**Bio-Inspired Optimization Algorithm**

**N-Queen’s Problem:**

Optimization techniques for N-Queen’s problem By Genetic Algorithm

**Source Code:**

using System;

import java.util.ArrayList;

import java.util.Random;

public class NQueen

{

private static final int START\_SIZE = 75; // Population size at start.

private static final int MAX\_EPOCHS = 1000; // Arbitrary number of test cycles.

private static final double MATING\_PROBABILITY = 0.7; // Probability of two chromosomes mating. Range: 0.0 < MATING\_PROBABILITY < 1.0

private static final double MUTATION\_RATE = 0.001; // Mutation Rate. Range: 0.0 < MUTATION\_RATE < 1.0

private static final int MIN\_SELECT = 10; // Minimum parents allowed for selection.

private static final int MAX\_SELECT = 50; // Maximum parents allowed for selection. Range: MIN\_SELECT < MAX\_SELECT < START\_SIZE

private static final int OFFSPRING\_PER\_GENERATION = 20; // New offspring created per generation. Range: 0 < OFFSPRING\_PER\_GENERATION < MAX\_SELECT.

private static final int MINIMUM\_SHUFFLES = 8; // For randomizing starting chromosomes

private static final int MAXIMUM\_SHUFFLES = 20;

private static final int PBC\_MAX = 4; // Maximum Position-Based Crossover points. Range: 0 < PBC\_MAX < 8 (> 8 isn't good).

private static final int MAX\_LENGTH = 10; // chess board width.

private static int epoch = 0;

private static int childCount = 0;

private static int nextMutation = 0; // For scheduling mutations.

private static int mutations = 0;

private static ArrayList<Chromosome> population = new ArrayList<Chromosome>();

private static void algorithm()

{

int popSize = 0;

Chromosome thisChromo = null;

boolean done = false;

initializeChromosomes();

mutations = 0;

nextMutation = getRandomNumber(0, (int)Math.round(1.0 / MUTATION\_RATE));

while (!done)

{

popSize = population.size();

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

{

thisChromo = population.get(i);

if ((thisChromo.conflicts() == 0) || epoch == MAX\_EPOCHS)

{

done = true;

}

}

getFitness();

rouletteSelection();

mating();

prepNextEpoch();

epoch++;

// This is here simply to show the runtime status.

System.out.println("Epoch: " + epoch);

}

System.out.println("done.");

if (epoch != MAX\_EPOCHS)

{

popSize = population.size();

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

{

thisChromo = population.get(i);

if (thisChromo.conflicts() == 0)

{

printbestSolution(thisChromo);

}

}

}

System.out.println("Completed " + epoch + " epochs.");

System.out.println("Encountered " + mutations + " mutations in " + childCount + " offspring.");

return;

}

private static void getFitness()

{

// Lowest errors = 100%, Highest errors = 0%

int popSize = population.size();

Chromosome thisChromo = null;

double bestScore = 0;

double worstScore = 0;

// The worst score would be the one with the highest energy, best would be lowest.

worstScore = population.get(maximum()).conflicts();

// Convert to a weighted percentage.

bestScore = worstScore - population.get(minimum()).conflicts();

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

{

thisChromo = population.get(i);

thisChromo.fitness((worstScore - thisChromo.conflicts()) \* 100.0 / bestScore);

}

return;

}

private static void rouletteSelection()

{

int j = 0;

int popSize = population.size();

double genTotal = 0.0;

double selTotal = 0.0;

int maximumToSelect = getRandomNumber(MIN\_SELECT, MAX\_SELECT);

double rouletteSpin = 0.0;

Chromosome thisChromo = null;

Chromosome thatChromo = null;

boolean done = false;

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

{

thisChromo = population.get(i);

genTotal += thisChromo.fitness();

}

genTotal \*= 0.01;

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

{

thisChromo = population.get(i);

thisChromo.selectionProbability(thisChromo.fitness() / genTotal);

