**Genetic Algorithm**: Finding the Best Path for a Maze Solution
**Team Members:** Xingyao Wu(001937257) Zhiwei Zhang(001833899)
**Team Number:** 320
**Final Draft:** April 13, 2018

---

**Problem Inspiration:**
By inspiration the maze games on the Internet, I and my teammate have decided to use genetic algorithms to find an optimistic path given the starting point(entrance) and the ending point(exit) shown as the graph below:

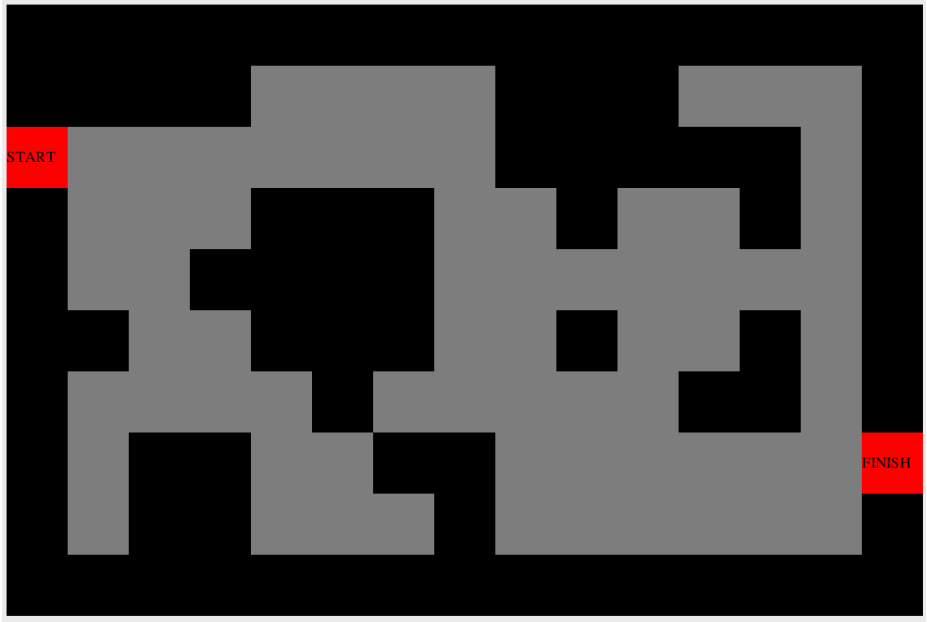



***Figure 1: GUI of the maze***

In our application, users are given freedom to configure these independent variables:

1. Starting Point
2. Ending Point
3. Map of the maze using a 2-dimensional integer array
4. Mutation Probability
5. Crossover Probability
6. Chromosome size
7. Gene Size

**Design Implementation:**
*Genetic code:* As a typical gene type, our gene object has all the essential fields such as the length of the gene, an integer array representing the genotype which contains the behavior of this gene. Lastly our gene has the fitness score.
Below is a table which representing the rule of movement(genotype) we implemented:

| First value stored in the index | Second value stored by the index | Behavior |
|---|---|---|
| 0 | 0 | Moving Up |
| 0 | 1 | Moving Down |
| 1 | 0 | Moving Left |
| 1 | 1 | Moving Right |

*Gene Expression:* Our program iterates the gene sequence from the first index till the last index to parse the movement.

*Crossover and Mutation Step*: At beginning, we initialize the gene group by using the chromosome population number that user defined. Each gene is assigned with random integers ranging 0 to 1 which means random path solutions are generated for each gene in the first phase of the algorithm. Of course, these solutions are nearly impossible to be the correct path users looking for so that further mutation and crossover steps are applied to find the fittest path.

