# **LAMPIRAN**

## **A. Kode Sumber *CloudSimulationGA.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.File; import java.io.FileNotFoundException; import java.io.FileReader; import java.io.FileWriter; import java.text.DecimalFormat; import java.util.ArrayList; import java.util.Calendar; import java.util.DoubleSummaryStatistics; import java.util.LinkedList; import java.util.List; import java.util.Scanner; import java.util.stream.DoubleStream;  import org.cloudbus.cloudsim.Cloudlet; import org.cloudbus.cloudsim.CloudletSchedulerSpaceShared; import org.cloudbus.cloudsim.DatacenterBroker; import org.cloudbus.cloudsim.DatacenterCharacteristics; import org.cloudbus.cloudsim.Log; import org.cloudbus.cloudsim.Pe; import org.cloudbus.cloudsim.Storage; import org.cloudbus.cloudsim.UtilizationModel; import org.cloudbus.cloudsim.UtilizationModelFull; import org.cloudbus.cloudsim.Vm; import org.cloudbus.cloudsim.VmSchedulerTimeShared; import org.cloudbus.cloudsim.core.CloudSim; import org.cloudbus.cloudsim.power.PowerHost; import org.cloudbus.cloudsim.power.PowerDatacenter; import org.cloudbus.cloudsim.power.PowerHostUtilizationHistory; import org.cloudbus.cloudsim.power.PowerVmAllocationPolicySimple; import org.cloudbus.cloudsim.power.models.PowerModelLinear; import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple; import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple; import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;  public class CloudSimulationGA {   private static PowerDatacenter datacenter1, datacenter2, datacenter3, datacenter4, datacenter5, datacenter6;  /\*\* The cloudlet list. \*/  private static List<Cloudlet> cloudletList;   /\*\* The vmlist. \*/  private static List<Vm> vmlist;   private static List<Vm> createVM(int userId, int vms) {   //Creates a container to store VMs.   //This list is passed to the broker later  LinkedList<Vm> list = new LinkedList<Vm>();   //VM Parameters  long size = 10000; //Image size (MB)  int[] ram = {512,1024,2048}; //VM memory (MB)  int[] mips = {400,500,600}; //VM processing power (MIPS)  long bw = 1000; //VM bandwidth  int pesNumber = 1; //Number of cpus  String vmm = "Xen"; //VMM name    //create VMs  Vm[] vm = new Vm[vms];   for(int i=0;i<vms;i++){  //For loop to create a VM with a time shared scheduling policy for cloudlets:  vm[i] = new Vm(i, userId, mips[i%3], pesNumber, ram[i%3], bw, size, vmm, new CloudletSchedulerSpaceShared());  list.add(vm[i]);  }    return list;  }   private static ArrayList<Double> getSeedValue(int cloudletcount){    // Creating an arraylist to store Cloudlet Datasets  ArrayList<Double> seed = new ArrayList<Double>();  Log.printLine(System.getProperty("user.dir")+ "/SDSCDataset.txt");    try{  // Opening and scanning the file  File fobj = new File(System.getProperty("user.dir")+ "/SDSCDataset.txt");  java.util.Scanner readFile = new java.util.Scanner(fobj);    while(readFile.hasNextLine() && cloudletcount>0)  {  // Adding the file to the arraylist  seed.add(readFile.nextDouble());  cloudletcount--;  }  readFile.close();    } catch (FileNotFoundException e) {  e.printStackTrace();  }    return seed;  }   private static List<Cloudlet> createCloudlet(int userId, int cloudlets){    ArrayList<Double> randomSeed = getSeedValue(cloudlets);    // Creates a container to store Cloudlets  LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();   //Cloudlet parameters  long length = 0; // Cloudlet length (MI) - 0 for SDSC  //long length = 1000; // Cloudlet length (MI) - 1000 for Random Dataset  long fileSize = 300; // Cloudlet file size (MB)  long outputSize = 300; // Cloudlet file size (MB)  int pesNumber = 1; // Cloudlet CPU needed to process  UtilizationModel utilizationModel = new UtilizationModelFull();   Cloudlet[] cloudlet = new Cloudlet[cloudlets];   for(int i=0;i<cloudlets;i++){  long finalLen = length + Double.valueOf(randomSeed.get(i)).longValue();  // Creating the cloudlet with all the parameter listed  cloudlet[i] = new Cloudlet(i, finalLen, pesNumber, fileSize, outputSize, utilizationModel, utilizationModel, utilizationModel);    // setting the owner of these Cloudlets  cloudlet[i].setUserId(userId);  list.add(cloudlet[i]);  }   return list;  }   /\*\*  \* Creates main() to run this example  \*/  public static void main(String[] args) {  Log.printLine("Starting Cloud Simulation Example...");   try {  // First step: Initialize the CloudSim package. It should be called  // before creating any entities.  int num\_user = 1; // Number of grid users  Calendar calendar = Calendar.getInstance();  boolean trace\_flag = false; // Mean trace events  int hostId=0; // Starting host ID  BufferedWriter outputWriter = null;  outputWriter = new BufferedWriter(new FileWriter("filename.txt")); //Save output to text file  int vmNumber = 54; // The number of VMs created  int cloudletNumber = 7395; // The number of Tasks created   // Initialize the CloudSim library  CloudSim.init(num\_user, calendar, trace\_flag);      //Second step: Create Data Centers  //Datacenters are the resource providers in CloudSim. We need at least one of them to run a CloudSim simulation  datacenter1 = createDatacenter("DataCenter\_1", hostId);  hostId = 3;  datacenter2 = createDatacenter("DataCenter\_2", hostId);  hostId = 6;  datacenter3 = createDatacenter("DataCenter\_3", hostId);  hostId = 9;  datacenter4 = createDatacenter("DataCenter\_4", hostId);  hostId = 12;  datacenter5 = createDatacenter("DataCenter\_5", hostId);  hostId = 15;  datacenter6 = createDatacenter("DataCenter\_6", hostId);      //Third step: Create Broker  DatacenterBroker broker = createBroker();  int brokerId = broker.getId();     //Fourth step: Create VMs and Cloudlets   vmlist = createVM(brokerId,vmNumber); //Creating vms  cloudletList = createCloudlet(brokerId,cloudletNumber); // Creating cloudlets      //Fifth step: Send VMs and Cloudlets to broker   broker.submitVmList(vmlist);  broker.submitCloudletList(cloudletList);      //Sixth step: Use Genetic Algorithm  int chromosomeLength = 9; //number of genes inside a chromosome  int cloudletLoopingNumber = cloudletNumber/vmNumber - 1; //number of iteration needed to process the dataset    for (int cloudletIterator=0; cloudletIterator<=cloudletLoopingNumber; cloudletIterator++)  {  System.out.println("Cloudlet Iteration Number " + cloudletIterator);  for (int dataCenterIterator = 1; dataCenterIterator <= 6; dataCenterIterator++)  {  // Initialize Genetic Algorithm  GeneticAlgorithm ga = new GeneticAlgorithm(20, 0.3, 0.95, 2, cloudletList, vmlist);    // Initialize population  System.out.println("Datacenter " + dataCenterIterator + " Population Initialization");  Population population = ga.initPopulation(chromosomeLength, dataCenterIterator);    // Evaluate population  ga.evalPopulation(population, dataCenterIterator, cloudletIterator);    // Genetic Algorithm Iteration  int iteration = 1;  while (iteration <= 15)   {   // get fittest individual from population in every iteration  Individual fit = population.getFittest(0);    System.out.print("Fittest: ");  for(int j=0;j<9;j++) {  System.out.print(fit.chromosome[j] + " ");  }  System.out.println(" fitness => " + fit.getFitness());   // Apply crossover  population = ga.crossoverPopulation(population, dataCenterIterator);   // Apply mutation  population = ga.mutatePopulation(population, dataCenterIterator);   // Evaluate population  ga.evalPopulation(population, dataCenterIterator, cloudletIterator);   // Increment the current generation  iteration++;  }  // Get the fittest individual from Genetic Algorithm System.out.println("Best solution of GA: " + population.getFittest(0) + " For Datacenter-" + dataCenterIterator);  System.out.println("Highest Fitness Achieved: " + population.getFittest(0).getFitness());    // Assign Cloudlet to their respective VMs according to the fittest individual's chromosome  //outputWriter.write("{"); for (int assigner=0+(dataCenterIterator-1)\*9 + cloudletIterator\*54; assigner<9+(dataCenterIterator-1)\*9 + cloudletIterator\*54; assigner++)  {  broker.bindCloudletToVm(assigner, population.getFittest(0).getGene(assigner%9));    outputWriter.write(Long.toString(population.getFittest(0).getGene(assigner%9)%9)); // Print Assigned VM ID %  outputWriter.write(" ");  //if (assigner%9<8)  //{  // outputWriter.write(",");  //}  }   outputWriter.newLine();  //outputWriter.write("}");  }  }      // Seventh step: Starts the simulation  CloudSim.startSimulation();   outputWriter.flush();  outputWriter.close();  // Final step: Print results when simulation is over  List<Cloudlet> newList = broker.getCloudletReceivedList();    CloudSim.stopSimulation();   printCloudletList(newList);   Log.printLine("Cloud Simulation Example finished!");  }  catch (Exception e)  {  e.printStackTrace();  Log.printLine("The simulation has been terminated due to an unexpected error");  }  }       private static PowerDatacenter createDatacenter(String name, int hostId){   // Here are the steps needed to create a PowerDatacenter:  // 1. We need to create a list to store one or more machines  List<PowerHost> hostList = new ArrayList<PowerHost>();     // 2. A Machine contains one or more PEs or CPUs/Cores. Therefore, should  // create a list to store these PEs before creating a Machine.  