**Evolutionary Comp – CS5320 Project 2 Report**

**Spring - 2016**

**Kruthika Holla**

For this project, I have a data structure class: Data.java . Also, I have used 11 bits to encode, where the most significant bit is the signed bit since the range is -10 to +10.

Data.java:

**package** proj2;

**import** java.util.Random;

**public** **class** Data {

**private** **int** n;

**private** **int** bitStringLength;

**private** String individual;

//Constructor

**public** Data(**int** x){

setAlleleNumber(x);

bitStringLength = 33; //11 bits in each individual; total 33

setIndividual();

}

//Copy constructor

**public** Data (**int** x, String vec){

setAlleleNumber(x);

**this**.individual = vec;

}

//setter for individual

**public** **void** setIndividual(){

individual = "";

**for**(**int** i = 0; i < bitStringLength; i++){

//get bit

Random randInt = **new** Random();

**int** temp = randInt.nextInt(2) ;

individual += String.*valueOf*(temp);

}

}

//getter for individual

**public** String getIndividual(){

**return** individual;

}

//get length of each individual

**public** **int** getIndividualLength(){

**return** individual.length();

}

//replacement for individual at index i

**public** **void** replaceIdividualAtIndex(**int** index, **char** x){

individual = changeCharInPosition(index, x, individual);

}

**public** String changeCharInPosition(**int** position, **char** ch, String str){

**char**[] charArray = str.toCharArray();

charArray[position] = ch;

**return** **new** String(charArray);

}

//get gene at index i

**public** **char** getComponentAtIndex(**int** index){

**char** a\_char = individual.charAt(index);

**return** a\_char;

}

//convert bit strings to doubles

**public** **double**[] convertVector(String data){

//separate the super string into three individual strings

String x1 = data.substring(0, 11);

String x2 = data.substring(11, 22);

String x3 = data.substring(22);

//Convert x1, x2, x3 to double

**double** convertedX1 = convertIndividual(x1);

**double** convertedX2 = convertIndividual(x2);

**double** convertedX3 = convertIndividual(x3);

**double**[] returnVar = **new** **double**[n];

returnVar[0] = convertedX1;

returnVar[1] = convertedX2;

returnVar[2] = convertedX3;

**return** returnVar;

}

//encoding

**public** **double** convertIndividual(String ind){

String zero = "0";

String one = "1";

**int** signed = 0;

**char** z = zero.charAt(0);

**char** o = one.charAt(0);

String temp = **new** String();

temp = ind;

//If negative

**if** (temp.charAt(0) == o){

signed = 1;

temp = changeCharInPosition(0, z, temp);

**for**(**int** i = 1; i < temp.length(); i++){

**if** (temp.charAt(i) == o){

temp = changeCharInPosition(i, z, temp);

}

**else**{

temp = changeCharInPosition(i, o, temp);

}

}

}

**int** byteToInt = Integer.*parseInt*(temp, 2);

**double** x = ((**double**)byteToInt/1023.0) \* 10.0;

**if** (signed == 1){

x = (-1.0)\*x;

}

**return** x;

}

//setter for allele number

**public** **void** setAlleleNumber(**int** x){

n = x;

}

//getter for allele number

**public** **int** getAlleleNumber(){

**return** n;

}

//tostring

**public** String toString(){

String geneString = "";

geneString = getIndividual();

**return** geneString;

}

}

proj2.java:

**package** proj2;

**import** java.util.\*;

//import proj1.Data;

**public** **class** proj2 {

**public** **static** **void** main(String[] args) {

**int** alleleNumber = 3; //Number of individual components

**int** N = 50; //pop size

**int** t\_max = 100; //max generations

**double** bestOfRun = 300.0; //best of run

Data bestOfRunVector = **new** Data(alleleNumber);

**double** prob = 0.0;

**double** min = 0.5, max = 1.5;

**int** tournSize = 0;

**double**[] fitnessDistr = **new** **double**[N]; //fitness holder

ArrayList<Data> data = **new** ArrayList<Data>(N);

**int** t = 0;

//Initialize Pop(0)

**for**(**int** i = 0; i < N; i++){

data.add(**new** Data(alleleNumber));

}

//Get fitness

fitnessDistr = *getFitnessDistr*(data, N);

**double** bestOfGeneration = fitnessDistr[0];

**int** index = 0;

**double** avgOfGen = fitnessDistr[0];

//System.out.println("Fitness at gen 0 "+ fitnessDistr[0]+" corresponding to vector: "+data.get(index).getIndividual());

