Final Project – Robotic Controller using Genetic Algorithm

INFO6205 – Program Structures and Algorithms

Project Group 515

Aakash Deogaonkar – 001465733

Mrunal Ghorpade – 001475183

Yash Lekhwani – 001477242

Contents

Abstract

The report explores the application of genetic algorithm in the navigation of a robotic controller autonomously in an unknown environment. The robot performs actions/moves based on sensor inputs. The robot has six sensors attached to it – three in the front, one on the left, right and one at the back. For example, if a robot has *n* sensors to detect anomalies and accordingly navigate in a maze, it takes decisions based on an action sequence 2^n with 2 being the base since the sensor output is always binary – 0 or 1. Our goal is to utilize genetic algorithm and build a robotic controller that can respond to any given input actions without any external stimuli. In this project, we use an 8 by 8 maze as the testing environment for the robot.

Introduction

Genetic algorithms are a type of search optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a function. These algorithms are often applied to robotics as a method of designing sophisticated robotic controllers that enable a robot to perform complex tasks and behaviours, eliminating the need to manually program a complicated robotic controller. The problem with autonomous robotic navigation is how to link the sensor data to motor actions so that the robot can navigate in an environment. The field of artificial intelligence where genetic algorithms, and more generally, the ideas of Darwinian evolution are applied to robotics is referred to as evolutionary robotics. Typically, a genetic algorithm evaluates a large population of individuals to locate the best individuals for the next generation. Evaluating an individual is done by running a fitness function that gauges the performance of an individual based on certain pre-defined criteria. However, applying genetic algorithms and their fitness functions to physical robots gives rise to a new challenge; physically evaluating each robotic controller isn't feasible for large populations. This is due to the difficulty in physically testing each robotic controller and the time taken for such testing. For this reason, robotic controllers are typically evaluated by applying them to simulated models of real, physical robots and environments. This enables quick evaluations of each controller in software that later can be applied to their physical counterparts. In this project, we use our knowledge of binary genetic algorithms to design a robotic controller and begin applying it to a virtual robot in a virtual environment.

Section 1 explains what makes up a genetic algorithm and how they operate. Section 2 will explain about the details behind our testing environment and robot navigation problem. Section 3 demonstrates how we go about solving the problem – the flow of our proposed genetic algorithm. Section 4 explains the reason we followed this certain approach. Section 5 discusses the test cases we executed. Section 6 illustrates our observations and Section 7 concludes our project.

Key Terms

Individual: In genetic algorithm, an individual is an entity which can be compared to genes on which the fitness function is applied, also known as genome. In our problem, the individual or genome is action performed by robot. The value of fitness function for an individual is its fitness score.

Population: Population is a subset of candidate solutions/individuals in the current generation. It can also be defined as a set of chromosomes. For example, in our problem, the population is 200 and the variable in fitness function is maze matrix. Population is the collection where all genetic operators such as mutation and crossover are applied.

Generation: In Genetic Algorithms, new sets of hypotheses are formed from previous sets of hypotheses, either by selecting some full chromosome (generally of high fitness) to move forward to a new generation unscathed (selection), by flipping a bit of an existing full chromosome and moving it forward to a new generation (mutation) or by breeding child chromosomes for the new generation by using an existing set's genes as parents. A generation, thus, is simply the full set of the results of a single iteration. In our case, we are running 500 generations.

Chromosome: A possible solution to a given problem on which fitness function is applied to find optimum solution.

Gene: Gene is an element position of chromosome which can also be referred as parent.

Genotype: Genotype is the population present in computation space which can easily be manipulated. In our problem Genotype expression will be sensors data which is in binary value.

Phenotype: Phenotype is the population in the real-world solution space in which solutions cannot be manipulated. In this problem phenotype is the action performed by robot after getting the command.

Fitness Function: A fitness function is function which checks the suitability of the candidate solution as output.

Genetic Operators: These operators manipulate the genetic composition of population or offspring. For example, crossover, mutation, selection, etc.