List<Pe> peList1 = new ArrayList<Pe>();  List<Pe> peList2 = new ArrayList<Pe>();  List<Pe> peList3 = new ArrayList<Pe>();   int mipsunused= 300; // Unused core, only 3 cores will be able to process Cloudlets for this simulation  int mips1 = 400; // The MIPS Must be bigger than the VMs  int mips2 = 500;  int mips3 = 600;     // 3. Create PEs and add these into the list.  //for a quad-core machine, a list of 4 PEs is required:  peList1.add(new Pe(0, new PeProvisionerSimple(mips1))); // need to store Pe id and MIPS Rating, Must be bigger than the VMs  peList1.add(new Pe(1, new PeProvisionerSimple(mips1)));  peList1.add(new Pe(2, new PeProvisionerSimple(mips1)));  peList1.add(new Pe(3, new PeProvisionerSimple(mipsunused)));  peList2.add(new Pe(4, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(5, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(6, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(7, new PeProvisionerSimple(mipsunused)));  peList3.add(new Pe(8, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(9, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(10, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(11, new PeProvisionerSimple(mipsunused)));     //4. Create Hosts with its id and list of PEs and add them to the list of machines  int ram = 128000 ; //Host memory (MB), Must be bigger than the VMs  long storage = 1000000; //Host storage (MB)  int bw = 10000; //Host bandwidth  int maxpower = 117; // Host Max Power  int staticPowerPercentage = 50; // Host Static Power Percentage   hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList1,   new VmSchedulerTimeShared(peList1),  new PowerModelLinear(maxpower, staticPowerPercentage)));  hostId++;    hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList2,   new VmSchedulerTimeShared(peList2),  new PowerModelLinear(maxpower, staticPowerPercentage)));  hostId++;    hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList3,   new VmSchedulerTimeShared(peList3),  new PowerModelLinear(maxpower, staticPowerPercentage)));    // 5. Create a DatacenterCharacteristics object that stores the  // properties of a data center: architecture, OS, list of  // Machines, allocation policy: time- or space-shared, time zone  // and its price (G$/Pe time unit).  String arch = "x86"; // System architecture  String os = "Linux"; // Operating system  String vmm = "Xen"; // Name  double time\_zone = 10.0; // Time zone this resource located  double cost = 3.0; // The cost of using processing in this resource  double costPerMem = 0.05; // The cost of using memory in this resource  double costPerStorage = 0.1; // The cost of using storage in this resource  double costPerBw = 0.1; // The cost of using bw in this resource  LinkedList<Storage> storageList = new LinkedList<Storage>();   DatacenterCharacteristics characteristics = new DatacenterCharacteristics(  arch, os, vmm, hostList, time\_zone, cost, costPerMem, costPerStorage, costPerBw);    // 6. Finally, we need to create a PowerDatacenter object.  PowerDatacenter datacenter = null;  try {  datacenter = new PowerDatacenter(name, characteristics, new PowerVmAllocationPolicySimple(hostList), storageList, 9);   } catch (Exception e) {  e.printStackTrace();  }   return datacenter;  }       private static DatacenterBroker createBroker(){   DatacenterBroker broker = null;  try {  broker = new DatacenterBroker("Broker");  } catch (Exception e) {  e.printStackTrace();  return null;  }  return broker;  }      /\*\*  \* Prints the Cloudlet objects  \* @param list list of Cloudlets  \* @throws FileNotFoundException   \*/  private static void printCloudletList(List<Cloudlet> list) throws FileNotFoundException {    // Initializing the printed output to zero  int size = list.size();  Cloudlet cloudlet = null;   String indent = " ";  Log.printLine();  Log.printLine("========== OUTPUT ==========");  Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +  "Data center ID" + indent + "VM ID" + indent + "Time"   + indent + "Start Time" + indent + "Finish Time"+ indent + "Waiting Time");    double waitTimeSum = 0.0;  double CPUTimeSum = 0.0;  int totalValues = 0;  DecimalFormat dft = new DecimalFormat("###.##");    double response\_time[] = new double[size];    // Printing all the status of the Cloudlets  for (int i = 0; i < size; i++) {  cloudlet = list.get(i);  Log.print(cloudlet.getCloudletId() + indent + indent);    if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){  Log.print("SUCCESS");  CPUTimeSum = CPUTimeSum + cloudlet.getActualCPUTime();  waitTimeSum = waitTimeSum + cloudlet.getWaitingTime();  Log.printLine(indent + indent + indent + (cloudlet.getResourceId()-1) + indent + indent + indent + cloudlet.getVmId() +  indent + indent + dft.format(cloudlet.getActualCPUTime()) + indent + indent + dft.format(cloudlet.getExecStartTime())+  indent + indent + dft.format(cloudlet.getFinishTime())+ indent + indent + indent + dft.format(cloudlet.getWaitingTime()));  totalValues++;     response\_time[i] = cloudlet.getActualCPUTime();  }  }  DoubleSummaryStatistics stats = DoubleStream.of(response\_time).summaryStatistics();    // Show the parameters and print them out  Log.printLine();  System.out.println("min = " + stats.getMin());  System.out.println("Response\_Time: " + CPUTimeSum/totalValues);    Log.printLine();  Log.printLine("TotalCPUTime : " + CPUTimeSum);  Log.printLine("TotalWaitTime : " + waitTimeSum);  Log.printLine("TotalCloudletsFinished : " + totalValues);  Log.printLine();  Log.printLine();     //Average Cloudlets Finished   Log.printLine("AverageCloudletsFinished : " + (CPUTimeSum/ totalValues));   //Average Start Time  double totalStartTime =0.0;  for (int i = 0; i < size; i++) {  totalStartTime = cloudletList.get(i).getExecStartTime();   }  double avgStartTime = totalStartTime/size;  System.out.println("Average StartTime: " + avgStartTime );    //Average Execution Time  double ExecTime =0.0;  for (int i = 0; i < size; i++) {  ExecTime = cloudletList.get(i).getActualCPUTime();   }  double avgExecTime = ExecTime/size;  System.out.println("Average Execution Time: " + avgExecTime );    //Average Finish Time  double totalTime =0.0;  for (int i = 0; i < size; i++) {  totalTime = cloudletList.get(i).getFinishTime();   }  double avgTAT = totalTime/size;  System.out.println("Average FinishTime: " + avgTAT );    //Average Waiting Time  double avgWT = cloudlet.getWaitingTime()/size;  System.out.println("Average Waiting time: " + avgWT);    Log.printLine();  Log.printLine();    //Throughput  double maxFT =0.0;  for (int i = 0; i < size; i++) {  double currentFT = cloudletList.get(i).getFinishTime();   if (currentFT > maxFT) {  maxFT = currentFT;  }   }  double throughput = size/maxFT;  System.out.println("Throughput: " + throughput );    //Makespan  double makespan =0.0;  double makespan\_total = makespan + cloudlet.getFinishTime();  System.out.println("Makespan: " + makespan\_total);    //Imbalance Degree  double degree\_of\_imbalance = (stats.getMax() - stats.getMin())/(CPUTimeSum/ totalValues);  System.out.println("Imbalance Degree: " + degree\_of\_imbalance);    //Scheduling Length  double scheduling\_length = waitTimeSum + makespan\_total;  Log.printLine("Total Scheduling Length: " + scheduling\_length);    //CPU Resource Utilization  double resource\_utilization = (CPUTimeSum / (makespan\_total \* 54)) \* 100;  Log.printLine("Resource Utilization: " + resource\_utilization);    //Energy Consumption  Log.printLine(String.format("Total Energy Consumption: %.2f kWh", (datacenter1.getPower() + datacenter2.getPower()+ datacenter3.getPower()+ datacenter4.getPower()+ datacenter5.getPower()+ datacenter6.getPower())/ (3600\*1000)));  }   } |

