MAZE SOLVING
REPORT

MOHANA KARTHIK IANALA

Title:

Solving Mazes from an input Image using Artificial Intelligence

Abstract:

In this Project, a Black and White image of maze is taken as input and stored in the memory. A black pixel represents wall and a white pixel represents path

Start pixel is at the top and End pixel is at the bottom. All mazes used are entirely surrounded black walls except the start and the end white pixels.

When it is traversing through a path pixel, it checks for all the possible neighbouring path pixels where it can go further. This goes on till it reaches the end pixel

Keywords:

Breadth-First Algorithm: It is a traversing Algorithm which starts from the selected node(Start Node) and explore all its children nodes level wise till the goal node is reached

Depth-First Algorithm: It is a traversing Algorithm which starts from the selected node(Start Node) and traverse through one of the child node to the deepest level and if goalstate not found, it backtracks to the start and explores other child nodes

Pillow Library: It is a **Python library** which deals with image processing and visualization. This library is a replacement to **PIL(Python Imaging Library)** which is discontinued.

PNG(Portable Network Graphics): It is an Image format and it uses Lossless Compression

JPEG(Joint Photographic Experts Group): It is also an Image format but it uses Lossy Compression

JPEG tends to loss some data of the image where are PNG doesn’t

Introduction:

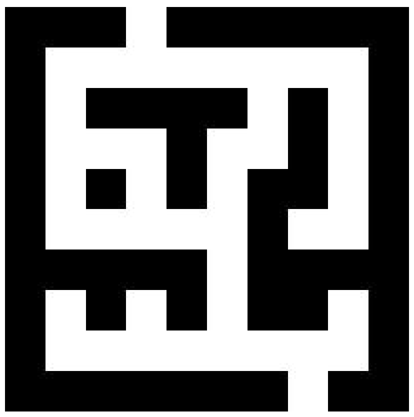
Maze Solving is a retro game which used to be published in newspapers and articles for children. In this project, when the user inputs a black and white image of a maze, the program traverses through all the possible outcomes and presents the output as an image with the path indicated with a series of colours ranging from blue to red.

Rules considered:

- Images should be Black and White
- Black pixel is wall
- White pixel is path
- Maze is completely surrounded by the wall except for the Start and end
- Entrance and exit should be at top and bottom respectively

**Image uploaded should avoid jpeg compression and most preferably .png format*

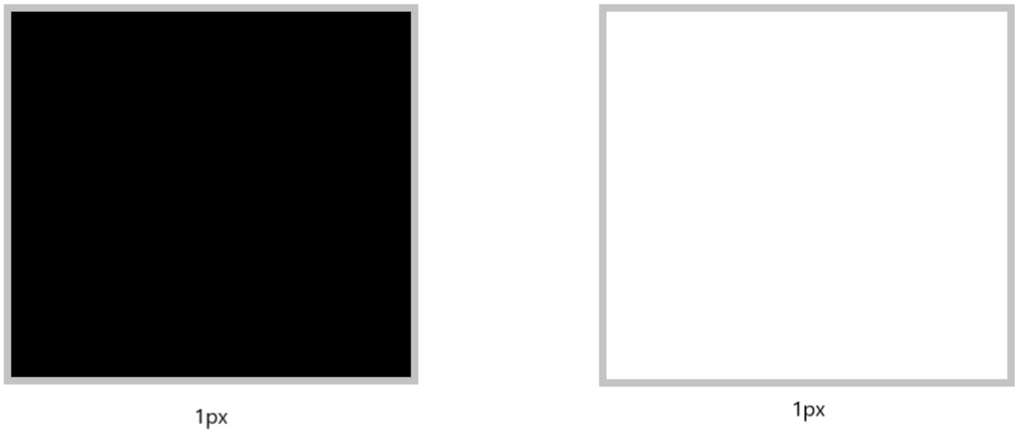
Sample Maze:



The image of the maze following the above rules is put into the memory, where it analyses the black walls and path.

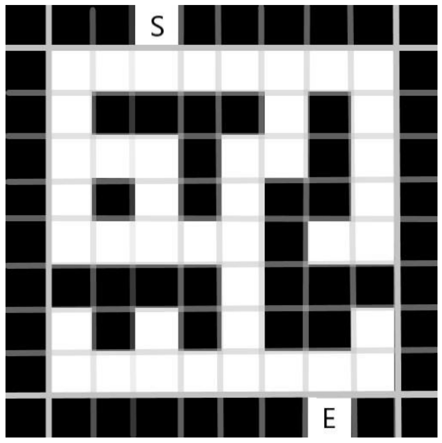
There it identifies the start and creates the starting node. From there, it checks all the neighbours and if we can go there, it follows the path and creates another node there. This goes on until end is reached

In the maze each wall and each path is 1pixel



In this image grey boundary is just for understanding

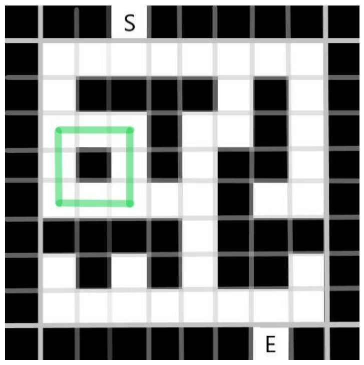
The maze looks like a grid for better understanding



