Genetic Algorithm Implementation of Robotic Controllers

Amanda Shen, Eunjun Choo, and Peng Gu

December 20, 2019

Contents

1 Introduction & Main Problem

1.1 Introduction

This paper strives to explain how we used Genetic Algorithm to train the robot controller. Closely studying Jacobson's and Kanber's "Genetic Algorithm in Java Basic", we present how Genetic Algorithm can be used to train a robot controller.

Genetic Algorithm is not about finding the most optimal solution but a possible one as a default solution for the problem. By utilizing the Darwin theory of finding the most fittest individual among the population, Genetic Algorithm similarly works by calculating how well the individual candidate fits within the population and prioritizing to select the successful individual to the next generation.

By using this algorithm, we train a hypothetical robot sensor and its movement. The robot controller is analogous to training a physical robots like a robot vacuum. Physical robot vacuum will have sensors around itself to indicate if it is nearby a wall and navigate to an open route. Similarly, we will be using a robot instruction to test the robot in a maze represented by 2-D array. The array maze represents a simulated model for the robot to navigate and evaluate its sensor.

1.2 Main Problem

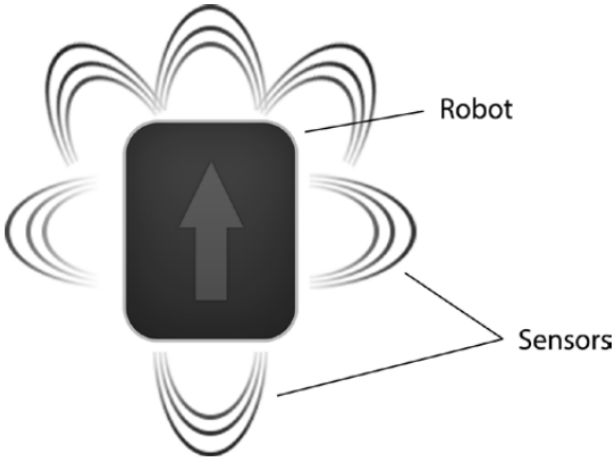

Figure 1: Our Robot can only move front. However, the robot has six sensors around itself to turn left and right.[2]

In this application, we utilize the binary genetic representation for the robotic controller. The binary representation is used because of its simplicity of encoding when the chromosome goes through evolution.

To begin with, Our robot will have six sensors: three on the front, two on the side, and one on the back. The sensors will activate whenever the robot detects a wall adjacent to the sensor. For example, if the robot faces a wall in front of itself, the robot's front

[2]Taken from Jacobson and Kanber, page 48.

sensor will activate and each of the six sensors as either "0" or "1" indicates whether or not it's facing a wall. To explain, when the front of the sensor activates, it indicates that the robot is facing a wall and we will record it as "1" while if the robot does not face a wall, the sensor will be "0". Before each movement, we will have all sensor to input a binary representation of sensing result and then we will make prediction of the movement based on that. Because there are total of six different on/off sensors, we will have total of $2^6 = 64$ possible combinations of movements consist of : move forward, turn left, turn right, or stay in its place. The four actions are represented in binary as below:

- move forward: "01"
- turn left: "10"
- turn right: "11"
- stay in its place: "00" .

Since each of the action requires two bits and we have 64 possible combinations of sensor inputs, our controller requires 128 bits of storage and we will use 128 bits to represent the different parts of the robot sensors.

In the next two sections, the specific implementation of Genetic algorithm for the Robot Controller and the Robot & Maze will be introduced.

2 Genetic Algorithm

2.1 Overview

This section contains class that involved with genetic algorithm, which involved the following classes:

- **Individual class**: this class represents a single candidate solution and its chromosome. It will be represented with an array in Java.
- **Population class**: this class represents a population or a generation of Individuals. This class is an array of individual class.
- **Genetic Algorithm class**: this class abstracts the Genetic Algorithm itself. It will provide problem-specific implementations of interface methods, such as crossover, mutation, fitness function, and termination condition checking.