**for** (**int** i = 1; i < N; i++){

//System.out.println("Fitness at gen 0 "+ fitnessDistr[i]+" corresponding to vector: "+data.get(i).getIndividual());

avgOfGen += fitnessDistr[i];

**if** (fitnessDistr[i] < bestOfGeneration){

bestOfGeneration = fitnessDistr[i];

index = i;

}

}

**double**[] bestOfGen = **new** **double**[(t\_max/10) + 1];

**double**[] AvOfGen = **new** **double**[(t\_max/10) + 1];

**int** j =0;

Scanner in = **new** Scanner(System.***in***);

**int** choice;

System.***out***.println("Please enter the selection method that you want to use");

System.***out***.print("Enter 1 for probabability, 2 for tournament, and 3 for ranking: ");

choice = in.nextInt();

**if**(choice == 2){

System.***out***.print("What tournament size would you like to play (Typical values 2-5): ");

tournSize = in.nextInt();

**if**(tournSize == 2){

System.***out***.print("Enter probability of selecting winner: ");

prob = in.nextDouble();

}

}

**if**(choice == 3){

System.***out***.println("Please enter the values for min and max");

System.***out***.print("min: ");

min = in.nextDouble();

System.***out***.print("max: ");

max = in.nextDouble();

}

System.***out***.println("Generation 0");

System.***out***.println("Best of generation: "+ bestOfGeneration +" corresponding to vector: "+data.get(index));

bestOfGen[j] = bestOfGeneration;

System.***out***.println("Average of generation: "+ avgOfGen/N);

AvOfGen[j] = avgOfGen/N;

System.***out***.println("--------------------------------------------------------");

**while**(t < t\_max){

ArrayList<Data> afterCrossover = **new** ArrayList<Data>(1);

ArrayList<Data> afterMutation = **new** ArrayList<Data>(1);

**for** (**int** i = 0; i < N; i++){

//select pop(t+1)

**if**(choice == 1){

**int** x = *selection*(fitnessDistr);

**int** y = *selection*(fitnessDistr);

//crossover

Data crossoverVector = *crossover*(data.get(x), data.get(y));

afterCrossover.add(crossoverVector);

**double** fit = *fitness*(crossoverVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = crossoverVector;

}

//mutation

Data mutationVector = *mutationApplied*(crossoverVector);

afterMutation.add(mutationVector);

fit = *fitness*(mutationVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = mutationVector;

}

}

**if**(choice == 2){

Data x = *tournamentSelection*(fitnessDistr, tournSize, prob, data);

Data y = *tournamentSelection*(fitnessDistr, tournSize, prob, data);

//crossover

Data crossoverVector = *crossover*(x, y);

afterCrossover.add(crossoverVector);

**double** fit = *fitness*(crossoverVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = crossoverVector;

}

//mutation

Data mutationVector = *mutationApplied*(crossoverVector);

afterMutation.add(mutationVector);

fit = *fitness*(mutationVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = mutationVector;

}

}

//select pop(t+1)

**if**(choice == 3){

**int** x = *rankingSelection*(min, max, fitnessDistr, data);

**int** y = *rankingSelection*(min, max, fitnessDistr, data);

//crossover

Data crossoverVector = *crossover*(data.get(x), data.get(y));

afterCrossover.add(crossoverVector);

**double** fit = *fitness*(crossoverVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = crossoverVector;

}

//mutation

Data mutationVector = *mutationApplied*(crossoverVector);

afterMutation.add(mutationVector);

fit = *fitness*(mutationVector);

//best of Run

**if** (fit < bestOfRun){

bestOfRun = fit;

bestOfRunVector = mutationVector;

}

}

}

data = afterMutation; //update data vector

fitnessDistr = *getFitnessDistr*(data, N); //update fitness distribution

//Get best and average at every 10th generation

**if** ((t+1) % 10 == 0){

bestOfGeneration = fitnessDistr[0];

avgOfGen = fitnessDistr[0];

index = 0;

**for** (**int** i = 1; i < N; i++){

//System.out.println("Fitness at gen " +(t+1) +"::: "+ fitnessDistr[i]+" corresponding to vector: "+data.get(index));

avgOfGen += fitnessDistr[i];

**if** (fitnessDistr[i] < bestOfGeneration){

bestOfGeneration = fitnessDistr[i];

index = i;

}

}

j++;

avgOfGen = avgOfGen / N;

System.***out***.println("Generation "+ (t+1));