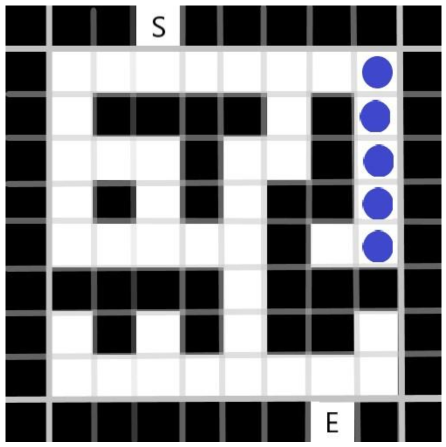
S is Start and E is End

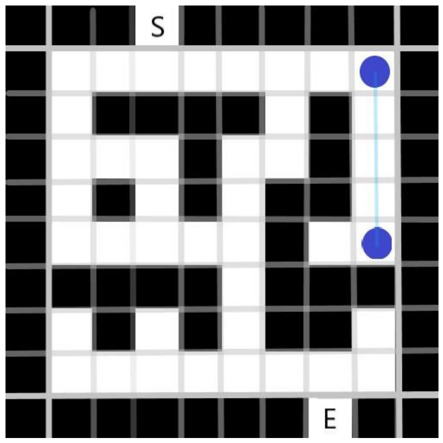
Optimality:

To avoid looping of path or going to the previous nodes, another array of the same size of the maze's grid is taken, which is basically true or false, where true means already visited node and false means unvisited node.



If every path pixel is taken into consideration and made a node, it's space complexity increases exponentially. To avoid this only the end of a straight path nodes are created





Instead of 5 nodes, only two end nodes are considered, thus decreasing the space complexity.

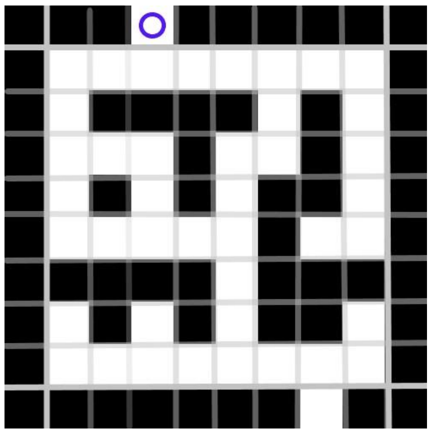
Objective:

This Project is a Path Finding Algorithm which solves Mazes which are taken input. These mazes inputed are in image format. These algorithm analyse all the path pixels and then find the optimal path connecting Start and Stop.

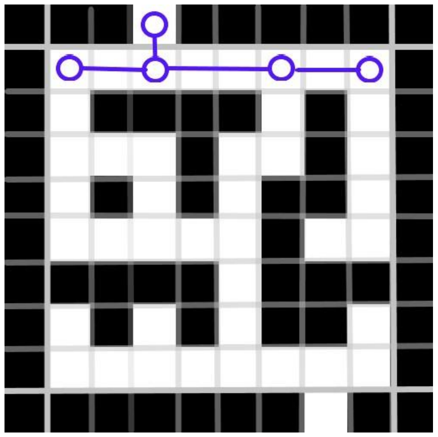
Methodology:

Mazes are taken as input in the image format, so **Python Imaging Library(PIL)** is used. When the maze is put into the memory, it analyses every pixel and forms an array

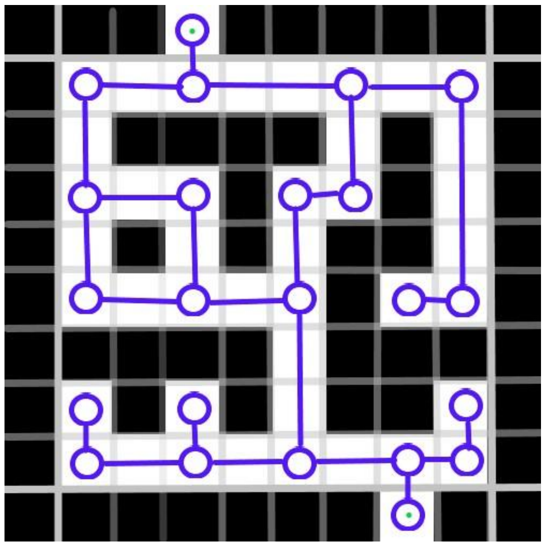
We traverse through the path pixels column wise and create nodes. It starts by identifying the Start node.



Then all the possible paths are explored, and nodes are created where there are more than 1 path neighbours and nodes are created.



This process is repeated until the last node or end node is found. So the final node graph is



If we have taken all the path nodes instead of straight line end nodes, there would be more nodes stored in the memory which are useless. Step count is also reduced by this.