2.2 Individual Class

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. In our case, a possible solution is 128 bit of instruction for our robot controller.

```
    ...
    ...
    // constructor for input type : int[]
    public Individual(int[] newchromosome) {
      this.chromosome = newchromosome;
    }

    // constructor for input type : length
    public Individual(int chromosomeLength) {
        int[] temp = new int[chromosomeLength];
        this.chromosome = RandomSetGene(temp);
    }

    // a starting point for the new gene
    // will be adjusted as generation progresses
    public int[] RandomSetGene (int[] chromosome) {
        for (int i = 0; i < chromosome.length; i++) {
            double random = Math.random();
            if (random < 0.5) {
                chromosome[i] = 0;
            } else {
                chromosome[i] = 1;
            }
        }
        return chromosome;
    }
    ...
    ...
```

An individual also has a "fitness" score; this is a number that represents how good a solution to the problem this individual is. The "fitness" score of the individual will be evaluated and adjusted in Genetic Algorithm class.

```
public class Individual {
    // primary variables for Individual class

    private int id;
    private int[] chromosome;
   private double fitness = -1;

   ...
   ...
   ...
   }
```

2.3 Population Class

A population is an abstraction of a collection of individuals. This population class is used to perform a group-level operation such as selecting individual genes and grouping individual genes for mutation and cross over from the GeneticAlgorithm class.

For example, we used our population class to initialize the collection of individuals (robot control instruction).

```
...
...
// Constructor: given size
// in our problem, the size will be given by main class in Robot Controller
 public Population(int population Size) {
  this.population = new Individual[populationSize];
}
// Constructor: given size and chromosome length
 public Population(int populationSize, int chromosomeLength) {
  this.population = new Individual[populationSize];
  for (int i = 0; i < populationSize; i++) {
     Individual individual = new Individual(chromosomeLength);
     this.population[i] = individual;
  }
}
...
...
```

We use function such as **SortByFit** to organize our population grouping.

```
// Function to organize the grouping of individual from highest fitness to
    lowest.
public void SortByFit() {
   Arrays.sort(this.population, new Comparator<Individual>() {
      public int compare(Individual o1, Individual o2) {
         if (o1.getFitness() > o2.getFitness()) {
            return -1;
         } else if (o1.getFitness() < o2.getFitness()) {
            return 1;
         }
         return 0;
      }
     }
     );
}
```

2.4 Genetic Algorithm

This Genetic Algorithm class is our main part that manages the operations of the genetic algorithm. This class includes the essential parts to the Genetic Algorithm like mutation

and crossover but those two functions are specifically designed to solve our "Robot Controller" problem.

First, we have our constructors for Genetic Algorithm. Each constructor is used throughout the **GeneticAlgorithm** class to update its value after functions like **BestParent**, **mutate**, and **crossover**.

```
public class GeneticAlgorithm {
    //pop_size is total number of individual candidate to hold
    private int pop_size;

    private int good_chromosome_count;
    private int t_size;
    private double m_rate;
    private double c_rate;
    ...
    ...
    }
```

We list couple of important functions below.

To calculate the fitness of individual, we use **Individual_fitnessScore** function, which first initializes a new Robot class from an individual instructor and evaluate its performance in the given maze.

```
    public double Individual_fitnessScore (Individual individual, Maze maze) {
        int[] current = individual.getChromosome();
        Robot robot = new Robot(current, maze, 100); // 100 is maxmoves
        robot.run();
        double result = maze.scoreRoute(robot.getRoute());
        individual.setFitness(result);
        return result;
    }
```

To evaluate the whole population, we use **Population_fitnessScore**. This function loops over each individual in the population, calculate the fitness for each, and then calculate the entire population's fitness. While the population's fitness isn't that important for our robot controlling problem but it's important to make sure each individual instruction is evaluated. In our case, each instruction is evaluated in the function **Individual_fitnessScore** inside the for loop.

