Java Framework for Genetic Algorithm to solve Multi Depot Multi Traveling Salesman Problem

TIM DEVELOPER

NURALAMSAH ZULKARNAIM
SULFAYANTI
NAHYA NUR
MAHMUDDIN

Universitas Sulawesi Barat

Java Framework for Genetic Algorithm to solve Multi Depot Multi Traveling Salesman Problem

Pustaka kerangka kerja yang dibangun menggunakan bahasa pemrograman java untuk mengimplementasikan algoritma Genetika 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

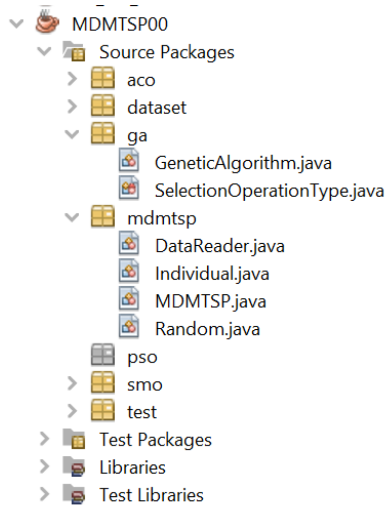
Nature Inspired Algorithms (NIAs) atau yang kita sebut dengan algoritma terispirasi alam merupakan kelompok algoritma yang terdiri dari serangkaian teknik penyelesaian masalah baru yang terinspirasi oleh fenomena alam. Secara umum, masalah optimisasi di dunia nyata sangatlah kompleks dengan karakteristik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

Souce Code

Java Framework for **Genetic Algorithm** to solve **Multi Depot Multi Traveling Salesman Problem**

Struktur Directory



```
MDMTSP00
  Source Packages
    aco
    dataset
    ga
      GeneticAlgorithm.java
      SelectionOperationType.java
    mdmtsp
      DataReader.java
      Individual.java
      MDMTSP.java
      Random.java
    pso
    smo
    test
  Test Packages
  Libraries
  Test Libraries
```