Mutation: The process in which genes in a candidate solution are randomly altered to maintain the diversity in two successive generations.

Crossover: The process in which chromosomes are combined to create a new candidate solution. This is sometimes referred to as recombination. There are various type of crossover e.g. Single point crossover, double point crossover. In our problem, we are using single point crossover.

Selection: This is the technique of selecting candidate solutions to breed the next generation of solutions.

Fitness Score: A score which measures the extent to which a candidate solution is adapted to suit a given problem.

Parameters

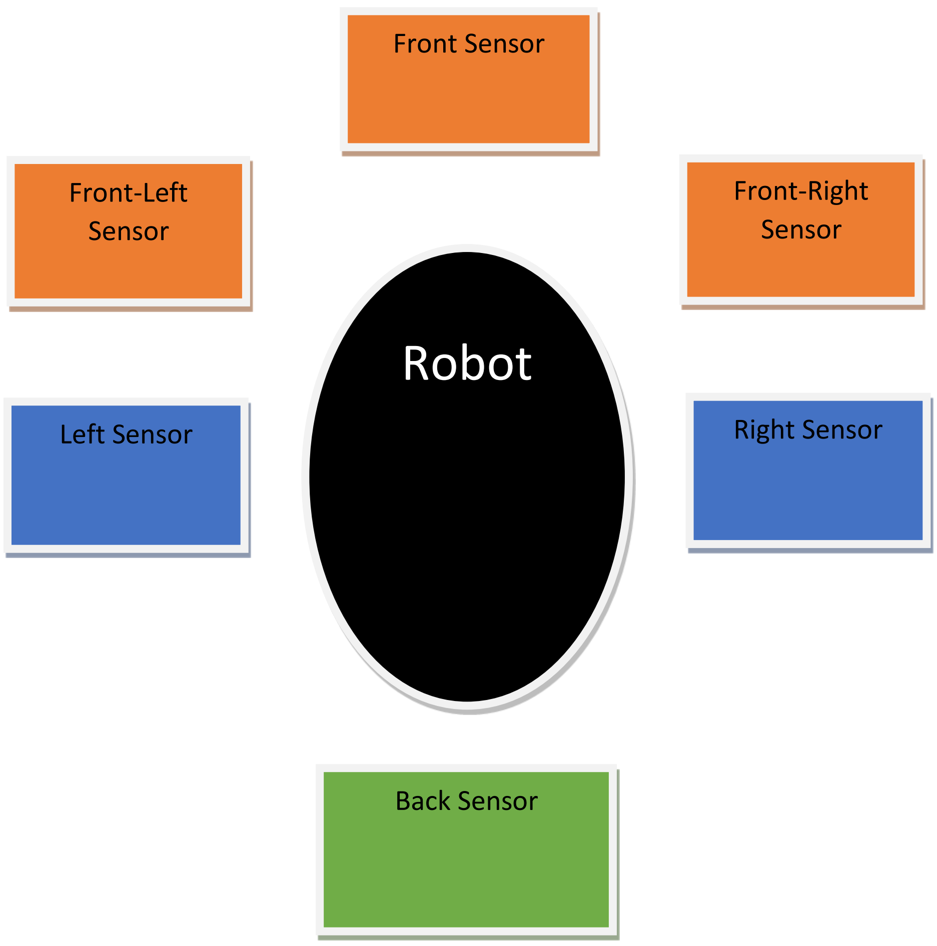
Mutation Rate: It is the probability in which a parent gene of a chromosome will be mutated. If mutation rate is too low, the algorithm will take longer time to find the same solution.

Crossover Rate: It is the frequency of crossover operation on one population. Higher the frequency the rate of finding new superior genes is higher.

Population Size: It is the number of individuals present in one generation.

Details

Robot Configuration:



The problem we addressed is designing a robotic controller that can use the robot's sensors to navigate a robot successfully through a maze. The robot can take four actions: move one-step forward, turn left, turn right, or, rarely, do nothing. The robot also has six sensors: three on the front, one on the left, one on the right and one on the back.

