**INTRODUCTION**

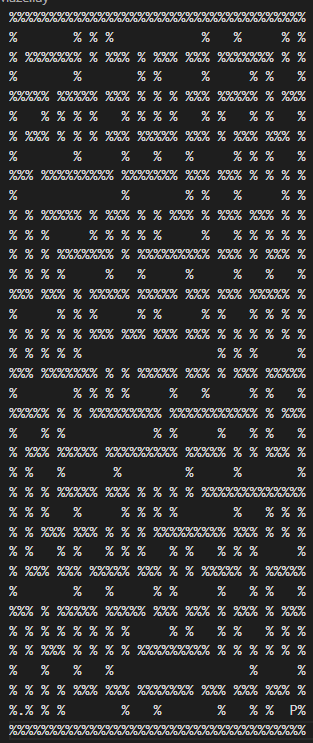
In this assignment, the problem of finding a path through the maze from a given start position to an end position the maze layout will be given to us in the form of “*. lay*” files where the *“%”* stands for wall or border and *“P”*stands for starting position and **“.”** stands for ending position. We will select one searching algorithm from “*Depth first search*” and “*Breath first search*” and “*A\**” algorithm with Manhattan distance with heuristic function to solve with different mazes

We will run each of the selected algorithm on all the four different mazes of various sizes “*Small*, *Medium*, *Big* and *Open*” for each problem. We will search the path using different algorithms we have implemented and we will report the solution, and it part costs, number of nodes it has explored and the maximum depth of the tree and the maximum size of the fringe the final solution will be described by putting a dot in every node position visited on the path and using pygame we will visualize the solution.

This practical experience in implementing analysis different search algorithms and understanding their strengths and limitations in solving different types of problems .

**MAZE REPRESENTATION**

The mazes are represented in “. lay” format where the **“%”** symbol denotes the wall or border and the **“P”** denotes the starting point of the pacman and **“.”** denotes the ending point to be reached and we will convert this “. lay” file into a matrix to work on it



**SEARCHING ALGORITHMS**

We will be implementing three searching algorithms, namely DFS, BFS and A\* search. In this Both DFS&BFS are Uninformed search algorithms. While A\* is an informed search algorithm. For A\* algorithm, we will be using Manhattan distance with the help of heuristic function.

How **DFS(DepthFirstSearch)** works DFS can be implemented with recursion. It starts from the source cell and explores the adjacent cells in 4 directions (up, down, left, right). The adjacent cells are accessed by increasing or decreasing the indices of the row or the column by 1 at a time. If the cell’s value is pass, it continues further. If a cell’s all un-visited adjacent cells are blocked or beyond the edge, this is a dead end. Then the recursive method returns by the order of call stacks and explores other routes.

How **BFS (BreadthFirstSearch)** works BFS uses queue to traverse nodes or vertex of a graph or tree. BFS first pushes all neighbour nodes of the current depth into the queue and then visits them by following the FIFO (First in First Out) pattern. While visiting a node, BFS simultaneously adds its unvisited neighbours to the queue. Since the queue follows the FIFO pattern, the vertex at the highest level is popped and visited first. More is the depth of the nodes later it will be processed by the queue, hence implementing breadth-first search traversal.

How **A\*** works and what is **Manhattan distance** A\* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes It will be used for the shortest path finding the extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A\* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. Manhattan distance is nothing but the sum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively.

**Heuristic Function:**

h = abs (current\_cell.x – goal.x) + abs (current\_cell.y – goal.y)

**WORKING OF** **FUNCTIONS:**

**layToMatrix(path)**

This function reads the path which leads to the “. lay” files and converts into 2D Matrix of characters, where each character represents a cell in the maze.

This function first opens the file using the with open statement, which automatically opens the file and reads all the lines in the file using the “readlines()” function and then it converts each line into a list of character. And finally, it returns a list of lists representing the 2D matters of the characters.

**findStartAndEnd(mat)**

This function helps to find the starting position of the pacman and the ending position of the maze.

This function iterates through all the lists inside the Matrix and iterate over the characters in the list.

If the character is equal to “P” it will initiate Start as the indexes 0f (i, j) in to tuple and if the character is equal to “.” it will initialize end as indexes of (i, j) in to tuple and returns the start and end.

**neighbours(mazeMat, node)**

This function takes 2 parameters. mazeMat and node. were mazeMat is a representation of matrix. And the node is a tuple that consist the current position to which we want to find out the neighbours.

This function returns a list of tuples representing all the valid neighbours from the current position in the maze. It is done by iterating in a predefined “direction” list which contains tuple representing X&Y values neighbour nodes. i.e., up, down, right, left for each neighbour. It checks if the neighbours within the boundary of the matrix and the neighbour is valid, it will be added to the list of neighbours.

**findPathUsingDFS(mazeMat)**

The function “findPathUsingDFS” is an implementation of depth first search algorithm to find a path through the maze from the given start position to the end position. The input for this function is given in the format of two-dimensional matrix.

This function starts by finding the start and end position of the matrix by calling the function “findStartAndEnd” Function. After that, initiate a “node\_expanded” as zero, “max\_depth” has zero, and max\_fringe has zero and initializes the stacked data structure. With a tuple containing the start node and the list containing the start node, and initializes set named has “visitedPath” to keep the track of visited nodes.

The function then enters the loop that continues While the stack is not empty, end goal is found, In this iteration of the loop. It pops the top node and the path from the stack. And increment the node expanded value of by 1. And also updated the maximum depth and the maximum freeze by keep track in the maximum death and the size of the freeze. The function then calls the draw function to draw the path in pygame console using the colour red. if the popped node is equals to end node, it returns the path and path cost, number of nodes expanded, and the maximum depth of the tree, and the maximum size of the films. If the popped node is not in the visited set, it adds in the visited set and generate list of unvisited neighbours from the popped node using the ‘neighbours’ function. It then appends each unvisited neighbour and its corresponding path to the stack. Finally, the goal is not found, and the stack is empty. The function returns None for all output parameters.

**findPathUsingBFS(mazeMat)**

The function “findPathUsingBFS” is an implementation of breadth first search algorithm to find a path through the maze from the given start position to the end position. The input for this function is given in the format of two-dimensional matrix.

This function starts by finding the start and end position of the matrix by calling the function “findStartAndEnd” Function. After that, initiate a “node\_expanded” as zero, “max\_depth” has zero, and max\_fringe has zero and initializes the queue(deque) data structure. With a tuple containing the start node and the list containing the start node, and initializes set named has “visitedPath” to keep the track of visited nodes.

The function then enters the loop that continues While the deque is not empty, end goal is found, In this iteration of the loop. It pops the node and the path from the deque. And increment the node expanded value of by 1. And also updated the maximum depth and the maximum freeze by keep track in the maximum death and the size of the freeze. The function then calls the draw function to draw the path in pygame console using the colour red. if the popped node is equals to end node, it returns the path and path cost, number of nodes expanded, and the maximum depth of the tree, and the maximum size of the films. If the popped node is not in the visited set, it adds in the visited set and generate list of unvisited neighbours from the popped node using the ‘neighbours’ function. It then appends each unvisited neighbour and its corresponding path to the deque. Finally, the goal is not found, and the stack is empty. The function returns None for all output parameter

**findPathUsingAStar(mazeMat)**

The function “findPathUsingAStar” is an implementation of A\* algorithm to find a path through the maze from the given start position to the end position. The input for this function is given in the format of two-dimensional matrix.

This function starts by finding the start and end position of the matrix by calling the function “findStartAndEnd” Function. After that, initiate a “node\_expanded” as zero, “max\_depth” has zero, and max\_fringe has zero and initializes the heap (priority queue) data structure. With a tuple containing the start node and the list containing the start node, and initializes set named has “visitedPath” to keep the track of visited nodes.

The cost of the path is sum of the length of the path and the Manhattan distance from the current node to the end node

The function then enters the loop that continues While the heap is not empty, end goal is found, In this iteration of the loop. It pops lowest cost and path from heap and increment the node expanded value of by 1. And also updated the maximum depth and the maximum fringe by keep track in the maximum death and the size of the freeze. The function then calls the draw function to draw the path in pygame console using the colour red. if the popped node is equals to end node, it returns the path and path cost, number of nodes expanded, and the maximum depth of the tree, and the maximum size of the films. If the popped node is not in the visited set, it adds in the visited set and generate list of unvisited neighbours from the popped node using the ‘neighbours’ function. It then appends each unvisited neighbour and its corresponding path and cost to the heap. Finally, the goal is not found, and the stack is empty. The function returns None for all output parameters.

**draw(path, color)**

The function draw is used to visualize the search path on the maze it takes 2 arguments, path and color. The path is the argument that consists of the nodes representing the path to be visualized and color to be drawn.

The function iterates through each node in the path and draws a rectangle on the screen. Using the given color, the position and the size are determined for the nodes position in the maze and the cell variable, which is representing the size of the node After drawing each rectangle, the function updates the display using “pygame.diplay.flip()” and waits for the short amount of time using the “clock.tick(120)”

**main()**

The function “main “initialize the python library creates the game window and sets the dimension of the window, and also display the options for the users to choose from 0 to 3 the user display determines which of algorithm to be run if the user chooses “0” the original matrix is displayed on the screen. If The user chooses option “1”. The DFS algorithm will run and the matrix will be displayed and path will be printed on the console, and the path generated by the algorithm is given on the screen in the colour of green. The “pygame.dispaly.update()” after the algorithm is run. And the game windows remain open until the user closes.