## **B. Kode Sumber *CloudSimulationANN.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.File; import java.io.FileNotFoundException; import java.io.FileReader; import java.io.FileWriter; import java.io.PrintStream; import java.text.DecimalFormat; import java.util.ArrayList; import java.util.Calendar; import java.util.DoubleSummaryStatistics; import java.util.LinkedList; import java.util.List; import java.util.Scanner; import java.util.stream.DoubleStream;  import org.cloudbus.cloudsim.Cloudlet; import org.cloudbus.cloudsim.CloudletSchedulerSpaceShared; import org.cloudbus.cloudsim.DatacenterBroker; import org.cloudbus.cloudsim.DatacenterCharacteristics; import org.cloudbus.cloudsim.Log; import org.cloudbus.cloudsim.Pe; import org.cloudbus.cloudsim.Storage; import org.cloudbus.cloudsim.UtilizationModel; import org.cloudbus.cloudsim.UtilizationModelFull; import org.cloudbus.cloudsim.Vm; import org.cloudbus.cloudsim.VmSchedulerTimeShared; import org.cloudbus.cloudsim.core.CloudSim; import org.cloudbus.cloudsim.power.PowerHost; import org.cloudbus.cloudsim.power.PowerDatacenter; import org.cloudbus.cloudsim.power.PowerHostUtilizationHistory; import org.cloudbus.cloudsim.power.PowerVmAllocationPolicySimple; import org.cloudbus.cloudsim.power.models.PowerModelLinear; import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple; import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple; import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple; import org.encog.ml.data.MLData; import org.encog.ml.data.MLDataPair; import org.encog.ml.data.MLDataSet; import org.encog.ml.data.basic.BasicMLData; import org.encog.ml.data.basic.BasicMLDataSet; import org.encog.neural.networks.BasicNetwork; import org.encog.neural.networks.training.lma.LevenbergMarquardtTraining; import org.encog.neural.networks.training.propagation.back.Backpropagation; import org.encog.neural.networks.training.propagation.manhattan.ManhattanPropagation; import org.encog.neural.networks.training.propagation.quick.QuickPropagation; import org.encog.neural.networks.training.propagation.resilient.ResilientPropagation; import org.encog.neural.networks.training.propagation.scg.ScaledConjugateGradient; import org.encog.persist.EncogDirectoryPersistence; import org.encog.util.arrayutil.NormalizationAction; import org.encog.util.arrayutil.NormalizedField;  public class CloudSimulationANN {    // Double Array to hold the raw length data  public static double LENGTH\_RAW\_DATA[][];    // Double Array to hold the raw target data  public static double TARGET\_RAW\_DATA[][];     public static double[][] Reading2DArrayFromFileLength()  {  Scanner scannerLength;  int rows = 163; // Number of rows to be scanned  int columns = 9; // Number of columns to be scanned  double [][] arrayLength = new double[rows][columns];    try   { scannerLength = new Scanner(new BufferedReader(new FileReader(System.getProperty("user.dir")+ "/test/TestLength-SDSC.txt")));  while(scannerLength.hasNextLine()) {  for (int i=0; i<arrayLength.length; i++) {  String[] line = scannerLength.nextLine().trim().split(" "); // Splitting the dataset  for (int j=0; j<line.length; j++){  arrayLength[i][j] = Integer.parseInt(line[j]); // Parsing String to Integer and save to array  }  }  }  } catch (FileNotFoundException e)   {  e.printStackTrace();  }  return arrayLength;  }    public static double[][] Reading2DArrayFromFileTarget()  {  Scanner scannerTarget;  int rows = 163; // Number of rows to be scanned  int columns = 9; // Number of columns to be scanned  double [][] arrayTarget = new double[rows][columns];    try   { scannerTarget = new Scanner(new BufferedReader(new FileReader(System.getProperty("user.dir")+ "/test/TestTarget-SDSC.txt")));  while(scannerTarget.hasNextLine()) {  for (int i=0; i<arrayTarget.length; i++) {  String[] line = scannerTarget.nextLine().trim().split(" "); // Splitting the dataset  for (int j=0; j<line.length; j++) {  arrayTarget[i][j] = Integer.parseInt(line[j]); // Parsing String to Integer and save to array  }  }  }  } catch (FileNotFoundException e)   {  e.printStackTrace();  }  return arrayTarget;  }    private static PowerDatacenter datacenter1, datacenter2, datacenter3, datacenter4, datacenter5, datacenter6;  /\*\* The cloudlet list. \*/  private static List<Cloudlet> cloudletList;   /\*\* The vmlist. \*/  private static List<Vm> vmlist;   private static List<Vm> createVM(int userId, int vms) {   //Creates a container to store VMs.   //This list is passed to the broker later  LinkedList<Vm> list = new LinkedList<Vm>();   //VM Parameters  long size = 10000; //Image size (MB)  int[] ram = {512,1024,2048}; //VM memory (MB)  int[] mips = {400,500,600}; //VM processing power (MIPS)  long bw = 1000; //VM bandwidth  int pesNumber = 1; //Number of cpus  String vmm = "Xen"; //VMM name    //create VMs  Vm[] vm = new Vm[vms];   for(int i=0;i<vms;i++){  //For loop to create a VM with a time shared scheduling policy for cloudlets: vm[i] = new Vm(i, userId, mips[i%3], pesNumber, ram[i%3], bw, size, vmm, new CloudletSchedulerSpaceShared());  list.add(vm[i]);  }    return list;  }   private static ArrayList<Double> getSeedValue(int cloudletcount){    // Creating an arraylist to store Cloudlet Datasets  ArrayList<Double> seed = new ArrayList<Double>();  Log.printLine(System.getProperty("user.dir")+ "/dataset/SDSCDatasetANN.txt");    try{  // Opening and scanning the file  File fobj = new File(System.getProperty("user.dir")+ "/dataset/SDSCDatasetANN.txt");  java.util.Scanner readFile = new java.util.Scanner(fobj);    while(readFile.hasNextLine() && cloudletcount>0)  {  // Adding the file to the arraylist  seed.add(readFile.nextDouble());  cloudletcount--;  }  readFile.close();    } catch (FileNotFoundException e) {  e.printStackTrace();  }    return seed;  }   private static List<Cloudlet> createCloudlet(int userId, int cloudlets){    ArrayList<Double> randomSeed = getSeedValue(cloudlets);    // Creates a container to store Cloudlets  LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();   //Cloudlet parameters  long length = 0; // Cloudlet length (MI) - 0 for SDSC  //long length = 1000; // Cloudlet length (MI) - 1000 for Random Dataset  long fileSize = 300; // Cloudlet file size (MB)  long outputSize = 300; // Cloudlet file size (MB)  int pesNumber = 1; // Cloudlet CPU needed to process  UtilizationModel utilizationModel = new UtilizationModelFull();   Cloudlet[] cloudlet = new Cloudlet[cloudlets];   for(int i=0;i<cloudlets;i++){  long finalLen = length + Double.valueOf(randomSeed.get(i)).longValue();  // Creating the cloudlet with all the parameter listed cloudlet[i] = new Cloudlet(i, finalLen, pesNumber, fileSize, outputSize, utilizationModel, utilizationModel, utilizationModel);    // setting the owner of these Cloudlets  cloudlet[i].setUserId(userId);  list.add(cloudlet[i]);  }   return list;  }   /\*\*  \* Creates main() to run this example  \*/  public static void main(String[] args) {  Log.printLine("Starting Cloud Simulation Example...");   try {  // First step: Initialize the CloudSim package. It should be called  // before creating any entities.  int num\_user = 1; // Number of grid users  Calendar calendar = Calendar.getInstance();  boolean trace\_flag = false; // Mean trace events  int hostId=0; // Starting host ID  int vmNumber = 54; // The number of VMs created  int cloudletNumber = 1479; // The number of Tasks created   // Initialize the CloudSim library  CloudSim.init(num\_user, calendar, trace\_flag);      //Second step: Create Data Centers //Datacenters are the resource providers in CloudSim. We need at least one of them to run a CloudSim simulation  datacenter1 = createDatacenter("DataCenter\_1", hostId);  hostId = 3;  datacenter r2 = createDatacenter("DataCenter\_2", hostId);  hostId = 6;  datacenter3 = createDatacenter("DataCenter\_3", hostId);  hostId = 9;  datacenter4 = createDatacenter("DataCenter\_4", hostId);  hostId = 12;  datacenter5 = createDatacenter("DataCenter\_5", hostId);  hostId = 15;  datacenter6 = createDatacenter("DataCenter\_6", hostId);      //Third step: Create Broker  DatacenterBroker broker = createBroker();  int brokerId = broker.getId();     //Fourth step: Create VMs and Cloudlets   vmlist = createVM(brokerId,vmNumber); //Creating vms  cloudletList = createCloudlet(brokerId,cloudletNumber); // Creating cloudlets      //Fifth step: Send VMs and Cloudlets to broker   broker.submitVmList(vmlist);  broker.submitCloudletList(cloudletList);      //Sixth step: Use ANN BasicNetwork network = (BasicNetwork)EncogDirectoryPersistence.loadObject(new File("ANNscheduler-SDSC.EG"));  LENGTH\_RAW\_DATA = Reading2DArrayFromFileLength();  TARGET\_RAW\_DATA = Reading2DArrayFromFileTarget();    // Creating a normalization rules //NormalizedField input = new NormalizedField(NormalizationAction.Normalize, null, 50000, 10000, 1, 0); //for Random Dataset NormalizedField input = new NormalizedField(NormalizationAction.Normalize, null, 8790000, 0, 1, 0); //for SDSC  NormalizedField output = new NormalizedField(NormalizationAction.Normalize, null, 10, 0, 1, 0);    // Doing normalization to the Input  for (int m=0; m<LENGTH\_RAW\_DATA.length; m++) {  for (int n=0; n<9; n++) {  LENGTH\_RAW\_DATA[m][n] = input.normalize(LENGTH\_RAW\_DATA[m][n]);  }  }    // Doing normalization to the Output  for (int m=0; m<TARGET\_RAW\_DATA.length; m++) {  for (int n=0; n<9; n++) {  TARGET\_RAW\_DATA[m][n] = output.normalize(TARGET\_RAW\_DATA[m][n]);  }  }    // Create data  MLDataSet trainingSet = new BasicMLDataSet(LENGTH\_RAW\_DATA, TARGET\_RAW\_DATA);  int iterator = 0; //Iterator for the Cloudlet IDs  Long