Maze:

1	**0**	**1**	**0**	**2**
1	**1**	**1**	**1**	**3**
1	**1**	**3**	**3**	**3**
3	**3**	**3**	**1**	**0**
4	**1**	**1**	**1**	**1**

Route specifications

0 = Empty

1 = Obstacle/wall

2 = Starting position

3 = Route

4 = Goal position

The maze we explored is comprised of walls that the robot cannot cross and will have an outlined route, shown in Figure, which we want the robot to follow. Our purpose is to automatically program a robot controller with six sensors so that it doesn't crash into walls; we're using the maze as a complicated environment in which we could test our robot controller

Flow

Our robot will have four actions: do nothing, move forward one step, turn left and turn right. These can be represented in binary as:

• "00": do nothing

• "01": move forward

• "10": turn left

• "11": turn right

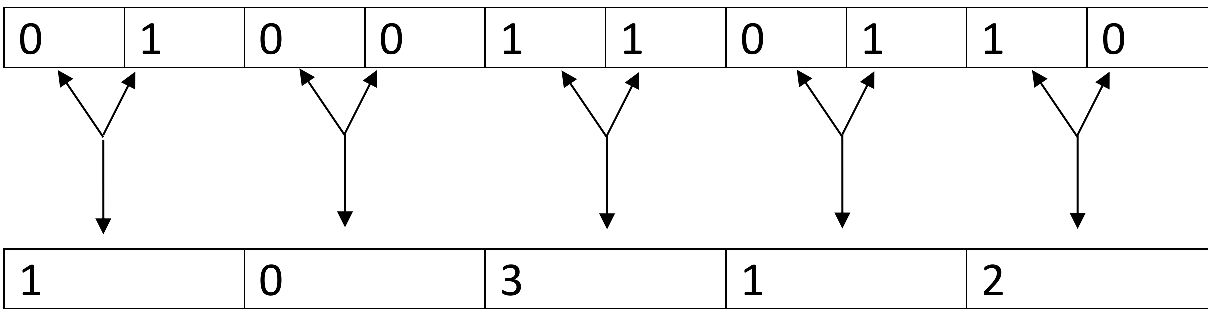
Following is the flow of the algorithm, we have used in the project. In this project, we have taken a sample maze to solve where the robot will traverse the path from one point to other as aforementioned. When the sensor data will give us the number of genes, we have 6 sensors which give values in binary form on (1) and off (0) giving 2^6 = 64 unique combinations. Since each combination encoding takes a memory of 2 bits, the storage for this problem will be 64 * 2 = 128 bits. In the algorithm, chromosomes are easiest to manipulate as an array. We have considered 128 as the number of chromosomes.

Explanation

1. **Initialize population**: In this problem, the genetic Algorithm starts with the population of 1000 generations which helps us to create the maze.
2. **Creation of Test Maze**: The test maze is created using the population given where 3 will guide us through the path that the robot has covered to reach from starting point at 2 to the goal at 4.
3. **The run function**: The run function is called with 128 chromosomes which it gets from the sensor inputs.
4. **Scoring the route**: It will evaluate the route and compare with the best route for fitness.
5. **Tournament Selection**: If the generation count is less than the maximum number of generations, it will go to tournament selection otherwise, it will give the result. Tournament selection provides a method for selecting individuals based on their fitness value. Thus, the higher an individual's fitness the greater the chance that the individual will be chosen for crossover.
6. **Crossover**: We have adopted the Single Point Crossover. Single point crossover is a very simple crossover method in which a single position in the genome is chosen at random to define which genes come from which parent.
7. **Mutation**: The mutation is used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next.