**PYTHON SOURCE CODE:**

from collections import deque

import heapq

import sys

import pygame

red=(255,0,0)

green =(0,255,0)

screen=''

cell=''

BREATH=''

LENGTH=''

clock=''

print("Enter Input of LayOut File Choice: ")

print("0 for bigMaze")

print("1 for mediunMaze ")

print("2 for smallMaze")

print("3 for openMaze")

layChoice = int(input())

#    all .lay files are stored in the layOutFiles

layOutFiles = ['bigMaze.lay','mediumMaze.lay','smallMaze.lay','openMaze.lay']

# layToMatrix function uses to convert .lay files to matrix

def layToMatrix(path):

    with open(path, 'r') as f:

        maze =  [list(line.strip()) for line in f.readlines()]

    return maze

mazeMat=layToMatrix(layOutFiles[layChoice])

# findStartAndEnd function helps to find the starting "PACMAN" position and ending position

def findStartAndEnd(mat):

    for i in range(len(mat)):

        for j in  range(len(mat[i])):

            if mat[i][j]=='P':

                start = (i,j)

            elif mat[i][j]=='.':

                end = (i,j)

    return start,end

# direction consists of direction of node to find neighbours

direction = [(0, 1), (0, -1), (-1, 0), (1, 0)]

#neighbours function helps to visit neighbour node i.e up ,down, left, right

def neighbours(mazeMat,node):

    neighbors = []

    for x,y in direction:

        xx=node[0]+x

        yy=node[1]+y

        if 0 <= xx < len(mazeMat) and 0 <= yy < len(mazeMat[0]) and mazeMat[xx][yy] != '%':

            neighbors.append((xx, yy))

    return neighbors

# findPathUsingDFS helps to find path using depth first search Algo

def findPathUsingDFS(mazeMat):

    node\_expanded=0

    max\_depth=0

    max\_fringe=0

    start,end = findStartAndEnd(mazeMat)

    stack = [(start,[start])]

    vistedPath = set()

    while stack:

        node, path = stack.pop()

        node\_expanded+=1

        max\_depth=max(max\_depth,len(path))

        max\_fringe=max(max\_fringe,len(stack))

        draw(path,red)

        pygame.display.flip()

        #print(node,path)

        if node ==end:

            return path,len(path)-1,node\_expanded,max\_depth,max\_fringe

        if node not in vistedPath:

            vistedPath.add(node)

            for \_ in neighbours(mazeMat , node):

                if \_ not in vistedPath:

                    stack.append((\_ , path+[\_]))

    return None,None, None,None,None

# findPathUsingBFS helps to find path using breath first search Algo

def findPathUsingBFS(mazeMat):

    node\_expanded=0

    max\_depth=0

    max\_fringe=0

    start,end = findStartAndEnd(mazeMat)

    q = deque([(start,[start])])

    vistedSet = set()

    while q:

        node ,path = q.popleft()

        node\_expanded+=1

        max\_depth=max(max\_depth,len(path))

        max\_fringe=max(max\_fringe,len(q))

        draw(path,red)

        if node == end:

            return path , len(path)-1,node\_expanded,max\_depth,max\_fringe

        if node not in vistedSet:

            vistedSet.add(node)

            for \_ in neighbours(mazeMat,node):

                if \_ not in vistedSet:

                    q.append((\_ , path+[\_]))

    return None,None,None,None,None

# findPathUsingAstar helps to find path using A\* Algo

def findPathUsingAStar(mazeMat):

    node\_expanded=0

    max\_depth=0

    max\_fringe=0

    start,end = findStartAndEnd(mazeMat)

    heap  = [(0,start,[start])]

    vistedSet = set()

    while heap:

        K, node, path = heapq.heappop(heap)

        node\_expanded+=1

        max\_depth=max(max\_depth,len(path))

        max\_fringe=max(max\_fringe,len(heap))

        draw(path,red)

        if node == end:

            return path , len(path)-1,node\_expanded,max\_depth,max\_fringe

        if node not in vistedSet:

            vistedSet.add(node)

            for \_ in neighbours(mazeMat,node):

                if \_ not in vistedSet:

                    cost  =len(path)+1

                    cost += abs(\_[0]-end[0])+abs(\_[1]-end[1])

                    heapq.heappush(heap,(cost, \_ ,path+[\_]))

    return None,None,None,None,None

# draw function helps in draw the path of visted nodes

def draw(path,color):

    global screen,cell,BREATH,LENGTH,clock

    for node in path:

        pygame.display.set\_caption("Maze Route")

        pygame.draw.rect(screen,color,(node[1] \*cell, node[0] \* cell, cell, cell),3)

        pygame.display.flip()

        if layChoice==3:

            clock.tick(600)

        else:

            clock.tick(90)

# main function helps to run the program using the input given by the user this

# function creates a pygame frame to visilatize the matrix if the

# choice is 0 to show original matrix

# choice is 1 dfs Algo runs

# choice is 2  bfs Algo runs

# choice is 3 a\* Algo runs

def main():

    global screen,cell,BREATH,LENGTH,clock

    print("Enter Input of Choice: ")

    print("0 for original matrix")

    print("1 for DFS Algo path ")

    print("2 for BFS Algo path")

    print("3 for A\* Algo path")

    choice =int(input())

    pygame.init()

    cell=20

    BREATH=len(mazeMat)\*cell

    LENGTH=len(mazeMat[0])\*cell

    screen = pygame.display.set\_mode((LENGTH, BREATH),pygame.RESIZABLE)

    clock = pygame.time.Clock()

    flag=True

    pygame.display.set\_caption("Maze Route")

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                sys.exit()

        if flag:

            for i in range(len(mazeMat)):

                for j in range(len(mazeMat[0])):

                    if mazeMat[i][j]=='%':

                        pygame.draw.rect(screen,(0,0,0),(j \*cell, i \* cell, cell, cell))

                    elif mazeMat[i][j]=='P':

                        pygame.draw.rect(screen,(255,0,0),(j \*cell, i \* cell, cell, cell))

                    elif mazeMat[i][j]=='.':

                        pygame.draw.rect(screen,(0,255,0),(j \*cell, i \* cell, cell, cell))

                    else:

                        pygame.draw.rect(screen,(112,128,144),(j \*cell, i \* cell, cell, cell))

        if choice==0:

            pygame.display.flip()

        if choice == 1 :

            dfs\_path,dfs\_cost,nodeExpanded,maxDepth,maxFringe = findPathUsingDFS(mazeMat)

            print("DFS path:", dfs\_path)

            print("DFS cost:", dfs\_cost)

            print("DFS nodeExpanded:", nodeExpanded)

            print("DFS maxDepth:", maxDepth)

            print("DFS maxFringe:", maxFringe)

            draw(dfs\_path,green)

            pygame.display.flip()

            choice=0

        if choice == 2 :

            bfs\_path,bfs\_cost,nodeExpanded,maxDepth,maxFringe  = findPathUsingBFS(mazeMat)

            print("BFS path:", bfs\_path)

            print("BFS cost:", bfs\_cost)

            print("BFS nodeExpanded:", nodeExpanded)

            print("BFS maxDepth:", maxDepth)

            print("BFS maxFringe:", maxFringe)

            draw(bfs\_path,green)

            pygame.display.flip()

            choice=0

        if choice == 3 :

            Astar\_path,Astar\_cost,nodeExpanded,maxDepth,maxFringe  = findPathUsingAStar(mazeMat)

            print("A\* path:", Astar\_path)

            print("A\* cost:", Astar\_cost)

            print("A\* nodeExpanded:", nodeExpanded)

            print("A\* maxDepth:", maxDepth)

            print("A\* maxFringe:", maxFringe)

            draw(Astar\_path,green)

            pygame.display.flip()