Class GeneticAlgorithm

```
package ga;

import java.util.ArrayList;
import mdmtsp.Individual;
import mdmtsp.MDMTSP;
import mdmtsp.Random;

public class GeneticAlgorithm {

    //GA parameters
    private final int MAX_GENERATION;
    private int populationSize;
    private SelectionOperationType selection =
SelectionOperationType.tournament;
    private int numberOfIndividualsSelected;
    private double mutationRate = 0.5;

    // variables
    private MDMTSP mdmtsp = null;
    private Individual bestSolution = null;
    private double bestFitness = 0;

    public GeneticAlgorithm(MDMTSP mdmtsp, int populationSize, int
MAX_GENERATION, SelectionOperationType selection, int
numberOfIndividualsSelected, double mutationRate) {
        this.mdmtsp = mdmtsp;
        this.populationSize = populationSize;
        this.MAX_GENERATION = MAX_GENERATION;
        this.selection = selection;
        this.numberOfIndividualsSelected = numberOfIndividualsSelected;
        this.mutationRate = mutationRate;
    }

    public boolean init() {
        boolean status = false;
        if (this.mdmtsp != null
                && this.populationSize > 0
                && this.MAX_GENERATION >= 1
                && this.mutationRate >= 0) {
            if (this.populationSize < 2) {
                this.populationSize = 2;
            }
            if (numberOfIndividualsSelected <= 0) {
                numberOfIndividualsSelected = 2;
            } else if (numberOfIndividualsSelected > populationSize) {
                numberOfIndividualsSelected = (int)
Math.ceil(populationSize / 2.0);
            }
            status = true;
        }
        return status;
    }

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

    public void process() {
        if (init()) {
            Individual[] population = new Individual[populationSize];

            //Initialize random population
            for (int i = 0; i < population.length; i++) {
                population[i] = new Individual();
                population[i].setMDMTSP(mdmtsp);
                population[i].generateRandomChromosome();
                population[i].calculateFitness();
                // ELITISM
                if (population[i].getFitness() > bestFitness) {
                    bestSolution = population[i];
                    bestFitness = bestSolution.getFitness();
                }
            }

            // Evolution Process
            for (int g = 1; g <= this.MAX_GENERATION; g++) {
                Individual[] newPopulation = new
Individual[populationSize];

                // SELECTION--------------------------------------------------
-----
                if (selection == SelectionOperationType.tournament) {
                    // Tournament Selection
                    // sort population base on their fitness values
                    double[][] fitness = new
double[populationSize][2];//[index | fitness]
                    for (int i = 0; i < populationSize; i++) {
                        fitness[i][0] = i;
                        fitness[i][1] = population[i].getFitness();
                    }
                    // sort
                    for (int i = 0; i < fitness.length - 1; i++) {
                        int iMAX = i;
                        double MAX_FITNESS = fitness[i][1];
                        for (int j = i + 1; j < fitness.length; j++) {
                            if (fitness[j][1] > MAX_FITNESS) {
                                MAX_FITNESS = fitness[j][1];
                                iMAX = j;
                            }
                        }
                        if (iMAX > i) {
                            // swap
                            double temp0 = fitness[i][0];
                            double temp1 = fitness[i][1];
                            fitness[i][0] = fitness[iMAX][0];
                            fitness[i][1] = fitness[iMAX][1];
                            fitness[iMAX][0] = temp0;
                            fitness[iMAX][1] = temp1;
                        }
                    }
                    // save selected individual
                    for (int i = 0; i < numberOfIndividualsSelected; i++) {
                        int index = (int) fitness[i][0];
                        newPopulation[i] = population[index].clone();
                    }

                } else if (selection ==
SelectionOperationType.roulette_wheel) {
                    // Roulette Wheel Selection
                    double[][] fitness = new
double[populationSize][2];//[index | fitness]
                    double totalFitness = 0;
                    for (int i = 0; i < populationSize; i++) {
                        fitness[i][0] = i;
                        fitness[i][1] = population[i].getFitness();
                        totalFitness += fitness[i][1];
                    }
                    // cumulative probability
                    double top = 0;
                    double[] cumulativeProbability = new
double[populationSize];
                    for (int i = 0; i < populationSize; i++) {
                        double relativeFitness = fitness[i][1] /
totalFitness;
                        top += relativeFitness;
                        cumulativeProbability[i] = top;
                    }

                    // save selected individual
                    for (int i = 0; i < numberOfIndividualsSelected; i++) {
                        // random
                        double rs = Random.getRandomUniform();
                        for (int j = 0; j < populationSize; j++) {
                            if (cumulativeProbability[j] >= rs) {
                                newPopulation[i] = population[j].clone();
                                break;
                            }
                        }

                    }

                }

                // CROSSOVER-----------------------------------------------
-----
                // The Operation for crossover is Partially Mapped
Crossover (PMX)
```

```
                int k = numberOfIndividualsSelected;
                while (k < populationSize) {
                    // select random parents
                    int indexParent1 = Random.getRandomBetween(0,
numberOfIndividualsSelected - 1);
                    int indexParent2 = indexParent1;
                    while (indexParent2 == indexParent1) {
                        indexParent2 = Random.getRandomBetween(0,
numberOfIndividualsSelected - 1);
                    }
                    int[] parent1 =
newPopulation[indexParent1].getChromosome()[0];
                    int[] parent2 =
newPopulation[indexParent2].getChromosome()[0];
                    // random two crossover points
                    int genSize = parent1.length;
                    int point1 = Random.getRandomBetween(0, genSize - 1);
                    int point2 = point1;
                    while (point2 == point1) {
                        point2 = Random.getRandomBetween(0, genSize - 1);
                    }
                    if (point1 > point2) {
                        int temp = point1;
                        point1 = point2;
                        point2 = temp;
                    }
                    // prepare offspring
                    int[] offspring1 = new int[genSize];
                    int[] offspring2 = new int[genSize];
                    for (int i = 0; i < genSize; i++) {
                        offspring1[i] = parent1[i];
                        offspring2[i] = parent2[i];
                    }
                    // do PMX
                    for (int i = point1; i <= point2; i++) {
                        int value1 = offspring1[i];
                        int value2 = offspring2[i];
                        for (int j = 0; j < genSize; j++) {
                            if (offspring1[j] == value2) {
                                offspring1[j] = value1;
                            }
                            if (offspring2[j] == value1) {
                                offspring2[j] = value2;
                            }
                        }
                        // crossover
                        offspring1[i] = value2;
                        offspring2[i] = value1;
                    }
                    //end of PMX
                    // set new individu as ofspring
                    // offspring_1
                    if (k < populationSize) {
                        newPopulation[k] = new Individual();
                        newPopulation[k].setMDMTSP(mdmtsp);
                        int[][] chromosome = new int[2][genSize];
                        chromosome[0] = offspring1;
                        chromosome[1] =
newPopulation[indexParent1].getChromosome()[1];
                        newPopulation[k].setChromosome(chromosome);
                        newPopulation[k].calculateFitness();
                        k++;
                    }
                    // offspring_2
                    if (k < populationSize) {
                        newPopulation[k] = new Individual();
                        newPopulation[k].setMDMTSP(mdmtsp);
                        int[][] chromosome = new int[2][genSize];
                        chromosome[0] = offspring2;
                        chromosome[1] =
newPopulation[indexParent2].getChromosome()[1];
                        newPopulation[k].setChromosome(chromosome);
                        newPopulation[k].calculateFitness();
                        k++;
                    }
                }

                // MUTATION--------------------------------------------------
-----
                for (int i = 0; i < populationSize; i++) {
                    double rm = Random.getRandomUniform();
                    if (rm > mutationRate) {
                        Individual mutant = newPopulation[i].clone();
                        // customerMutation
                        int index = Random.getRandomBetween(0,
mutant.getChromosome()[0].length - 1);
                        mutant.customerMutation(index);
                        mutant.calculateFitness();
                        if (mutant.getFitness() >
newPopulation[i].getFitness()) {
                            newPopulation[i] = mutant.clone();
                        }
                        // salesmanMutation
                        index = Random.getRandomBetween(0, 
mutant.getOriginDepot().length - 1);
                        mutant.salesmanMutation(index);
                        mutant.calculateFitness();
                        if (mutant.getFitness() > 
newPopulation[i].getFitness()) {
                            newPopulation[i] = mutant.clone();
                        }
                    }
                }

                // SET NEW POPULATION
                for (int i = 0; i < populationSize; i++) {
                    population[i] = newPopulation[i].clone();
                    // ELITISM
                    if (population[i].getFitness() > bestFitness) {
                        bestSolution = population[i];
                        bestFitness = bestSolution.getFitness();
                    }
                }
            }//end of evolution
        }
    }

}
```

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;
    }
}
```