Genetic algorithm for maze solving bot

Dexter Shepherd

February 2021

1 Introduction

In this report I shall experiment with genetic algorithms solving mazes, in comparison to brute force maze solving algorithms. Genetic algorithms [4] generate a number of solutions and follow the solutions which work best, mutating at each interval. In a maze it will generate instructions in the form left-right-forward-forward. My experiment is to see how genetic algorithms solve a maze against brute force algorithms [2].

2 Implementation

I will use Python [3] to conduct my experiment due to its syntactic simplicity for complex algorithms. I will be testing this on one maze size due to time constraints.

2.1 Maze

The maze is 10 by 10, where a space bar character represents walkway and 'X' represents a wall. 'S' will mark the start position and 'W' will mark the win position. The aim is to get through the maze in as few iterations as possible. This maze will be held as a list of lists in the Python implementation.

X	X	X	X	X	X	X	X	X	X
X	S	X							X
X		X		X	X	X	X		X
X		X		X					X
X		X	X	X	X	X	X	X	X
X									X
X		X	X	X	X	X	X		X
X		X		X			X		X
X							X	W	X
X	X	X	X	X	X	X	X	X	X

Figure 1: This maze shows the format of a maze

In Python the maze is an object. This allows me to develop methods to interact with the maze internally, without effecting the maze layout.

```
class maze:
    def __init__(self):
        self.maze=[
            ["X","X","X","X","X","X","X","X","X","X"],
            ["X","S","X"," "," "," "," "," "," ","X"],
            ["X"," ","X"," ","X","X","X","X"," ","X"],
            ["X"," ","X"," ","X"," "," "," "," ","X"],
            ["X"," ","X"," ","X","X","X","X","X","X"],
            ["X"," "," "," "," "," "," "," "," ","X"],
            ["X"," ","X","X","X","X","X","X"," ","X"],
            ["X"," ","X"," ","X"," "," ","X"," ","X"],
            ["X"," "," "," "," "," "," ","X","W","X"],
            ["X","X","X","X","X","X","X","X","X","X"]
            ]
        self.pos=[0,0]
        for i, row in enumerate(self.maze):
            for j, place in enumerate(row):
                if place=="S":
                    self.pos=[i,j]
```

The current position of the navigating entity is set coordinates. Although it is in coordinates [1,1] the algorithm will still calculate the position in the event of different mazes being added.

Following this initialization method, a rule based method to say whether or not the next move is valid is implemented. This will be called by the genetic algorithm when deciding on places.

2.2 Brute Force Algorithm

A brute force algorithm is an algorithm which explores every avenue till it finds the correct path. Depending on the algorithm, it has complexities O(1) where it gets it right the first time and (f(x)) where f(x) is a complete function being run exhaustively. Brute force can be terribly inefficient. I decided to use a recursive

backtracking algorithm [1] which is a more refined method of brute force.

```
def solveMaze( Maze , position , N ):
    # returns a list of the paths taken
    num=1
    if position == ( N[0], N[1]):
        return num,[ ( N - 1 , N - 1 ) ]
    x , y = position
    if x + 1 < N[0] and Maze[x+1][y] != "X":
        numa,a = solveMaze( Maze , ( x + 1 , y ) , N )
        if a != None:
            return num,[ (x , y ) ] + a
        else:
            num+=numa
    if y + 1 < N[1] and Maze[x][y+1] != "X":
        numa,b = solveMaze( Maze , (x , y + 1) , N )

        if  b != None:
            return num,[ ( x , y ) ] + b
        else:
            num+=numa
    return num,None
```

2.3 Genetic Algorithm

The genetic algorithm approach I took was to generate 20 random pathways containing 100 different instructions. On initialization this will all be created. This means we have 2000 different possible steps. Each evaluation will find how far the program got without making a false move. If it discovers a solution it will log this. Size 100 was appropriate because there are not more than 100 steps to getting a solution.

The best four series of instructions are used again and mutated at the point it went wrong, following to the end of the array.

```
class GenBot:
    def __init__(self):
        self.paths=[]
        self.possibleMoves=["L","R","U","D"]
        for i in range(20): #10 different paths
            t=[]
            for j in range(100): #each can hold up to 100 possible
                                          outcomes
                t.append(random.choice(self.possibleMoves))
            self.paths.append(t)
        self.win=[]
    def evaluate(self,moves):
        count=0
        self.m=maze()
        while count<100 and self.m.NextMove(moves[count]):
            count+=1
            if self.m.win():
                self.win=moves[0:count]
                break
        return count #return how far the variation got
```

```
    def evaluateAll(self):
        nums=[]
        index=[]
        for i in self.paths:
            nums.append(self.evaluate(i))
        if self.win!=[]: return True
        codes=[]
        a1=max(nums)
        codes.append(self.paths[nums.index(a1)])
        nums.remove(a1)
        self.paths.remove(codes[0])
        a2=max(nums)
        codes.append(self.paths[nums.index(a2)])
        self.paths.remove(codes[1])
        nums.remove(a2)
        a3=max(nums)
        codes.append(self.paths[nums.index(a3)])
        self.paths.remove(codes[2])
        nums.remove(a3)
        a4=max(nums)
        codes.append(self.paths[nums.index(a4)])
        self.paths.remove(codes[3])
        nums.remove(a4)
        top=[a1,a2,a3,a4] #gather the top 4 solutions and feed them
                                    back into the algorithm
        self.paths=[]
        for i in range(4): #loop through best
            for j in range(5):  #set quatrer of paths to be this
                t=codes[i][0:top[i]].copy()
                if top[i]==100:
                    top[i]=0 #if inefficient solution
                    t=[]
                for k in range(top[i],100):
                    move=random.choice(self.possibleMoves)
                    t.append(move)
                self.paths.append(t)

        return False
    def findEnd(self):
        count=0
        while self.evaluateAll()==False:
            count+=1
        return count
```

3 Discussion

The brute force algorithm would make 22 iterations to get to the solution, whereas the genetic algorithm would take from a recorded low of 4, to a high of 285. In a sample size of 50 there was an average iterative cycles of 72.

The brute force has a standard iteration size based on the way in which the algorithm travels (biased to right or downwards). My genetic code was only to find a path in the shortest time, and not to find the shortest path, which the brute force does. This experiment was to find the quickest.

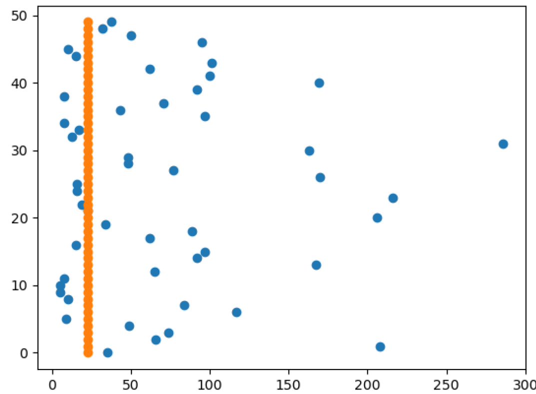

Figure 2: The y axis represents the samples 1 to 50, and the x-axis represents the number of iterations

The data shows a large grouping of data which is less than 100 iterations, indeed 30 percent of my samples were quicker than the brute force.

This was quite a low score. After making alterations to the algorithm so that it would not retrace it's steps, and when it gets stuck it would start again, the algorithm improved to a 48 percent success rate. This makes it on average 35 iterations. This is still 13 iterations above the brute force.

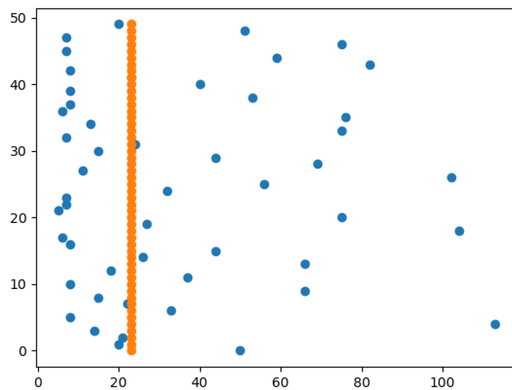

Figure 3: The y axis represents the samples 1 to 50, and the x-axis represents the number of iterations

4 Conclusion

From the results I gathered, I conclude that the genetic algorithm can find the route quicker than brute force, however on average brute force is quicker with the sample maze I created. The algorithm performed much better once rules were applied to its process. These rules were invented from looking at what slowed the system down. Rules such as this could be drawn from a background occurring machine learning algorithm working with the genetic algorithm.

My experiment only tested on one maze size. To improve the credibility of the results I would need to test these algorithms on larger mazes.

References

[1] Recursive backtracking. https://brilliant.org/wiki/recursive-backtracking/.

[2] Brute force search, 2009. Accessed: 2021-02-14.

[3] Python Software Foundation. Python language reference, version 3.8. http://www.python.org.

[4] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1996.