NNGA Project : Bio-inspired Maze solving : Genetic Algorithm vs Rule Based

1. Preface

The problem we want to solve here is the solving of a Maze. I was inspired by the solving of mazes by *Physarum polycephalum* which is an organic being that outperforms human solutions when it comes to solving complex pathfinding problems like mazes. I wanted to test it myself in programming : using a human solution rule-based and an organic-like solution. Here the human solution I chose is the “right hand rule” and the organic-like solution is a genetic algorithm.

The first step of this problem solving is to program an algorithm that follows the right hand rule to find the exit of the maze. The second step will be to use the genetic algorithms logic and create individuals which will try to solve the maze.

Regarding the results, we will test three maze structures : a very simple one, a medium complexity one and a very complex one to validate that both the right hand rule and the genetic algorithm can solve multiple types of maze. The expected results are that both solutions will be very close in terms of performance for the simple and medium maze but the genetic algorithm should be a lot more efficient regarding the complex one. Here what we call efficiency will not be a matter of time but the number of steps to find the exit of the maze.

The code to this project is also attached, every file is self defined by the name. But it is possible to find more details on the github page.

2. Preparing the code

In order to prepare the code, I created a Maze class which will be used in both solutions. This class defines a Maze and some actions that we can perform on it. Let me go through it.

```
class Maze():
   def __init__(self):
      self.maze = [
            [1,1,1,0,1,1,1],
            [1,0,0,0,0,0,1],
            [1,0,1,1,1,1,1],
            [0,0,0,0,1,1,1],
            [1,1,1,0,0,0,1],
```

```
            [1,1,1,1,1,0,1],
            [1,1,1,0,0,0,1],
            [1,1,1,1,1,1,1]
        ]
        self.colorizer = ColorTranslator()
        self.px = 0
        self.py = 3
        self.orientation = 1
        self.arrows = ["\u2191", "\u2192", "\u2193", "\u2190"]
```

The class defines three mazes (only one displayed here), a simple one, a medium one and a more complex one, we can uncomment the one we want to test. A “1” is a wall while a “0” is a free path. We give it two initial coordinates (0,3) which is the entry of the maze in the three mazes, where our piece starts. Finally, we give it an orientation and arrows that will be used to print where the piece is looking at.

```
def printMaze(self):
        for y,line in enumerate(self.maze):
            for x,cell in enumerate(line):
                if x == self.px and y == self.py:
                    arrow = self.arrows[self.orientation]

print(self.colorizer.colorfulPrint(f"{arrow}{arrow}", "red"), end="")
                elif cell == 1:
                      print(self.colorizer.colorfulPrint("\u2588\u2588",
"white"), end="")
                else:
                     print(self.colorizer.colorfulPrint("  ", "white"),
end="")
            print()
```

Then we define printMaze() which is a method to print the maze in real time in the terminal. Basically, it goes through each line and cell of the maze. If the current cell coordinate corresponds to the piece coordinate, then we print the piece in red with the correct arrow. If a cell is a wall, we print two unicode characters that look like a wall, otherwise we print two blank spaces.

```
def updateOrientation(self, orientation):
        self.orientation = orientation

    def getX(self):
        return self.px
```

```
  def getY(self):
      return self.py

  def getMaze(self):
      return self.maze

  def updateX(self, newX):
      self.px = newX

  def updateY(self, newY):
      self.py = newY

  def clear(self):
      os.system('clear')
```

Some basic methods to get/update positions and one method to clear the terminal content allowing a smooth display.

3. Right-hand rule

a. Implementing the algorithm

As stated at the beginning, our first solution is the right-hand rule. This solution is defined by wikipedia as :

“keeping one hand in contact with one wall of the maze the solver is guaranteed not to get lost and will reach a different exit if there is one; otherwise, the algorithm will return to the entrance having traversed every corridor “

Knowing that, we can define a pseudo algorithm that we will then implement in Python to solve the maze. The algorithm shall be the following.

```
While not exit:
    (1) is there a wall on the right?
        no :
            turn right
        yes :
            go automatically to (2)

    (2) is there a wall in front of me?
        no :
            make a step in front
        yes :
            go automatically to (3)
```

```
    (3) is there a wall on the left:
        yes :
            turn around (the left wall becomes a right path)
            go back to (1)
        no :
            turn around (dead-end)
            go back to (1)
```

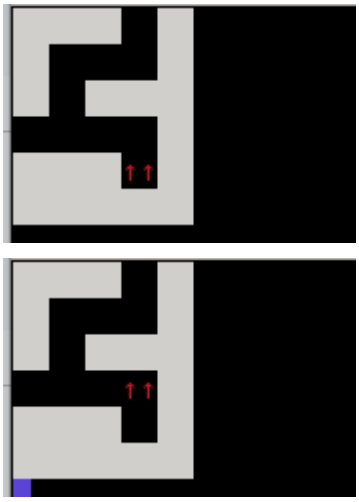
To explain it more in detail, this algorithm follows the right hand rule meaning that we only look for a right or straight path because those are the only way of “keeping” our hand on our right, imagine having our hand stuck to the right wall. The left and dead-end scenario comes to the same conclusion because in order to turn left we need to transform the left into right by turning around. Basically, if there is a path on the left, we turn around, the left becomes our right and the algorithm goes back to (1) which will work. Otherwise, if there is no path on the left, we turn around, again the left becomes a right and we go forward which corresponds to (2)

Then we implement this algorithm in Python as follows.

```python
def executeRigthtHand():
    mazeObject.clear()
    time.sleep(0.3)

    mazeObject.clear()
    mazeObject.printMaze()
    time.sleep(0.3)

    x = mazeObject.getX()
    y = mazeObject.getY()
    maze = mazeObject.getMaze()

    # define directions
    directions = [
        (0, -1), # 0 = North
        (1, 0),  # 1 = East
        (0, 1),  # 2 = South
        (-1, 0), # 3 = West
    ]

    # at the beginning we look to the right (east)
    orientation = 1
    steps = 0
```

We start by printing the maze once to show the initial state. Then we prepare our variables : x, y and the maze board. Defining the directions will be used as a compass to know where we are looking. Depending on where we look, the right is different so it is important to define that. We start by looking East because the entrance of the maze is on the west side.