Relation Between Genotype and Phenotype

128-bit array – Sensor Values

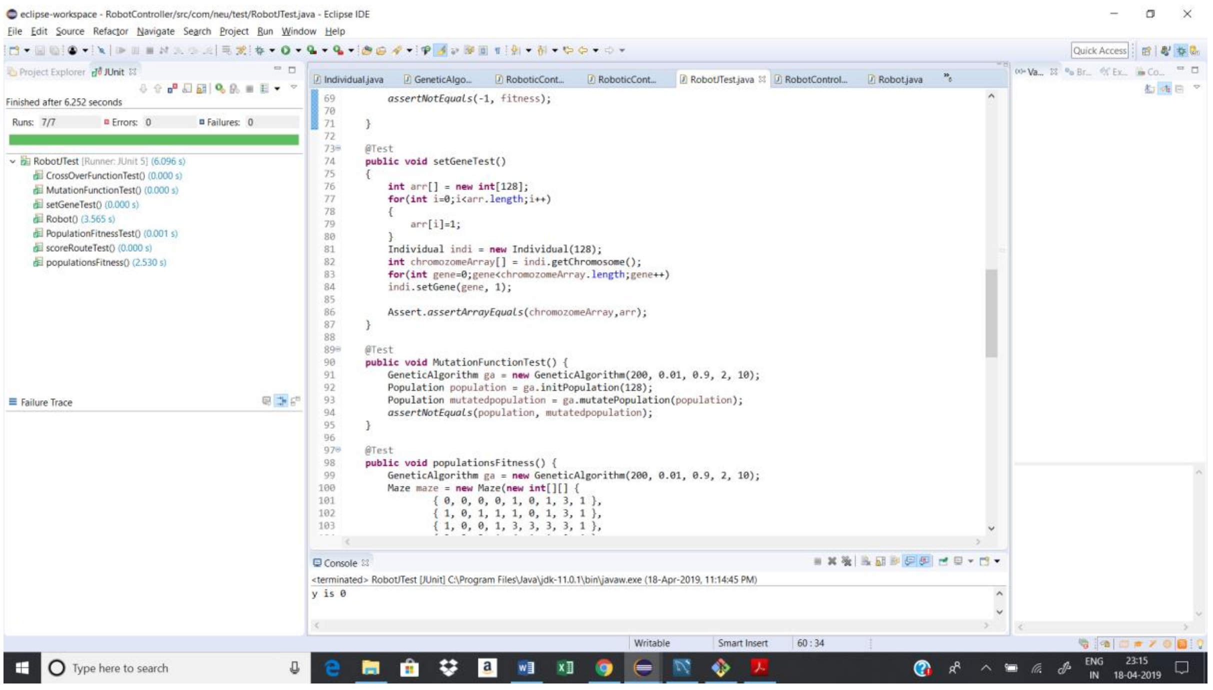


Sensor Actions

Here, sensor values are input in binary and form the genotype and sensor actions form the phenotype. Above shown is our mapping between the genotype and phenotype.

Test Cases:

To verify the correctness of our algorithm and approach, we designed 6 unique test cases (refer: https://github.com/aakashdeogaonkar/INFO6205_515). The test cases were tested against each function used in the genetic algorithm. The test case results are as follows:


eclipse-workspace - RobotController/src/com/neu/test/RobotJTest.java - Eclipse IDE

```
Finished after 6.252 seconds
Runs: 7/7    Errors: 0    Failures: 0

RobotJTest [Runner: JUnit 5] (6.096 s)
    CrossOverFunctionTest() (0.000 s)
    MutationFunctionTest() (0.000 s)
    setGeneTest() (0.000 s)
    Robot() (3.565 s)
    PopulationFitnessTest() (0.001 s)
    scoreRouteTest() (0.000 s)
    populationsFitness() (2.530 s)
```

```java
        assertNotEquals(-1, fitness);

    }

    @Test
    public void setGeneTest()
    {
        int arr[] = new int[128];
        for(int i=0;i<arr.length;i++)
        {
            arr[i]=1;
        }
        Individual indi = new Individual(128);
        int chromozomeArray[] = indi.getChromosome();
        for(int gene=0;gene<chromozomeArray.length;gene++)
        indi.setGene(gene, 1);

        Assert.assertArrayEquals(chromozomeArray,arr);
    }

    @Test
    public void MutationFunctionTest() {
        GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.01, 0.9, 2, 10);
        Population population = ga.initPopulation(128);
        Population mutatedpopulation = ga.mutatePopulation(population);
        assertNotEquals(population, mutatedpopulation);
    }

    @Test
    public void populationsFitness() {
        GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.01, 0.9, 2, 10);
        Maze maze = new Maze(new int[][] {
                { 0, 0, 0, 0, 1, 0, 1, 3, 1 },
                { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
                { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
```

```
<terminated> RobotJTest [JUnit] C:\Program Files\Java\jdk-11.0.1\bin\javaw.exe (18-Apr-2019, 11:14:45 PM)
y is 0
```