placeholderLong; //Placeholder to convert long to integer    // Testing the ANN  for(MLDataPair pair: trainingSet ) {  final MLData outputData = network.compute(pair.getInput());  System.out.println("");  System.out.println("For Input:");  for (int a=0 ; a<9; a++) {  System.out.print(Math.round(input.deNormalize(pair.getInput().getData(a))) + " ");  }  System.out.println("");  System.out.println("Actual Result:");  for (int b=0 ; b<9; b++) {  System.out.print(Math.round(output.deNormalize(outputData.getData(b))) + " ");  }  System.out.println("");  System.out.println("Assignment:");  for (int c=0 ; c<9; c++) {  placeholderLong = new Long(Math.round(output.deNormalize(outputData.getData(c))));  int VMidOutput = placeholderLong.intValue();  System.out.print(cloudletList.get(iterator\*9+c).getCloudletId() + " Assigned to ");  System.out.print((VMidOutput + iterator\*9)%54); broker.bindCloudletToVm(cloudletList.get(iterator\*9+c).getCloudletId(), (VMidOutput + iterator\*9)%54);  System.out.println("");  }  System.out.println("");  iterator++;  }      // Seventh step: Starts the simulation  CloudSim.startSimulation();     // Final step: Print results when simulation is over  List<Cloudlet> newList = broker.getCloudletReceivedList();    CloudSim.stopSimulation();   printCloudletList(newList);   Log.printLine("Cloud Simulation Example finished!");  }  catch (Exception e)  {  e.printStackTrace();  Log.printLine("The simulation has been terminated due to an unexpected error");  }  }       private static PowerDatacenter createDatacenter(String name, int hostId){   // Here are the steps needed to create a PowerDatacenter:  // 1. We need to create a list to store one or more machines  List<PowerHost> hostList = new ArrayList<PowerHost>();     // 2. A Machine contains one or more PEs or CPUs/Cores. Therefore, should  // create a list to store these PEs before creating a Machine.  List<Pe> peList1 = new ArrayList<Pe>();  List<Pe> peList2 = new ArrayList<Pe>();  List<Pe> peList3 = new ArrayList<Pe>();   int mipsunused= 300; // Unused core, only 3 cores will be able to process Cloudlets for this simulation  int mips1 = 400; // The MIPS Must be bigger than the VMs  int mips2 = 500;  int mips3 = 600;     // 3. Create PEs and add these into the list.  //for a quad-core machine, a list of 4 PEs is required: peList1.add(new Pe(0, new PeProvisionerSimple(mips1))); // need to store Pe id and MIPS Rating, Must be bigger than the VMs  peList1.add(new Pe(1, new PeProvisionerSimple(mips1)));  peList1.add(new Pe(2, new PeProvisionerSimple(mips1)));  peList1.add(new Pe(3, new PeProvisionerSimple(mipsunused)));  peList2.add(new Pe(4, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(5, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(6, new PeProvisionerSimple(mips2)));  peList2.add(new Pe(7, new PeProvisionerSimple(mipsunused)));  peList3.add(new Pe(8, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(9, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(10, new PeProvisionerSimple(mips3)));  peList3.add(new Pe(11, new PeProvisionerSimple(mipsunused)));     //4. Create Hosts with its id and list of PEs and add them to the list of machines  int ram = 128000 ; //Host memory (MB), Must be bigger than the VMs  long storage = 1000000; //Host storage (MB)  int bw = 10000; //Host bandwidth  int maxpower = 117; // Host Max Power  int staticPowerPercentage = 50; // Host Static Power Percentage   hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList1,   new VmSchedulerTimeShared(peList1),  new PowerModelLinear(maxpower, staticPowerPercentage)));  hostId++;    hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList2,   new VmSchedulerTimeShared(peList2),  new PowerModelLinear(maxpower, staticPowerPercentage)));  hostId++;    hostList.add(  new PowerHostUtilizationHistory(  hostId, new RamProvisionerSimple(ram),   new BwProvisionerSimple(bw),  storage,   peList3,   new VmSchedulerTimeShared(peList3),  new PowerModelLinear(maxpower, staticPowerPercentage)));    // 5. Create a DatacenterCharacteristics object that stores the  // properties of a data center: architecture, OS, list of  // Machines, allocation policy: time- or space-shared, time zone  // and its price (G$/Pe time unit).  String arch = "x86"; // System architecture  String os = "Linux"; // Operating system  String vmm = "Xen"; // Name  double time\_zone = 10.0; // Time zone this resource located  double cost = 3.0; // The cost of using processing in this resource  double costPerMem = 0.05; // The cost of using memory in this resource  double costPerStorage = 0.1; // The cost of using storage in this resource  double costPerBw = 0.1; // The cost of using bw in this resource  LinkedList<Storage> storageList = new LinkedList<Storage>();   DatacenterCharacteristics characteristics = new DatacenterCharacteristics(  arch, os, vmm, hostList, time\_zone, cost, costPerMem, costPerStorage, costPerBw);    // 6. Finally, we need to create a PowerDatacenter object.  PowerDatacenter datacenter = null;  try { datacenter = new PowerDatacenter(name, characteristics, new PowerVmAllocationPolicySimple(hostList), storageList, 9);  } catch (Exception e) {  e.printStackTrace();  }   return datacenter;  }       private static DatacenterBroker createBroker(){   DatacenterBroker broker = null;  try {  broker = new DatacenterBroker("Broker");  } catch (Exception e) {  e.printStackTrace();  return null;  }  return broker;  }      /\*\*  \* Prints the Cloudlet objects  \* @param list list of Cloudlets  \* @throws FileNotFoundException   \*/  private static void printCloudletList(List<Cloudlet> list) throws FileNotFoundException {    // Initializing the printed output to zero  int size = list.size();  Cloudlet cloudlet = null;   String indent = " ";  Log.printLine();  Log.printLine("========== OUTPUT ==========");  Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +  "Data center ID" + indent + "VM ID" + indent + "Time"   + indent + "Start Time" + indent + "Finish Time"+ indent + "Waiting Time");    double waitTimeSum = 0.0;  double CPUTimeSum = 0.0;  int totalValues = 0;  DecimalFormat dft = new DecimalFormat("###.##");    double response\_time[] = new double[size];    // Printing all the status of the Cloudlets  for (int i = 0; i < size; i++) {  cloudlet = list.get(i);  Log.print(cloudlet.getCloudletId() + indent + indent);    if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){  Log.print("SUCCESS");  CPUTimeSum = CPUTimeSum + cloudlet.getActualCPUTime();  waitTimeSum = waitTimeSum + cloudlet.getWaitingTime();  Log.printLine(indent + indent + indent + (cloudlet.getResourceId()-1) + indent + indent + indent + cloudlet.getVmId() +  indent + indent + dft.format(cloudlet.getActualCPUTime()) + indent + indent + dft.format(cloudlet.getExecStartTime())+  indent + indent + dft.format(cloudlet.getFinishTime())+ indent + indent + indent + dft.format(cloudlet.getWaitingTime()));  totalValues++;     response\_time[i] = cloudlet.getActualCPUTime();  }  }  DoubleSummaryStatistics stats = DoubleStream.of(response\_time).summaryStatistics();    // Show the parameters and print them out  Log.printLine();  System.out.println("min = " + stats.getMin());  System.out.println("Response\_Time: " + CPUTimeSum/totalValues);    Log.printLine();  Log.printLine("TotalCPUTime : " + CPUTimeSum);  Log.printLine("TotalWaitTime : " + waitTimeSum);  Log.printLine("TotalCloudletsFinished : " + totalValues);  Log.printLine();  Log.printLine();     //Average Cloudlets Finished   Log.printLine("AverageCloudletsFinished : " + (CPUTimeSum/ totalValues));   //Average Start Time  double totalStartTime =0.0;  for (int i = 0; i < size; i++) {  totalStartTime = cloudletList.get(i).getExecStartTime();   }  double avgStartTime = totalStartTime/size;  System.out.println("Average StartTime: " + avgStartTime );    //Average Execution Time  double ExecTime =0.0;  for (int i = 0; i < size; i++) {  ExecTime = cloudletList.get(i).getActualCPUTime();   }  double avgExecTime = ExecTime/size;  System.out.println("Average Execution Time: " + avgExecTime );    //Average Finish Time  double totalTime =0.0;  for (int i = 0; i < size; i++) {  totalTime = cloudletList.get(i).getFinishTime();   }  double avgTAT = totalTime/size;  System.out.println("Average FinishTime: " + avgTAT );    //Average Waiting Time  double avgWT = cloudlet.getWaitingTime()/size;  System.out.println("Average Waiting time: " + avgWT);    Log.printLine();  Log.printLine();    //Throughput  double maxFT =0.0;  for (int i = 0; i < size; i++) {  double currentFT = cloudletList.get(i).getFinishTime();   if (currentFT > maxFT) {  maxFT = currentFT;  }   }  double throughput = size/maxFT;  System.out.println("Throughput: " + throughput );    //Makespan  double makespan =0.0;  double makespan\_total = makespan + cloudlet.getFinishTime();  System.out.println("Makespan: " + makespan\_total);    //Imbalance Degree  double degree\_of\_imbalance = (stats.getMax() - stats.getMin())/(CPUTimeSum/ totalValues);  System.out.println("Imbalance Degree: " + degree\_of\_imbalance);    //Scheduling Length  double scheduling\_length = waitTimeSum + makespan\_total;  Log.printLine("Total Scheduling Length: " + scheduling\_length);    //CPU Resource Utilization  double resource\_utilization = (CPUTimeSum / (makespan\_total \* 54)) \* 100;  Log.printLine("Resource Utilization: " + resource\_utilization);    //Energy Consumption  Log.printLine(String.format("Total Energy Consumption: %.2f kWh", (datacenter1.getPower() + datacenter2.getPower()+ datacenter3.getPower()+ datacenter4.getPower()+ datacenter5.getPower()+ datacenter6.getPower())/ (3600\*1000)));  }   } |