Now we have all that are necessary: start node, path nodes, and an end node. So we apply **Breadth First Search(BFS**) or **Depth First Search (DFS)** to traverse through every node possible connecting **Start node** and **End node(Goal State).**

Code:

As both BFS and DFS are used, the program is made Modular for easy understanding.

solve.py # This is the Main program where we take inputs and send them to respected module

```
from PIL import Image
import time
from mazes import Maze
from factory import SolverFactory
Image.MAX_IMAGE_PIXELS = None

# Read command line arguments - the python argparse class is convenient
here.
import argparse

def solve(factory, method, output_file):
    # Load Image
    print ("Loading Image")
    im = Image.open("m_4.png")

    # Create the maze (and time it) - for many mazes this is more time
consuming than solving the maze
    print ("Creating Maze")
    t0 = time.time()
    maze = Maze(im)
    t1 = time.time()
    print ("Node Count:", maze.count)
    total = t1-t0
    print ("Time elapsed:", total, "\n")

    # Create and run solver
    [title, solver] = factory.createsolver(method)
    print ("Starting Solve:", title)
    t0 = time.time()
    [result, stats] = solver(maze)
    t1 = time.time()

    total = t1-t0

    # Print solve stats
    print ("Nodes explored: ", stats[0])
    if (stats[2]):
        print ("Path found, length", stats[1])
    else:
        print ("No Path Found")
    print ("Time elapsed: ", total, "\n")

    """
    Create and save the output image.
    This is simple drawing code that travels between each node in turn,
drawing either
    a horizontal or vertical line as required. Line colour is roughly
interpolated between
    blue and red depending on how far down the path this section is.
    """

    print ("Saving Image")
    im = im.convert('RGB')
    impixels = im.load()

    resultpath = [n.Position for n in result]

    length = len(resultpath)

    for i in range(0, length - 1):
        a = resultpath[i]
        b = resultpath[i+1]

        # Blue - red
        r = int((i / length) * 255)
        px = (r, 0, 255 - r)

        if a[0] == b[0]:
            # Ys equal - horizontal line
            for x in range(min(a[1],b[1]), max(a[1],b[1])):
                impixels[x,a[0]] = px
        elif a[1] == b[1]:
            # Xs equal - vertical line
            for y in range(min(a[0],b[0]), max(a[0],b[0]) + 1):
                impixels[a[1],y] = px

    im.save(output_file)


def main():
    sf = SolverFactory()
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--method", nargs='?', const=sf.Default,
default=sf.Default,
                        choices=sf.Choices)
    parser.add_argument("output_file")
    args = parser.parse_args()

    solve(sf, args.method, args.output_file)

if __name__ == "__main__":
    main()
```

factory.py and profile.py

#These are Quality of Life Modules where you can select which Searching Algo to be used and have some default mazes

profile.py

```
import tempfile
from solve import solve
from factory import SolverFactory

methods = [
    "breadthfirst",
    "depthfirst",

]
inputs = [
    "tiny",
    "small",
    "normal",
    "perfect2k",

]

def profile():
    for m in methods:
        for i in inputs:
            with tempfile.NamedTemporaryFile(suffix='.png') as tmp:
                solve(SolverFactory(), m, "examples/%s.png" % i, tmp.name)

profiler = BProfile('profiler.png')
with profiler:
    profile()
```

factory.py

```
# A simple factory class that imports and returns a relevant solver when
provided a string
# Not hugely necessary, but reduces the code in solve.py, making it easier to
read.

class SolverFactory:
  def __init__(self):
    self.Default = "breadthfirst"
    self.Choices = ["breadthfirst","depthfirst"]

  def createsolver(self, type):
    if type == "depthfirst":
      import depthfirst
      return ["Depth first search", depthfirst.solve]
    else:
      import breadthfirst
      return ["Breadth first search", breadthfirst.solve]
```