Test Case 1 (PopulationFitnessTest())

- **Use Case**: To test our customized Fitness Function in the Genetic Algorithm

- **Condition given**: The default fitness will be updated by running calcFitness()
- **Expected Result**: Fitness value should not equal default value

Test Case 2 (setGeneTest())

- **Use Case**: To test our customized setGene() Function in the Genetic Algorithm
- **Condition given**: Check the initialization of gene.
- **Expected Result**: The chromosome array should equal to the user-defined gene array

Test Case 3 (MutationFunctionTest())

- **Use Case**: To test our customized Mutation Function in the Genetic Algorithm
- **Condition given**: A new population is generated by running the mutateFunction().
- **Expected Result**: The newly generated population must not equal old population

Test Case 4 (CrossoverFunctionTest())

- **Use Case**: To test our customized Crossover Function in the Genetic Algorithm
- **Condition given**: A new population is generated by running the crossoverPopulation().
- **Expected Result**: The newly generated population must not equal the old population.

Test Case 5 (populationsFitness())

- **Use Case**: To test our customized Fitness Function in the Genetic Algorithm
- **Condition given**: Genetic Algorithm is run with required parameter hence the end fitness value is best score route of the maze. This is not to infer best path taken but the most optimum chromosome is created.
- **Expected Result**: Fittest individual in the population will have best score route of given maze. Best score means the route taken by the robot is what we have designed in the input maze.

Test Case 6 (Robot ())

- **Use Case**: To test Run () method in the Robot class in our Genetic Algorithm
- **Condition given**: Robot is run using best chromosome generated by GA in new maze i.e. the designed controller is now put into new test environment to check if robot is not hitting any obstacle
- **Expected Result**: Robot route final path must not be an obstacle/wall, so it is inferred that robot has avoid the obstacle in its course of travel.

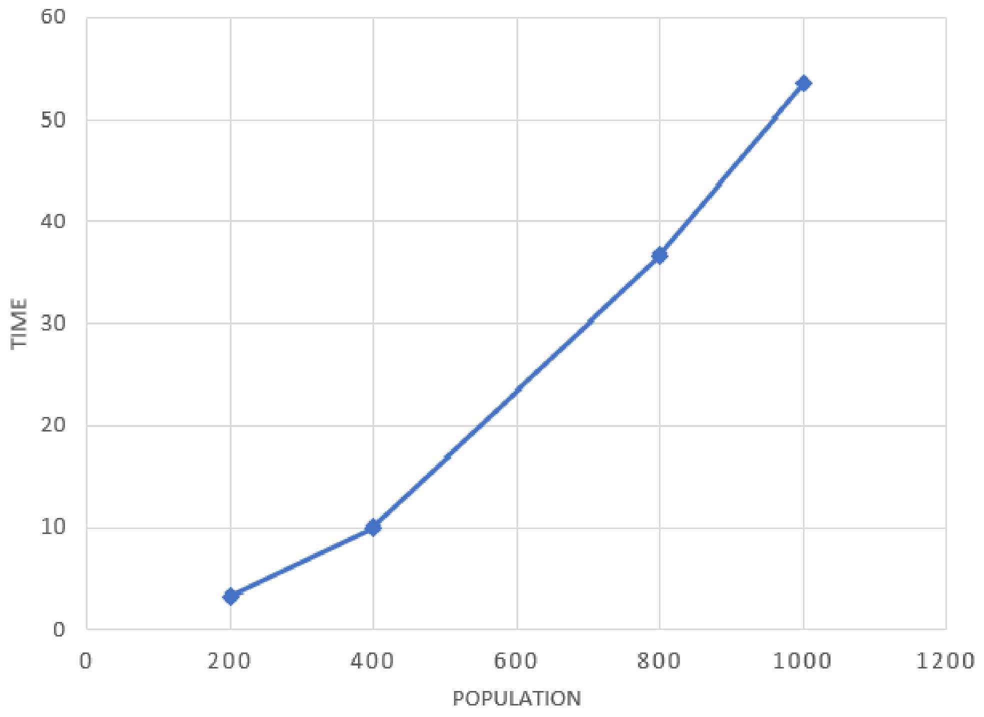
Test Case 7 (scoreRouteTest())