System.***out***.println("Best of generation is: "+ bestOfGeneration+" corresponding to vector: "+data.get(index) );

System.***out***.println("Average of generation is: "+ avgOfGen);

bestOfGen[j] = bestOfGeneration;

AvOfGen[j] = avgOfGen;

System.***out***.println("Best so far: "+ bestOfRun + " corresponding to vector: "+bestOfRunVector);

System.***out***.println("--------------------------------------------------------");

}

t++;

}

System.***out***.println("Best of run is: "+ bestOfRun + " corresponding to vector: "+bestOfRunVector);

System.***out***.println("The Best of generations in an array:");

System.***out***.println(Arrays.*toString*(bestOfGen));

System.***out***.println("The Average of generations in an array:");

System.***out***.println(Arrays.*toString*(AvOfGen));

in.close();

}

//function to get fitness distribution

**public** **static** **double**[] getFitnessDistr(ArrayList<Data> data, **int** popSize){

**double**[] fitnessDistr = **new** **double**[popSize];

**for** (**int** i = 0; i< popSize; i++){

**double** fit = *fitness*(data.get(i));

fitnessDistr[i] = fit;

//System.out.println(fitnessDistr[i]);

}

**return** fitnessDistr;

}

//calculate fitness

**public** **static** **double** fitness(Data data){

**double** f = 0.0;

**int** n = data.getAlleleNumber();

String currentbitString = data.getIndividual() ;

//System.out.println(currentbitString);

**double**[] currentVector = data.convertVector(currentbitString);

**for** (**int** i = 0; i< n; i++){

**double** x = currentVector[i]\*currentVector[i];

f += x;

}

**return** f;

}

//fitness proportionate selection

**public** **static** **int** selection(**double**[] array){

**double** sum = 0.0;

**double**[] probability = **new** **double**[array.length];

//get probability of selection for each vector

**for**(**int** i = 0; i < array.length; i++){

**double** s = array[i] + 1;

**double** x = 1/s;

sum += x;

}

**for**(**int** i = 0; i < array.length; i++){

**double** s = array[i] + 1;

**double** x = 1/s;

probability[i] = x /sum;

}

//selection

sum = 0.0;

Random rand = **new** Random();

**double** randNum = rand.nextDouble();

**int** i = 0;

sum = probability[i];

**while**(sum < randNum){

i++;

sum += probability[i];

}

**return** i;

}

//tournament selection

**public** **static** Data tournamentSelection(**double**[] array, **int** times, **double** prob, ArrayList<Data> data){

**int** size = array.length-1;

**double**[] tournamentPlayers = **new** **double**[times];

ArrayList<Data> temp = **new** ArrayList<Data>();

//generate random numbers to choose the tournament players

**for**(**int** i = 0; i < times; i++){

Random rand = **new** Random();

**int** randomNum = rand.nextInt(((size-1) - 0) + 1) + 1;

tournamentPlayers[i] = array[randomNum];

temp.add(data.get(randomNum));

}

//get winner of tournament

**for**(**int** i = 0; i < times-1; i++){

**int** index = i;

**for**(**int** j = i + 1; j < times; j++){

**if** (tournamentPlayers[j] < tournamentPlayers[index])

index = j;

**double** smallerNumber = tournamentPlayers[index];

Data tempSave = temp.get(index);

tournamentPlayers[index] = tournamentPlayers[i];

temp.set(index, temp.get(i));

tournamentPlayers[i] = smallerNumber;

temp.set(i, tempSave);

}

}

**if**(times == 2){

Random rand = **new** Random();

**double** randNum = rand.nextDouble();

**if**(randNum < prob){

**return** temp.get(0);

}

**else**{

**return** temp.get(1);

}

}

**else**{

**return** temp.get(0);

}

}

//Ranking selection

**public** **static** **int** rankingSelection(**double** min, **double** max, **double**[] array, ArrayList<Data> data){

ArrayList<Data> temp = **new** ArrayList<Data>();

**double**[] sortFitness = **new** **double**[array.length];

//sort individuals based on fitness

**for**(**int** i = 0; i < array.length-1; i++){

**int** index = i;

**for**(**int** j = i + 1; j < array.length; j++){

**if** (sortFitness[j] < sortFitness[index])

index = j;

**double** smallerNumber = sortFitness[index];

Data tempSave = data.get(index);

sortFitness[index] = sortFitness[i];

sortFitness[i] = smallerNumber;

temp.add(tempSave);