## **C. Kode Sumber *GeneticAlgorithm.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  import java.util.List;  import org.cloudbus.cloudsim.Cloudlet; import org.cloudbus.cloudsim.Vm;  /\*\*  \* The GeneticAlgorithm class is our main abstraction for managing the  \* operations of the genetic algorithm. This class is meant to be  \* problem-specific, meaning that (for instance) the "calcFitness" method may  \* need to change from problem to problem.  \* \*/ public class GeneticAlgorithm {  private int populationSize;  public static List<Cloudlet> cloudletList;  public static List<Vm> vmlist;   /\*\*  \* Mutation rate is the fractional probability than an individual gene will  \* mutate randomly in a given generation. The range is 0.0-1.0, but is  \* generally small (on the order of 0.1 or less).  \*/  private double mutationRate;   /\*\*  \* Crossover rate is the fractional probability that two individuals will  \* "mate" with each other, sharing genetic information, and creating  \* offspring with traits of each of the parents. Like mutation rate the  \* range is 0.0-1.0 but small.  \*/  private double crossoverRate;   /\*\*  \* Elitism is the concept that the strongest members of the population  \* should be preserved from generation to generation. If an individual is  \* one of the elite, it will not be mutated or crossover.  \*/  private int elitismCount;   @SuppressWarnings("static-access")  public GeneticAlgorithm(int populationSize, double mutationRate, double crossoverRate, int elitismCount, List<Cloudlet> cloudletList, List<Vm> vmlist) {  this.populationSize = populationSize;  this.mutationRate = mutationRate;  this.crossoverRate = crossoverRate;  this.elitismCount = elitismCount;  this.cloudletList = cloudletList;  this.vmlist = vmlist;  }   /\*\*  \* Initialize population  \*   \* @param chromosomeLength  \* The length of the individual's chromosome  \* @return population The initial population generated  \*/  public Population initPopulation(int chromosomeLength, int dataCenterIterator) {  // Initialize population  Population population = new Population(this.populationSize, chromosomeLength, dataCenterIterator);  return population;  }    /\*\*  \* Get Poisson Distribution for the chances of VM to fail  \*   \* @param lambda  \* The fault rate at which a VM is going to fail  \* @param x  \* x is going to 0, the number of VM we want to fail   \* @return the chances of 0 VM to fail  \*/  static int factorial(int n)  {   if (n == 0)   return 1;   else   return(n \* factorial(n-1));   }     public static int getRandomPoisson(double lambda)   {  double L = Math.exp(-lambda);  double p = 1.0;  int k = 0;   do   {  k++;  p \*= Math.random();  } while (p > L);   return k - 1;  }    public static double getPoisson(double lambda, int x, int n)   {  double L = Math.exp(-lambda) \* n;  double p = Math.pow(lambda \* n, x);  int k = factorial(x);  double result;    result = L \* p / k;   return result;  }   /\*\*  \* Calculate fitness for an individual.  \*   \* In this case, the fitness score is very simple: it's the number of ones  \* in the chromosome. Don't forget that this method, and this whole  \* GeneticAlgorithm class, is meant to solve the problem in the "AllOnesGA"  \* class and example. For different problems, you'll need to create a  \* different version of this method to appropriately calculate the fitness  \* of an individual.  \*   \* @param individual  \* the individual to evaluate  \* @return double The fitness value for individual  \*/  public double calcFitness(Individual individual, int dataCenterIterator, int cloudletIteration) {   double totalExecutionTime = 0;  double mips = 0;  double failureRate = 0.04847468455;  int iterator=0;  dataCenterIterator = dataCenterIterator-1;    for (int i=0 + dataCenterIterator\*9 + cloudletIteration\*54; i<9 + dataCenterIterator\*9 + cloudletIteration\*54; i++)  {  int gene = individual.getGene(iterator);  if (gene%9 == 0 || gene%9 == 3 || gene%9 == 6)   {  mips = 400;  }else if (gene%9 == 1 || gene%9 == 4 || gene%9 == 7)   {  mips = 500;  }else if (gene%9 == 2 || gene%9 == 5 || gene%9 == 8)  {  mips = 600;  }else break;    //Log.printLine("Gene " + gene);  totalExecutionTime = totalExecutionTime + cloudletList.get(i).getCloudletLength() / mips;  iterator++;  }    int random = getRandomPoisson(failureRate);  double poisson=(getPoisson(failureRate, random, 9));   // Calculate fitness  double fitness = 0.90 \* (1/totalExecutionTime) + 0.1 \* (1/poisson);  //Log.printLine("Fitness " + fitness);   // Store fitness  individual.setFitness(fitness);   return fitness;  }   /\*\*  \* Evaluate the whole population  \*   \* Essentially, loop over the individuals in the population, calculate the  \* fitness for each, and then calculate the entire population's fitness. The  \* population's fitness may or may not be important, but what is important  \* Here is making sure that each individual gets evaluated.  \*   \* @param population  \* the population to evaluate  \*/  public void evalPopulation(Population population, int dataCenterIterator, int cloudletIteration) {    // Loop over population evaluating individuals and summing population fitness  double populationFitness=0;   for (Individual individual : population.getIndividuals()) {   double individualFitness = calcFitness(individual, dataCenterIterator, cloudletIteration);   individual.setFitness(individualFitness);  populationFitness+=individualFitness;    }  population.setPopulationFitness(populationFitness);   }   /\*\*  \* Check if population has met termination condition  \*   \* For this simple problem, we know what a perfect solution looks like, so  \* we can simply stop evolving once we've reached a fitness level.  \*   \* @param population  \* @return boolean True if termination condition met, otherwise, false  \*/    /\*\*  \* Select parent for crossover  \*   \* @param population  \* The population to select parent from  \* @return The individual selected as a parent  \*/  public Individual selectParent(Population population) {  // Get individuals  Individual individuals[] = population.getIndividuals();    // Spin roulette wheel  double populationFitness = population.getPopulationFitness();  double rouletteWheelPosition = Math.random() \* populationFitness;   // Find parent  double spinWheel = 0;  for (Individual individual : individuals) {  spinWheel += individual.getFitness();  if (spinWheel >= rouletteWheelPosition) {  return individual;  }  }  return individuals[population.size() - 1];  }   /\*\*  \* Apply crossover to population  \*   \* Crossover, more colloquially considered "mating", takes the population  \* and blends individuals to create new offspring. It is hoped that when two  \* individuals crossover that their offspring will have the strongest  \* qualities of each of the parents. Of course, it's possible that an  \* offspring will end up with the weakest qualities of each parent.  \*   \* This method considers both the GeneticAlgorithm instance's crossoverRate  \* and the elitismCount.  \*   \* The type of crossover we perform depends on the problem domain. We don't  \* want to create invalid solutions with crossover, so this method will need  \* to be changed for different types of problems.  \*   \* This particular crossover method selects random genes from each parent.  \*   \* @param population  \* The population to apply crossover to  \* @return The new population  \*/  public Population crossoverPopulation(Population population, int dataCenterIterator) {  // Create new population  Population newPopulation = new Population(population.size());   // Loop over current population by fitness  for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {  Individual parent1 = population.getFittest(populationIndex);   // Apply crossover to this individual?  if (this.crossoverRate > Math.random()&& populationIndex > this.elitismCount ) {  // Initialize offspring  Individual offspring = new Individual(parent1.getChromosomeLength(), dataCenterIterator);    // Find second parent  Individual parent2 = selectParent(population);   // Loop over genome  for (int geneIndex = 0; geneIndex < parent1.getChromosomeLength(); geneIndex++) {  // Use half of parent1's genes and half of parent2's genes  if (0.5 > Math.random()) {  offspring.setGene(geneIndex, parent1.getGene(geneIndex));  } else {  offspring.setGene(geneIndex, parent2.getGene(geneIndex));  }  }   // Add offspring to new population  newPopulation.setIndividual(populationIndex, offspring);  } else {  // Add individual to new population without applying crossover  newPopulation.setIndividual(populationIndex, parent1);  }  }   return newPopulation;  }   /\*\*  \* Apply mutation to population  \*   \* Mutation affects individuals rather than the population. We look at each  \* individual in the population, and if they're lucky enough (or unlucky, as  \* it were), apply some randomness to their chromosome. Like crossover, the  \* The type of mutation applied depends on the specific problem we're solving.  \* In this case, we simply randomly flip 0s to 1s and vice versa.  \*   \* This method will consider the GeneticAlgorithm instance's mutationRate  \* and elitismCount  \*   \* @param population  \* The population to apply mutation to  \* @return The mutated population  \*/  public Population mutatePopulation(Population population, int dataCenterIterator) {  // Initialize new population  Population newPopulation = new Population(this.populationSize);  dataCenterIterator = dataCenterIterator - 1;   // Loop over current population by fitness  for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {  Individual individual = population.getFittest(populationIndex);   // Loop over individual's genes  for (int geneIndex = 0; geneIndex < individual.getChromosomeLength(); geneIndex++) {  // Skip mutation if this is an elite individual  if (populationIndex > this.elitismCount) {  // Does this gene need mutation?  if (this.mutationRate > Math.random()) {  // Get new gene  int newGene=0 + 9 \* dataCenterIterator;  if (individual.getGene(geneIndex)%9 == 0) {  double r=Math.random();  if(r<0.125)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 1) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 2) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 3) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 4) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=5 + 9 \* dataCenterIterator;   }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 5) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=6 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 6) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=7 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 7) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=6 + 9 \* dataCenterIterator;  }  else  {  newGene=8 + 9 \* dataCenterIterator;  }  }  else if (individual.getGene(geneIndex)%9 == 8) {  double r=Math.random();  if(r<0.125)  {  newGene=0 + 9 \* dataCenterIterator;  }  else if(r>0.125 && r<0.250)  {  newGene=1 + 9 \* dataCenterIterator;  }  else if(r>0.250 && r<0.375)  {  newGene=2 + 9 \* dataCenterIterator;  }  else if(r>0.375 && r<0.5)  {  newGene=3 + 9 \* dataCenterIterator;  }  else if(r>0.5 && r<0.625)  {  newGene=4 + 9 \* dataCenterIterator;  }  else if(r>0.625 && r<0.75)  {  newGene=5 + 9 \* dataCenterIterator;  }  else if(r>0.75 && r<0.875)  {  newGene=6 + 9 \* dataCenterIterator;  }  else  {  newGene=7 + 9 \* dataCenterIterator;  }  }  // Mutate gene  individual.setGene(geneIndex, newGene);  }  }  }   // Add individual to population  newPopulation.setIndividual(populationIndex, individual);  }   // Return mutated population  return newPopulation;  }  } |