mazes.py

```
   #This module is a key module where all the traversing and nodes
  are created, it also checks whether the path is taken already or not
class Maze:
  class Node:
    def __init__(self, position):
      self.Position = position
      self.Neighbours = [None, None, None, None]
      #self.Weights = [0, 0, 0, 0]

  def __init__(self, im):

    width = im.size[0]
```

```
        height = im.size[1]
        data = list(im.getdata(0))

        self.start = None
        self.end = None

        # Top row buffer
        topnodes = [None] * width
        count = 0

        # Start row
        for x in range (1, width - 1):
            if data[x] > 0:
                self.start = Maze.Node((0,x))
                topnodes[x] = self.start
                count += 1
                break

        for y in range (1, height - 1):
            #print ("row", str(y)) # Uncomment this line to keep a track of row
progress

            rowoffset = y * width
            rowaboveoffset = rowoffset - width
            rowbelowoffset = rowoffset + width

            # Initialise previous, current and next values
            prv = False
            cur = False
            nxt = data[rowoffset + 1] > 0

            leftnode = None

            for x in range (1, width - 1):
                # Move prev, current and next onwards. This way we read from the
image once per pixel, marginal optimisation
                prv = cur
            cur = nxt
            nxt = data[rowoffset + x + 1] > 0

            n = None

            if cur == False:
                # ON WALL - No action
                continue

            if prv == True:
                if nxt == True:
                    # PATH PATH PATH
                    # Create node only if paths above or below
                    if data[rowaboveoffset + x] > 0 or data[rowbelowoffset + x] > 0:
                        n = Maze.Node((y,x))
                        leftnode.Neighbours[1] = n
                        n.Neighbours[3] = leftnode
                        leftnode = n
                else:
                    # PATH PATH WALL
                    # Create path at end of corridor
                    n = Maze.Node((y,x))
                    leftnode.Neighbours[1] = n
                    n.Neighbours[3] = leftnode
                    leftnode = None
            else:
                if nxt == True:
                    # WALL PATH PATH
                    # Create path at start of corridor
                    n = Maze.Node((y,x))
                    leftnode = n
                else:
                    # WALL PATH WALL
                    # Create node only if in dead end
                    if (data[rowaboveoffset + x] == 0) or (data[rowbelowoffset + x]
== 0):
                        #print ("Create Node in dead end")
                        n = Maze.Node((y,x))

            # If node isn't none, we can assume we can connect N-S
somewhere
            if n != None:
                # Clear above, connect to waiting top node
                if (data[rowaboveoffset + x] > 0):
                    t = topnodes[x]
                    t.Neighbours[2] = n
                    n.Neighbours[0] = t

                # If clear below, put this new node in the top row for the next
connection
                if (data[rowbelowoffset + x] > 0):
                    topnodes[x] = n
                else:
                    topnodes[x] = None

                count += 1

    # End row
    rowoffset = (height - 1) * width
    for x in range (1, width - 1):
        if data[rowoffset + x] > 0:
            self.end = Maze.Node((height - 1,x))
            t = topnodes[x]
            t.Neighbours[2] = self.end
            self.end.Neighbours[0] = t
            count += 1
            break

    self.count = count
    self.width = width
    self.height = height
```

breadthfirst.py

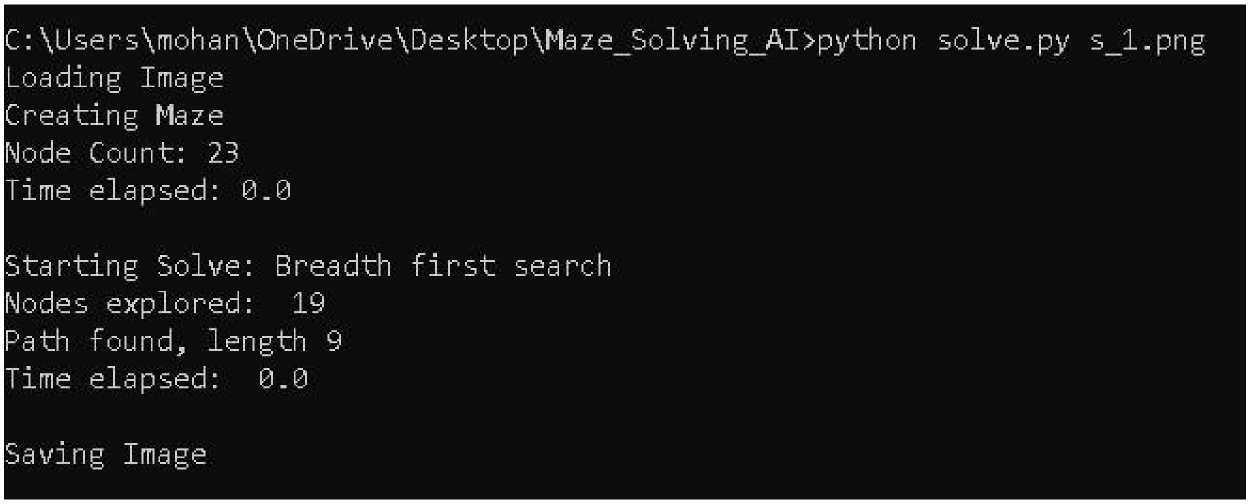
```
  # This module is used to find the optimal path from start to end by traversing
                                                          the nodes with BFS Algo
from collections import deque

def solve(maze):
    start = maze.start
    end = maze.end

    width = maze.width

    queue = deque([start])
    shape = (maze.height, maze.width)
    prev = [None] * (maze.width * maze.height)
    visited = [False] * (maze.width * maze.height)

    count = 0

    completed = False

    visited[start.Position[0] * width + start.Position[1]] = True

    while queue:
        count += 1
        current = queue.pop()

        if current == end:
            completed = True
            break

        for n in current.Neighbours:
            if n != None:
                npos = n.Position[0] * width + n.Position[1]
                if visited[npos] == False:
                    queue.appendleft(n)
                    visited[npos] = True
                    prev[npos] = current
    path = deque()
    current = end
    while (current != None):
        path.appendleft(current)
        current = prev[current.Position[0] * width + current.Position[1]]

    return [path, [count, len(path), completed]]
```

