Genetic Algorithm Implementation in Pyamaze Module

Abiam Asif Khalid

7 min read · Apr 2

Autonomous path planning, a classical programming problem implemented through various algorithms and techniques. It has multiple practical uses in industrial application, delivering packages, surveillance and can even safe human lives in natural disasters.

We are diving into this same problem using **Genetic Algorithm** (Search Heuristic inspired by Theory of Evolution by Natural Selection) with the help of **pyamaze** for building the maze and the GUI for the task. The programming language to be used is Python version 3.10.5

Genetic Algorithm solves this problem by the following steps:

- Creating a random population of chromosome
- Evaluating the most fittest using fitness function
- Selecting the parent chromosomes and producing daughter chromosomes using crossover
- Mutation in population
- Repeating bullets 2~4 until the solution is found.

Path Planning

The most important aspect of ***Solving Maze*** is the path planning. For the purpose of this project we mainly used two different types of path, both implemented to both square [n*n] and rectangular [m*n] mazes. The path basically is derived from the **chromosomes** in the population for that we must first discuss the chromosome. The two methods of path planning are C***olumn First*** and ***Row First***.

> ***Chromosome: The chromosome is represented by a list (list object in python), containing different genes. In each chromosome the list index represents the column and value of the***

> *element at each index represents row. Each gene represents the milestone for path.*

Column First : For column first, the path moves in columns first and if the columns are same then proceeds to move in rows.

Such that the chromosome **[1 1 3 6 9 9 9 9 10 5 7 7 7 12 15]** can be represented as X, and the intermediate steps are represented as O.

+	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	X	X													
2		O													
3		O	X												
4			O												
5			O						O	X					
6			O	X					O	O					
7				O					O	O	X	X	X		
8				O					O				O		
9				O	X	X	X	X	O				O		
10								O	X				O		
11													O		
12													O	X	
13														O	
14														O	
15														O	X

Column Wise Path for above chromosome in a 15×15 Grid (maze)

Row First : In row first, we follow the same representation but the path checks for the row first and then changes the column.

Such that the chromosome **[1 1 3 6 9 9 9 9 10 5 7 7 7 12 15]** can be represented as X, and the intermediate steps are represented as O.

+	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	X	X	O												
2			O												
3			X	O											
4				O											
5				O						X	O				
6				X	O					O	O				
7					O					O	X	X	X	O	
8					O					O				O	
9					X	X	X	X	O	O				O	
10									X	O				O	
11														O	
12														X	O
13															O
14															O
15															X

Row Wise Path for above chromosome in a 15×15 Grid (maze)

Functions of Genetic Algorithm

Now we will focus on how the program actually functions. The maze is created by using pyamaze module. Agent (or the character) that moves on the grid by following the path from Genetic Algorithm is also created using pyamaze.

```
from pyamaze import maze,COLOR,agent

# Creating/Loading the maze and agent with starting point(1,1)
m=maze(row, column)

m.CreateMaze(row, column, theme=COLOR.dark, loopPercent=100)

a=agent(m, x=1, y=1, footprints = True)

# After creating the path through GA
m.tracePath({a:path})
m.run()
```

For more information about **pyamaze** please refer to this **blog** by the author of the module.

Generating Random Population of Chromosomes:

This function generates the random population for the maze using built in random module of python. Each chromosome has a length of n-2 (column-2). This is done because the maze starts at position (1,1) and ends at (m, n) for a m*n sized grid. But as we will see later on, the random mutation takes place. To prevent any issues to the path generated we will add these values using a separate function (**chromosomemaker**) when needed.

The actual values of the list (**chromosome**) range from 0 to m-1. Function returns a complete population of chromosomes, implemented as a list of lists.

```
def generate_population(rows, columns, popsize):
    for _ in range (0, pop_size):
        sublist =[]
        sublist= [random.randrange(0,rows) for _ in range(columns - 2)]
        population.append(sublist)

    return population
```

Generating Direction of Chromosomes:

This function works similar to the above mentioned function, but generates a number to decide the direction of path. Where 0 → column first and 1 → row first.

```
def generate_direction():

    direction = [random.randrange(2) for _ in range (pop_size)]
    return direction
```

Completing the Chromosome:

Chromosome maker adds the starting and ending genes for each chromosome for creation of accurate paths. The function takes chromosome (incomplete) as input and returns the completed one.

```
def chromosomemaker(chromosome):
    # chromosome --> each chromosome in population to be completed
    chromo = []
    chromo.append(0)

    for gene in chromosome:
        chromo.append(gene)

    chromo.append(Grid_rows-1)
    return chromo
```

Path Evaluator:

The path evaluator function evaluates the path that was produced for each column and fills the three lists containing data about infeasible steps (moves that cause the path to jump over maze barrier), turns in path, and path length for each chromosome.

Arguments of the function are population, direction and map maze (produced by the pyamaze module).

```
#for checking infeasible steps
        for k in range (0,step-1):
            row,col = path[k]
            if mapmaze[path[k]]['E']==0 and path[k+1]==(row,col+1):
                infeas+=0.25
            elif mapmaze[path[k]]['W']==0 and path[k+1]==(row,col-1):
                infeas+=0.25
            elif mapmaze[path[k]]['N']==0 and path[k+1]==(row-1,col):
                infeas+=0.25
            elif mapmaze[path[k]]['S']==0 and path[k+1]==(row+1,col):
                infeas+=0.25
        inf.append(infeas)
```

The **pathvar1** and **pathvar2** function are used to create path for each chromosome. **pathvar1** creates path for a given chromosome with column first(0), whereas **pathvar2** creates path with row first(1).

Fitness Function and Sorting Functions:

Fitness function calculates the fitness of each chromosome as determined by infeasible steps, turns and path length. Normalizes the inf, turns and steps and evaluates the fitness with different weights for each attribute.

Normalizes here means the scaling the values between 0 and 1. Whereas the weight of the infeasible steps = 5, weight of turns and steps is both = 2. In below code block maxinf = maximum infeasible steps in the population, and mininf = minimum infeasible steps (taken as 0, meaning the solution).

Sorting in the algorithm takes place depending on the fitness value and is in descending order. [Most Fit (e.g. 500) → Least Fit (e.g. 0)]

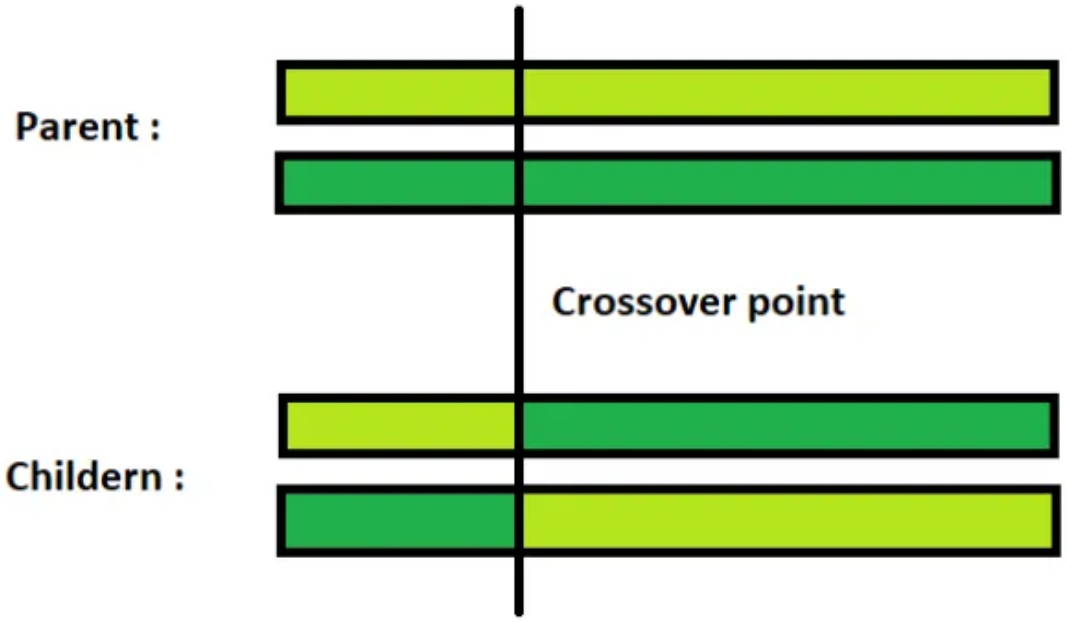
```
def fitnessfn(inf, turns, steps):
     maxinf, maxturns, maxsteps = max(inf), max(turns), max(steps)
     mininf, minturns, minsteps = 0, min(turns), min(steps)
     for j in range (0,pop_size):
        finf = 1 - ((inf[j] - mininf) / (maxinf - mininf))
        fturn = 1 - ((turns[j] - minturns) / (maxturns - minturns))
        flength = 1 - ((steps[j] - minsteps) / (maxsteps - minsteps))

        fitval = 5 * 100 * finf * ((2 * fturn + 2 * flength) / (2 + 2))
        fitness.append(fitval)
    return fitness

    #Sort function can be used as follows
    sortt=[(fitness[i],population[i]) for i in range (0,pop_size)]
    sortt.sort(reverse=True)
    #then unpack the sortt into seperate list for fitness and population and
    #return it.
```

Cross Over Function:

Crossover involves selecting a part of parent 1 and concatenating it with parent 2. Such as to create two new chromosomes as shown below.



The crossover point for our algorithm is midpoint of each chromosome. The crossover takes place between most fit chromosomes. This creates the new generation for the next iteration of the Genetic Algorithm.

Mutation and Finding Solution:

Mutation is placing a random value at a random index of each chromosome. This adds the element of mutation that exists in nature too. Mutation rate for our algorithm is **50%.**

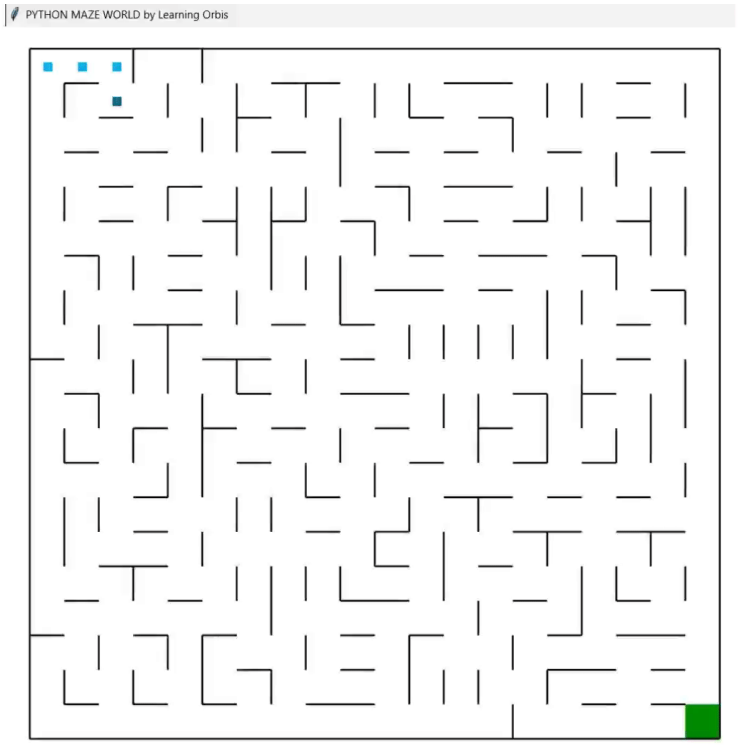
Solution is determined by locating the chromosome with zero infeasible steps, this method prioritizes finding the solution rather than looking for most optimal one.

```
def mutation(population):
    index, val = 0,0
    for i in range (0,pop_size,2):
        index = random.randint(0,Grid_columns-3)
        val = random.randint(0,Grid_rows-1)
```

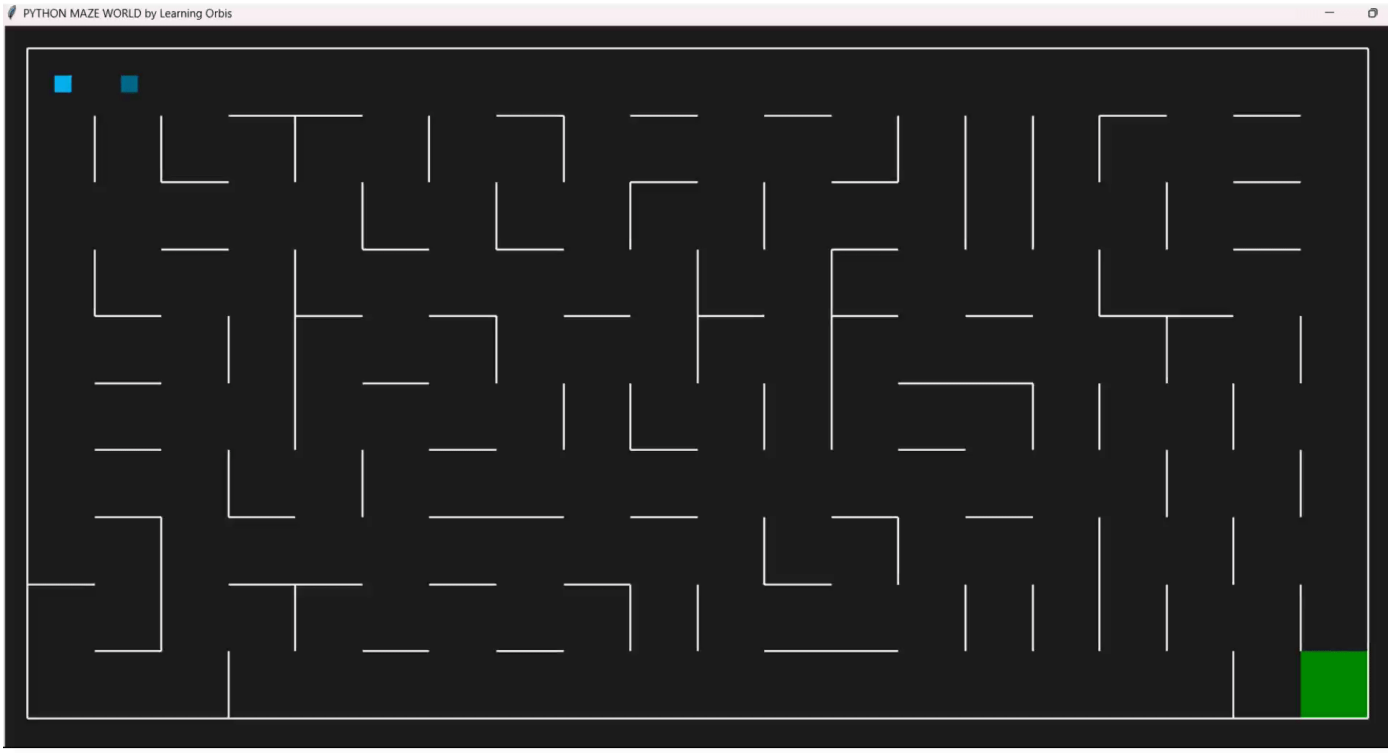
```
            population[i].insert(index,val)
        return population


    def solution(inf):
        for i in range(0,pop_size):
            if (inf[i]==0.00) :
                return i
        return 0
```

Solution



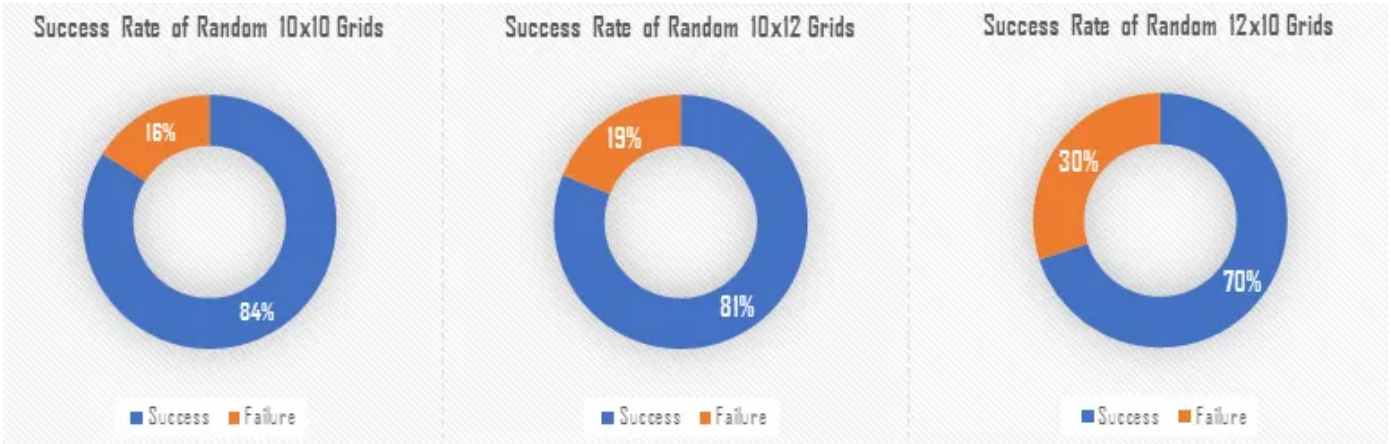
20*20 Maze Solution through the GA

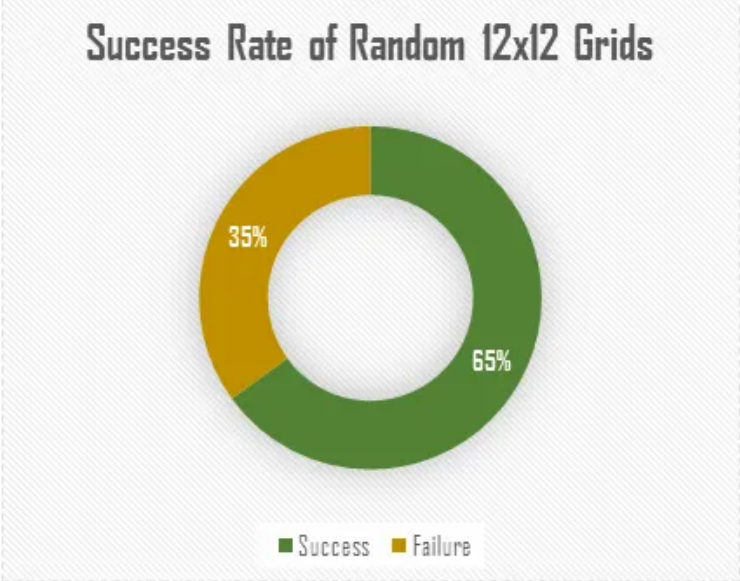


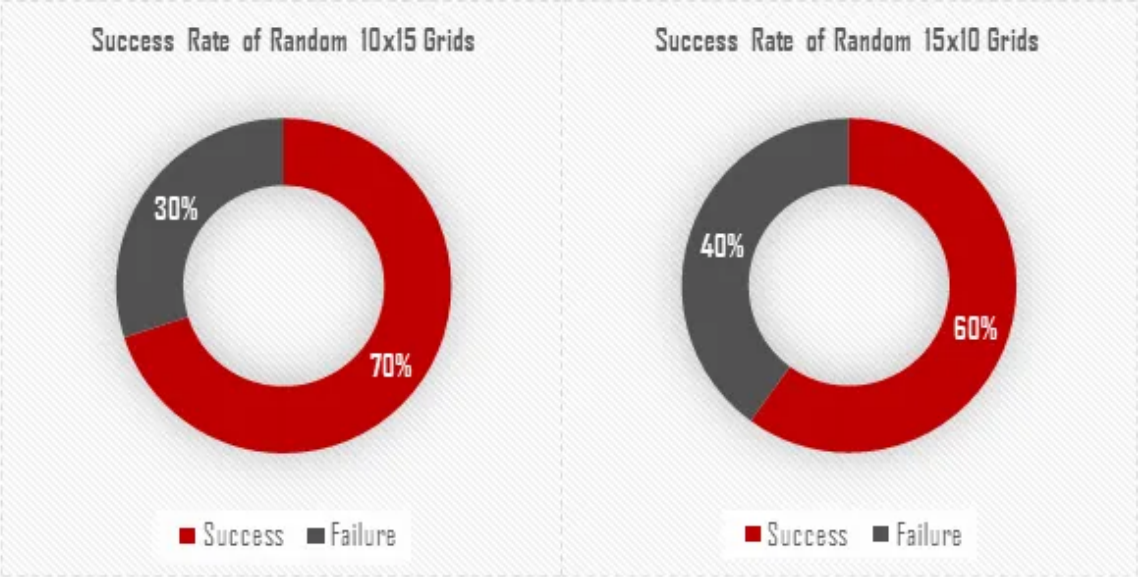
10*20 Maze Solution through GA

Comparison In Random Grids:

The below charts show how the success rate of algorithm varies for random mazes. The data for these grids were collected for 100 iterations. The average generation in which the solution was found follows a similar trend as the success rate.

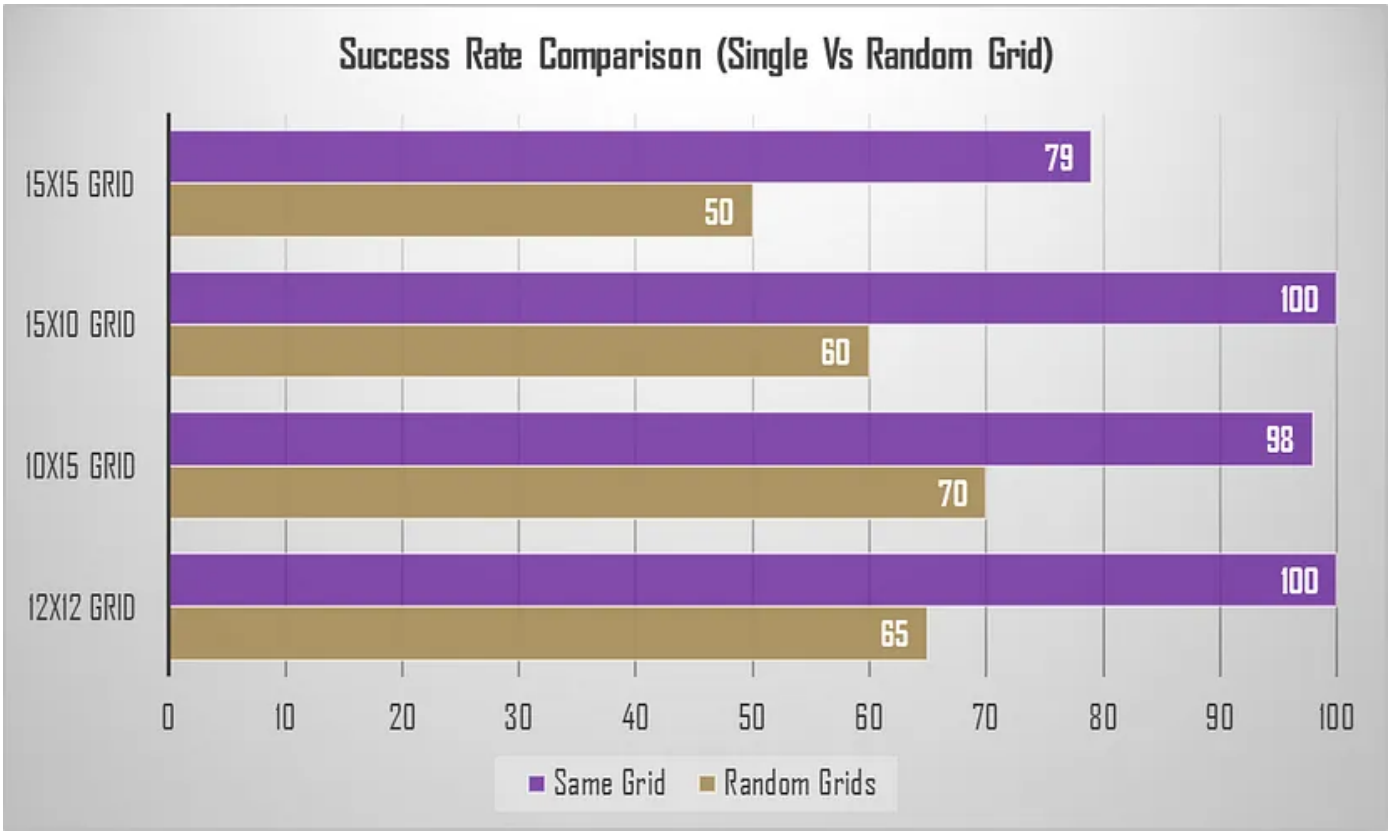






This shows how the the succession rate of random mazes decreases as the size of the grid changes.

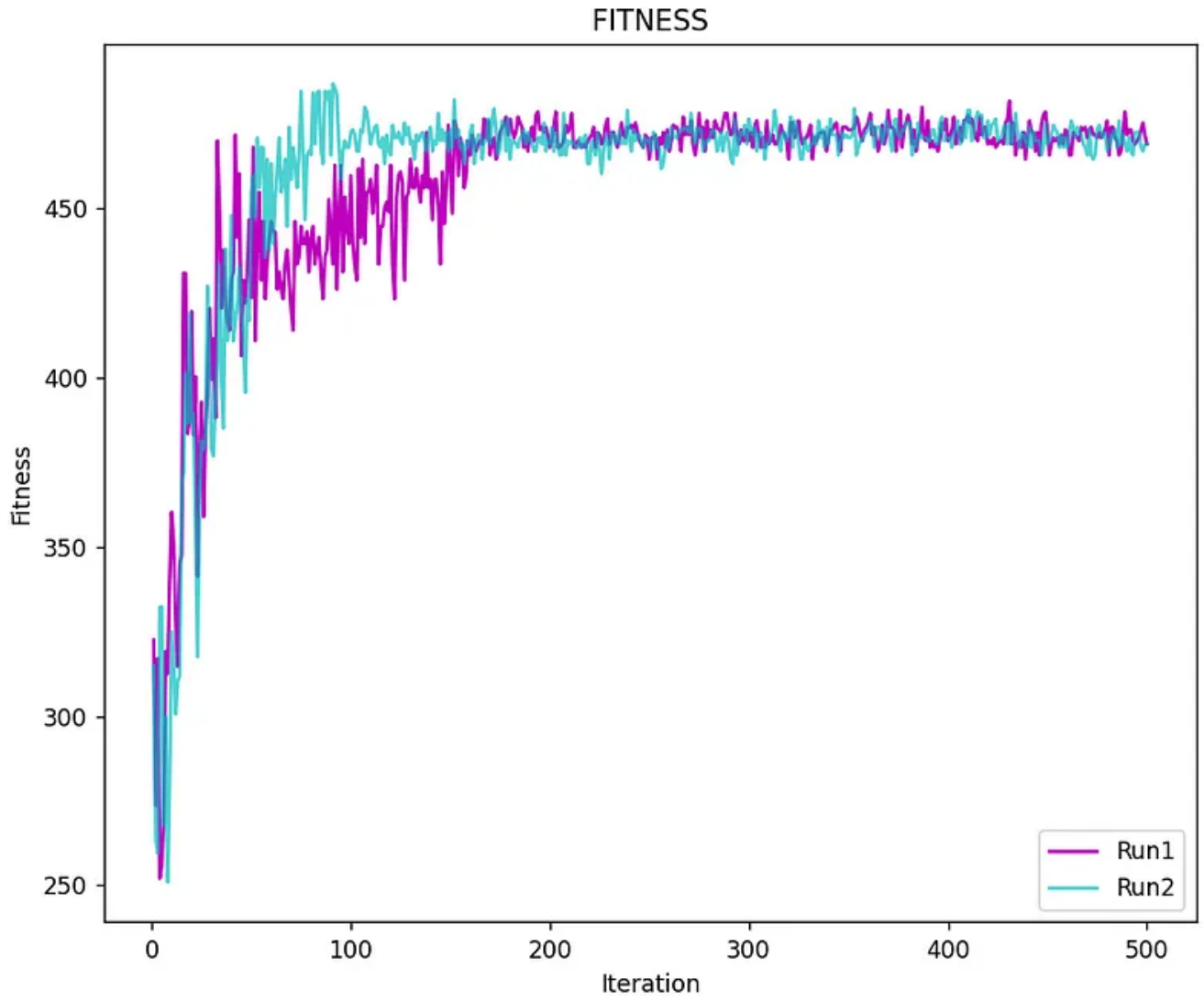
Comparison between Random and Same Grids:

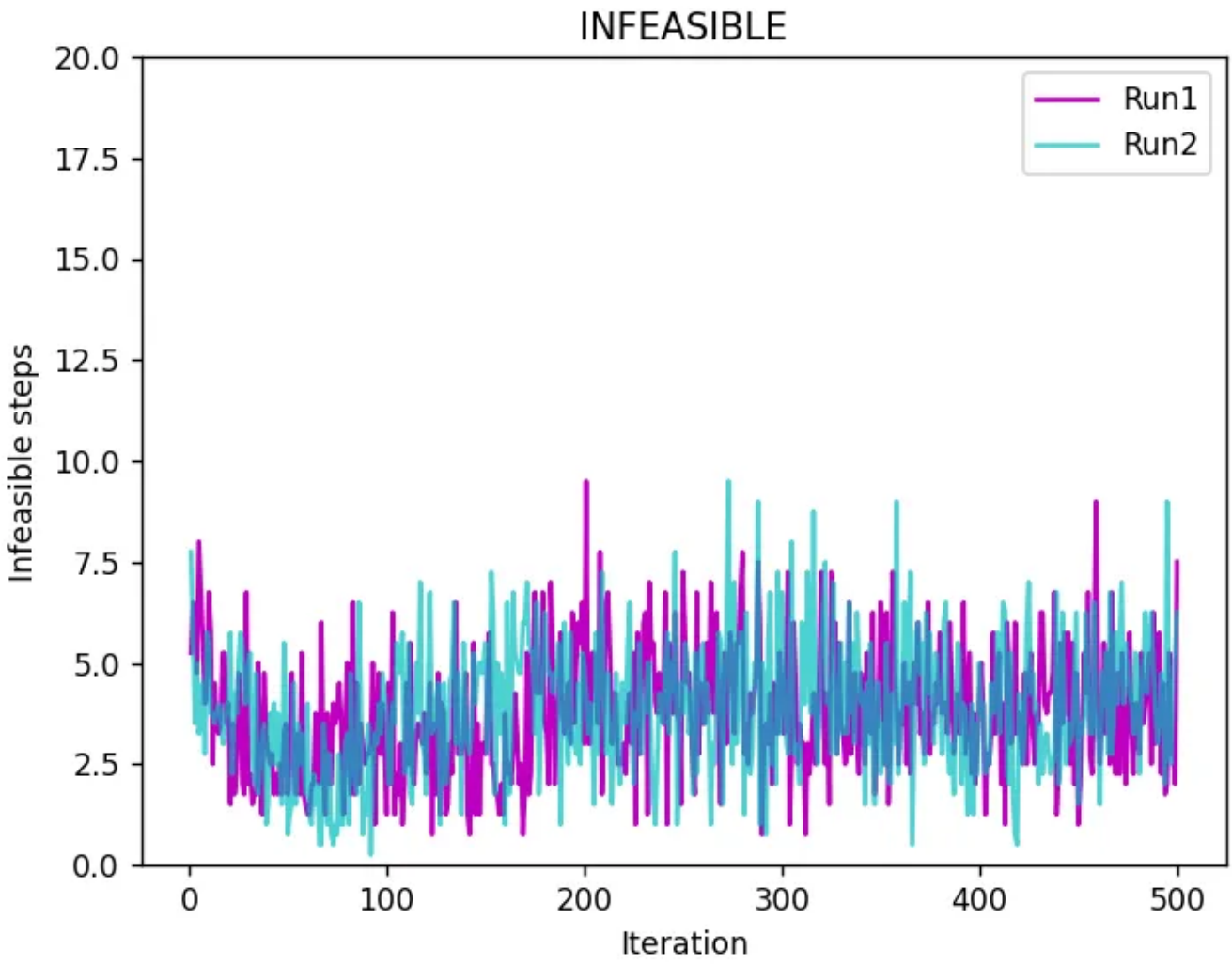


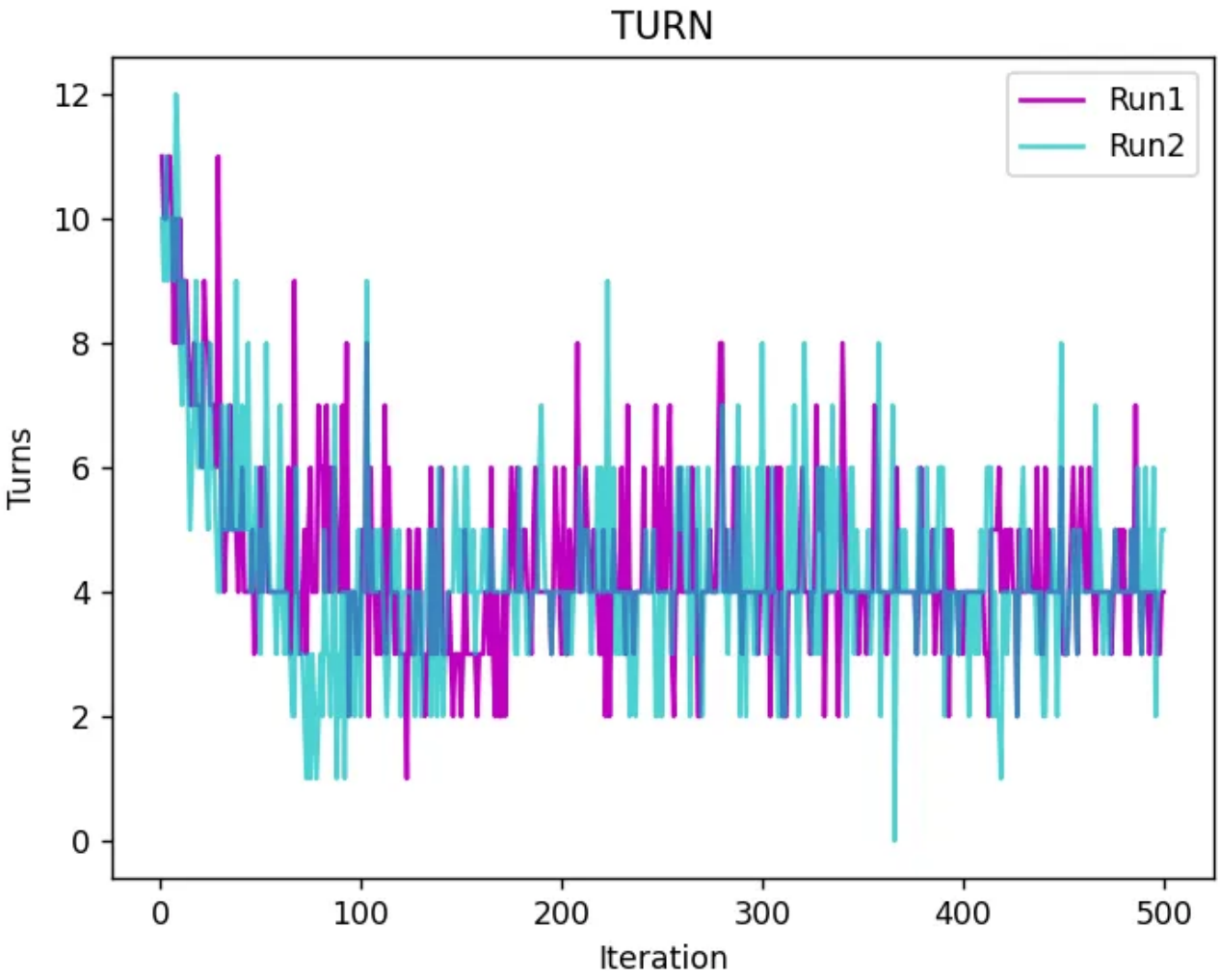
As seen from the above graphs the program is much more efficient in finding the solution if same maze is provided to it, whereas in random grids the algorithm is sometimes unable to find a optimal solution.

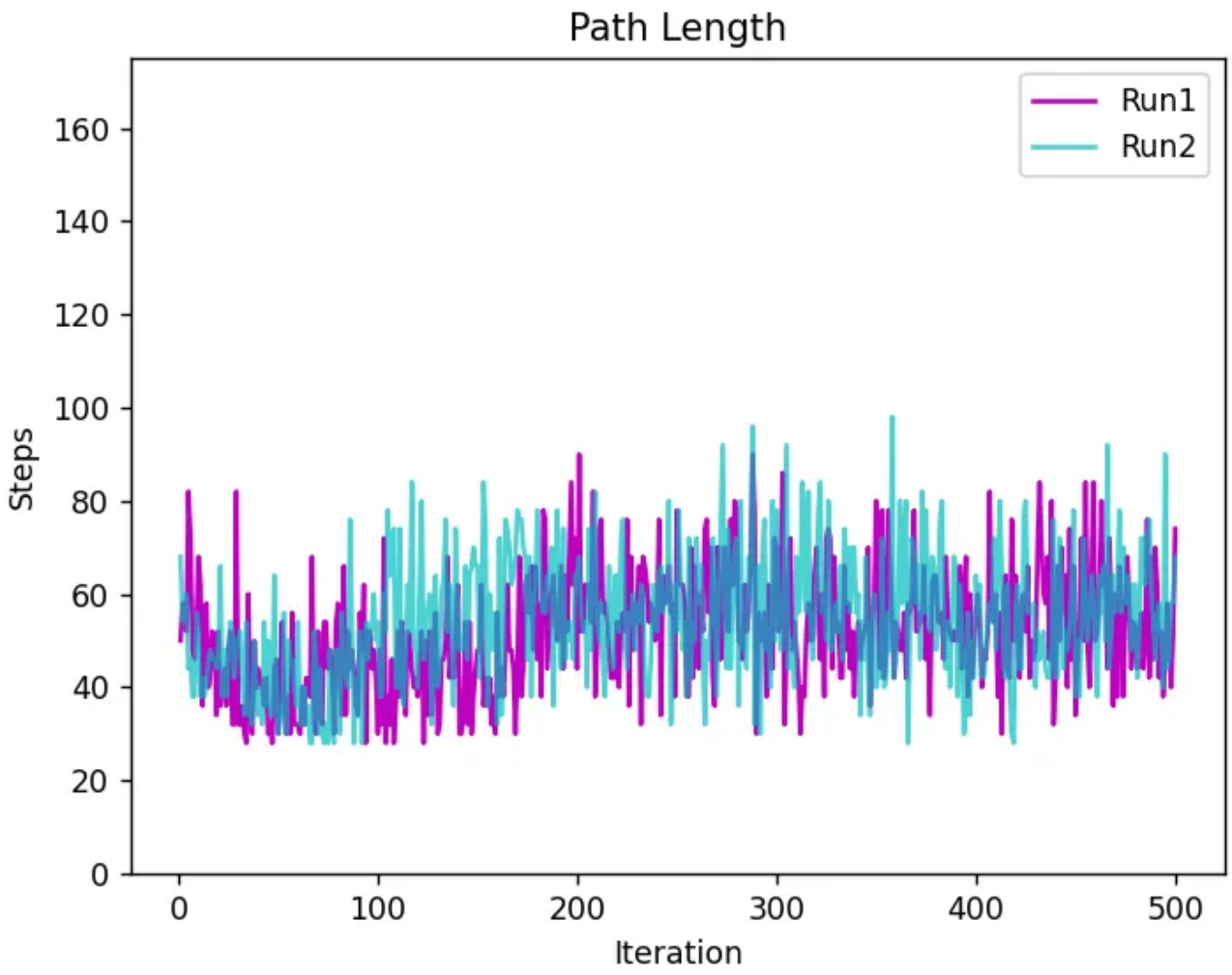
Fitness Trends:

The graphs were plotted using matplotlib package of python.


FITNESS
Fitness
Iteration
Run1
Run2
250
300
350
400
450
0
100
200
300
400
500


INFEASIBLE
Run1
Run2
Infeasible steps
Iteration
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
100
200
300
400
500


TURN
Run1
Run2
Turns
Iteration
0
2
4
6
8
10
12
0
100
200
300
400
500



Code:

The complete code can be found on GitHub using the following link. **Code**

https://gist.github.com/AbiamAsifKhalid/c3b76957f5e5738403ceab4e331d5eee

Python Programming Genetic Algorithm Mechatronics Engineering