of
t=================================================
                t++;//increment t = t+1
            }//end of while while(t<=I);
            bestSolution = this.globalLeader.clone();
        }
    }

}
```

Class Operation

```
package smo;

import java.util.ArrayList;

public class Operation {

    public int[] swap(int[] chromosome, int index1, int index2) {
        if (chromosome != null
                && index1 >= 0 && index1 < chromosome.length
                && index2 >= 0 && index2 < chromosome.length
                && chromosome[index1] != chromosome[index2]) {
            int temp = chromosome[index1];
            chromosome[index1] = chromosome[index2];
            chromosome[index2] = temp;
        }
        return chromosome;
    }

    public int[] add(int[] chromosome, SO so) {
        return swap(chromosome, so.index1, so.index2);
    }

    public int[][] subtraction(int[] chromosome1, int[] chromosome2) {
        int[][] swapsequence = null;
        if (chromosome1 != null && chromosome2 != null &&
chromosome1.length == chromosome2.length) {
            ArrayList<SO> listSwapOperation = new ArrayList<>();
            int[] chromosomeOperation = cloneChromosome(chromosome1);
            for (int i = 0; i < chromosome2.length; i++) {
                int value = chromosome2[i];
```

```
                for (int j = i; j < chromosomeOperation.length; j++) {
                    if (value == chromosomeOperation[j] && i != j) {
                        SO so = new SO(i, j);
                        chromosomeOperation = add(chromosomeOperation, so);
                        listSwapOperation.add(so);
                        break;
                    }
                }

                /*
                if (chromosomeOperation[i] != value) {
                    for (int j = i + 1; j < chromosomeOperation.length;
j++) {
                        if (value == chromosomeOperation[j]) {
                            SO so = new SO(i, j);
                            chromosomeOperation = add(chromosomeOperation,
so);
                            listSwapOperation.add(so);
                            break;
                        }
                    }
                }
                 */
            }
            if (!listSwapOperation.isEmpty()) {
                swapsequence = new int[listSwapOperation.size()][2];
                for (int i = 0; i < swapsequence.length; i++) {
                    SO so = listSwapOperation.get(i);
                    swapsequence[i][0] = so.index1;
                    swapsequence[i][1] = so.index2;
                }
            }
        }
        return swapsequence;
    }

    public int[][] merge(int[][] swapsequence1, int[][] swapsequence2) {
        int[][] result = null;
        if (swapsequence1 != null && swapsequence2 != null) {
            result = new int[swapsequence1.length +
swapsequence2.length][2];
            int k = 0;
            for (int i = 0; i < swapsequence1.length; i++) {
                result[k][0] = swapsequence1[i][0];
                result[k][1] = swapsequence1[i][1];
                k++;
```

```
            }
            for (int i = 0; i < swapsequence2.length; i++) {
                result[k][0] = swapsequence2[i][0];
                result[k][1] = swapsequence2[i][1];
                k++;
            }
        } else if (swapsequence1 != null) {
            result = new int[swapsequence1.length][2];
            for (int i = 0; i < swapsequence1.length; i++) {
                result[i][0] = swapsequence1[i][0];
                result[i][1] = swapsequence1[i][1];
            }
        } else if (swapsequence2 != null) {
            result = new int[swapsequence2.length][2];
            for (int i = 0; i < swapsequence2.length; i++) {
                result[i][0] = swapsequence2[i][0];
                result[i][1] = swapsequence2[i][1];
            }
        }
        return result;
    }

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

    public class SO {

        int index1;
        int index2;

        public SO(int index1, int index2) {
            this.index1 = index1;
            this.index2 = index2;
        }

    }
}
```

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;
                    }
                }
                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;
    }
}
```