depthfirst.py

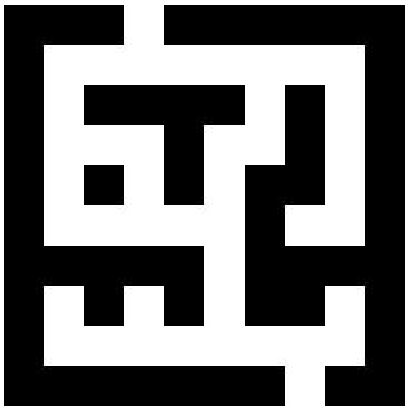
```
   #This module is used to find the optimal path from start to end by traversing
                                                         the nodes with DFS Algo
from collections import deque

def solve(maze):
    start = maze.start
    end = maze.end
    width = maze.width
    stack = deque([start])
    shape = (maze.height, maze.width)
    prev = [None] * (maze.width * maze.height)
    visited = [False] * (maze.width * maze.height)
    count = 0

    completed = False
    while stack:
        count += 1
        current = stack.pop()

        if current == end:
            completed = True
            break

        visited[current.Position[0] * width + current.Position[1]] = True

        #import code
        #code.interact(local=locals())
    for n in current.Neighbours:
        if n != None:
            npos = n.Position[0] * width + n.Position[1]
            if visited[npos] == False:
                stack.append(n)
                prev[npos] = current

path = deque()
current = end
while (current != None):
    path.appendleft(current)
    current = prev[current.Position[0] * width + current.Position[1]]

return [path, [count, len(path), completed]]
```

Result and Discussion:

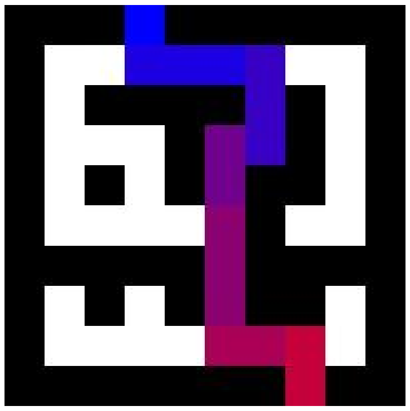
❖ Maze 1(Small)



```
C:\Users\mohan\OneDrive\Desktop\Maze_Solving_AI>python solve.py s_1.png
Loading Image
Creating Maze
Node Count: 23
Time elapsed: 0.0

Starting Solve: Breadth first search
Nodes explored:  19
Path found, length 9
Time elapsed:  0.0

Saving Image
```