## **D. Kode Sumber *Population.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  import java.util.Arrays; import java.util.Comparator; import java.util.Random;  /\*\*  \* A population is an abstraction of a collection of individuals. The population  \* class is generally used to perform group-level operations on its individuals,  \* such as finding the strongest individuals, collecting stats on the population  \* as a whole, and selecting individuals to mutate or crossover.  \*/ public class Population {  public Individual population[];  public double populationFitness = -1;   /\*\*  \* Initializes blank population of individuals  \*   \* @param populationSize  \* The number of individuals in the population  \*/  public Population(int populationSize) {  // Initial population  this.population = new Individual[populationSize];  }   /\*\*  \* Initializes population of individuals  \*   \* @param populationSize  \* The number of individuals in the population  \* @param chromosomeLength  \* The size of each individual's chromosome  \*/  public Population(int populationSize, int chromosomeLength, int dataCenterIterator) {  // Initialize the population as an array of individuals  this.population = new Individual[populationSize];   // Create each individual in turn  for (int individualCount = 0; individualCount < populationSize; individualCount++) {  // Create an individual, initializing its chromosome to the given length  Individual individual = new Individual(chromosomeLength, dataCenterIterator);  // Add individual to population  this.population[individualCount] = individual;  }  }   /\*\*  \* Get individuals from the population  \*   \* @return individuals Individuals in population  \*/  public Individual[] getIndividuals() {  return this.population;  }   /\*\*  \* Find an individual in the population by its fitness  \*   \* This method lets you select an individual in order of its fitness. This  \* can be used to find the single strongest individual (eg, if you're  \* testing for a solution), but it can also be used to find weak individuals  \* (if you're looking to cull the population) or some of the strongest  \* individuals (if you're using "elitism").  \*   \* @param offset  \* The offset of the individual you want, sorted by fitness. 0 is  \* the strongest, population.length - 1 is the weakest.  \* @return individual Individual at offset  \*/  public Individual getFittest(int offset) {  // Order population by fitness  Arrays.sort(this.population, new Comparator<Individual>() {  @Override  public int compare(Individual o1, Individual o2) {  if (o1.getFitness() < o2.getFitness()) {  return 1;  } else if (o1.getFitness() > o2.getFitness()) {  return -1;  }  return 0;  }  });   // Return the fittest individual  return this.population[offset];  }   /\*\*  \* Set population's group fitness  \*   \* @param fitness  \* The population's total fitness  \*/  public void setPopulationFitness(double fitness) {  this.populationFitness = fitness;  }   /\*\*  \* Get population's group fitness  \*   \* @return populationFitness The population's total fitness  \*/  public double getPopulationFitness() {  return this.populationFitness;  }   /\*\*  \* Get population's size  \*   \* @return size The population's size  \*/  public int size() {  return this.population.length;  }   /\*\*  \* Set individual at offset  \*   \* @param individual  \* @param offset  \* @return individual  \*/  public Individual setIndividual(int offset, Individual individual) {  return population[offset] = individual;  }   /\*\*  \* Get individual at offset  \*   \* @param offset  \* @return individual  \*/  public Individual getIndividual(int offset) {  return population[offset];  }    /\*\*  \* Shuffles the population in-place  \*   \* @param void  \* @return void  \*/  public void shuffle() {  Random rnd = new Random();  for (int i = population.length - 1; i > 0; i--) {  int index = rnd.nextInt(i + 1);  Individual a = population[index];  population[index] = population[i];  population[i] = a;  }  } } |