- **Use Case**: To test our customized scoreRoute() Function in the Genetic Algorithm
- **Condition given**: Best route is given to the function, so it must return no of best tiles
- **Expected Result**: Returned score must be equal to no of best tiles that we set in the maze.

Results:

Outcome 1: Population vs Time graph

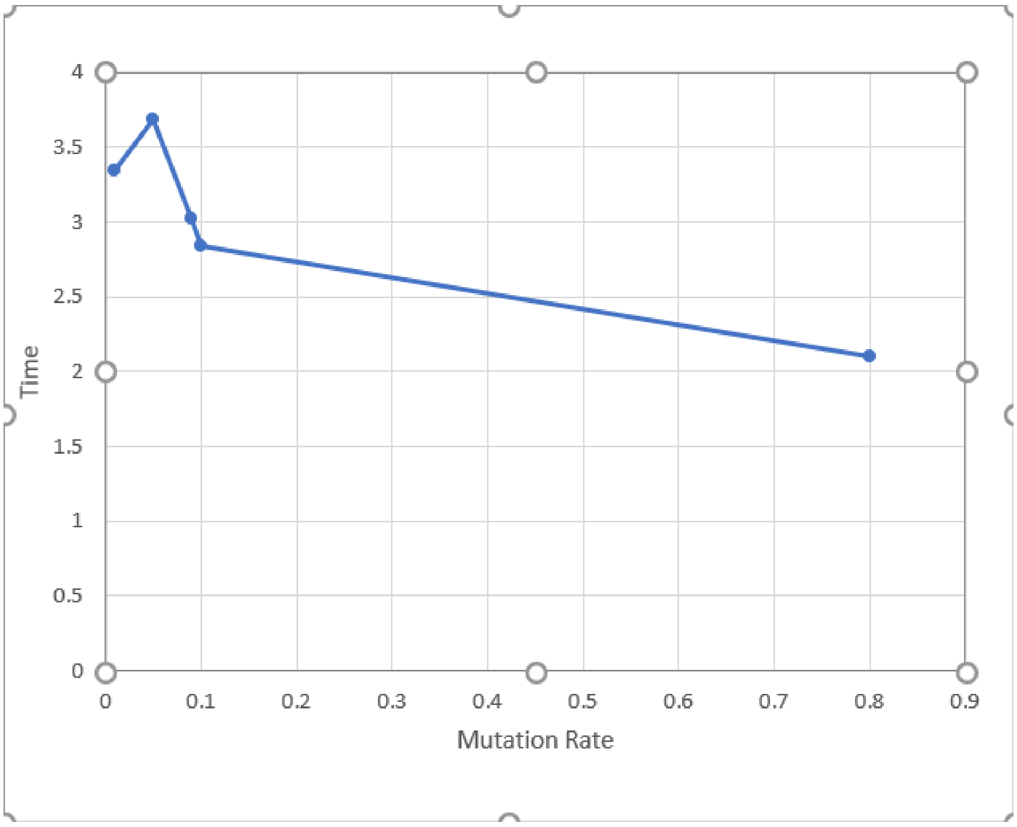
Population	**Time (in sec)**
200	3.153
400	10.02
800	36.602
1000	53.503



Observation: It is observed that as the number of populations increased, the time of execution also catapulted. Although, the best fit solution was found in the earlier generations itself when implemented with the bigger population.