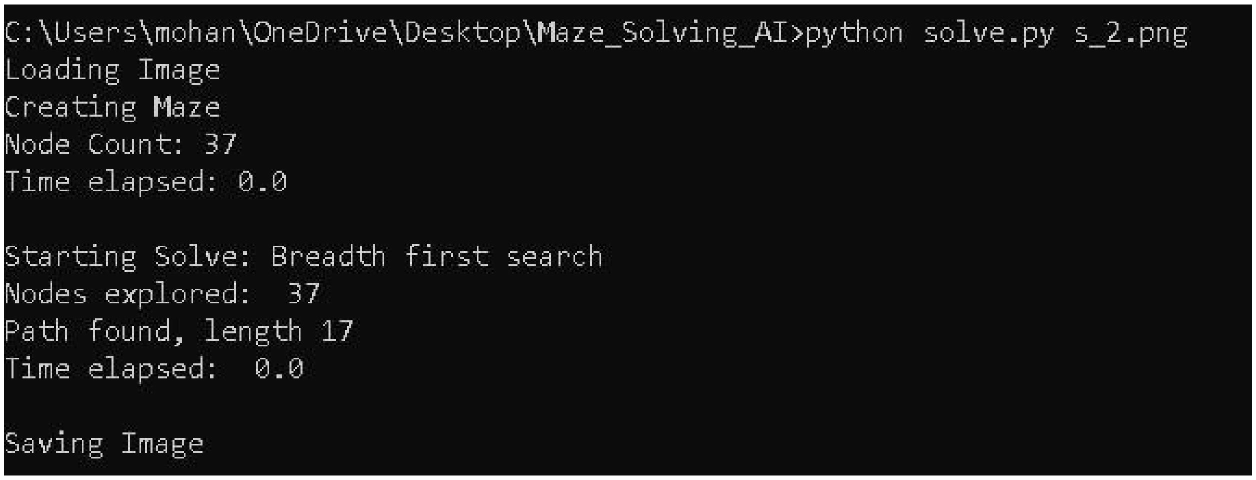


⇒

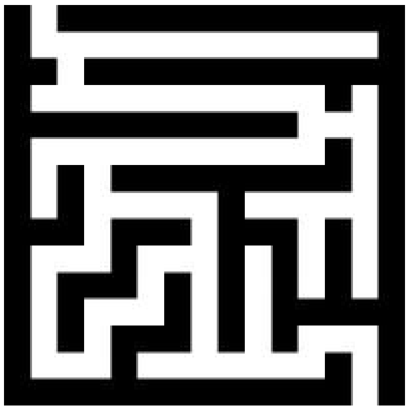


Maze Solution

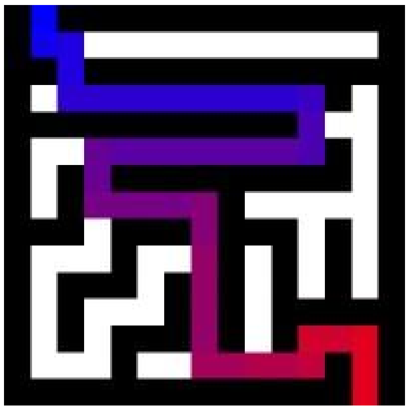
❖ Maze 2 (Normal):



```
C:\Users\mohan\OneDrive\Desktop\Maze_Solving_AI>python solve.py s_2.png
Loading Image
Creating Maze
Node Count: 37
Time elapsed: 0.0

Starting Solve: Breadth first search
Nodes explored:  37
Path found, length 17
Time elapsed:  0.0

Saving Image
```



⇒

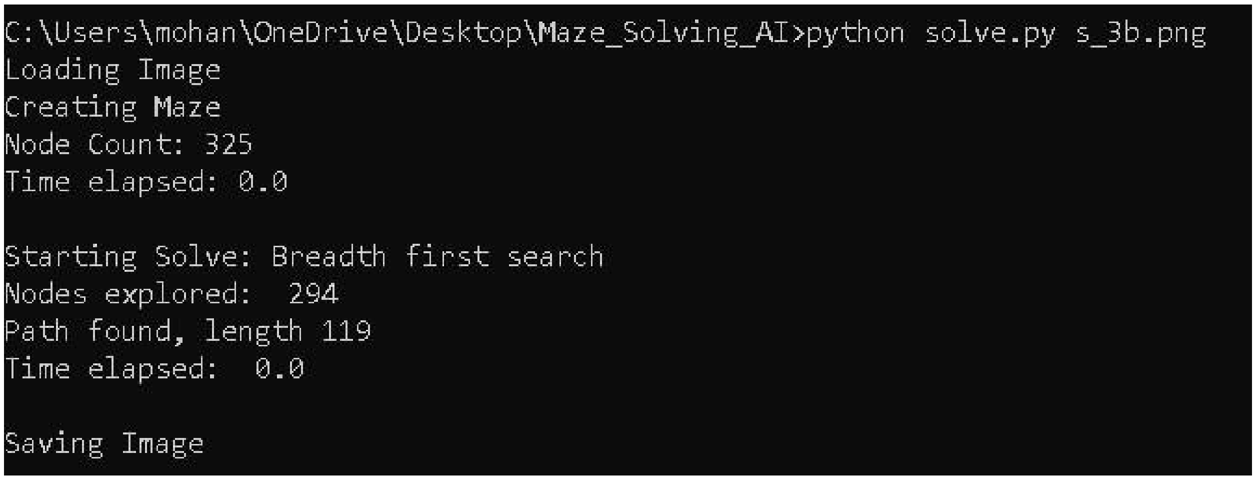


Maze

Solution

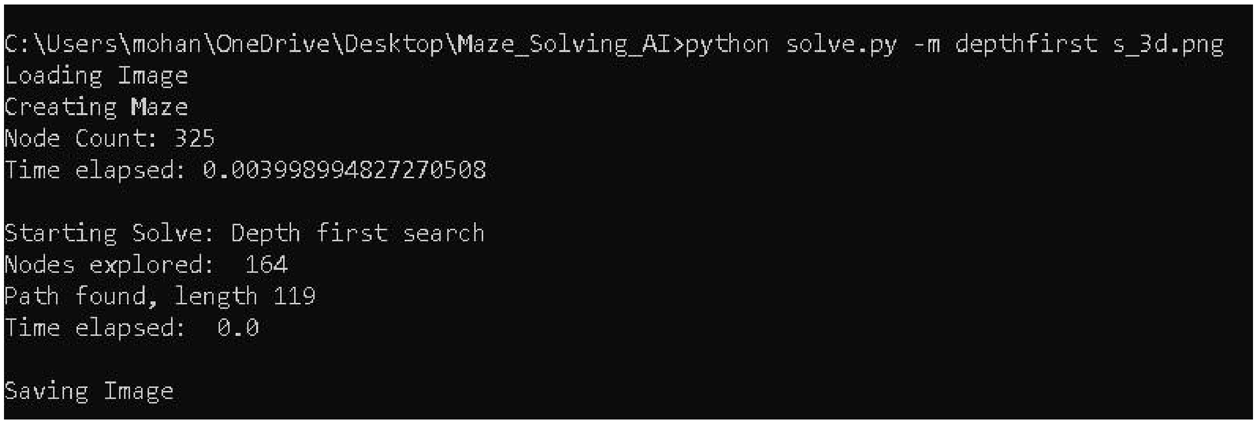
❖ Maze 3 (Large):

BFS:

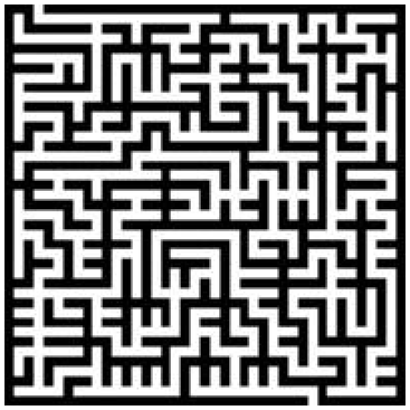


```
C:\Users\mohan\OneDrive\Desktop\Maze_Solving_AI>python solve.py s_3b.png
Loading Image
Creating Maze
Node Count: 325
Time elapsed: 0.0

Starting Solve: Breadth first search
Nodes explored:  294
Path found, length 119
Time elapsed:  0.0

Saving Image
```

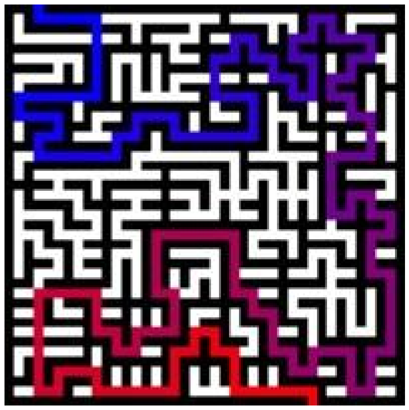
DFS:



```
C:\Users\mohan\OneDrive\Desktop\Maze_Solving_AI>python solve.py -m depthfirst s_3d.png
Loading Image
Creating Maze
Node Count: 325
Time elapsed: 0.003998994827270508

Starting Solve: Depth first search
Nodes explored:  164
Path found, length 119
Time elapsed:  0.0

Saving Image
```



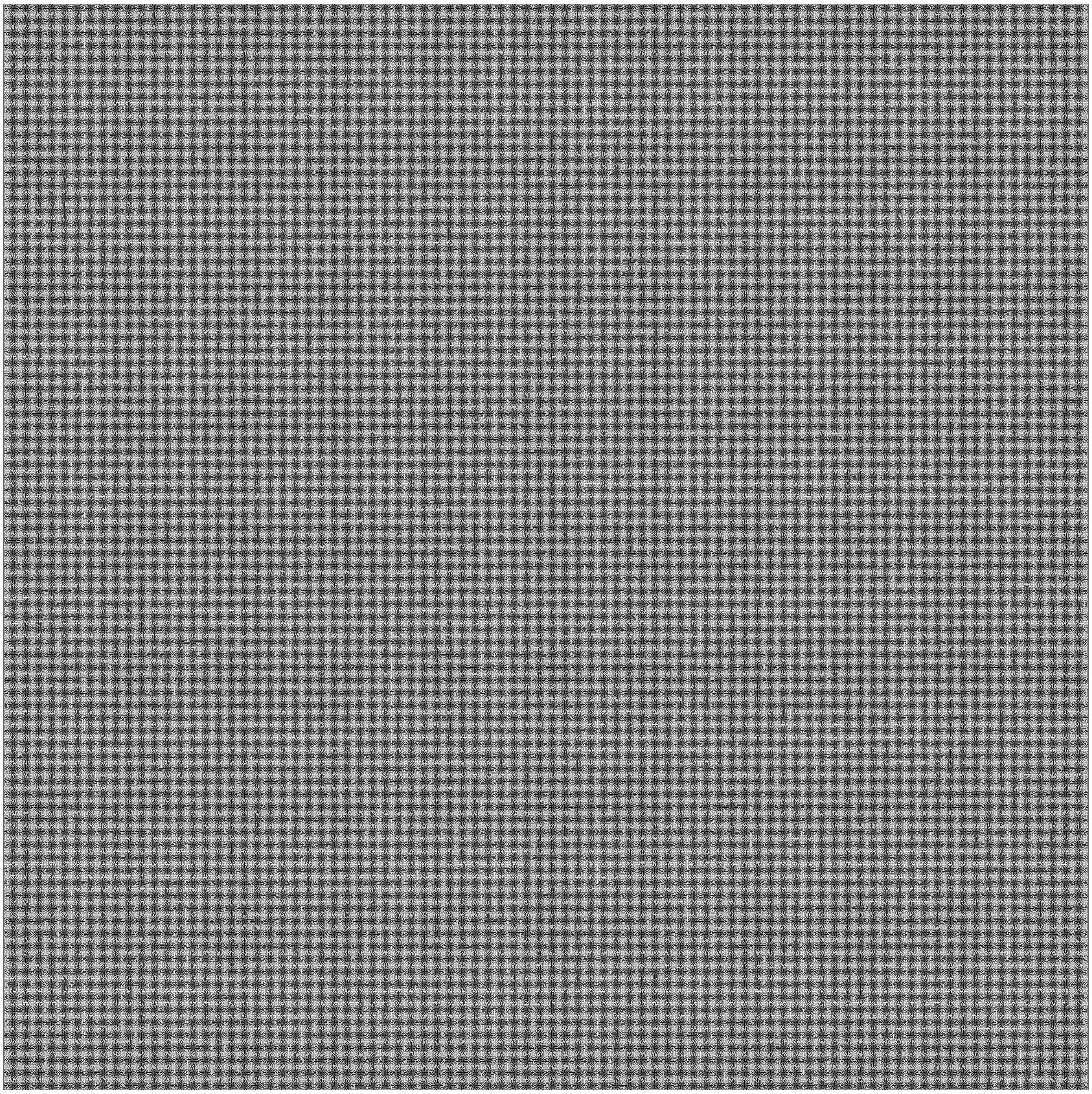
⇒

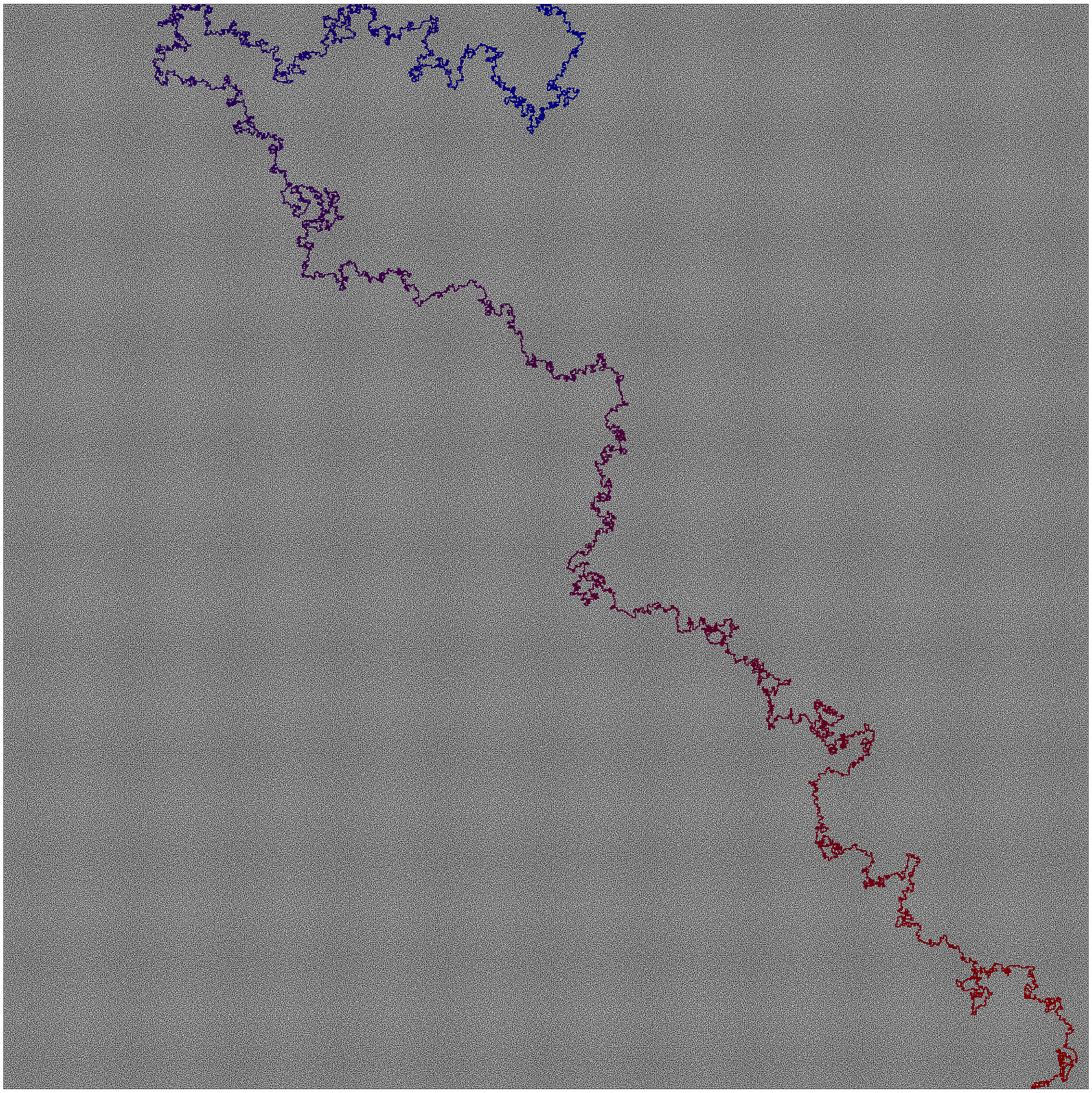


Maze   Solution

This can solve large mazes too which follow the states rules but need high computational power

I have tried a 20 megapixels image and it has solved it but the images are too big to be clearly shown.





Conclusion:

This Project is a Path Finding Algorithm which visualizes Mazes which are taken as input images. It can solve all the mazes which follow the above stated rules and also gives the optimal solution. This algorithm analyses the type of pixel where it is traversing and differentiate all the path pixels from wall pixels and then find the optimal path connecting Start and Stop.

This project is made modular so that it is easy to understand and also we can implement other searching algos in the future.

THANK YOU