To take an example, when we will be on the coordinate `(x,y) = (3,4)` our algorithm states that we will head north (up), thus we will do `(x,y) = (3+0 , 4-1)`. This a schema of what will actually happen.



First state, x = 3 and y = 4

Compute : x = x - 0 = 3 + 0 = 3
y = y + (-1) = 4 - 1 = 3

Second state, x = 3 and y = 3

Then we can enter the main loop of the algorithm that will not stop until we reach the exit.

```
while x != 3 or y != 0:
        right_direction = (orientation+1) % 4
        dx_d, dy_d = directions[right_direction]

        right_cell_x = x + dx_d
        right_cell_y = y + dy_d
        try:
            if maze[right_cell_y][right_cell_x] == 0:
                orientation = right_direction
                x = right_cell_x
                y = right_cell_y
                steps +=1


            else :
                dx_d, dy_d = directions[orientation]
                cell_infront_x = x + dx_d
                cell_infront_y = y + dy_d
```

```
            if maze[cell_infront_y][cell_infront_x] == 0:

                x = cell_infront_x
                y = cell_infront_y
                steps +=1
            else:
                orientation = (orientation-1) % 4

    except IndexError:
        orientation = (orientation -1) %4

    mazeObject.updateOrientation(orientation)
    mazeObject.updateX(x)
    mazeObject.updateY(y)

    mazeObject.clear()
    mazeObject.printMaze()
    time.sleep(0.3)
```

The condition to exit the loop is based on the exit coordinate. The first thing we want to test is if there is a path on the right. To do this, we compute the right direction based on our current orientation. To obtain our current right we just need to do +1. For example, if we are looking south (2) our right is to the west because `(2+1)%4 = 3`. Then, we get *dx_d and dy_d* that will be used to calculate our next coordinate based on the directions we defined. After that we compute the coordinate of the cell on our right.

Now we enter the conditions, using the coordinate of the cell on our right, we check if it is a free path. If it is the case, we go there. Our orientation becomes our right and the coordinates of x,y becomes the coordinates of this cell we were looking at.

Otherwise, we look in front of us. *dx_d and dy_d* becomes the value of our current orientation (e.g. if we are looking east, `dx_d = 1 and dy_d = 0`). We compute the coordinates of the cell in front of us and if this cell is free we move there.

If both right and front are obstructed, our current orientation rotates, basically making a 90° turn. As an example, if we were looking east with our hand on south, we compute `(orientation-1)%4` which gives us an orientation to north with a hand on east. From that, the algorithm starts again. We use a try-except in case we are looking out of the maze we will once again turn around. All this code is included in a function in order to be used later for the comparison.

At the end of each loop, we update orientation, x and y to perform a print of the current state, in some cases those will not move because we will simply turn and that will be clear by looking at the arrows orientation.

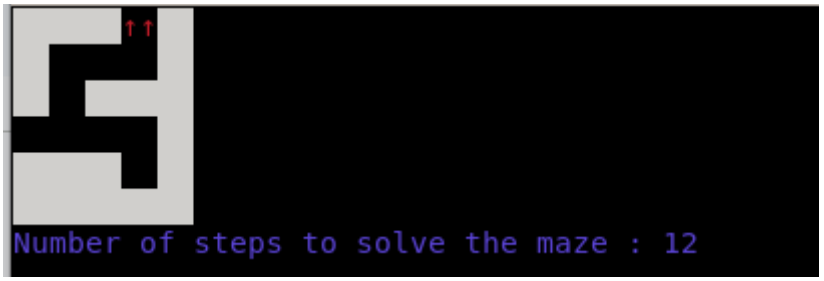
```
print(f"Number of steps to resolve the maze : {steps}")
return steps
```

Finally, at the end of the solving, we print the results and return it. We will see how those are used later.

b. Testing the solution

Now that the algorithm is ready we can test it on our three mazes. As stated in the preface, we will compute the number of steps done to see “how good” the algorithm is.
In the previous code, a variable step was incremented every time we make a step (turning around is not counted). This will allow us to see how many steps were done by the right-hand rule and if the genetic algorithm has less this will mean that the right-hand rule makes useless steps.

The **simple** maze is solved like this.



It is also possible to watch the solving in video using the following link simple_maze_right_hand.mp4. As we can see, the maze is solved in 12 steps, we will remember this for the future.

The **medium** maze is solved like this.

As we can see this one takes more than twice the steps of the simple one, this is due to the structure of the maze which is more complex. Once again it is possible to watch the full solving with this link medium_maze_right_hand.mp4.

The **complex** maze is solved like this.

Here the results start to show the limits of the right hand rule, the more complex the maze is the more steps it needs to find an exit even if there is an obvious path it will not test it first because of the algorithm rules. Once again it is possible to watch the solving with this link complex_maze_right_hand.mp4.

To conclude this part on the right hand solution, the simple maze is solved in 12 steps, the medium one is solved in 28 steps and the complex one is solved in 38 steps. In all three cases we can find the exit but it feels like a lot of steps considering the structure of the mazes that we can see as a human. For example, in all three cases we could turn left almost instantly to get near the exit. This solution is guaranteed to work but not in the best way possible.
The genetic algorithm which mimics a true organic being is supposed to work as a trial-and-error process. It will not just follow a wall but seek the best solution possible, the final generation should have the shortest way of going through the maze. Much like *Physarum polycephalum* which inspired this project, the genetic algorithm will look for the shortest path between start and exit. By the end of the next part, we should be able to prove that organic-like trial and error outperforms a simple rule based algorithm.

4. Genetic algorithm solution

a. Implementing the algorithm

Our second solution is the genetic algorithm. Let us define the different steps that the algorithm will follow. The numbers used below are for the easy and medium maze, we will see that for the complex we will need to change those a bit in order to solve the maze.

```
Step 1 :
create a base population of 100 individuals. Each individual
will have a genome of 10 genes since we know the mazes can be
solved in 6 steps. The genes correspond to a direction to go,
for example a genome could be [1,0,3,3,...] meaning [go east,
go north, go west, go west,...].

Step 2 :
Fitness : We proceed to create 10 generations. For each
individual, loop through their genes (moves) and if the move
leads to a reachable spot, move there and increase steps. If
it reaches the end we simply stop the moves. After the moves
are completed we compute a score based on closest distance to
the maze and number of steps done (more details later).

Step 3 :
The crossover/next generation, the best individual from
generation n is kept in n+1, thus we create 99 new
individuals. To create those, we select 2 parents among the
20 best of n. We create a child which receives those parents
genes and can have mutations.

Step 4 :
return the best score for each generation and the final best
genome.
```

Let us go through the code step by step. Once again this code is in genetic_algorithm.py. Some irrelevant parts will not be included here.

```python
def executeGA():
   population_size = 100 # for easy and medium maze
   # population_size = 600 # for complex maze
   genome_length = 10 # for easy and medium maze
   # genome_length = 80 # for complex maze
   mutation_rate = 0.05
   best_score_per_gen = []
   best_genome = None
   f = open("ga_results.txt", "w")
   f.write("--- GENETIC ALGORITHM RESULTS BY GENERATION---\n\n")

   population = np.random.randint(0, 4, size=(population_size,
   genome_length))

      directions = [
       (0, -1), # 0 = North
       (1, 0),  # 1 = East
       (0, 1),  # 2 = South
```

```
        (-1, 0), # 3 = West
    ]

    # useful variables
    maze = mazeObject.getMaze()
    start_x = mazeObject.getX()
    start_y = mazeObject.getY()
```

We start by setting our parameters and variables. We then create a population of population_size individuals each having a genome of genome_length. Once again we define x, y, maze and directions. best_score_per_gen is used to store the best steps by generations. best_genome will be used later to store our final best individual.

```
for gen in range(10): # for easy and medium maze
# for gen in range(15): # for complex maze
        # to store distances
        all_scores = []

        # for each individual
        for p in population:
            # initialize x and y
            x = start_x
            y = start_y
            steps = 0

            # storing closest distance from the exit
            min_dist = abs(x-3) + abs(y-0)

            # for each gene (move)
            for move in p:
                # getting the moves based on directions
                dx_d, dy_d = directions[move]
                # computing the moves
                next_X = x + dx_d
                next_Y = y + dy_d

                # checking that we are not out of the maze
               if 0 <= next_Y < len(maze) and 0 <= next_X <
                  len(maze[0]):

                    if maze[next_Y][next_X] == 0:
                        x = next_X
```

```
                    y = next_Y
                    steps +=1

                    # updating closest distance
                    current_dist = abs(x-3) + abs(y-0)
                    if current_dist < min_dist:
                        min_dist = current_dist

            # if the end was reached, no need to keep going
            if x == 3 and y == 0:
                break

        if x == 3 and y == 0:
            # if the exit was found, we only use steps as a score
            score = steps *0.01
        else:
            score = min_dist + 10 + (genome_length - steps) *0.01
```

Then this is the fitness part. Here we essentially take each individual and make them go through the maze. We store the closest they were from the exit and update it as they go closer to the exit with min_dist.
At the end we compute the score, there are two conditions to determine the score :

- If the exit was reached, the score is simply the number of steps, ensuring that individuals who found the exit have a better score while still reaching the exit in next generations.
- If the exit was not reached, the score is computed as following :
    - closest distance from exit + 10 : ensuring that individuals who didn't find the exit have a bad score.
    - The number of steps that were made in total, this tells the algorithm that exploring is better than staying in place for many genes. With this we avoid having an individual who stays at the same spot for 10 moves.