The GeneticAlgorithm uses the Robot class. The Robot class will be described more thoroughly in the next section.

```
    public double Population_fitnessScore (Population population, Maze maze) {
        double result = 0;
        for (Individual each : population.getIndividuals()) {
            result += this.Individual_fitnessScore(each, maze);
        }
```

```
        population.setPopulationFitness(result);
        return result;
    }

    ...
    ...
    public double Individual_fitnessScore (Individual individual, Maze maze) {
        int[] current = individual.getChromosome();
        Robot robot = new Robot(current, maze, 100); // 100 is maxmoves
        robot.run();
        double result = maze.scoreRoute(robot.getRoute());
        individual.setFitness(result);
        return result;
    }
```

After setting up the population and the individual, we can start evolving the population by using crossover and mutation. Crossover function works by taking the individual instructions to create a new offspring. We use both "c_rate" and "good_chromosome_count" variables to take account of both crossover rate and the elitism count. Then using the probability, we crossover if the "Math.random()" is less than the crossover rate. In our case, we use a single-point crossover, in which a single position in the genome is chosen at random and the genetic information of the parent1 will occupy from index 0 to the single position while parent2's genetic information will occupy from index of single position until the end.

```
public Population crossover (Population population) {
        Population newPop = new Population(this.pop_size);

        for (int i = 0; i < population.size(); i++) {
            // get current chromosome as individual
            Individual p1 = population.getFittest(i);
            // loop throught its gene and mutate accordingly
            if (i >= this.good_chromosome_count) {
                if (this.c_rate >= Math.random()) {
                    Individual child = new Individual(p1.getChromosomeLength());
                    Individual p2 = this.BestParent(population);
                    int swapPoint = (int) (Math.random() *
                        (p1.getChromosomeLength() + 1));
                    //cross over:
                for (int j = 0; j < p1.getChromosomeLength(); j++) {
                        if (j < swapPoint) {
                            child.setGene(j, p1.getGene(j));
                        } else {
                            child.setGene(j, p2.getGene(j));
                        }
                    }
                    // add new child to new population
                newPop.setIndividual(i, child);
```

```
                }
            } else {
                newPop.setIndividual(i, p1);
            }
        }
        return newPop;
    }
}
```

The last step is mutation. This function works by looping over each individual's bit and using the mutation rate, we mutate the gene based on if the gene is a good or "ok" gene (elitism) and on the probability.

By mutating, we either mutate the gene to 0 or 1.

```
public Population mutate (Population population) {
        Population newPop = new Population(this.pop_size);

        for (int i = 0; i < population.size(); i++) {
            // get current chromosome as individual
            Individual indi = population.getFittest(i);
            // loop throught its gene and mutate accordingly
            for (int j = 0; j < indi.getChromosomeLength(); j++) {
                if (i >= this.good_chromosome_count) {
                    if (this.m_rate >= Math.random()) {
                        int newGene = 0;
                        if (indi.getGene(j) == 1) {
                            newGene = 0;
                        }
                        indi.setGene(j, newGene);
                    }

                }

            }
            // add new possible mutated chromosome back to population
            newPop.setIndividual(i, indi);
        }

        return newPop;
    }
```

3 Training a robot

3.1 Overview

We now apply the Genetic Algorithm to the robotic controllers. While there are other algorithms like breadth-first and depth-first search algorithms to solve mazes, we are not interested in solving and finding the shortest path of the maze. Instead, we are interesting in training the robot controller to navigate through the maze and not crash into walls. We use maze as a complicated environment to train and test the robot sensors.

To do this, we have three additional main classes:

- **Maze class**: this class abstracts the array representation of maze and interprets the value of each element in the maze array accordingly. Moreover, this class also evaluates the individual candidate's path through the "scoreRoute" function.
- **Robot class**: using the "individual" class from the Genetic Algorithm section, each candidate is converted to a "robot" with each sensor control from the individual's binary string
- **RobotController class**: this is the main function for our Robot control. The class calls all the necessary Genetic Algorithm class while also instantiating necessary individual candidate robots.

3.2 Maze Class

The maze class uses a constructor to create a new maze from array and use public methods to get the start position, position's value, and a route score through the maze. The maze class's a 2-D array of integers to represent the maze. Each position value is defined as below:

- "0" = Empty position
- "1" = Wall
- "3" = Route
- "4" = Ending position .