## **E. Kode Sumber *Individual.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  /\*\*  \* An "Individual" represents a single candidate solution. The core piece of  \* information about an individual is its "chromosome", which is an encoding of  \* a possible solution to the problem at hand. A chromosome can be a string, an  \* array, a list, etc -- in this class, the chromosome is an integer array.   \*   \* An individual position in the chromosome is called a gene, and these are the  \* atomic pieces of the solution that can be manipulated or mutated. When the  \* chromosome is a string, as in this case, each character or set of characters  \* can be a gene.  \*   \* An individual also has a "fitness" score; this is a number that represents  \* how good a solution to the problem this individual is. The meaning of the  \* fitness score will vary based on the problem at hand.  \*/ public class Individual {  public int[] chromosome;  private double fitness = -1;   /\*\*  \* Initializes individual with specific chromosome  \*   \* @param chromosome  \* The chromosome to give individual  \*/  public Individual(int[] chromosome) {  // Create individual chromosome  this.chromosome = chromosome;  }   /\*\*  \* Initializes random individuals.  \*   \* This constructor assumes that the chromosome is made entirely of 0s and  \* 1s, which may not always be the case, so make sure to modify as  \* necessary. This constructor also assumes that a "random" chromosome means  \* simply picking random zeroes and ones, which also may not be the case  \* (for instance, in a traveling salesman problem, this would be an invalid  \* solution).  \*   \* @param chromosomeLength  \* The length of the individual's chromosome  \*/  public Individual(int chromosomeLength, int dataCenterIterator) {   this.chromosome = new int[chromosomeLength];  dataCenterIterator = dataCenterIterator-1;  int max = 8 + 9 \* dataCenterIterator;  int min = 0 + 9 \* dataCenterIterator;  int range = max - min + 1;    // generate random numbers within 0 to 8  for (int gene = 0; gene < chromosomeLength; gene++) {  int rand = (int)(Math.random() \* range) + min;  this.setGene(gene, rand);  }  }   /\*\*  \* Gets individual's chromosome  \*   \* @return The individual's chromosome  \*/  public int[] getChromosome() {  return this.chromosome;  }   /\*\*  \* Gets individual's chromosome length  \*   \* @return The individual's chromosome length  \*/  public int getChromosomeLength() {  return this.chromosome.length;  }   /\*\*  \* Set gene at offset  \*   \* @param gene  \* @param offset  \* @return gene  \*/  public void setGene(int offset, int gene) {  this.chromosome[offset] = gene;  }   /\*\*  \* Get gene at offset  \*   \* @param offset  \* @return gene  \*/  public int getGene(int offset) {  return this.chromosome[offset];  }   /\*\*  \* Store individual's fitness  \*   \* @param fitness  \* The individual's fitness  \*/  public void setFitness(double fitness) {  this.fitness = fitness;  }   /\*\*  \* Gets individual's fitness  \*   \* @return The individual's fitness  \*/  public double getFitness() {  return this.fitness;  }      /\*\*  \* Display the chromosome as a string.  \*   \* @return string representation of the chromosome  \*/  public String toString() {  String output = "";  for (int gene = 0; gene < this.chromosome.length; gene++) {  output += this.chromosome[gene];  }  return output;  } } |