            choice=0

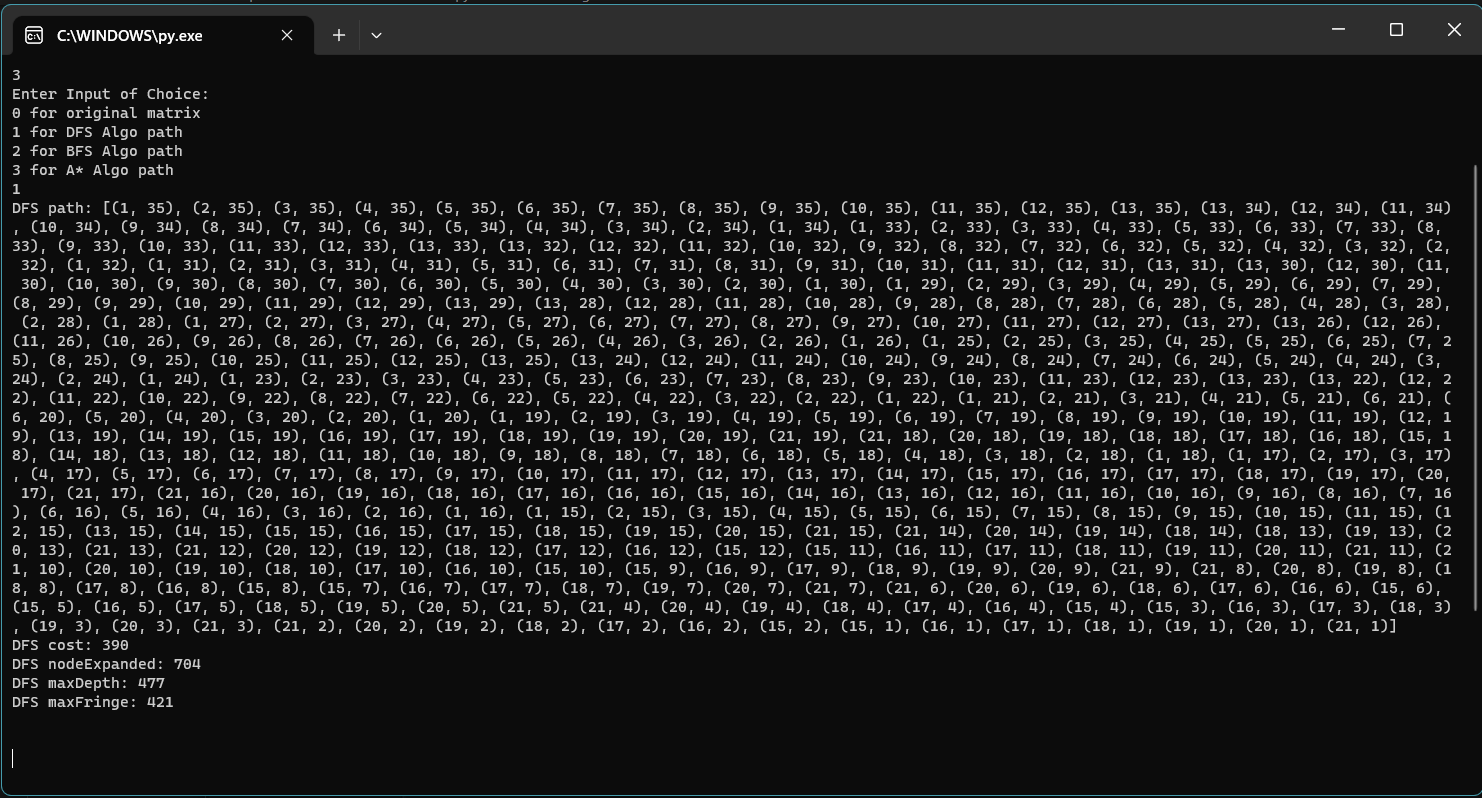
        pygame.display.flip()

        flag=False

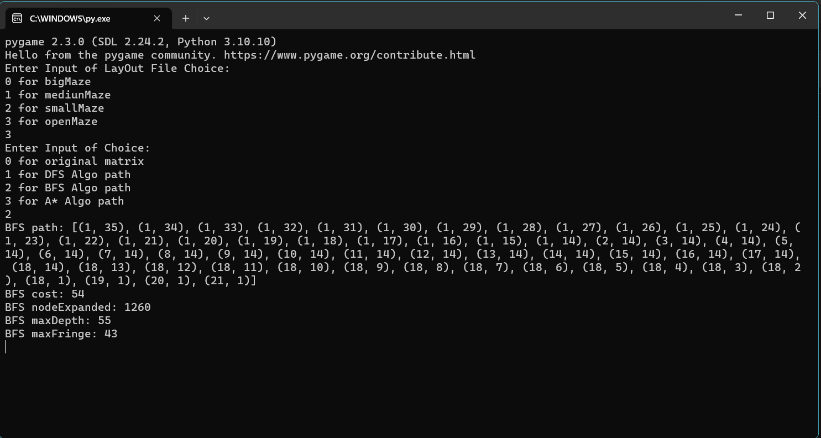
main()

**Results**

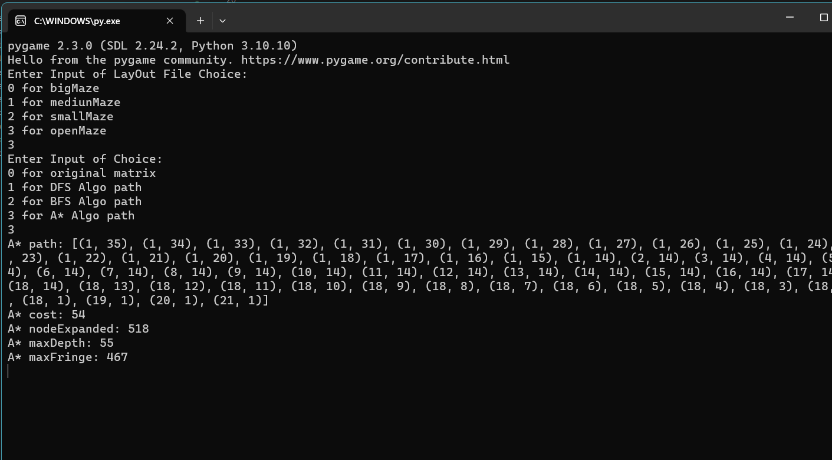
DFS ON OPENMAZE:

Console Path:

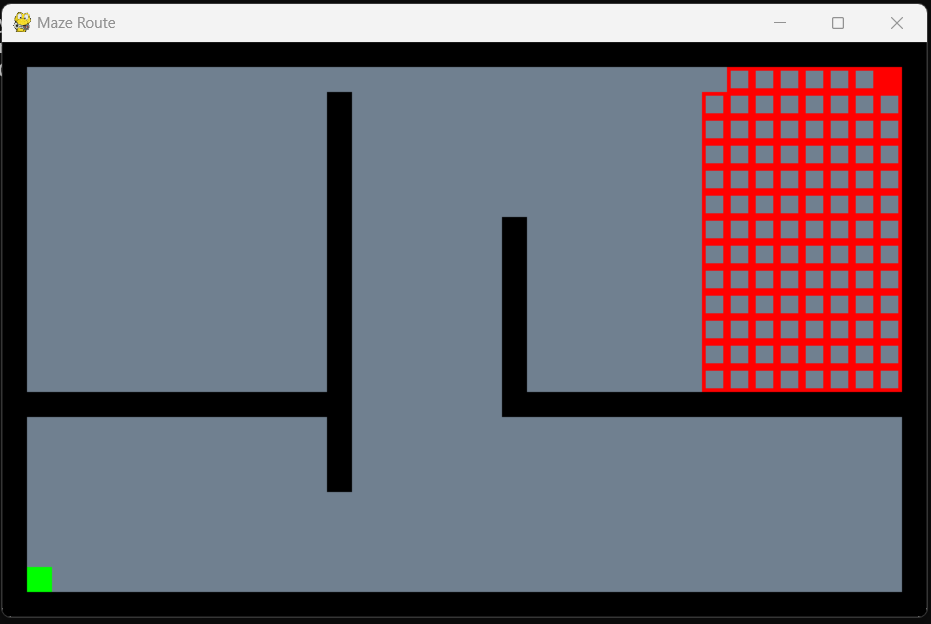
BFS ON OPENMAZE:

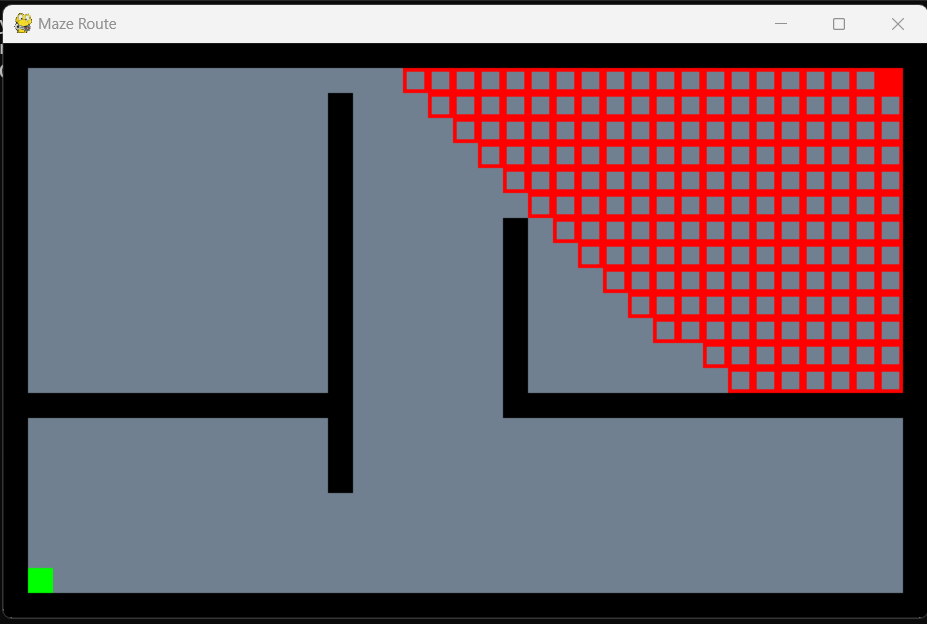
Console Path:

A\* ON OPENMAZE:

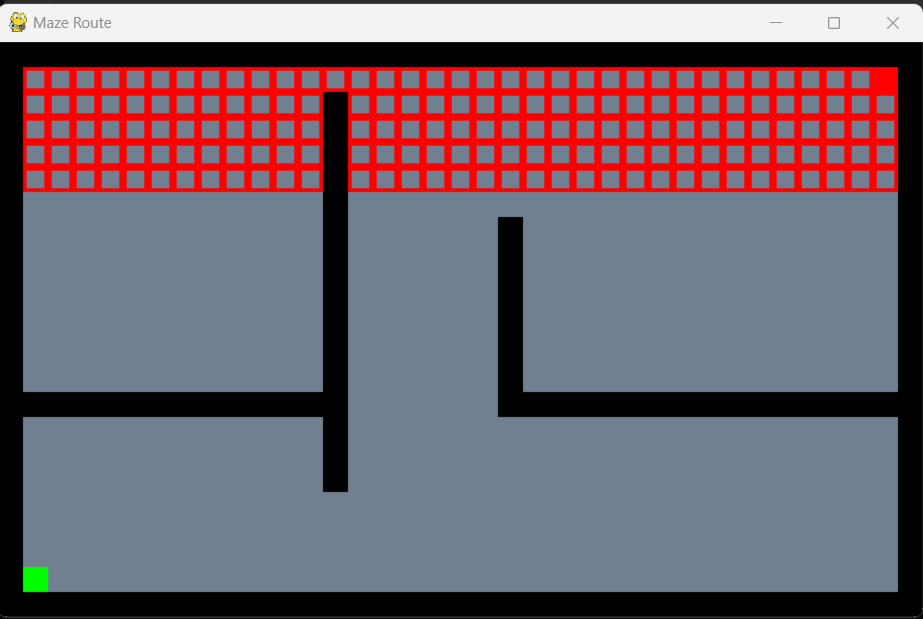
Console Path:

VISUALIZATION ON OPENMAZE( DFS ):

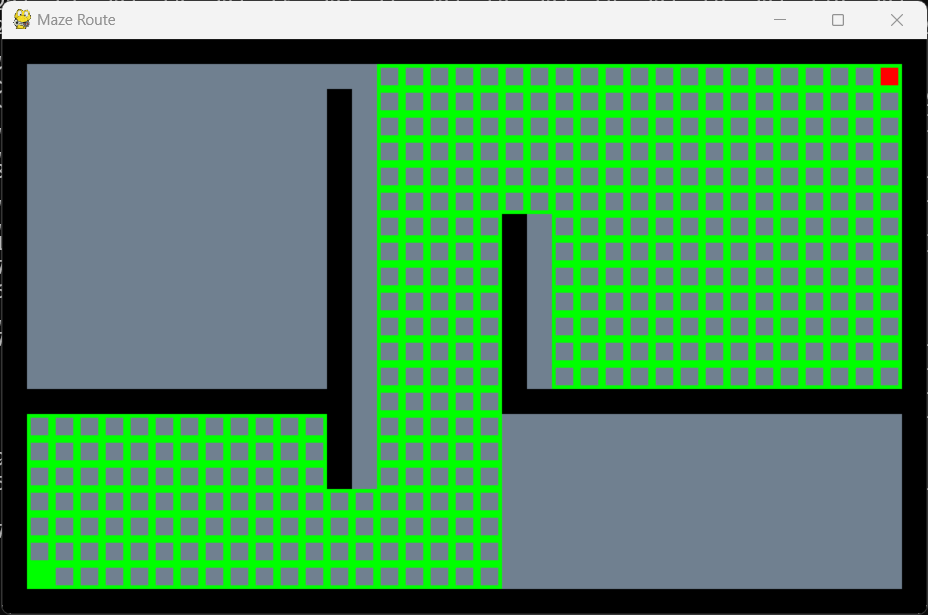


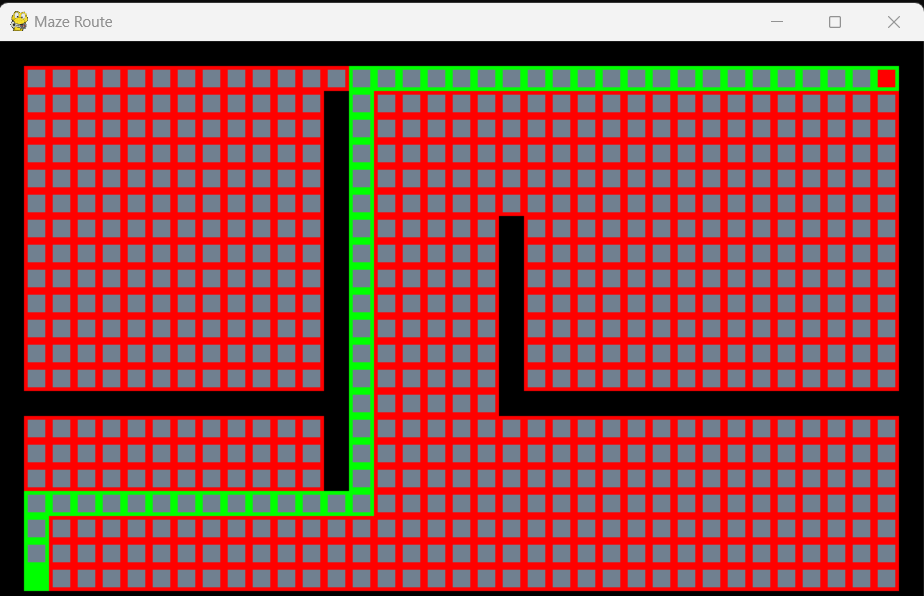
VISUALIZATION ON OPENMAZE(BFS):

VISUALIZATION ON OPENMAZE( A\* ):

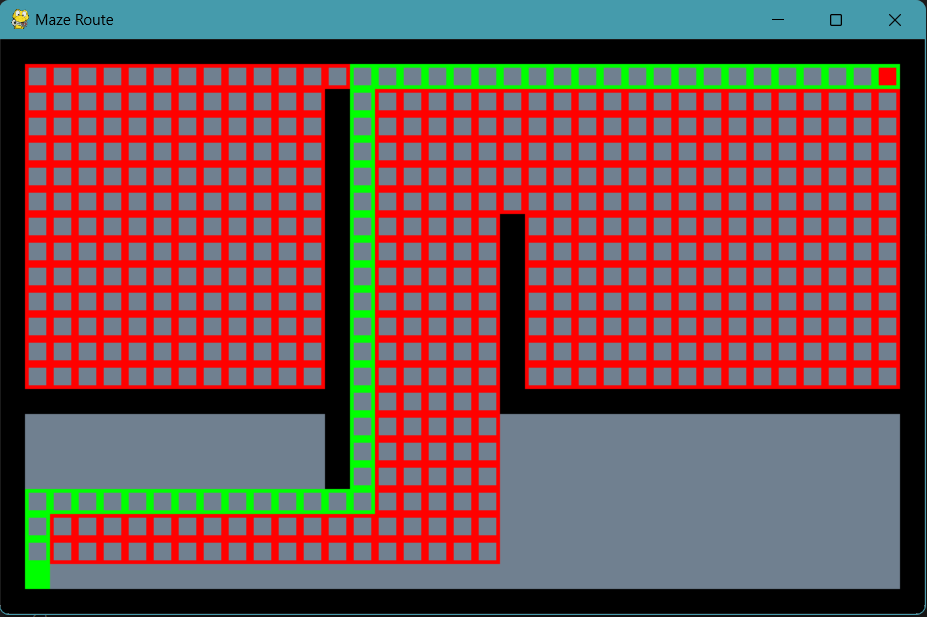


FINALPATH ON OPENMAZE(DFS):

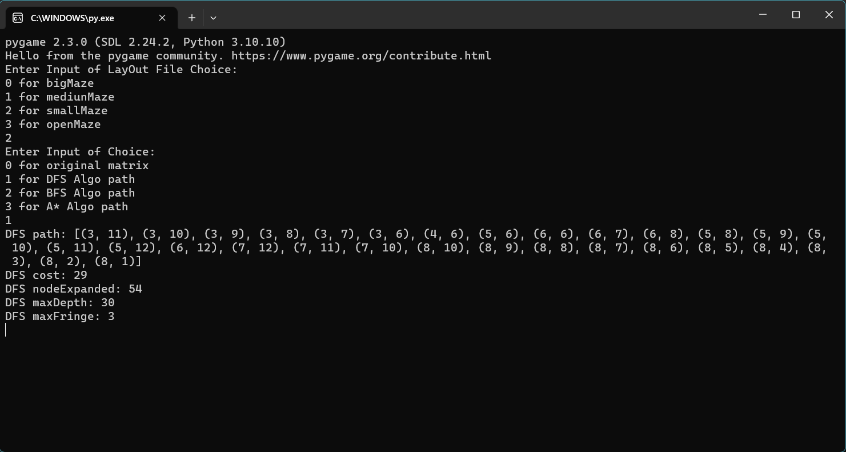


FINALPATH ON OPENMAZE(BFS):