}

}

**double** expectedR = 0.0;

**double** totalExpectedCopies = 0.0;

**double**[] expectedCopies = **new** **double**[array.length];

//Calculate E(r)

**for**(**int** i = 0; i < array.length; i++){

expectedR = min + ((max - min)/(array.length-1))\*(i); //i is r-1

expectedCopies[i] = expectedR;

totalExpectedCopies += expectedR;

}

**double**[] probailitiesOfRanks = **new** **double**[array.length];

//Set the probabilities with which the individuals get selected

**for**(**int** i =0; i < array.length; i++){

probailitiesOfRanks[i] = expectedCopies[i]/totalExpectedCopies;

}

//selection

**double** sum = 0.0;

Random rand = **new** Random();

**double** randNum = rand.nextDouble();

**int** i = 0;

sum = probailitiesOfRanks[i];

**while**(sum < randNum){

i++;

sum += probailitiesOfRanks[i];

}

**return** i;

}

//crossover

**public** **static** Data crossover(Data parentData1, Data parentData2){

**double** p\_c = 0.8; //Crossover probability

Random rand = **new** Random();

**double** randNum = rand.nextDouble();

Data childData1 = **new** Data(parentData1.getAlleleNumber(),parentData1.getIndividual()) ;

Data childData2 = **new** Data(parentData2.getAlleleNumber(),parentData2.getIndividual()) ;

//get cutpoint

Random randInt = **new** Random();

**int** cutPoint = randInt.nextInt((32-1)+1) + 1;

//apply crossover

**if** (randNum < p\_c){

childData1 = *crossoverHelper*(parentData1, parentData2, cutPoint);

}

//return the best

**else**{

**double** fitness1 = *fitness*(childData1);

**double** fitness2 = *fitness*(childData2);

**if** (fitness1 < fitness2){

**return** childData1;

}

**else**{

**return** childData2;

}

}

**return** childData1;

}

//Crossover applier function

**public** **static** Data crossoverHelper (Data data1, Data data2, **int** cutPoint){

Data childData1 = **new** Data(data1.getAlleleNumber(), data1.getIndividual());

Data childData2 = **new** Data(data2.getAlleleNumber(), data2.getIndividual());

//get children

**for** (**int** i = 0; i < childData1.getIndividualLength(); i++){

**if**(i >= cutPoint){

childData1.replaceIdividualAtIndex(i,data2.getComponentAtIndex(i));

childData2.replaceIdividualAtIndex(i, data1.getComponentAtIndex(i));

}

}

**double** fitness1 = *fitness*(childData1);

**double** fitness2 = *fitness*(childData2);

//return child with best fitness

**if** (fitness1 < fitness2){

**return** childData1;

}

**else**{

**return** childData2;

}

}

//Mutation

**public** **static** Data mutationApplied(Data data){

**double** p\_m = 0.01; //mutation rate

Data mutData = **new** Data(data.getAlleleNumber(), data.getIndividual());

String zero = "0";

String one = "1";

**char** z = zero.charAt(0);

**char** o = one.charAt(0);

//for each allele

**for** (**int** i = 0; i < data.getIndividualLength(); i++){

Random rand = **new** Random();

**double** randNum = rand.nextDouble();

**char** x = data.getComponentAtIndex(i); //get the allele

**if**(randNum < p\_m){

**if** (x == z)

mutData.replaceIdividualAtIndex(i, o);

**else**

mutData.replaceIdividualAtIndex(i, o);

}

}

**return** mutData;

}

}

Sample of 1 run:

Please enter the selection method that you want to use

Enter 1 for probabability, 2 for tournament, and 3 for ranking: 2

What tournament size would you like to play (Typical values 2-5): 3

Generation 0

Best of generation: 23.666329361154823 corresponding to vector: 111101111101100010100011101101100

Average of generation: 107.03145923333226

--------------------------------------------------------

Generation 10

Best of generation is: 0.9221913582901191 corresponding to vector: 111111111101111110011000001011111

Average of generation is: 9.814780096872614

Best so far: 0.8862630658108854 corresponding to vector: 111111111101111110011000001011101

--------------------------------------------------------

Generation 20

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 0.5943093789087547

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 30

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 2.8333242557062484

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 40

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 2.142650609777654

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 50

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 0.7047468345358803

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 60

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 1.102375567232249

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 70

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 1.0261731877473053

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 80

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 1.7211372069001425

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 90

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 0.39995069415181056

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Generation 100

Best of generation is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

Average of generation is: 1.2054783001713265

Best so far: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

--------------------------------------------------------

Best of run is: 0.3913890584971845 corresponding to vector: 111111111111111111111100001000000