Outcome 2: Mutation Rate vs Time graph

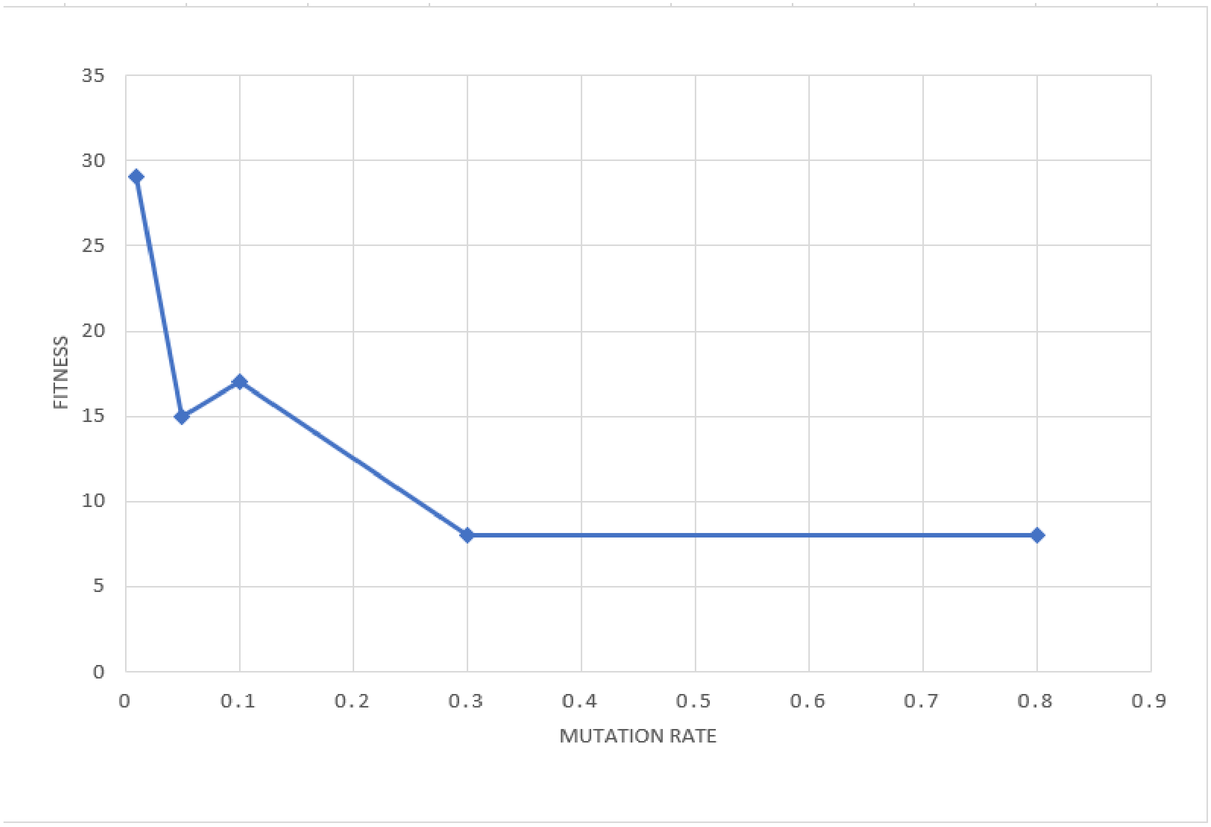
Mutation Rate	**Time (in sec)**
0.01	3.339
0.05	3.683
0.09	3.021
0.1	2.8363
0.8	2.105



Observation: It is observed that as the Mutation rate increased, the time required to run the code increased. Mutation is used to maintain the diversity in every generation and it is observed that mutation is an essential feature to converge in a Genetic Algorithm.