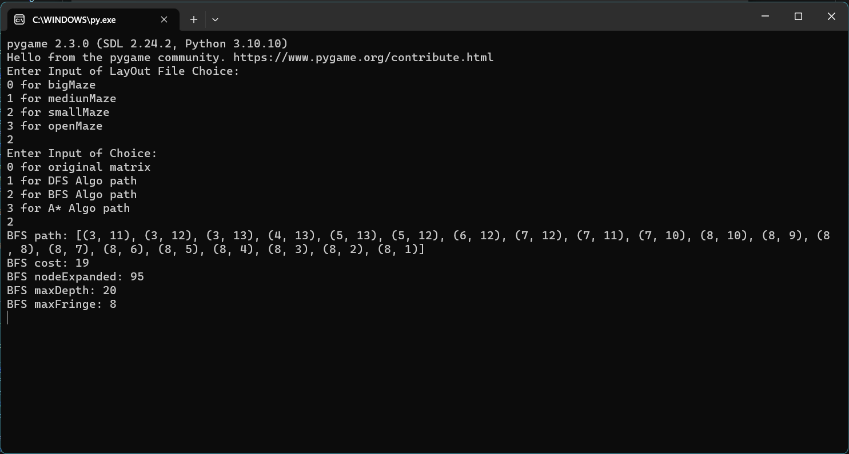
FINALPATH ON OPENMAZE(A\*):



DFS ON SMALLMAZE:

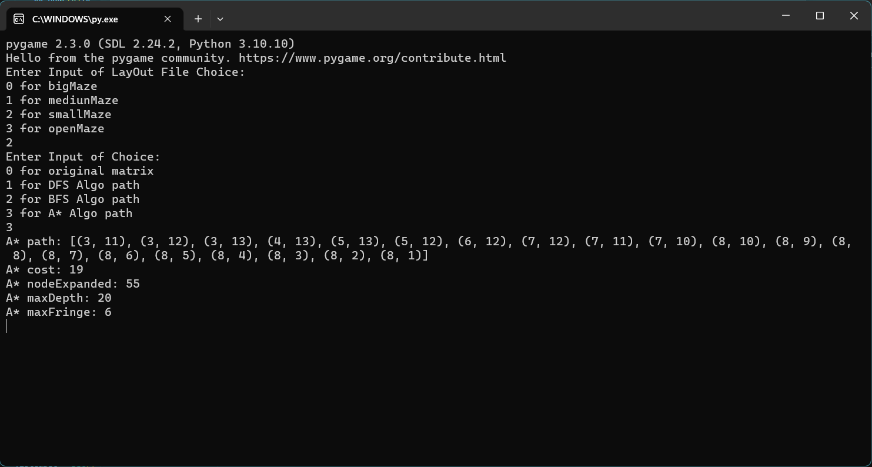
Console Path:

BFS ON SMALLMAZE:

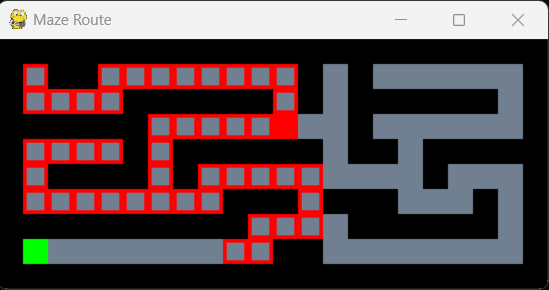
Console Path:

A\* ON SMALLMAZE:

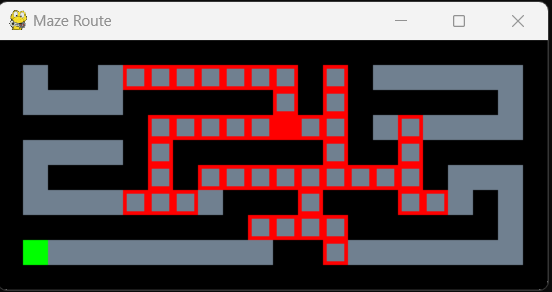
Console Path:



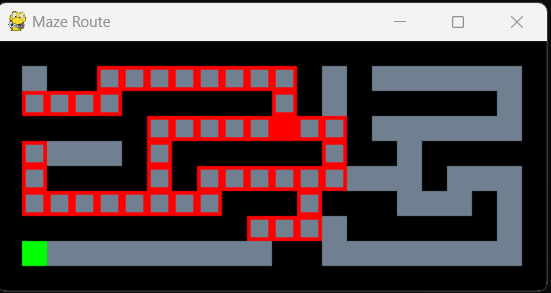
VISUALIZATION ON SMALLMAZE(DFS):



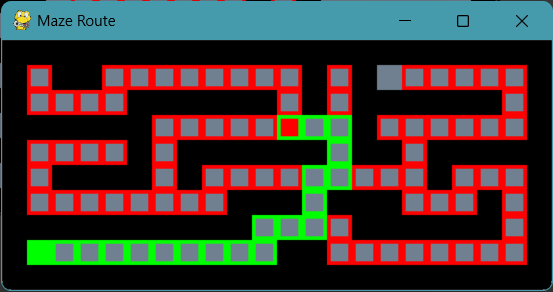
VISUALIZATION ON SMALLMAZE(BFS):



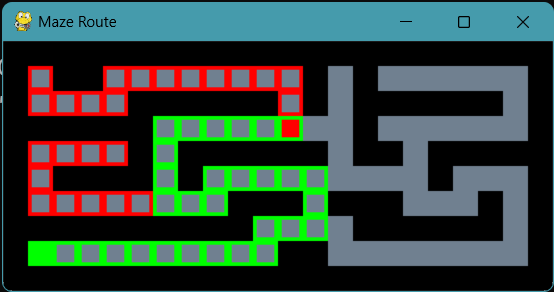
VISUALIZATION ON SMALLMAZE(A\*):



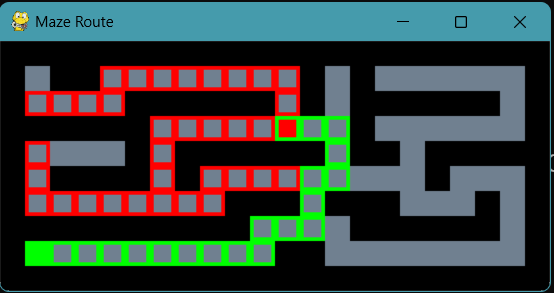
FINALPATH ON SMALLMAZE (DFS):



FINALPATH ON SMALLMAZE (DFS):

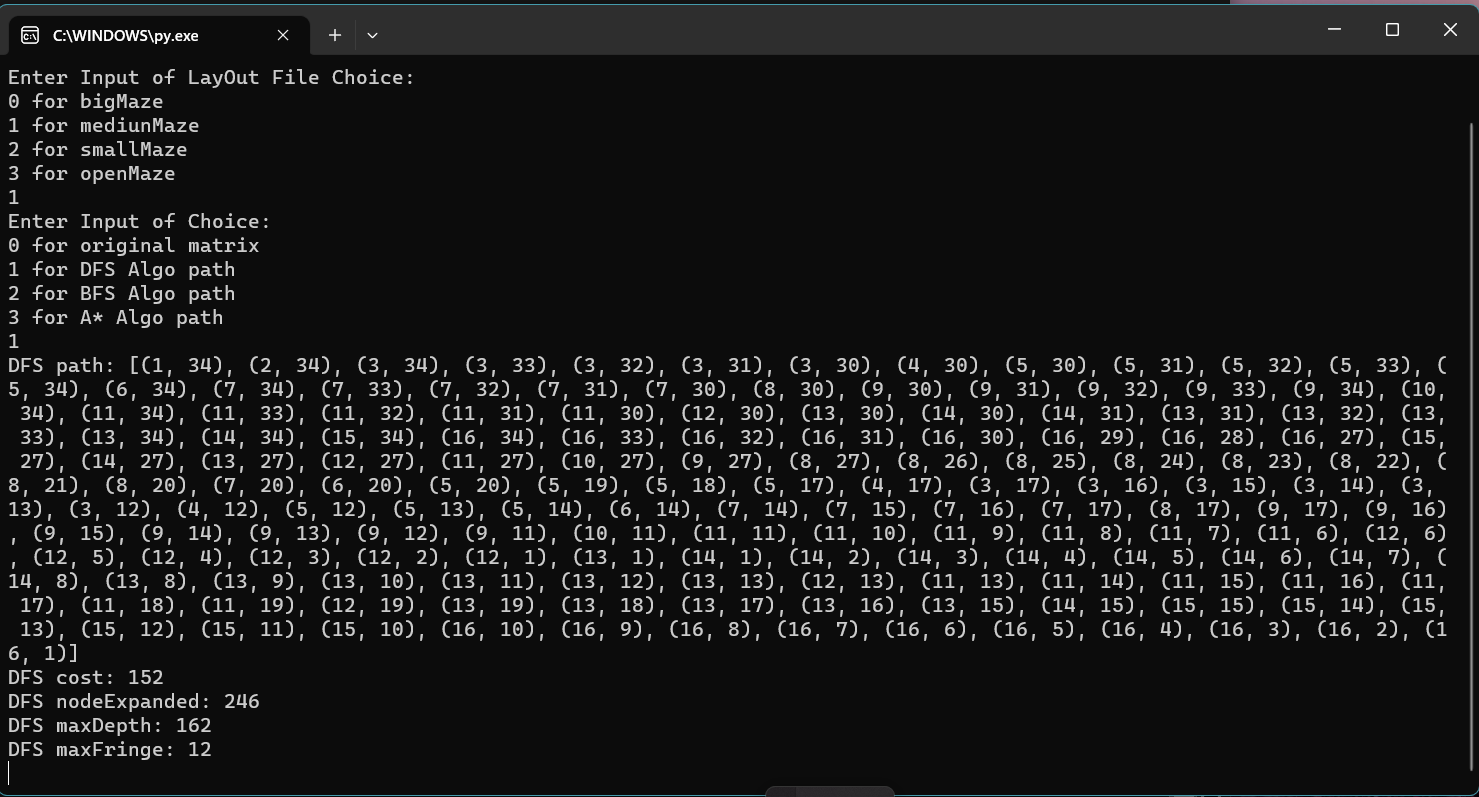


FINALPATH ON SMALLMAZE (DFS):