}

for (int i = 0; i < maximumToSelect; i++)

{

rouletteSpin = getRandomNumber(0, 99);

j = 0;

selTotal = 0;

done = false;

while (!done)

{

thisChromo = population.get(j);

selTotal += thisChromo.selectionProbability();

if (selTotal >= rouletteSpin)

{

if (j == 0)

{

thatChromo = population.get(j);

}

else if (j >= popSize - 1)

{

thatChromo = population.get(popSize - 1);

}

else

{

thatChromo = population.get(j - 1);

}

thatChromo.selected(true);

done = true;

}

else

{

j++;

}

}

}

return;

}

// This is where you can choose between options:

// To choose between crossover options, uncomment one of:

// partiallyMappedCrossover(),

// positionBasedCrossover(), while keeping the other two commented out.

// Keep in mind that the code will still run if(you try combinations or uncomment all of them,

// but this might hinder the algorithm in general.

// Of course, I could always be wrong, try it and find out!

private static void mating()

{

int getRand = 0;

int parentA = 0;

int parentB = 0;

int newIndex1 = 0;

int newIndex2 = 0;

Chromosome newChromo1 = null;

Chromosome newChromo2 = null;

for (int i = 0; i < OFFSPRING\_PER\_GENERATION; i++)

{

parentA = chooseParent();

// Test probability of mating.

getRand = getRandomNumber(0, 100);

if (getRand <= MATING\_PROBABILITY \* 100)

{

parentB = chooseParent(parentA);

newChromo1 = new Chromosome();

newChromo2 = new Chromosome();

population.add(newChromo1);

newIndex1 = population.indexOf(newChromo1);

population.add(newChromo2);

newIndex2 = population.indexOf(newChromo2);

// Choose either, or both of these:

partiallyMappedCrossover(parentA, parentB, newIndex1, newIndex2);

//positionBasedCrossover(parentA, parentB, newIndex1, newIndex2);

if (childCount - 1 == nextMutation)

{

exchangeMutation(newIndex1, 1);

}

else if (childCount == nextMutation)

{

exchangeMutation(newIndex2, 1);

}

population.get(newIndex1).computeConflicts();

population.get(newIndex2).computeConflicts();

childCount += 2;

// Schedule next mutation.

if (childCount % (int)Math.round(1.0 / MUTATION\_RATE) == 0)

{

nextMutation = childCount + getRandomNumber(0, (int)Math.round(1.0 / MUTATION\_RATE));

}

}

} // i

return;

}

private static void partiallyMappedCrossover(int chromA, int chromB, int child1, int child2)

{

int j = 0;

int item1 = 0;

int item2 = 0;

int pos1 = 0;

int pos2 = 0;

Chromosome thisChromo = population.get(chromA);

Chromosome thatChromo = population.get(chromB);

Chromosome newChromo1 = population.get(child1);

Chromosome newChromo2 = population.get(child2);

int crossPoint1 = getRandomNumber(0, MAX\_LENGTH - 1);

int crossPoint2 = getExclusiveRandomNumber(MAX\_LENGTH - 1, crossPoint1);

if (crossPoint2 < crossPoint1)

{

j = crossPoint1;

crossPoint1 = crossPoint2;

crossPoint2 = j;

}

// Copy Parent genes to offspring.

for (int i = 0; i < MAX\_LENGTH; i++)

{

newChromo1.data(i, thisChromo.data(i));

newChromo2.data(i, thatChromo.data(i));

}

for (int i = crossPoint1; i <= crossPoint2; i++)

{

// Get the two items to swap.

item1 = thisChromo.data(i);

item2 = thatChromo.data(i);

// Get the items// positions in the offspring.

for (j = 0; j < MAX\_LENGTH; j++)

{

if (newChromo1.data(j) == item1)

{

pos1 = j;

}

else if (newChromo1.data(j) == item2)