Outcome 3: Mutation Rate vs Fitness graph

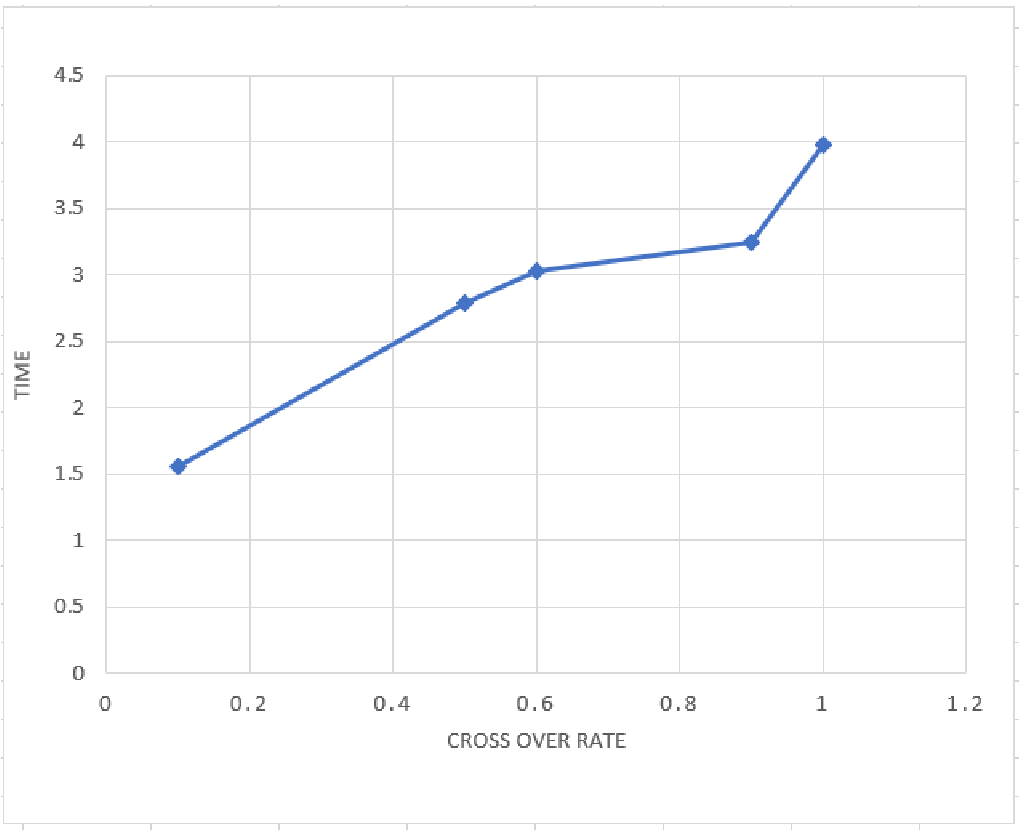
Mutation Rate	**Fitness**
0.8	8
0.01	29
0.05	15
0.1	17
0.3	8



Observation: It is observed that as the Mutation rate increased, the fitness of the algorithm reduces. Mutation is used to maintain the diversity in every generation and it is observed that mutation is an essential feature to converge in a Genetic Algorithm.

Outcome 4: Crossover Rate vs Time graph

Crossover Rate	Time (in sec)
0.9	3.247
0.1	1.553
0.5	2.787
1	3.981
0.6	3.03



Observation: It is observed that for a very low crossover rate, the genetic algorithm has a high execution time but after a certain rate, it is almost linear. This is what is inferred from the genetic algorithm: With a low crossover rate, the diversity is too low and thus, the evolution is at a very less rate.

Conclusion

We have now learned how we could apply a Genetic Algorithm to arrive to a solution with N-dimension problem space. Here, we had a 2-dimensional use case with four resultant sequences. Mapping for this problem requires a 2^N * 2 solution set. For a larger value of N, we find the genetic algorithm effective. With evolution theory we find a converging solution in few mins/sec instead of million or billion more years if it is manually commuted and solved.

References

- Jenna Carr (May 16 2014) Introduction to Genetic Algorithms
  https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf
- Lee Jacobson and Burak Kanber Genetic Algorithm in Java Basics
- Tutorialspoint
  https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm
- GeeksForGeeks
  https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm