Data Structures and Algorithms (CSE2003)

An Implementation of Common Maze Solving Algorithms

J Component Report

By:

Abhijeet Ambadekar (16BCE1156)

Hardik Ahuja (16BCE1148)

Harsh Seth (16BCE1054)

Introduction

There are lots of ways by which we can find a way out of a maze. The human mind is capable of doing so rather intuitively. But the computer, due to its incapability of creative thought, requires some predetermined generic instructions to solve the maze. The aim of the project is to analyze various algorithms, implement them and find their applications.

These are a few common algorithms to solve a maze

- Random Mouse Algorithm
- Wall Follower
- Pledge Algorithm
- Trémaux Algorithm
- Dead End Filling Algorithm
- Recursive Algorithm
- Maze-routing Algorithm
- Shortest Path Algorithm

The Random Mouse, Wall Follower, Pledge, and Trémaux's algorithms are designed to be used inside the maze by a traveler with no prior knowledge of the maze, whereas the Dead-End Filling and Shortest Path algorithms are designed to be used by a person or computer program that can see the whole maze at once.

Mazes containing no loops are known as "simply connected", or "perfect" mazes, and are equivalent to a tree in graph theory. Thus, many maze solving algorithms are closely related to graph theory. Intuitively, if one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble a tree.

We have considered two of the above listed algorithms for our project – Wall Follower Algorithm and the Recursive Algorithm.

Wall Follower Algorithm

Preface

The wall follower is probably the best-known rule for traversing mazes. If the maze is simply connected, that is, all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze, the solver is guaranteed to find a way out, if there is one; otherwise, he or she will return to the entrance having traversed every corridor next to that connected section of walls at least once.

To the human brain, this is the easiest way to solve a maze. Amusement fair goers use this algorithm every year to solve corn mazes. But if we go by the number of lines of code, this is one of the hardest maze-solving algorithms to implement on a computer, which is really interesting.

Algorithm

```
1. START
2. Define a Current position marker
3. Place this at the Start location on the maze
4. WHILE Current position not at End
    a. IF entity at right is not a wall
        i. Turn right
    b. ELSE IF entity in front is not a wall
        i. Move ahead
    c. ELSE IF entity at left is not a wall
        i. Turn left
    d. ELSE
        i. Turn around
    e. IF Current is Start
        i. EXIT, saying Maze Cannot Be Solved
5. END, saying Maze Solved
```

This algorithm can be modified to follow the left wall as well, by switching the conditions for steps 4a and 4c.

Complexity Analysis

$T(n) = O(l * b) \qquad ..(i)$

Since, in the worst case scenario, the algorithm must check each entity (the maze having maximum number of walls and the end being one position in the opposite direction of the search), complexity will come to be as O(n). Although on the average case, the algorithm might need to traverse n/2 entities to reach the end.

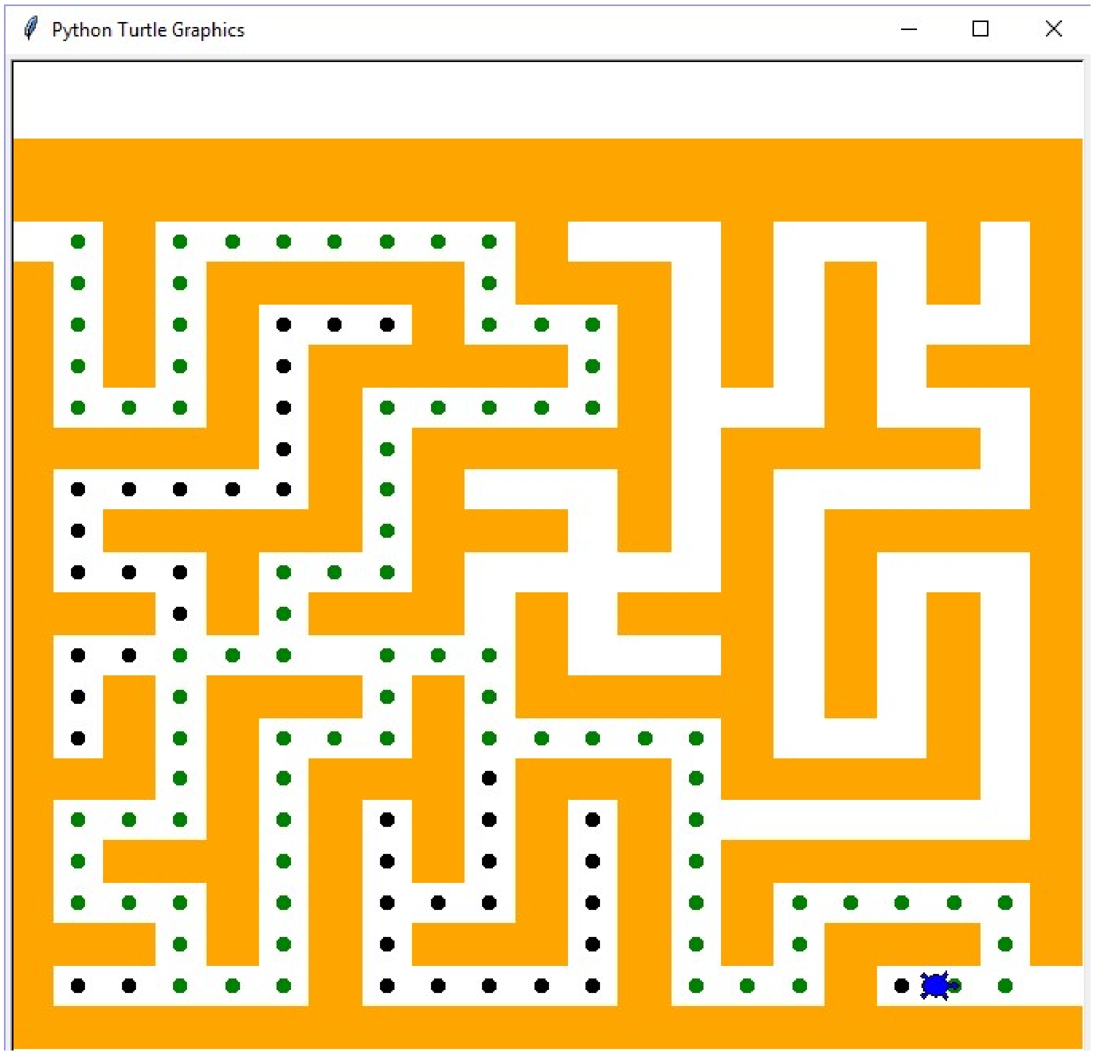
Implementation – Python Code

```python
import turtle

partofpath='O'
tried='.'
obstacle='+'
deadend='-'

class maze:
    def __init__(self,mazefile):
        rowsinmaze=0
        colsinmaze=0
        self.mazelist=[]
        maze_file=open(mazefile,'r')
        rowsinmaze=0
        for line in maze_file:
            rowlist=[]
            col=0
            for ch in line[:-1]:
                rowlist.append(ch)
                if ch=='S'  :
                    self.startrow=rowsinmaze
                    self.startcol=col
                col+=1
            rowsinmaze+=1
            self.mazelist.append(rowlist)
            colsinmaze=len(rowlist)
        self.rowsinmaze=rowsinmaze
        self.colsinmaze=colsinmaze
        self.xtranslate=-colsinmaze/2
        self.ytranslate=rowsinmaze/2
        self.t=turtle.Turtle()
        self.t.shape('turtle')
        self.wn=turtle.Screen()
        self.wn.setworldcoordinates(-(colsinmaze-1)/2-0.5,-(rowsinmaze-1)/2-0.5,
                                    (colsinmaze-1)/2,(rowsinmaze-1)/2)
    def drawmaze(self):
        self.t.speed(10000000000000000000)
        for y in range(self.rowsinmaze):
            for x in range(self.colsinmaze):
                if self.mazelist[y][x]==obstacle:
                    self.drawcenteredbox(x+self.xtranslate,-
y+self.ytranslate,'orange')
        self.t.color('black')
        self.t.fillcolor('blue')
        self.t.speed(1)

    def drawcenteredbox(self,x,y,color):
        self.t.up()
        self.t.goto(x-0.5,y-0.5)
        self.t.color(color)
        self.t.fillcolor(color)
        self.t.setheading(90)
        self.t.down()
        self.t.begin_fill()
        for i in range(4):
            self.t.forward(1)
            self.t.right(90)
        self.t.end_fill()

    def moveturtle(self,x,y):
        self.t.up()
        self.t.setheading(self.t.towards(x+self.xtranslate,-y+self.ytranslate))
        self.t.goto(x+self.xtranslate,-y+self.ytranslate)

    def dropbread(self,color):
        self.t.dot(10,color)

    def updatepos(self,row,col,val=None):
        if val:
            self.mazelist[row][col]=val
        self.moveturtle(col,row)
        if val==partofpath:
            color='green'
        elif val==obstacle:
            color='red'
        elif val==tried:
            color='black'
        else:
            color=None
        if color:
            self.dropbread(color)

    def isexit(self,row,col):
        return (row==0 or row==self.rowsinmaze-1 or col==0 or
col==self.colsinmaze-1)

    def getitem(self,idx):
        return self.mazelist(idx)



grid = """
++++++++++++++++++++++
S +       +   +   + ++
+ + +++++ +++ + + + ++
+ + +   +   + + +   ++
+ + + +++++ + + + ++++
+   + +     +   +   ++
+++++ + +++++ +++++ ++
+     + +   + +     ++
+ +++++ +++ + + ++++++
+   +   +     + +   ++
+++ + +++ + +++ + + ++
+         +   + + + ++
+ + +++ + +++++ + + ++
+ + +   +     +   + ++
+++ + +++ +++ +++++ ++
+   + + + + +       ++
+ +++ + + + + ++++++++
+   + +   + + +     ++
+++ + + +++ + + +++ ++
+     +     +   +    E
++++++++++++++++++++++
"""


grid = [list(row) for row in grid.splitlines()]


block = ["+","|","-","_","."]
path = [" ","o"]
end = ["E"]
```

```
start = ["S"]

curr =  (2,-1)
next_pos = (2,0)
last_next_pos = (2,-1)


def find_left():
      if curr[0]==next_pos[0] and curr[1]<next_pos[1]:
            return (curr[0]-1,next_pos[1])

      if curr[1]==next_pos[1] and curr[0]<next_pos[0]:
            return (curr[0]+1,next_pos[1]+1)

      if curr[0]==next_pos[0] and curr[1]>next_pos[1]:
            return (curr[0]+1,next_pos[1])

      if curr[1]==next_pos[1] and curr[0]>next_pos[0]:
            return (curr[0]-1,next_pos[1]-1)

def find_right():
      if curr[0]==next_pos[0] and curr[1]<next_pos[1]:
            return (curr[0]+1,next_pos[1])

      if curr[1]==next_pos[1] and curr[0]<next_pos[0]:
            return (curr[0]+1,next_pos[1]-1)

      if curr[0]==next_pos[0] and curr[1]>next_pos[1]:
            return (curr[0]-1,next_pos[1])

      if curr[1]==next_pos[1] and curr[0]>next_pos[0]:
            return (curr[0]-1,next_pos[1]+1)

def move_ahead():
      if curr[0]==next_pos[0] and curr[1]<next_pos[1]:
            return (curr[0],next_pos[1]+1)

      if curr[1]==next_pos[1] and curr[0]<next_pos[0]:
            return (next_pos[0]+1,next_pos[1])

      if curr[0]==next_pos[0] and curr[1]>next_pos[1]:
            return (curr[0],next_pos[1]-1)

      if curr[1]==next_pos[1] and curr[0]>next_pos[0]:
            return (next_pos[0]-1,next_pos[1])
```

```
def pos(l):
	if grid[l[0]][l[1]] in block or grid[l[0]][l[1]] in end or grid[l[0]][l[1]]
in path or grid[l[0]][l[1]] in start:
		return grid[l[0]][l[1]]

m = maze('maze1.txt')
m.drawmaze()
m.updatepos(m.startrow,m.startcol)

i = 0
print(pos(move_ahead()))
while ((pos(move_ahead()) !='E') and (pos(find_right()) !='E') and
(pos(find_left()) != 'E')):
	if ((pos(find_left()) =='S') or (pos(find_right()) =='S') or
(pos(move_ahead()) == 'S')):
		print("\nYou shall not escape this!")
		break
	i=i+1
	last_next_pos = next_pos
	if pos(find_right()) in path:
		grid[next_pos[0]][next_pos[1]]="o"
		next_pos = find_right()
		if (grid[next_pos[0]][next_pos[1]]=="o"):
			grid[curr[0]][curr[1]]="."
			m.updatepos(last_next_pos[0],last_next_pos[1],tried)
		else:
			m.updatepos(next_pos[0],next_pos[1],partofpath)
	elif pos(move_ahead()) in path:
		grid[next_pos[0]][next_pos[1]]="o"
		next_pos = move_ahead()
		if (grid[next_pos[0]][next_pos[1]]=="o"):
			grid[curr[0]][curr[1]]="."
			m.updatepos(last_next_pos[0],last_next_pos[1],tried)

		else:
			m.updatepos(next_pos[0],next_pos[1],partofpath)
	elif pos(find_left()) in path:
		grid[next_pos[0]][next_pos[1]]="o"
		next_pos = find_left()
		if (grid[next_pos[0]][next_pos[1]]=="o"):
			grid[curr[0]][curr[1]]="."
			m.updatepos(last_next_pos[0],last_next_pos[1],tried)
		else:
			m.updatepos(next_pos[0],next_pos[1],partofpath)
	else:
		m.updatepos(next_pos[0],next_pos[1],tried)
		grid[next_pos[0]][next_pos[1]] = "."
```

```
                grid[curr[0]][curr[1]] = " "
                next_pos = curr

        curr=last_next_pos

print("\n".join("".join(row) for row in grid))
```



Recursive Algorithm

Preface

If given an omniscient view of the maze, a simple recursive algorithm can tell one how to get to the end. The algorithm will be given a starting X and Y value. If the X and Y values do not represent a wall, the method will call itself with all adjacent X and Y values, making sure that it did not already use those X and Y values before. If the X and Y values are those of the end location, it will save all the previous instances of the method as the correct path.

As compared to most maze algorithms, this algorithm is the easier to implement. However, it comes at the cost of computational power. It will return the first complete path that it encounters.

Algorithm

```
1. START
2. Define a Current position marker
3. Place this at the Start location on the maze
4. IF Current position is a wall
     a. RETURN False
5. IF Current position is End
     a. Mark path
     b. RETURN True
6. ELSE
     a. FOR each entity adjacent to Current position
          i. Perform the algorithm on the entity
          ii. IF True
               i. Mark Path
               ii. Return True
7. END, saying Maze Solved
```

Complexity Analysis

$T(n) = 4\,T\left(\frac{n}{4}\right) + C \qquad ..(i)$

$a = 4, b = 4, f(n) = c \qquad ..(ii)$

$By\ Case\ I\ of\ Master's\ Theorem,$

$T(n) = O(l * b) \qquad ..(iii)$

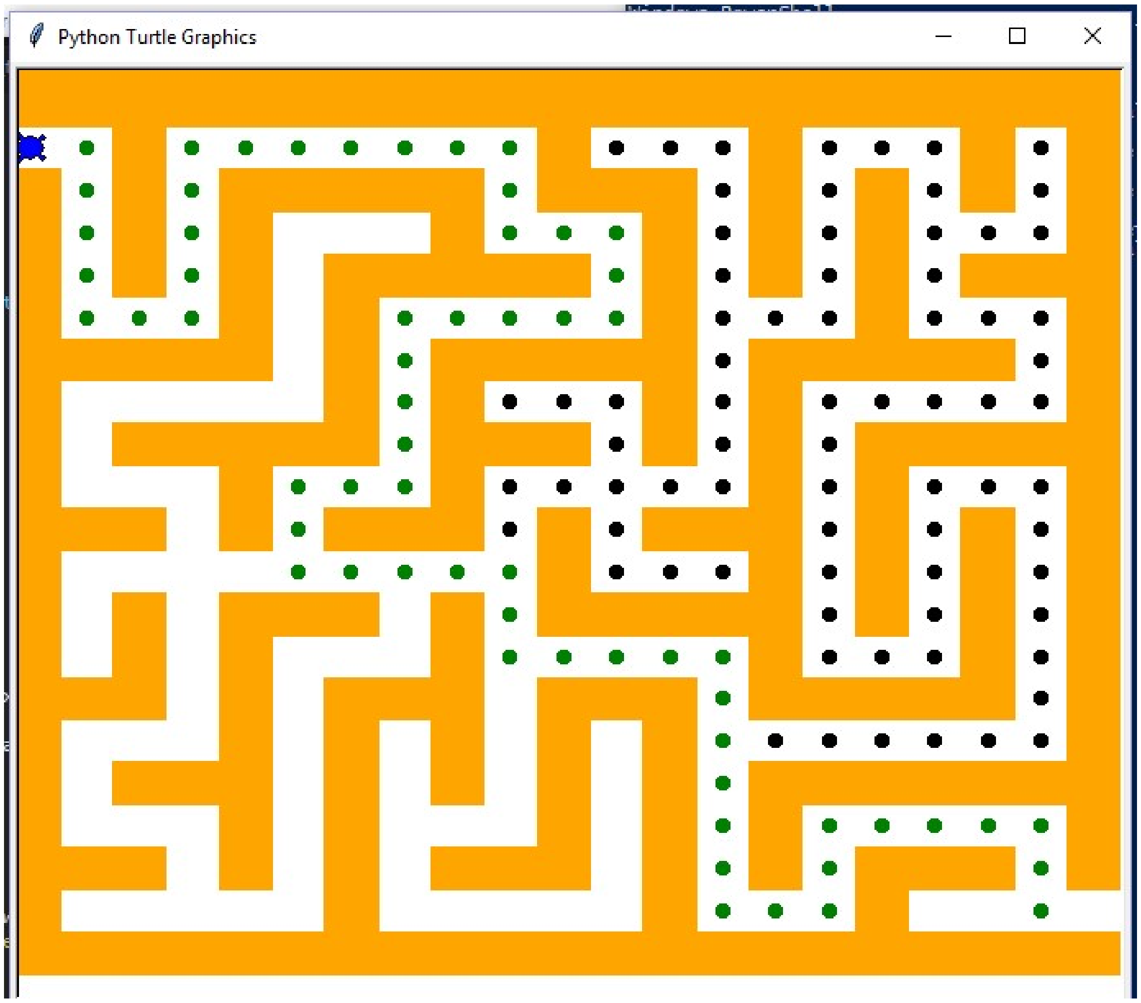
Since every recursion call will act on one adjacent entity of the Current position i.e. $\frac{1}{4}$ of the remaining positons, and since each call will in turn call 4 function calls (for 4 adjacent entities), we arrive at equation (i).