{

pos2 = j;

}

} // j

// Swap them.

if (item1 != item2)

{

newChromo1.data(pos1, item2);

newChromo1.data(pos2, item1);

}

// Get the items// positions in the offspring.

for (j = 0; j < MAX\_LENGTH; j++)

{

if (newChromo2.data(j) == item2)

{

pos1 = j;

}

else if (newChromo2.data(j) == item1)

{

pos2 = j;

}

} // j

// Swap them.

if (item1 != item2)

{

newChromo2.data(pos1, item1);

newChromo2.data(pos2, item2);

}

} // i

return;

}

private static void positionBasedCrossover(int chromA, int chromB, int child1, int child2)

{

int k = 0;

int numPoints = 0;

int tempArray1[] = new int[MAX\_LENGTH];

int tempArray2[] = new int[MAX\_LENGTH];

boolean matchFound = false;

Chromosome thisChromo = population.get(chromA);

Chromosome thatChromo = population.get(chromB);

Chromosome newChromo1 = population.get(child1);

Chromosome newChromo2 = population.get(child2);

// Choose and sort the crosspoints.

numPoints = getRandomNumber(0, PBC\_MAX);

int crossPoints[] = new int[numPoints];

for (int i = 0; i < numPoints; i++)

{

crossPoints[i] = getRandomNumber(0, MAX\_LENGTH - 1, crossPoints);

} // i

// Get non-chosens from parent 2

k = 0;

for (int i = 0; i < MAX\_LENGTH; i++)

{

matchFound = false;

for (int j = 0; j < numPoints; j++)

{

if (thatChromo.data(i) == thisChromo.data(crossPoints[j]))

{

matchFound = true;

}

} // j

if (matchFound == false)

{

tempArray1[k] = thatChromo.data(i);

k++;

}

} // i

// Insert chosens into child 1.

for (int i = 0; i < numPoints; i++)

{

newChromo1.data(crossPoints[i], thisChromo.data(crossPoints[i]));

}

// Fill in non-chosens to child 1.

k = 0;

for (int i = 0; i < MAX\_LENGTH; i++)

{

matchFound = false;

for (int j = 0; j < numPoints; j++)

{

if (i == crossPoints[j])

{

matchFound = true;

}

} // j

if (matchFound == false)

{

newChromo1.data(i, tempArray1[k]);

k++;

}

} // i

// Get non-chosens from parent 1

k = 0;

for (int i = 0; i < MAX\_LENGTH; i++)

{

matchFound = false;

for (int j = 0; j < numPoints; j++)

{

if (thisChromo.data(i) == thatChromo.data(crossPoints[j]))

{

matchFound = true;

}

} // j

if (matchFound == false)

{

tempArray2[k] = thisChromo.data(i);

k++;

}

} // i

// Insert chosens into child 2.

for (int i = 0; i < numPoints; i++)

{

newChromo2.data(crossPoints[i], thatChromo.data(crossPoints[i]));

}

// Fill in non-chosens to child 2.

k = 0;

for (int i = 0; i < MAX\_LENGTH; i++)

{

matchFound = false;

for (int j = 0; j < numPoints; j++)

{

if (i == crossPoints[j])

{

matchFound = true;

}

} // j

if (matchFound == false)

{

newChromo2.data(i, tempArray2[k]);

k++;

}

} // i

return;

}

private static void exchangeMutation(final int index, final int exchanges)

{

int i = 0;

int tempData = 0;

Chromosome thisChromo = null;

int gene1 = 0;

int gene2 = 0;

boolean done = false;

thisChromo = population.get(index);

while (!done)

{

gene1 = getRandomNumber(0, MAX\_LENGTH - 1);

gene2 = getExclusiveRandomNumber(MAX\_LENGTH - 1, gene1);

// Exchange the chosen genes.