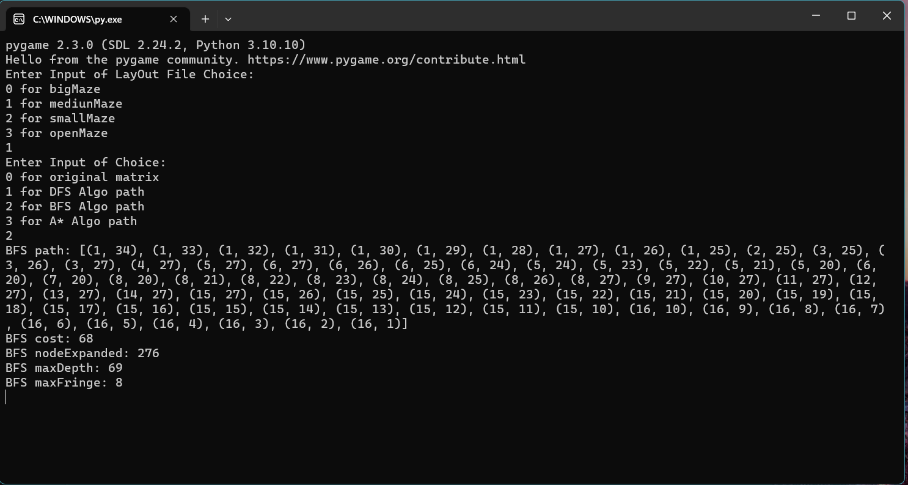


DFS ON MEDIUMMAZE:

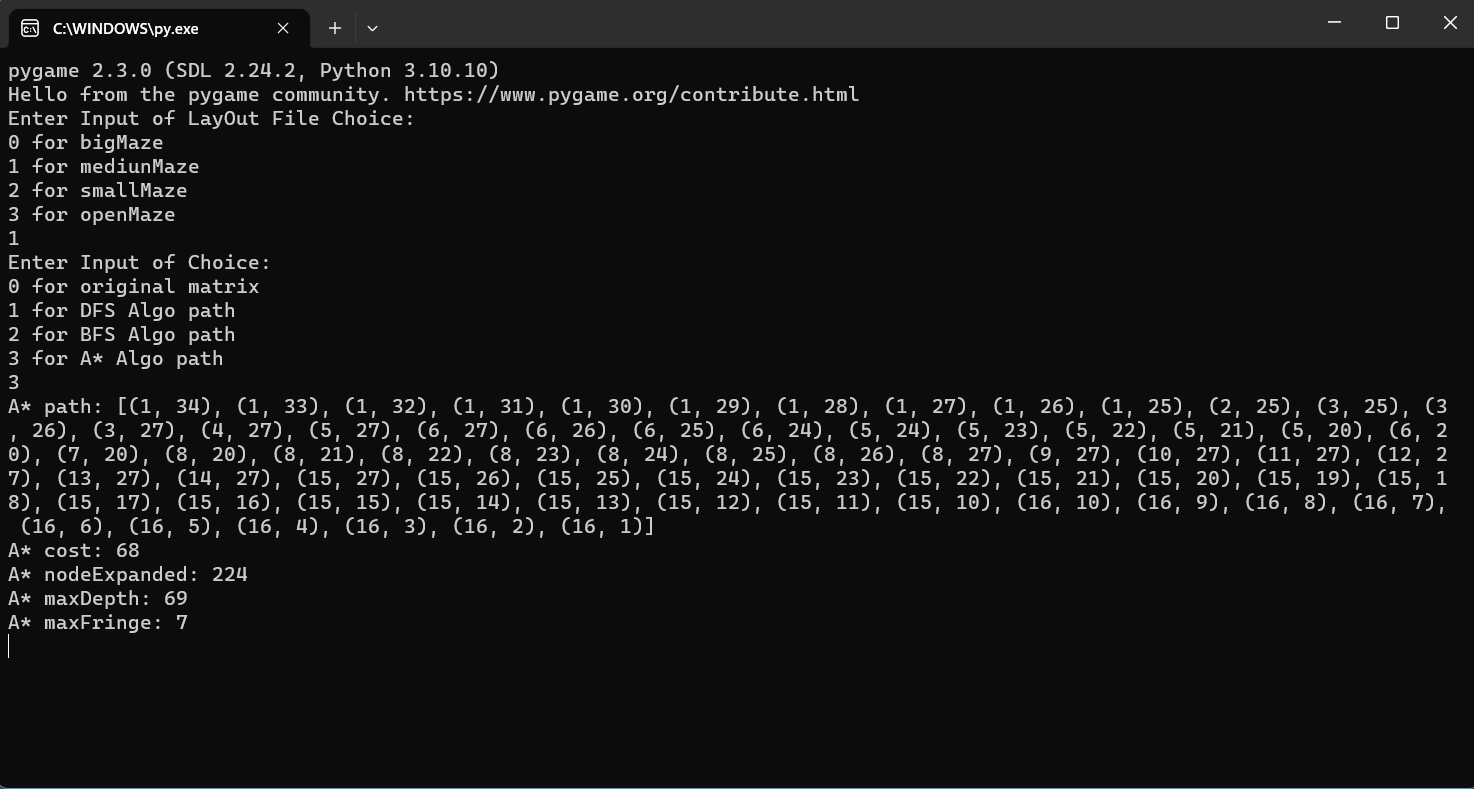
Console Path:

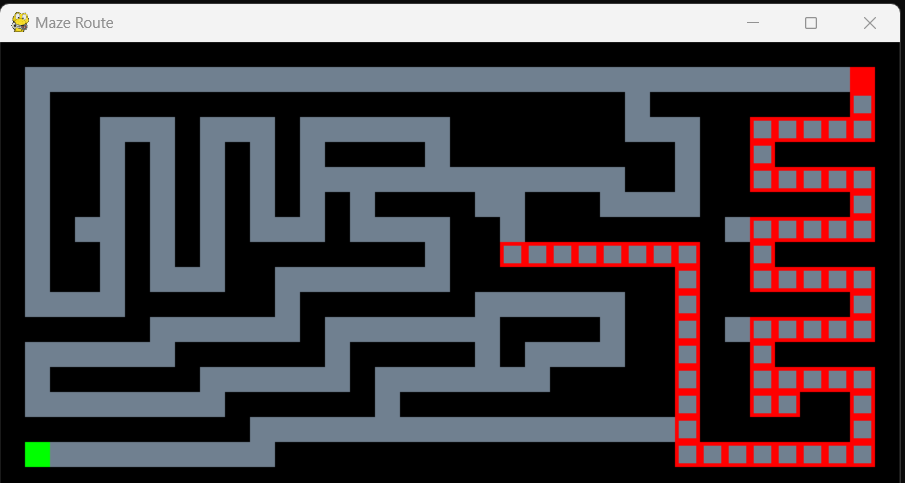


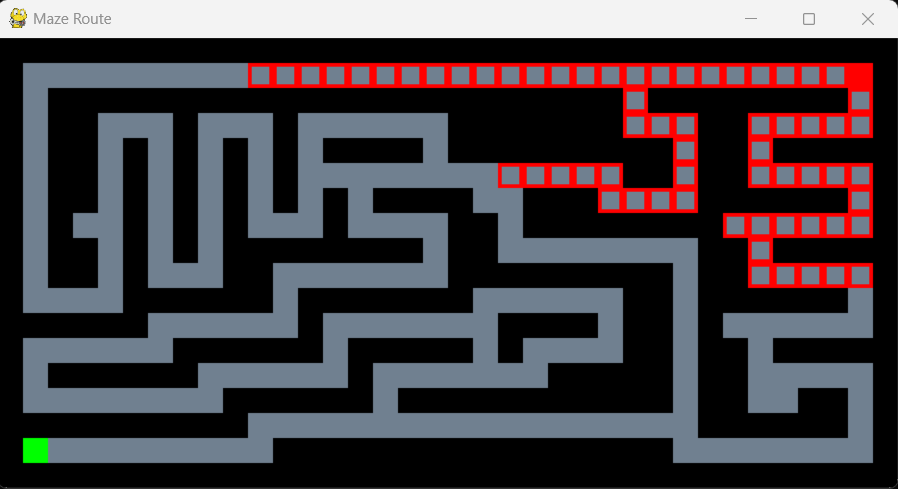
BFS ON MEDIUMMAZE:

Console Path:

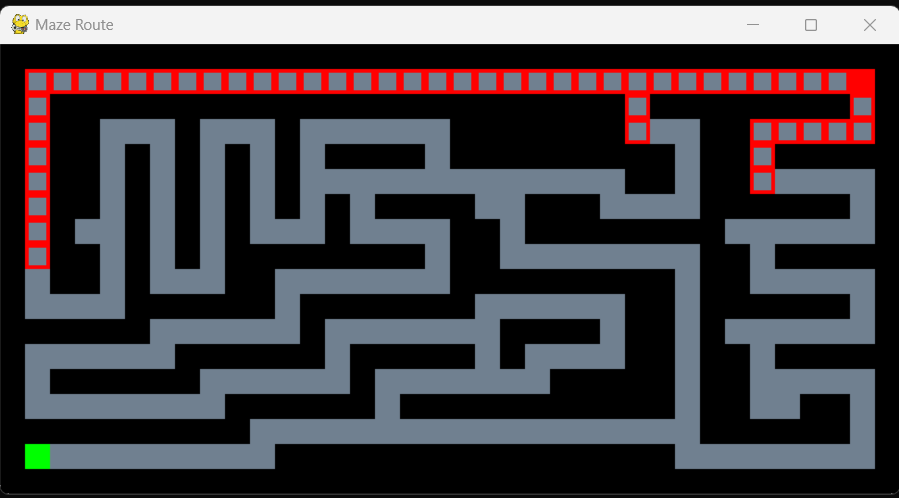
A\* ON MEDIUMMAZE:

Console Path:

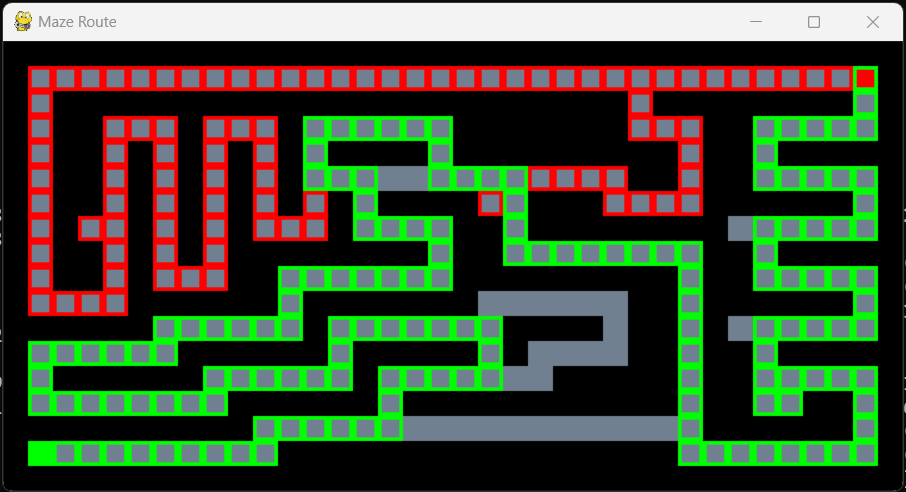
VISUALIZATION ON MEDIUMMAZE(DFS):

VISUALIZATION ON MEDIUMMAZE(BFS):

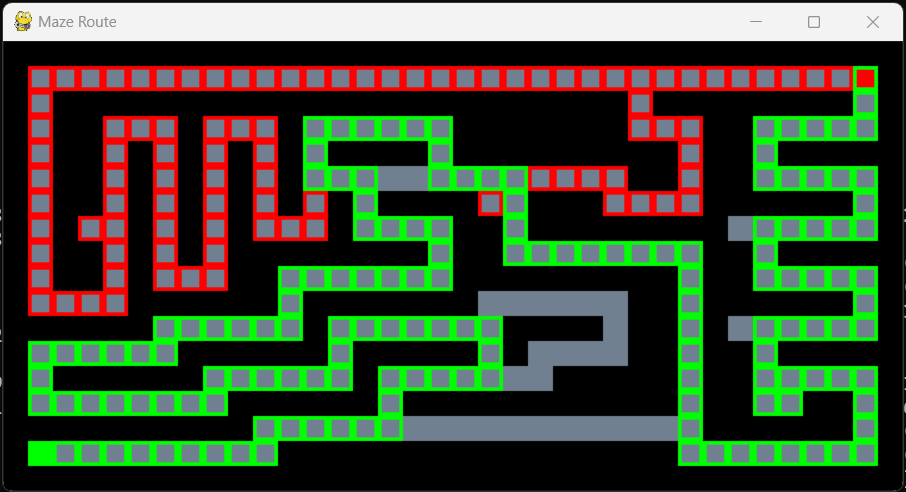
VISUALIZATION ON MEDIUMMAZE(A\*):



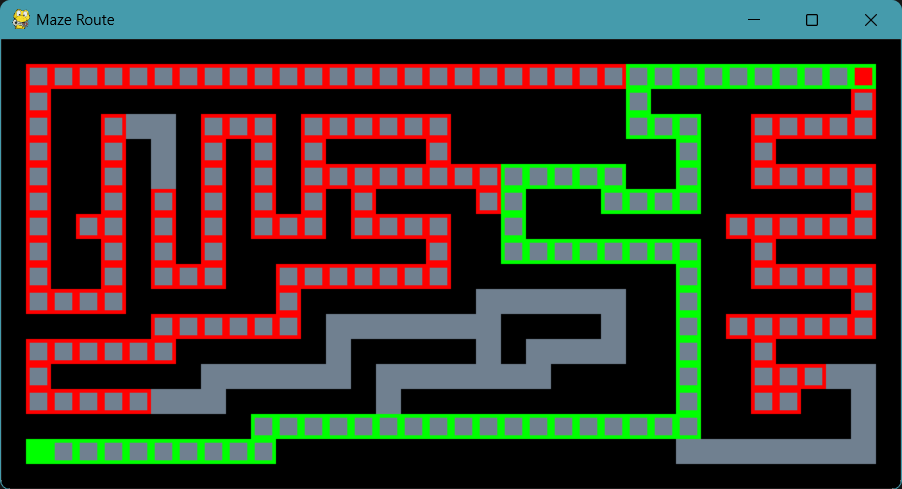
FINALPATH ON MEDIUMMAZE (DFS):



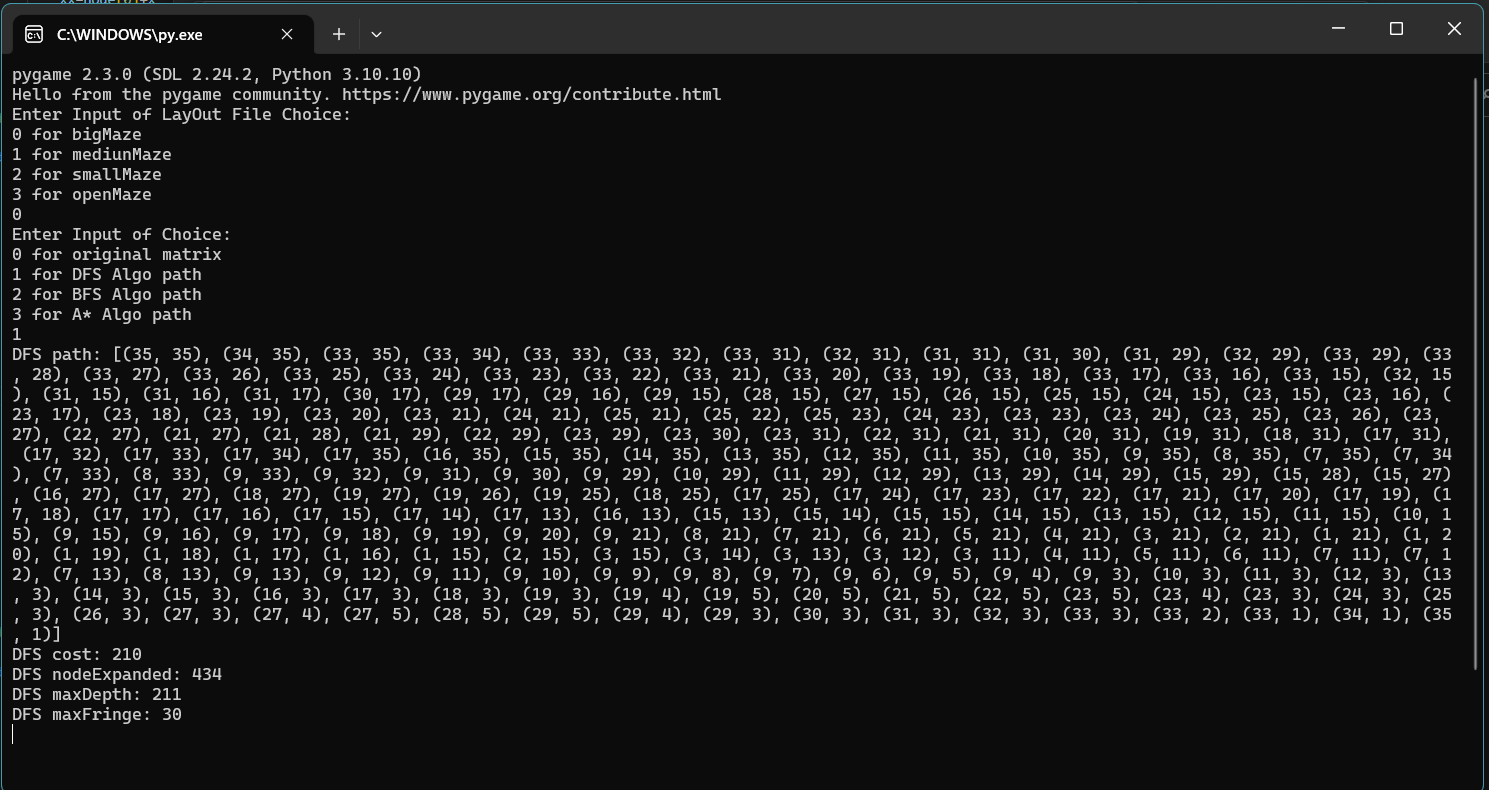
FINALPATH ON MEDIUMMAZE (BFS):