Implementation – Python Code

```
import turtle

partofpath='O'
tried='.'
obstacle='+'
deadend='-'

class maze:
    def __init__(self,mazefile):
        rowsinmaze=0
        colsinmaze=0
        self.mazelist=[]
        maze_file=open(mazefile,'r')
        rowsinmaze=0
        for line in maze_file:
            rowlist=[]
            col=0
            for ch in line[:-1]:
                rowlist.append(ch)
                if ch=='S'  :
                    self.startrow=rowsinmaze
                    self.startcol=col
                col+=1
            rowsinmaze+=1
            self.mazelist.append(rowlist)
            colsinmaze=len(rowlist)
        self.rowsinmaze=rowsinmaze
        self.colsinmaze=colsinmaze
        self.xtranslate=-colsinmaze/2
        self.ytranslate=rowsinmaze/2
        self.t=turtle.Turtle()
        self.t.shape('turtle')
        self.wn=turtle.Screen()
        self.wn.setworldcoordinates(-(colsinmaze-1)/2-0.5,-(rowsinmaze-1)/2-0.5,
                                     (colsinmaze-1)/2,(rowsinmaze-1)/2)
    def drawmaze(self):
        self.t.speed(10000000000000000000)
        for y in range(self.rowsinmaze):
            for x in range(self.colsinmaze):
                if self.mazelist[y][x]==obstacle:
                    self.drawcenteredbox(x+self.xtranslate,-
y+self.ytranslate,'orange')
        self.t.color('black')
        self.t.fillcolor('blue')
        self.t.speed(1)

    def drawcenteredbox(self,x,y,color):
        self.t.up()
        self.t.goto(x-0.5,y-0.5)
        self.t.color(color)
        self.t.fillcolor(color)
        self.t.setheading(90)
        self.t.down()
        self.t.begin_fill()
        for i in range(4):
            self.t.forward(1)
            self.t.right(90)
        self.t.end_fill()

    def moveturtle(self,x,y):
        self.t.up()
        self.t.setheading(self.t.towards(x+self.xtranslate,-y+self.ytranslate))
        self.t.goto(x+self.xtranslate,-y+self.ytranslate)

    def dropbread(self,color):
        self.t.dot(10,color)

    def updatepos(self,row,col,val=None):
        if val:
            self.mazelist[row][col]=val
        self.moveturtle(col,row)
        if val==partofpath:
            color='green'
        elif val==obstacle:
            color='red'
        elif val==tried:
            color='black'
        else:
            color=None
        if color:
            self.dropbread(color)

    def isexit(self,row,col):
        return (row==0 or row==self.rowsinmaze-1 or col==0 or
col==self.colsinmaze-1)
```

```
    def getitem(self,idx):
        return self.mazelist(idx)


m=maze('maze1.txt')
m.drawmaze()
m.updatepos(m.startrow,m.startcol)

grid = """
+-+-+-+-+-+-+-+-+-+-+
S |       |   |   | |
+ + +-+-+ +-+ + + + +
| | |   |   | | |   |
+ + + +-+-+ + + + +-+
|   | |     |   |   |
+-+-+ + +-+-+ +-+-+ +
|     | |   | |     |
+ +-+-+ +-+ + + +-+-+
|   |   |     | |   |
+-+ + +-+ + +-+ + + +
|         |   | | | |
+ + +-+ + +-+-+ + + +
| | |   |     |   | |
+-+ + +-+ +-+ +-+-+ +
|   | | | | |       |
+ +-+ + + + + +-+-+-+
|   | |   | | |     |
+-+ + + +-+ + + +-+ +
|     |     |   |   E
+-+-+-+-+-+-+-+-+-+-+
"""

g = [list(row) for row in grid.splitlines()]

for i in range(len(g)):
       for j in range(len(g[i])):
             if g[i][j]=="S":
                    start_x,start_y = i,j
             if g[i][j]=="E":
                    end_x,end_y = i,j
start,end,visit,sol,path = "SE.o "

def search(x,y):
       if g[x][y] in (start,path):
             g[x][y] = visit
```

```
            m.updatepos(x,y,tried)
            if search(x,y+1) or search(x,y-1) or search(x+1,y) or search(x-1,y):
                g[x][y] = sol
                m.updatepos(x,y,partofpath)
                return True
        elif g[x][y] == end:
            return True
        return False

search(start_x,start_y)
print("\n".join("".join(row) for row in g))
```


Python Turtle Graphics