In our mutation step, our program firstly picks up two genes and writing their gene data into two new integer arrays. Then our program uses a random function to determine whether crossover step should be applied based on crossover rate. If do, our program will exchange the gene series between the picked two genes shown as the graph below:

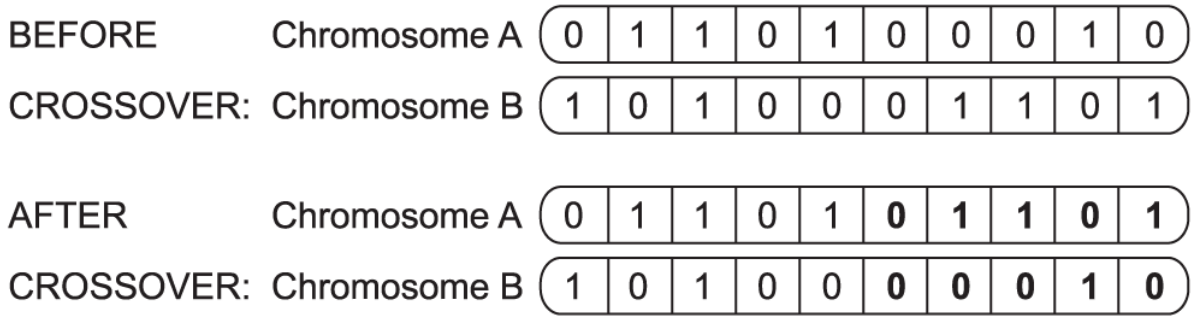



***Figure 2: Crossover solution***

Secondly, our program uses the same random function to determine if mutation should be applied based on the mutation probability user defined. If so, a random index from these two series is chosen to flip the gene series into an opposite value.
The graph below shows how the mutation step works:

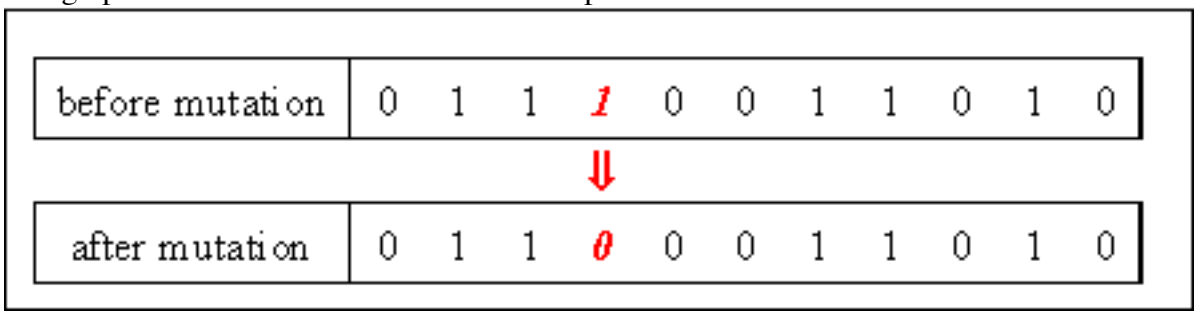



***Figure 3: Mutation solution***

*Fitness Function:*
The goal for our program is to find an optimistic path from the beginning point to the ending point, so our fitness function is designed to find how close our gene can search for the destination. In our fitness function, our program keep calculating the absolute distance between the current moving point and the destination, shown as the formula below: also to avoid the repeat step, we have defined variables to store times of duplicate step, if duplicate exists, score should be deducted.

```
double x = Math.abs(X - endingX);
double y = Math.abs(Y - endingY);
```

```
if ((x == 1 && y == 0) || (x == 0 && y == 1))
  return 1;
return (1 / (x + y + 1 + refind));
```

***Figure 4: Code for calculating the fitness score***

From the code above, variable “x”, “y” are the current 2-Dimension coordinate and variable “ending”, “endingY” are the 2-Dimension coordinate for the destination.