## **F. Kode SumberANNTest*.Java***

|  |
| --- |
| package org.cloudbus.cloudsim.examples;  import java.io.BufferedReader; import java.io.File; import java.io.FileNotFoundException; import java.io.FileReader; import java.util.Scanner;  import org.encog.Encog; import org.encog.engine.network.activation.ActivationReLU; import org.encog.engine.network.activation.ActivationSigmoid; import org.encog.ml.data.MLData; import org.encog.ml.data.MLDataPair; import org.encog.ml.data.MLDataSet; import org.encog.ml.data.basic.BasicMLDataSet; import org.encog.neural.networks.BasicNetwork; import org.encog.neural.networks.layers.BasicLayer; import org.encog.neural.networks.training.propagation.back.Backpropagation; import org.encog.neural.networks.training.propagation.manhattan.ManhattanPropagation; import org.encog.neural.networks.training.propagation.quick.QuickPropagation; import org.encog.persist.EncogDirectoryPersistence; import org.encog.util.arrayutil.NormalizationAction; import org.encog.util.arrayutil.NormalizedField;  public class ANNTest {    // Double Array to hold the raw length data  public static double LENGTH\_RAW\_DATA[][];    // Double Array to hold the raw target data  public static double TARGET\_RAW\_DATA[][];     public static double[][] Reading2DArrayFromFileLength()  {  Scanner scannerLength;  int rows = 653; // Number of rows to be scanned  int columns = 9; // Number of columns to be scanned  double [][] arrayLength = new double[rows][columns];    try   {  scannerLength = new Scanner(new BufferedReader(new FileReader(System.getProperty("user.dir")+ "/train/DatasetLength-SDSC.txt")));  while(scannerLength.hasNextLine()) {  for (int i=0; i<arrayLength.length; i++) {  String[] line = scannerLength.nextLine().trim().split(" "); // Splitting the dataset  for (int j=0; j<line.length; j++) {  arrayLength[i][j] = Integer.parseInt(line[j]); // Parsing String to Integer and save to array  }  }  }  } catch (FileNotFoundException e)   {  e.printStackTrace();  }  return arrayLength;  }    public static double[][] Reading2DArrayFromFileTarget()  {  Scanner scannerTarget;  int rows = 653; // Number of rows to be scanned  int columns = 9; // Number of columns to be scanned  double [][] arrayTarget = new double[rows][columns];    try   {  scannerTarget = new Scanner(new BufferedReader(new FileReader(System.getProperty("user.dir")+ "/train/DatasetTarget-SDSC.txt")));  while(scannerTarget.hasNextLine()) {  for (int i=0; i<arrayTarget.length; i++) {  String[] line = scannerTarget.nextLine().trim().split(" "); // Splitting the dataset  for (int j=0; j<line.length; j++) {  arrayTarget[i][j] = Integer.parseInt(line[j]); // Parsing String to Integer and save to array  }  }  }  } catch (FileNotFoundException e)   {  e.printStackTrace();  }  return arrayTarget;  }    /\*\*  \* The main method.  \* @param args No arguments are used.  \*/  public static void main(final String args[]) {    // Saving the data scanned into the double arrays  LENGTH\_RAW\_DATA = Reading2DArrayFromFileLength();  TARGET\_RAW\_DATA = Reading2DArrayFromFileTarget();    // Create a neural network  BasicNetwork network = new BasicNetwork();  network.addLayer(new BasicLayer(null,true,9)); // 9 input nodes  network.addLayer(new BasicLayer(new ActivationReLU(),true,18)); // 18 hidden nodes  network.addLayer(new BasicLayer(new ActivationSigmoid(),false,9)); // 9 output nodes  network.getStructure().finalizeStructure();  network.reset();    // Creating a normalization rules  //NormalizedField input = new NormalizedField(NormalizationAction.Normalize, null, 50000, 10000, 1, 0); //for Random Dataset  NormalizedField input = new NormalizedField(NormalizationAction.Normalize, null, 8790000, 0, 1, 0); //for SDSC  NormalizedField output = new NormalizedField(NormalizationAction.Normalize, null, 10, 0, 1, 0);    // Doing normalization to the Input  for (int m=0; m<LENGTH\_RAW\_DATA.length; m++) {  for (int n=0; n<9; n++) {  LENGTH\_RAW\_DATA[m][n] = input.normalize(LENGTH\_RAW\_DATA[m][n]);  }  }    // Doing normalization to the Output  for (int m=0; m<TARGET\_RAW\_DATA.length; m++) {  for (int n=0; n<9; n++) {  TARGET\_RAW\_DATA[m][n] = output.normalize(TARGET\_RAW\_DATA[m][n]);  }  }    // Create training data  MLDataSet trainingSet = new BasicMLDataSet(LENGTH\_RAW\_DATA, TARGET\_RAW\_DATA);    // Train the neural network  final ManhattanPropagation train = new ManhattanPropagation(network, trainingSet, 0.00001);  int epoch = 1;    do {  train.iteration();  System.out.println("Epoch #" + epoch + " Error:" + train.getError());  epoch++;  } while(epoch<100000 && train.getError()>0.12); // Epoch until 100000 or error below 12% (Best Fit for ANN)  train.finishTraining();    // Test the neural network  System.out.println("Neural Network Results:");  for(MLDataPair pair: trainingSet ) {  final MLData outputData = network.compute(pair.getInput());  System.out.println("");  System.out.println("For Input:");  for (int a=0 ; a<9; a++) {  System.out.print(Math.round(input.deNormalize(pair.getInput().getData(a))) + " ");  }  System.out.println("");  System.out.println("Actual Result:");  for (int b=0 ; b<9; b++) {  System.out.print(Math.round(output.deNormalize(outputData.getData(b))) + " ");  }  System.out.println("");  System.out.println("Ideal Result:");  for (int c=0 ; c<9; c++) {  System.out.print(Math.round(output.deNormalize(pair.getIdeal().getData(c))) + " ");  }  System.out.println("");  System.out.println("");  }    // Saving the neural network  EncogDirectoryPersistence.saveObject(new File("ANNscheduler.EG"), network);  Encog.getInstance().shutdown();  } } |

## **G. *Dataset Cloudlet***

Untuk *Dataset* yang digunakan sebagai Panjang *Cloudlet (Task)* di dalam Simulasi *Genetic Algorithm* bisa diakses pada <https://intip.in/TABryanDatasetCloudlet>

## **H. *Output Genetic Algorithm***

Untuk *Dataset* hasil *output* dari *Genetic Algorithm* sebelum dilakukan proses *Preprocessing* bisa diakses pada <https://intip.in/TABryanOutputGA>

## **I. *Dataset Artificial Neural Network***

Untuk *Dataset* yang digunakan sebagai Panjang *Cloudlet (Task)* di dalam Simulasi *Artificial Neural Network* bisa diakses pada <https://intip.in/TABryanDatasetANN>

## **J. *Dataset Train Artificial Neural Network***

Untuk *Dataset* yang digunakan sebagai pembelajaran untuk pembuatan model *Artificial Neural Network* bisa diakses pada <https://intip.in/TABryanDatasetTrain>

## **K. *Dataset Test Artificial Neural Network***

Untuk *Dataset* yang digunakan sebagai pengujian untuk pembuatan model *Artificial Neural Network* bisa diakses pada <https://intip.in/TABryanDatasetTest>

## **L. *Model Artificial Neural Network***

Untuk model *Artificial Neural Network* yang sudah dilakukan pembelajaran bisa diakses pada <https://intip.in/TABryanModelANN>