The constructor for the maze is illustrated below:

```
public Maze(int maze[][]) {
   this.maze = maze;
}
```

Since the program does not know the starting position beforehand, a **getStarting-Position** function loops through every element in the maze array to find its indices, and returns {-1, -1}, if not found.

```
 //constructors
private final int maze[][];
private int startPosition[] = { -1, -1 };
private int visited[][];
 ...
 ...

 public int[] getStartPosition() {
   if (startPosition[0]>=0) return startPosition;
   for (int i=0; i<this.maze.length; i++) {
      for (int j=0; j<this.maze[0].length; j++) {
         if (this.maze[i][j]==2) {
            startPosition[0] = i;
            startPosition[1] = j;
            return startPosition;
         }
      }
   }
   return startPosition;
 }
```

Moreover, we also used supplemental functions like **inMaze**, **getMaxX**, and **getPositionValue** to find the position and the length & width of the maze.

One of the most essential function in our Maze class is **scoreRoute** function. This function takes in the route that the individual robot took and gives a fitness score based on the correct steps.

```
public int scoreRoute(ArrayList<int[]> route) {
   this.visited = new int[maze.length][maze[0].length];
   int score = 0;
   for (Object p:route) {
      int[] pos = (int [])p;
      int posX = pos[0];
      int posY = pos[1];
      if (this.maze[posX][posY]==3 && this.visited[posX][posY]!=1) {
          // value 1 is true. We set  ‘‘this.visited[posX][posY] = 1’’
          // because we don’t want the robot to keep visiting the same place
          // and gain all the points
         this.visited[posX][posY] = 1;
         score ++;
      }
   }
   return score;
}
```

3.3 RobotController Class

Here comes the main class. Given the 2-D array of maze and the specified parameters, the RobotController class will use GeneticAlgorithm methods such as crossover and mutation on the population and the individual robot instruction. The main purpose of the RobotController class is to use a while loop to repeatedly evaluate the current population, increment the generation count, print the fittest individual, and perform crossover & mutation until the maximum number of loops is reached. The maximum number of loops is indicated by the variable maxGenerations:

```
// our constructor. We initialize to 1000.
public static int maxGenerations = 1000;


 public static void main(String[] args) {
    // Example of our maze

    // the total number of "3" is 29
    // the position value "3" is the correct tile to step


  Maze maze = new Maze(new int[][] {
  // our 9 x 9 maze
     { 0, 0, 0, 0, 1, 0, 1, 3, 2 },
     { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
     { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
     { 3, 3, 3, 1, 3, 1, 1, 0, 1 },
     { 3, 1, 3, 3, 3, 1, 1, 0, 0 },
     { 3, 3, 1, 1, 1, 1, 0, 1, 1 },
     { 1, 3, 0, 1, 3, 3, 3, 3, 3 },
     { 0, 3, 1, 1, 3, 1, 0, 1, 3 },
     { 1, 3, 3, 3, 3, 1, 1, 1, 4 }
  });



  // Create genetic algorithm
    ...
    ...
    ...
  GeneticAlgorithm ga = new GeneticAlgorithm(populationSize, elitismCount,
       tournamentSize, mutationRate, crossoverRate);

  Population population = ga.newPopulation(chromosomeLength);

  // Keep track of current generation
  int generation = 1;

  ga.Polulation_fitnessScore(population, maze);
```

```
// Start evolution loop
while (ga.ExceedMaxGeneration(generation, maxGenerations) == false) {
   // Print fittest individual from population
   Individual fittest = population.getFittest(0);
   System.out.printf("The fittest Individual after %d generations has
       fitness %.2f: %s\n", generation, fittest.getFitness(),
       fittest.toString());
   // Apply crossover
   population = ga.corssover(population);
   // Apply mutation
   population = ga.mutate(population);
   // Evaluate population
   ga.Polulation_fitnessScore(population, maze);
   // Increment the current generation
   generation++;
   }
  }
```

3.4 Robot Class

Given the instruction set and the maze from the RobotController class, we use the Robot class to test the individual candidate's instruction in the maze.