The Best of generations in an array:

[23.666329361154823, 0.9221913582901191, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845, 0.3913890584971845]

The Average of generations in an array:

[107.03145923333226, 9.814780096872614, 0.5943093789087547, 2.8333242557062484, 2.142650609777654, 0.7047468345358803, 1.102375567232249, 1.0261731877473053, 1.7211372069001425, 0.39995069415181056, 1.2054783001713265]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Run 1:

Fitness proportionate selection:

|  |  |  |
| --- | --- | --- |
| Generation | Best | Average |
| 0 | 0.479489818 | 99.34251378 |
| 10 | 3.82E-04 | 4.213731392 |
| 20 | 0 | 1.496970557 |
| 30 | 0 | 1.206241203 |
| 40 | 0 | 1.035761646 |
| 50 | 0 | 1.124581641 |
| 60 | 0 | 1.738805996 |
| 70 | 0 | 1.587616206 |
| 80 | 0 | 1.208293607 |
| 90 | 0 | 0.985422191 |
| 100 | 0 | 0.934636307 |
| AVERAGE | 0.04362473 | 10.44314314 |
| STD DEV | 0.144560141 | 28.12621056 |

Best of run is: 0.0 corresponding to vector: 111111111111111111111100000000000

N: 1000

Tmax: 100

Crossover Probability: 0.8

Mutation Probability: 0.02

Run 2:

Binary tournament selection:

Probability of selecting winner = 0.7

N: 1000

Tmax: 100

Crossover Probability: 0.8

Mutation Probability: 0.02

Best of run is: 0.0 corresponding to vector: 111111111111111111111111111111111

|  |  |  |
| --- | --- | --- |
| Generation | Best | Average |
| Gen 0 | 4.112833949 | 99.39487535 |
| Gen 10 | 9.56E-05 | 5.07349782 |
| Gen 20 | 0 | 0.869426265 |
| Gen 30 | 0 | 0 |
| Gen 40 | 0 | 0 |
| Gen 50 | 0 | 0 |
| Gen 60 | 0 | 0 |
| Gen 70 | 0 | 0 |
| Gen 80 | 0 | 0 |
| Gen 90 | 0 | 0 |
| Gen 100 | 0 | 0 |
| AVERAGE | 0.373902682 | 9.576163585 |
| STD DEV | 1.240063213 | 29.82801624 |

Run 3:

4-ary tournament:

N: 1000

Tmax: 100

Crossover Probability: 0.8

Mutation Probability: 0.02

Best of run is: 0.0 corresponding to vector: 000000000000000000000011111111111

|  |  |  |
| --- | --- | --- |
| Generation | Best | Average |
| Gen 0 | 1.18534699 | 99.53824882 |
| Gen 10 | 9.56E-05 | 6.09E+00 |
| Gen 20 | 0 | 2.990265248 |
| Gen 30 | 0 | 1.411670293 |
| Gen 40 | 0 | 0.947541444 |
| Gen 50 | 0 | 0.464690228 |
| Gen 60 | 0 | 1.494679842 |
| Gen 70 | 0 | 1.286267461 |
| Gen 80 | 0 | 0.525570338 |
| Gen 90 | 0 | 1.468874919 |
| Gen 100 | 0 | 1.207027421 |
| AVERAGE | 0.107767504 | 10.67516287 |
| STD DEV | 0.357392685 | 29.51483073 |

Run 4:

Ranking:

N: 50

Tmax: 100

Min = 0.5

Max = 1.5

Crossover Probability: 0.8

Mutation Probability: 0.02

Best of run is: 0.0 corresponding to vector: 111111111111111111111111111111111

|  |  |  |
| --- | --- | --- |
| Generation | Best | Average |
| Gen 0 | 6.402307055 | 96.35552001 |
| Gen 10 | 1.66E-01 | 6.55E+00 |
| Gen 20 | 0.001911079 | 0.023605653 |
| Gen 30 | 0.001528864 | 0.002721377 |
| Gen 40 | 0 | 0 |
| Gen 50 | 0 | 0 |
| Gen 60 | 0 | 0 |
| Gen 70 | 0 | 0 |
| Gen 80 | 0 | 0 |
| Gen 90 | 0 | 0 |
| Gen 100 | 0 | 0 |
| AVERAGE | 0.597429477 | 9.357242848 |
| STD DEV | 1.925901043 | 28.92079334 |