To take an example, if we have :
A = 10 + 10 + (80 - 50) * 0.01
B = 10 + 10 + (80 - 10) * 0.01
C = 5 + 10 + (80 - 50) * 0.01
D = 40 * 0.01
In this case A is better than B because it made a lot more exploration through the maze. C is better than A because it is closer to the exit. And D is better than C because it actually reached the exit.

```
    # converts the distance array to a numpy array
```

```
        scores_array = np.array(all_scores)
        # sort the results, best scored individuals first, gives an
        indexes array
        sorted_results = np.argsort(scores_array)
        best_score_per_gen.append(scores_array[sorted_results[0]])
        best_genome = population[sorted_results[0]]

        f.write(f"--- Generation {gen} ---\n")
        f.write(f"Best individual : #{sorted_results[0]}, with a score
            of {scores_array[sorted_results[0]]:.2f}.\n")
        f.write("-" *10+"\n")
```

Then we take all the scores and sort it (best score first) and we store the current best score and best genome. We also print this generation's best individual. Since sorted_results give us indexes we can simply print the index of the best individual that was found, after that we print its score. Since we know the number of steps is multiplied by 0.01, once we see a score below 0 we know that the exit was reached. For example, a score of 0.40 means we reached the exit in 40 steps.

```
        # Step 3 : Next Generation, the crossover
        # preparing next generation and making sure best individual
        survives
        new_generation = []
        new_generation.append(population[sorted_results[0]])

        # creating the new generation individuals
        # one is already kept as it is, we need 99 more
        for _ in range(population_size - 1):
            # taking two random parents among the 20 best
            index_parent1 = np.random.randint(0,20) # for easy and
                                                    medium maze
            index_parent2 = np.random.randint(0,20) # for easy and
                                                    medium maze
            # index_parent1 = np.random.randint(0,50) # for complex maze
            # index_parent2 = np.random.randint(0,50) # for complex maze

            parent1 = population[sorted_results[index_parent1]]
            parent2 = population[sorted_results[index_parent2]]

            # crossover : take a random index
            point = np.random.randint(1, genome_length)
            # creating a child, parent1 gives 0 to point genes and
```

```
            parent2 gives point to genome_length genes
            child = np.concatenate([parent1[:point], parent2[point:]])

            # mutation
            for i in range(genome_length):
                # takes a random number, if below mutation_rate a
                  mutation occurs
                if np.random.rand() <= mutation_rate:
                    child[i] = np.random.randint(0,4)

            new_generation.append(child)
        # the next population is set as the new one we created
        population = np.array(new_generation)
```

The next step is to create the next generation. We start by making the best current individual survive. Then we create 99 children, taking two random parents among the 20 best individuals. We then take a random point between 1 and genome_length. The child will receive :
1 →point : genes from parent1
point →genome_length : genes from parent2
Then we apply a simple mutation, we take a random digit between 0 and 1 and if it is below mutation_rate we change the gene we are currently inspecting. We finish by setting the new population.

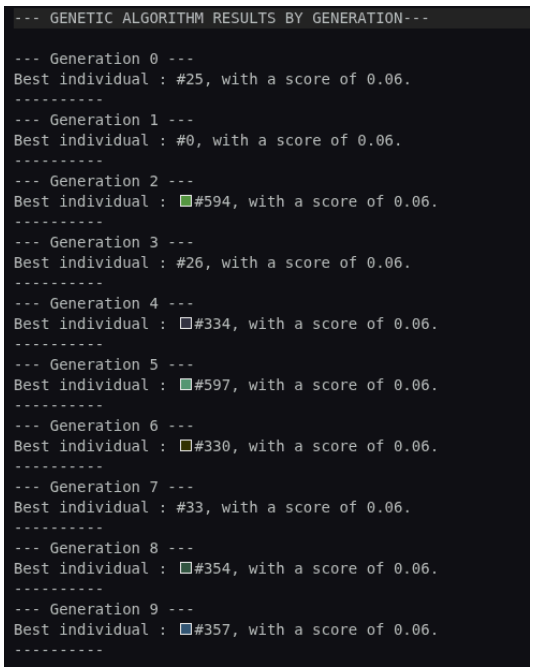
```
f.close()
return best_score_per_gen, best_genome
```

At the end of all generations we return both scores by generation (used for learning rate) and the best genome (used to display maze solving graphically). All the results are printed in a text file to make sure we can store them and not just have them in the terminal.

b. Testing the solution

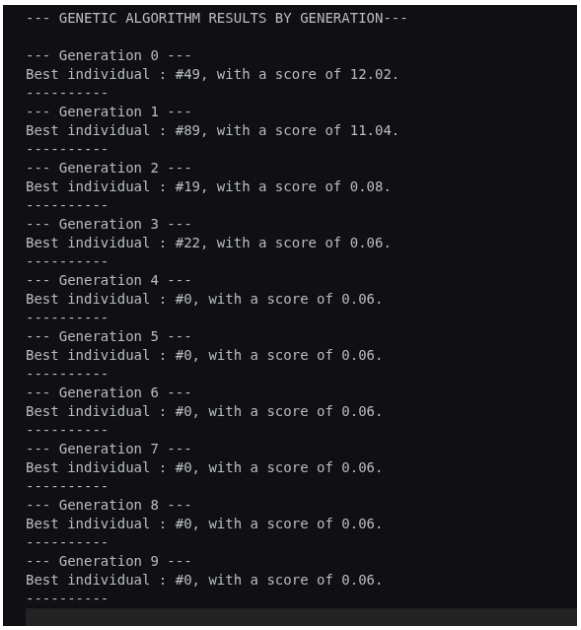
Now that our genetic algorithm is ready let us test it through the three mazes that we have. After that we will compare the results of the two solutions.

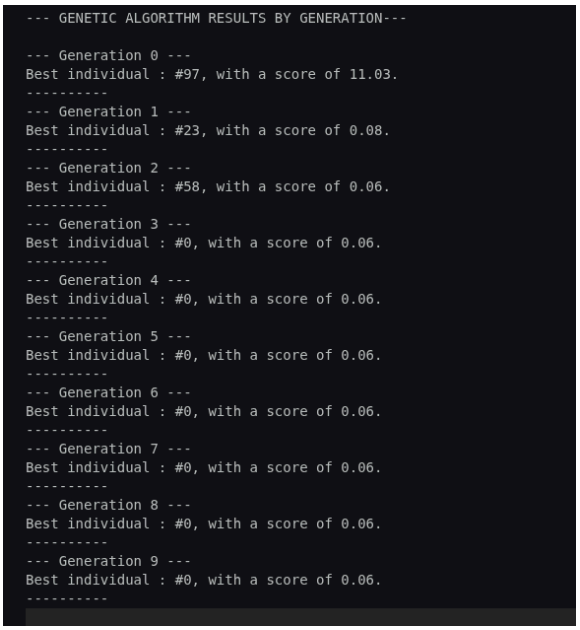
The **simple** maze is solved like this.



As we can see here the exit is found **instantly** in the first generation since a score of 0.06 means the exit was found within 6 steps. Of course the results can depend on the execution but for the simple one it is almost always the first or second generation which finds the exit. That is why above I displayed two executions to show that it can defer. Again it is possible to watch the solving at this link simple_maze_ga.mp4.

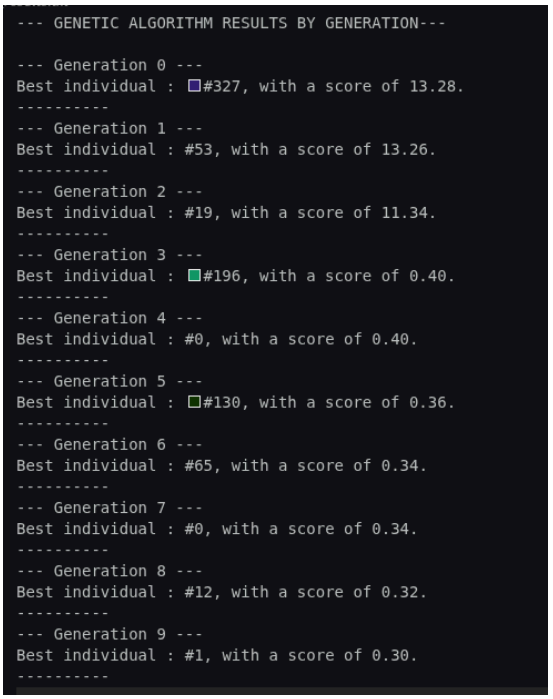
The **medium** maze is solved like this.

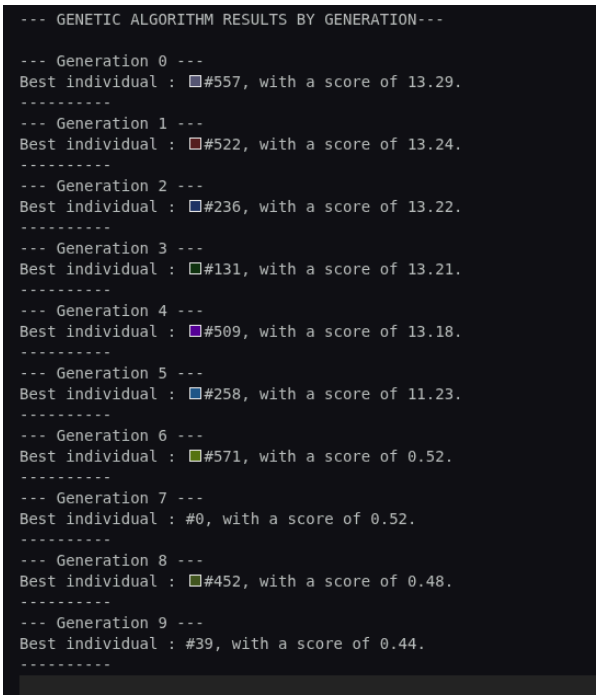




For the medium one the algorithm tends to take a little bit more time, again I displayed two examples. The results also depend on the execution, sometimes we are lucky and a perfect individual was born in the first generation. From all the executions I have done it takes on average **1 to 3** generations to find the exit in 6 steps. Again it is possible to watch the solving at this link medium_maze_ga.mp4.

The **complex** maze is solved like this.





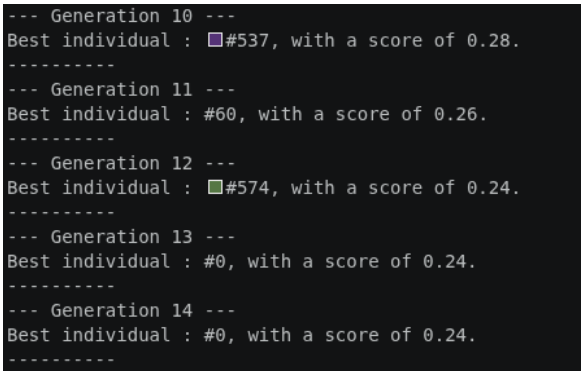
The case of the complex maze is a bit different since we need to increase some parameters. Let me go through them :

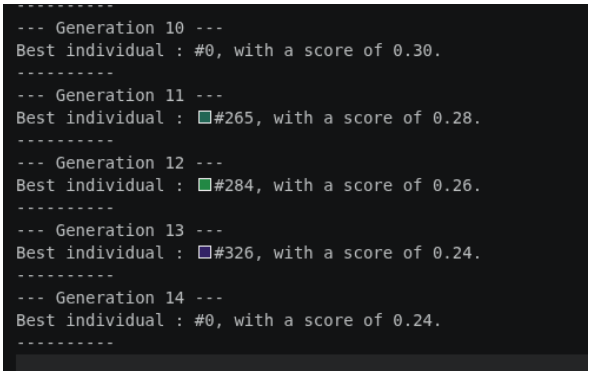
- population size : 100 → 600
- genome length : 10 → 80

- parents : 20 best → 50 best

Here the maze is designed to be solved in 24 steps which would be the best score. From the different executions I have made, it is very rare to reach the 24 steps in 10 generations, as the screenshot shows. Sometimes it is in fact not efficient at all and takes a lot of steps to reach the exit.

To counter that, let us increase the number of **generations to 15**. Those are the changes we can observe.





From the different executions I have made, it takes on average 12-13 generations to find the quickest path. Sometimes we can be very lucky and have the 24 steps at generation 10 but to guarantee the optimal path we must go to 15.

Again it is possible to watch the solving, here I will have two links.
The first one complex_maze_ga_not_optimal.mp4 shows a case where the algorithm is not optimal and takes more than 24 steps in fact 30 steps here (10 generations). The second one complex_maze_ga_optimal.mp4 shows a case where the algorithm is optimal and takes 24 steps (15 generations).

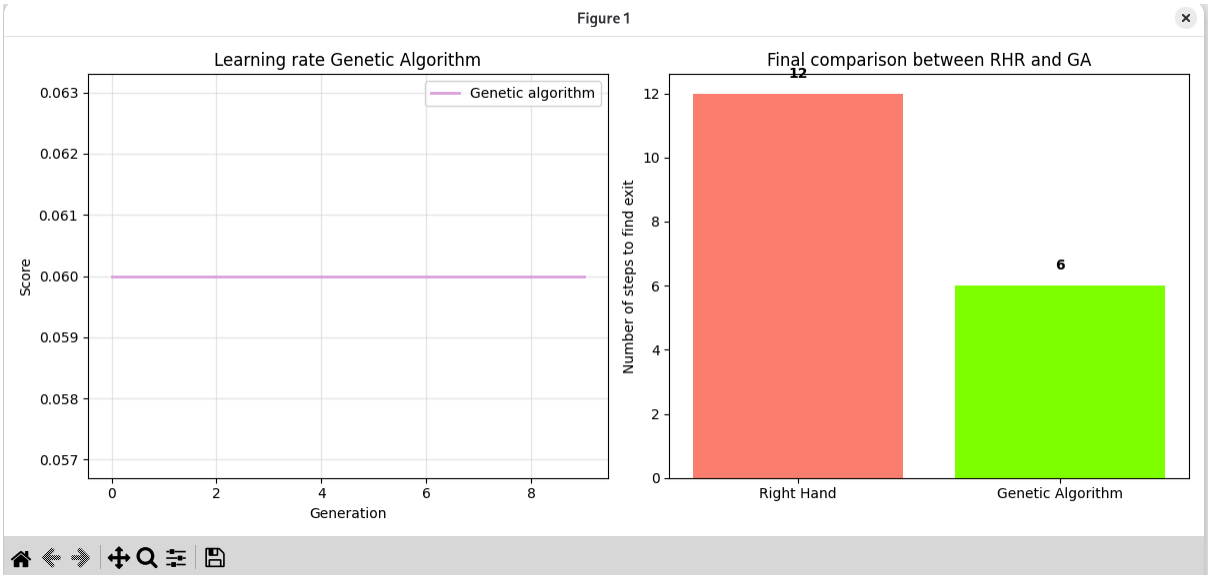
To conclude this part on genetic algorithm, we can note that this solution is a lot more efficient, in fact even in the simple maze the solution appears instantly which makes it almost excessive. The three mazes could be solved in their optimal number of steps respectively 6, 6 and 24. Only the complex maze needed to have adaptations because the parameters were too low to find an exit.
One thing is sure : the computing power needed is heavier because we need to create each generation 100 individuals with a total of 10-15 generations while the right hand rule simply applies rules. Now let us move on comparing those two solutions.

5. Results and conclusion

Now that our two solutions are ready and can be executed on their own, let us compare them. In order to make the comparison easier I made an interactive script, main.py, I will not go through it but it is an interactive script that will create results for us. The results below are based on the tests that were made in the respective parts of the two algorithms. To display the results I used matplotlib.

For each comparison I will display the graphs and the results it is based on.

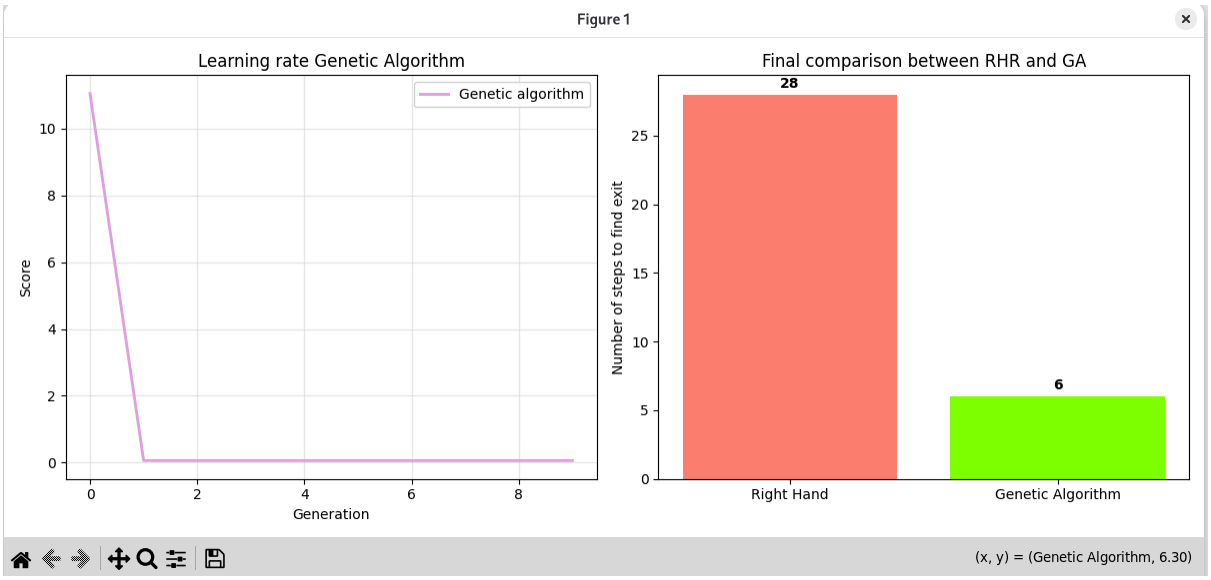
Firstly let us compare the cases of the **easy** maze.




```
--- GENETIC ALGORITHM RESULTS BY GENERATION---

--- Generation 0 ---
Best individual : #25, with a score of 0.06.
----------
--- Generation 1 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 2 ---
Best individual : #594, with a score of 0.06.
----------
--- Generation 3 ---
Best individual : #26, with a score of 0.06.
----------
--- Generation 4 ---
Best individual : #334, with a score of 0.06.
----------
--- Generation 5 ---
Best individual : #597, with a score of 0.06.
----------
--- Generation 6 ---
Best individual : #330, with a score of 0.06.
----------
--- Generation 7 ---
Best individual : #33, with a score of 0.06.
----------
--- Generation 8 ---
Best individual : #354, with a score of 0.06.
----------
--- Generation 9 ---
Best individual : #357, with a score of 0.06.
----------
```

What we can see here is that the genetic algorithm is far too powerful for the easy maze, there is actually almost no learning because it finds the perfect solution at the first generation. This is because the probability of having at least 1 individual out of 100 who can reach the exit is far too high. We just need 6 genes correct out of 10 so among 100 individuals it is almost sure to have one instantly. Sometimes it takes two generations to reach the best score which is 0.06 but it is very rare.

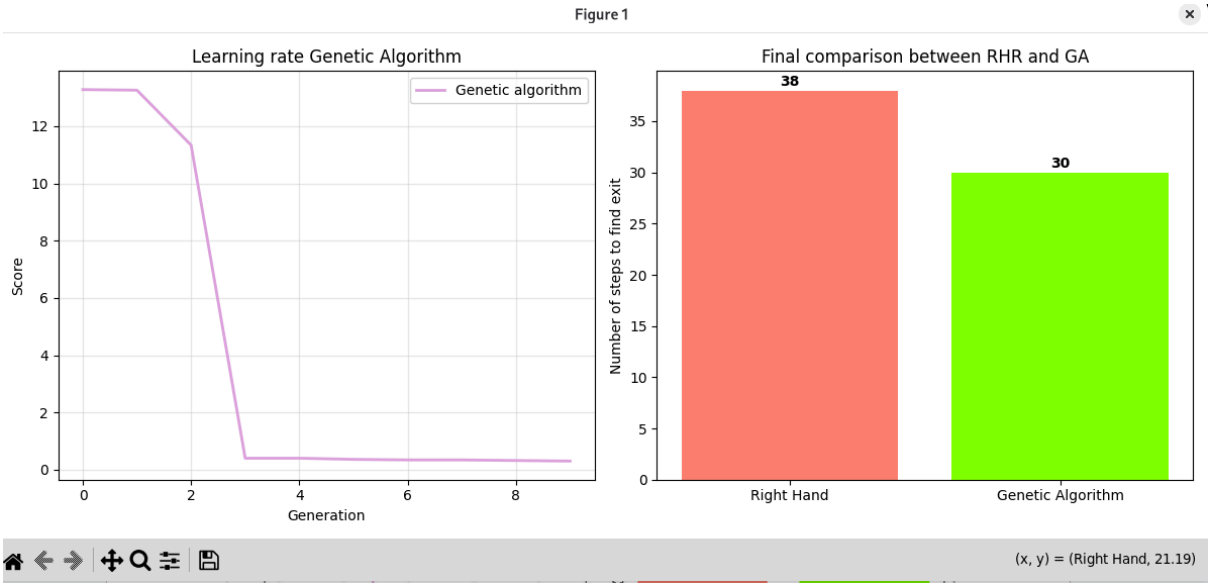
Secondly let us compare the cases of the **medium** maze.




```
--- GENETIC ALGORITHM RESULTS BY GENERATION---

--- Generation 0 ---
Best individual : #97, with a score of 11.03.
----------
--- Generation 1 ---
Best individual : #23, with a score of 0.08.
----------
--- Generation 2 ---
Best individual : #58, with a score of 0.06.
----------
--- Generation 3 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 4 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 5 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 6 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 7 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 8 ---
Best individual : #0, with a score of 0.06.
----------
--- Generation 9 ---
Best individual : #0, with a score of 0.06.
----------
```

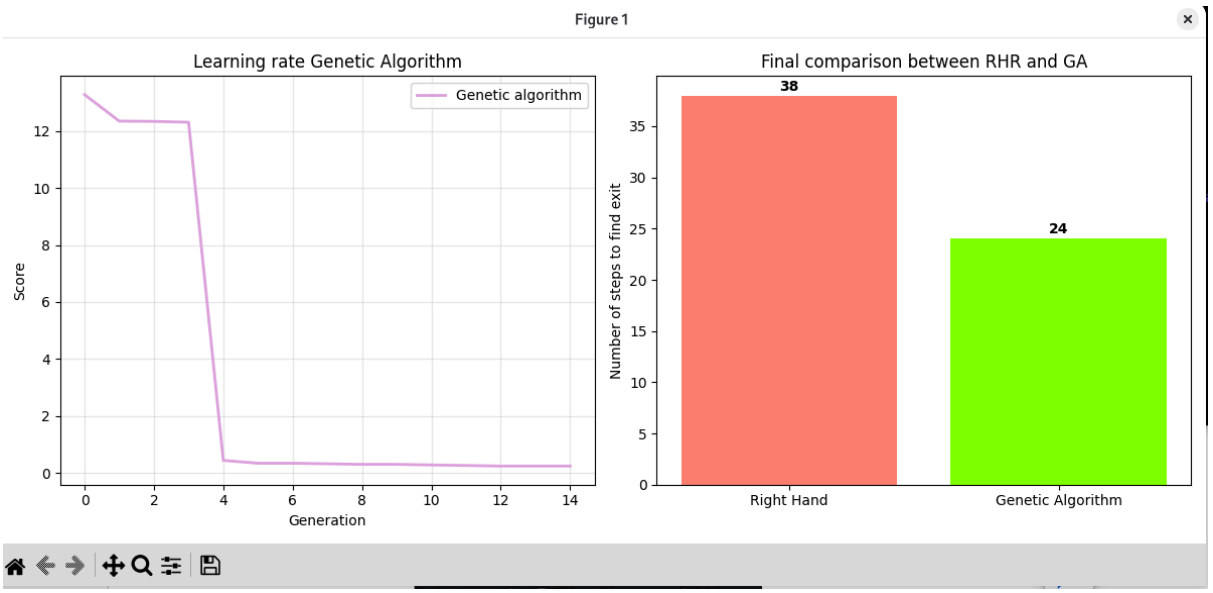
This is where the learning rate starts to appear even though it is very quick since the best path is the same.
What we want to write down here is that with a more complex maze, the genetic algorithm can keep the same best path of 6 while the right hand rule, stuck in its rules, has doubled the steps required to exit. The genetic algorithm is still too powerful since it finds an exit at generation 2.

Finally let us compare the cases of the **complex** maze. Here we will do it both with 10 and 15 generations.




```
--- GENETIC ALGORITHM RESULTS BY GENERATION---

--- Generation 0 ---
Best individual : #327, with a score of 13.28.
----------
--- Generation 1 ---
Best individual : #53, with a score of 13.26.
----------
--- Generation 2 ---
Best individual : #19, with a score of 11.34.
----------
--- Generation 3 ---
Best individual : #196, with a score of 0.40.
----------
--- Generation 4 ---
Best individual : #0, with a score of 0.40.
----------
--- Generation 5 ---
Best individual : #130, with a score of 0.36.
----------
--- Generation 6 ---
Best individual : #65, with a score of 0.34.
----------
--- Generation 7 ---
Best individual : #0, with a score of 0.34.
----------
--- Generation 8 ---
Best individual : #12, with a score of 0.32.
----------
--- Generation 9 ---
Best individual : #1, with a score of 0.30.
----------
```

Here the learning rate is clearly important, what we can see here is that the first generations are far from the exit and as generations goes by, we start to have individuals that find the exit but not in the optimal way. Here the genetic algorithm does not have enough time to find the best path. Thus, the results are very close to the ones of the right hand rule, it finds the exit but is supposed to be optimal. Let us see what happens if we increase to 15 generations.




```
--- Generation 10 ---
Best individual : #537, with a score of 0.28.
----------
--- Generation 11 ---
Best individual : #60, with a score of 0.26.
----------
--- Generation 12 ---
Best individual : #574, with a score of 0.24.
----------
--- Generation 13 ---
Best individual : #0, with a score of 0.24.
----------
--- Generation 14 ---
Best individual : #0, with a score of 0.24.
----------
```

Now that the genetic algorithm has more time to learn, it can reach the exit in 24 steps. The right hand rule is still stuck on 38 steps and that will stay forever but as we increase the parameters the genetic algorithm can become better and better in solving the maze.

To conclude, what we have seen throughout all of this is that both right hand rule and genetic algorithm have good and bad points.
As a final conclusion I want to set up a table that compares both solutions and see in which way one is better or not. They do not have the same use case and are both useful.

Solution Point of comparison	**Right Hand Rule**	**Genetic Algorithm**
Can reach the exit ?	always	always
Best path found	never	always (depends on parameters)
CPU cost	very fast, simply apply rules	slow, needs to compute the learning phase and create many individuals

Solution type	Deterministic (always the same path for a given maze)	evolves at each try/generation.
Learning	no	yes
Maze complexity impact	none	needs bigger population/genome

What we can say as a last word is :

- The right hand rule is a guarantee to find an exit no matter what the maze complexity is but it can take a lot of steps to find this exit. It is useful if our only goal is to find the exit.
- The genetic algorithm also ensures to find an exit while also looking for the quickest path to reach it. It is useful in case we want to go through the maze many times, then we will use the genetic algorithm to find the optimal path and save it for our use case. It is also important to note that here the mazes were rather simple, in some cases the genetic algorithm would require bigger parameters to actually find the exit. With the current parameters, some very complex mazes might not be solvable.

The right hand rule is here for security while the genetic algorithm is here for optimization.