We first initialize the each candidate's instruction set to the Robot controller:

```
public Robot(int[] sensorActions, Maze maze, int maxMoves){
    this.sensorActions = this.calcSensorActions(sensorActions);
    this.maze = maze;
    int startPos[] = this.maze.getStartPosition();
    this.xPos = startPos[0];
    this.yPos = startPos[1];
    this.sensorVal = -1;
    // our robot's front direction
    this.headDirect = Direction.EAST;
    this.maxMoves = maxMoves;
    this.movesCounter = 0;

    this.route = new ArrayList<int[]>() {{
     add(startPos);
    }};
}
```

Our variable such as "this.sensorActions" collects an integer array of 128 bits to an integer representing two-digit binary representation. Therefore, the two-digit binary representation such as "10" will be stored as integer "2" in the array.

Next, we use **run** to ensure that the car runs until finishing point or maximum steps is reached or the next step is staying at the current tile.

```
public void run(){
    // we keep the robot to run until runs too much or stops
    // not need to stop
    Boolean stop = false;
    while(!stop){
        this.movesCounter++;
        if (this.getNextAction() == 0 ||
            this.maze.getPositionValue(this.xPos, this.yPos) == 4 ||
            this.movesCounter > this.maxMoves) {
            stop = true;
        }
        this.NextAction();
    }
}
```

calcSensorActions, NextAction, getNextAction, getSensorValue are functions in charge of checking whether the sensor towards a certain direction detects a wall and move the car according to the direction output. For simplicity of this paper their codes will not be displayed.
There are a few other functions for getting specifics of the robot and printing it:

```
public int[] getPosition(){
    return new int[]{this.xPos, this.yPos};
}


private Direction getHeading(){
    return this.headDirect;
}

public ArrayList<int[]> getRoute(){
    return this.route;
}

public String printRoute(){
    String route = "";
    for (Object routeStep : this.route) {
        int step[] = (int[]) routeStep;
        route += "{" + step[0] + "," + step[1] + "}";
    }
    return route;
}
```

3.5 Result

Running our code with the following values for our parameters:

```
int populationSize = 200;
double mutationRate = 0.05;
double crossoverRate = 0.9;
int elitismCount = 2;
int tournamentSize = 10;
int chromosomeLength = 128;
```

Our printed result is below:

```
The fittest Individual after 1 generations has fitness 1.00
The fittest Individual after 2 generations has fitness 2.00
The fittest Individual after 3 generations has fitness 3.00
The fittest Individual after 4 generations has fitness 3.00
The fittest Individual after 5 generations has fitness 3.00
The fittest Individual after 6 generations has fitness 5.00
The fittest Individual after 7 generations has fitness 5.00
The fittest Individual after 8 generations has fitness 5.00
The fittest Individual after 9 generations has fitness 8.00
.....
.....
The fittest Individual after 48 generations has fitness 15.00
The fittest Individual after 49 generations has fitness 29.00
....
.....
.....

The fittest Individual after 998 generations has fitness 29.00
The fittest Individual after 999 generations has fitness 29.00
The fittest Individual after 1000 generations has fitness 29.00
```

We see that after about 48 to 49 generations, we have successfully trained the robot controller to navigate through the maze using their appropriate sensory reactions. From our construction of 9x9 maze, we had total of 29 number of tiles with position value of 3. Thus, we note that our robot controller does reach all the possible and correct tiles in the end.

4 Conclusion

In this project, we have explored a way to train the robot's sensor using Genetic Algorithm. While manually training the physical robot may be daunting, by extracting the robot's sensory instruction as binary bits and running its instruction through simulated maze, we can derive the robot's instruction that can navigate through a path without bumping to the wall.

5 Reference

References

[1] Jacobson, Lee, and Burak Kanber. Genetic Algorithms in Java Basics. Erscheinungsort nicht ermittelbar: Apress, 2015.