*Evolution*:
Our program uses rotary method to determine which genes are picked to crossover and mutate. Since genes are stored as data in the Array List in our Genetic Algorithm class, we determined the position of gene by firstly creating a random double ranging from 0 to 1 and then our program iterates the list of genes, if the fitness score of the iterated gene is greater than this double value, our program will choose the gene with this index. Then mutation and crossover procedure would begin after our program repeats the procedure described above again.
Below shows the code indicating how our program randomly chooses the genes:

```
Random random = new Random();
int position = 0;
double score, sum = 0, num = random.nextDouble(), total =
getTotalFitnessScore(geneGroup);
for (int i = 0; i < geneGroup.size(); i++) {
  Gene gene = geneGroup.get(i);
  score = gene.getFitnessScore();
  sum += score / total;
  if (sum > num)
    return i;
}
return position;
```

***Figure 5: Code for evolution***

*Sorting:*
During the evolution procedures, there might be more than one qualified gene that can find the correct path from the origin to the destination. Our program is designed to output the best route to the user, so we use a tree map to sort the qualified gene by the fitness score.

*Logging:*
We logged the procedures of mutation and crossover happen. Also, our program keeps tracking of the best fitness score.

*Junit Testing:*
Our program defines many helper functions in all classes, so we implemented Junit testing methods to test whether theses helper functions behave normally.
The snapshot of the testing is shown below:
Gene Testing:

Setup: we created four genes which contains solution to move towards up, down, left and right all the times i.e.: the sequence contains a repeat combination of “11”, “10”, “01” and “00”.

Result: All the four tests have passed.


***Figure 6: Prove of Gene Junit Test***

Genetic Algorithm testing:

Setup: for this testing, we are mainly focusing on testing mutation, crossover and getFitnessScore function which are the core part for this algorithm. We extract the part that is not randomized and pass with fixed parameters.

Result: All the three testings have passed.

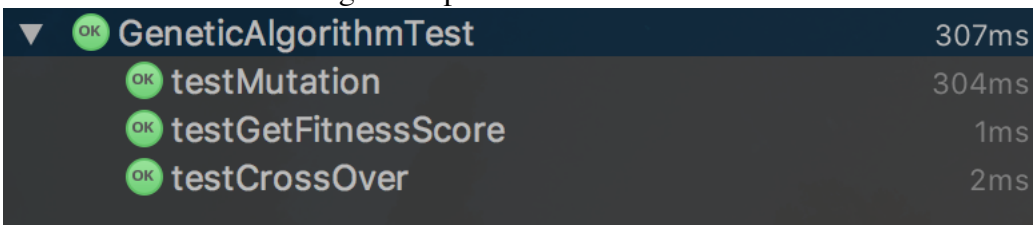



***Figure 7: Prove of GA Junit Test***

*GUI Output:*

Below snapshot is a typical optimistic solution our program found after mutation and crossover 50000 times:

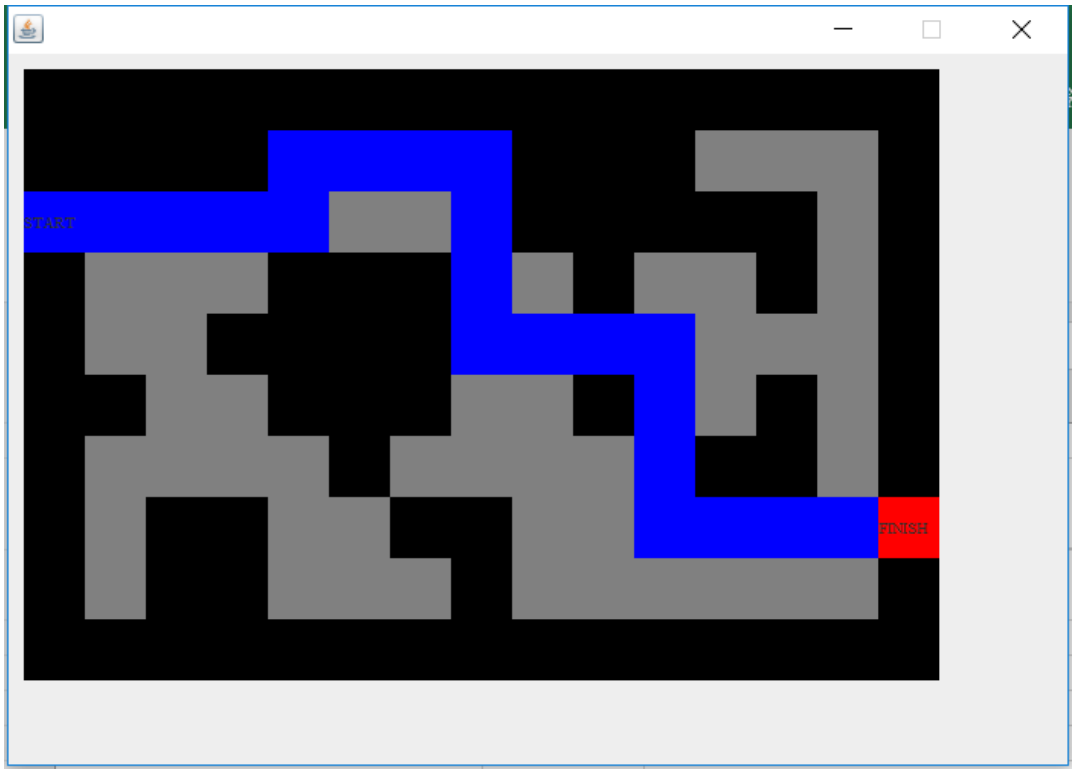
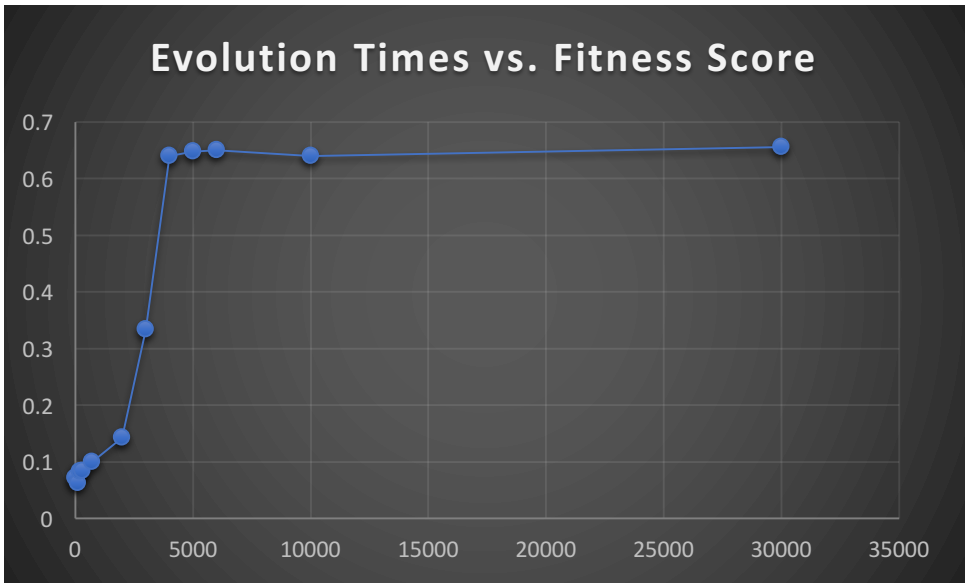


*Figure 8: GUI for the maze solution*

After experiments, we conclude that when the times of iterations increases, the more accurate and optimistic solution will be found.
The data is shown below:




. *Figure 9: Graph of Evolution vs Fitness Score*

From the graph, we can see the interval between 0 to 4000 evolution has a little increment comparing with the 4000 to 6000 intervals. From the actual graphic interface, we can see that before 4000 the program can find the correct path towards the destination. After 5000 intervals, our algorithm could find a correct path towards the destination however these are more repeated steps happen. After running towards 20000 and 30000 times, we found that the repeated steps have been reduced due to the higher score (the higher the score, the lower repeated steps will be occurred). So, I believe repeating the evolution around 20000 times is a optimistic balance between the running power and the calculated result.