tempData = thisChromo.data(gene1);

thisChromo.data(gene1, thisChromo.data(gene2));

thisChromo.data(gene2, tempData);

if (i == exchanges)

{

done = true;

}

i++;

}

mutations++;

return;

}

private static int chooseParent()

{

// Overloaded function, see also "chooseparent(ByVal parentA As Integer)".

int parent = 0;

Chromosome thisChromo = null;

boolean done = false;

while (!done)

{

// Randomly choose an eligible parent.

parent = getRandomNumber(0, population.size() - 1);

thisChromo = population.get(parent);

if (thisChromo.selected() == true)

{

done = true;

}

}

return parent;

}

private static int chooseParent(final int parentA)

{

// Overloaded function, see also "chooseparent()".

int parent = 0;

Chromosome thisChromo = null;

boolean done = false;

while (!done)

{

// Randomly choose an eligible parent.

parent = getRandomNumber(0, population.size() - 1);

if (parent != parentA)

{

thisChromo = population.get(parent);

if (thisChromo.selected() == true)

{

done = true;

}

}

}

return parent;

}

private static void prepNextEpoch()

{

int popSize = 0;

Chromosome thisChromo = null;

// Reset flags for selected individuals.

popSize = population.size();

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

{

thisChromo = population.get(i);

thisChromo.selected(false);

}

return;

}

private static void printbestSolution(Chromosome bestSolution)

{

String board[][] = new String[MAX\_LENGTH][MAX\_LENGTH];

// Clear the board.

for (int x = 0; x < MAX\_LENGTH; x++)

{

for (int y = 0; y < MAX\_LENGTH; y++)

{

board[x][y] = "";

}

}

for (int x = 0; x < MAX\_LENGTH; x++)

{

board[x][bestSolution.data(x)] = "Q";

}

// Display the board.

System.out.println("Board:");

for (int y = 0; y < MAX\_LENGTH; y++)

{

for (int x = 0; x < MAX\_LENGTH; x++)

{

if (board[x][y] == "Q")

{

System.out.print("Q ");

}

else

{

System.out.print(". ");

}

}

System.out.print("\n");

}

return;

}

private static int getRandomNumber(final int low, final int high)

{

return (int)Math.round((high - low) \* new Random().nextDouble() + low);

}

private static int getExclusiveRandomNumber(final int high, final int except)

{

boolean done = false;

int getRand = 0;

while (!done)

{

getRand = new Random().nextInt(high);

if (getRand != except)

{

done = true;

}

}

return getRand;

}

private static int getRandomNumber(int low, int high, int[] except)

{

boolean done = false;

int getRand = 0;

if (high != low)

{

while (!done)

{

done = true;

getRand = (int)Math.round((high - low) \* new Random().nextDouble() + low);

for (int i = 0; i < except.length; i++) //UBound(except)

{

if (getRand == except[i])

{

done = false;

}

} // i

}

return getRand;

}

else

{

return high; // or low (it doesn't matter).

}

}

private static int minimum()

{

// Returns an array index.

int popSize = 0;

Chromosome thisChromo = null;

Chromosome thatChromo = null;

int winner = 0;

boolean foundNewWinner = false;

boolean done = false;

while (!done)

{

foundNewWinner = false;

popSize = population.size();

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

{

if (i != winner)

{ // Avoid self-comparison.

thisChromo = population.get(i);

thatChromo = population.get(winner);

if (thisChromo.conflicts() < thatChromo.conflicts())

{

winner = i;

foundNewWinner = true;

}

}

}

if (foundNewWinner == false)

{

done = true;

}

}

return winner;

}

private static int maximum()

{

// Returns an array index.

int popSize = 0;

Chromosome thisChromo = null;

Chromosome thatChromo = null;

int winner = 0;

boolean foundNewWinner = false;

boolean done = false;

while (!done)

{

foundNewWinner = false;

popSize = population.size();

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

{

if (i != winner)

{ // Avoid self-comparison.

thisChromo = population.get(i);

thatChromo = population.get(winner);

if (thisChromo.conflicts() > thatChromo.conflicts())

{

winner = i;

foundNewWinner = true;

}

}

}

if (foundNewWinner == false)

{

done = true;

}

}

return winner;

}

private static void initializeChromosomes()

{

int shuffles = 0;

Chromosome newChromo = null;

int chromoIndex = 0;

for (int i = 0; i < START\_SIZE; i++)

{

newChromo = new Chromosome();

population.add(newChromo);

chromoIndex = population.indexOf(newChromo);

// Randomly choose the number of shuffles to perform.

shuffles = getRandomNumber(MINIMUM\_SHUFFLES, MAXIMUM\_SHUFFLES);

exchangeMutation(chromoIndex, shuffles);

population.get(chromoIndex).computeConflicts();

}

return;

}

private static class Chromosome

{

private int mData[] = new int[MAX\_LENGTH];

private double mFitness = 0.0;

private boolean mSelected = false;

private double mSelectionProbability = 0.0;

private int mConflicts = 0;

public Chromosome()

{

for (int i = 0; i < MAX\_LENGTH; i++)

{

this.mData[i] = i;

}

return;

}

public void computeConflicts()

{

int x = 0;

int y = 0;

int tempx = 0;

int tempy = 0;

String board[][] = new String[MAX\_LENGTH][MAX\_LENGTH];

int conflicts = 0;

int dx[] = new int[] { -1, 1, -1, 1 };

int dy[] = new int[] { -1, 1, 1, -1 };

boolean done = false;

// Clear the board.

for (int i = 0; i < MAX\_LENGTH; i++)

{

for (int j = 0; j < MAX\_LENGTH; j++)

{

board[i][j] = "";

}

}

for (int i = 0; i < MAX\_LENGTH; i++)

{

board[i][this.mData[i]] = "Q";

}

// Walk through each of the Queens and compute the number of conflicts.

for (int i = 0; i < MAX\_LENGTH; i++)

{

x = i;

y = this.mData[i];

// Check diagonals.

for (int j = 0; j <= 3; j++)

{

tempx = x;

tempy = y;

done = false;

while (!done)

{

tempx += dx[j];

tempy += dy[j];

if ((tempx < 0 || tempx >= MAX\_LENGTH) || (tempy < 0 || tempy >= MAX\_LENGTH))

{

done = true;

}

else

{

if (board[tempx][tempy].compareToIgnoreCase("Q") == 0)

{

conflicts++;

}

}

}

}

}

this.mConflicts = conflicts;

}

public void conflicts(int value)

{

this.mConflicts = value;

return;

}

public int conflicts()

{

return this.mConflicts;

}

public double selectionProbability()

{

return mSelectionProbability;

}

public void selectionProbability(final double SelProb)

{

mSelectionProbability = SelProb;

return;

}

public boolean selected()

{

return mSelected;

}

public void selected(final boolean sValue)

{

mSelected = sValue;

return;

}

public double fitness()

{

return mFitness;

}

public void fitness(final double score)

{

mFitness = score;

return;

}

public int data(final int index)

{

return mData[index];

}

public void data(final int index, final int value)

{

mData[index] = value;

return;

}

} // Chromosome

public static void main(String[] args)

{

algorithm();

return;

}

}

**Output:**

**By Partially Mapped Crossover**

A screen shot of a computer

Description automatically generated

**By Position Based Crossover**

**A screen shot of a computer

Description automatically generated**

**Comparison**:

The N-Queens problem entails the placement of queens on a chess board so that none are in jeopardy. The difficulty of the problem increases with the number of queens involved.

Partially Mapped Crossover completes in 244 epochs and 7 mutations in 6822 offspring.

Position Based Crossover Completes in 247 epochs and 7 mutations in 6896 offspring, thus Partially Mapped Crossover is more optimised.