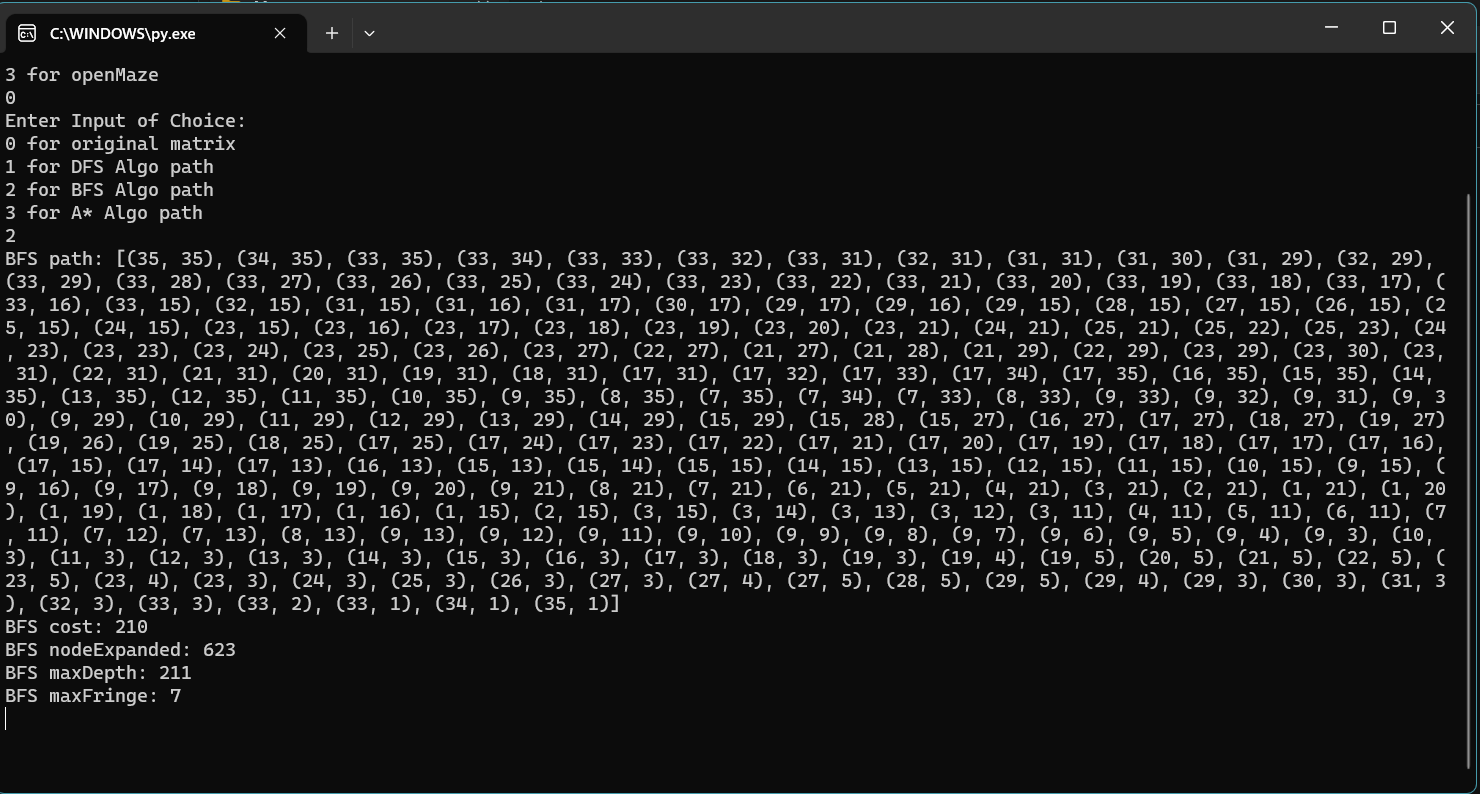
FINALPATH ON MEDIUMMAZE (A\*):



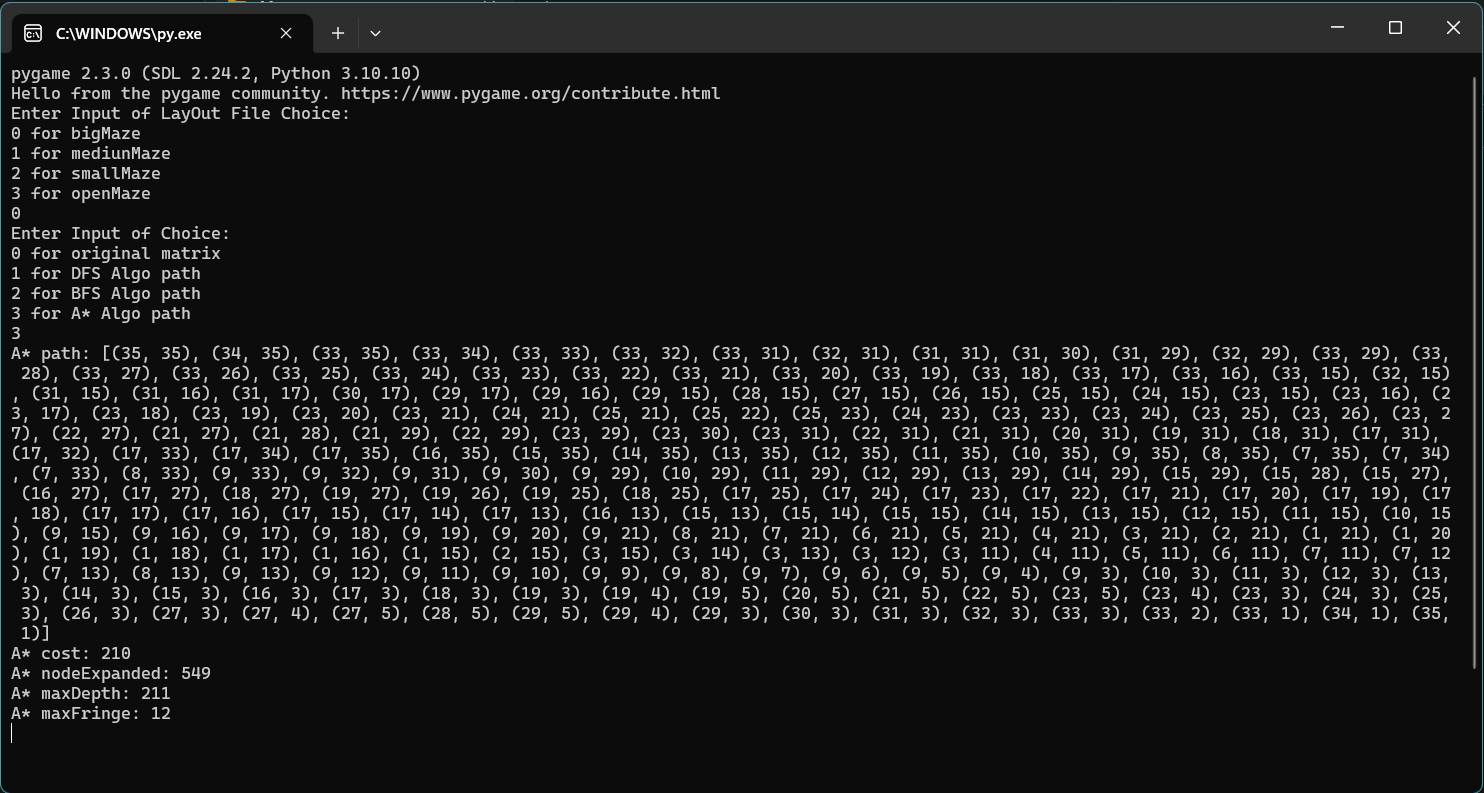
DFS ON BIGMAZE:

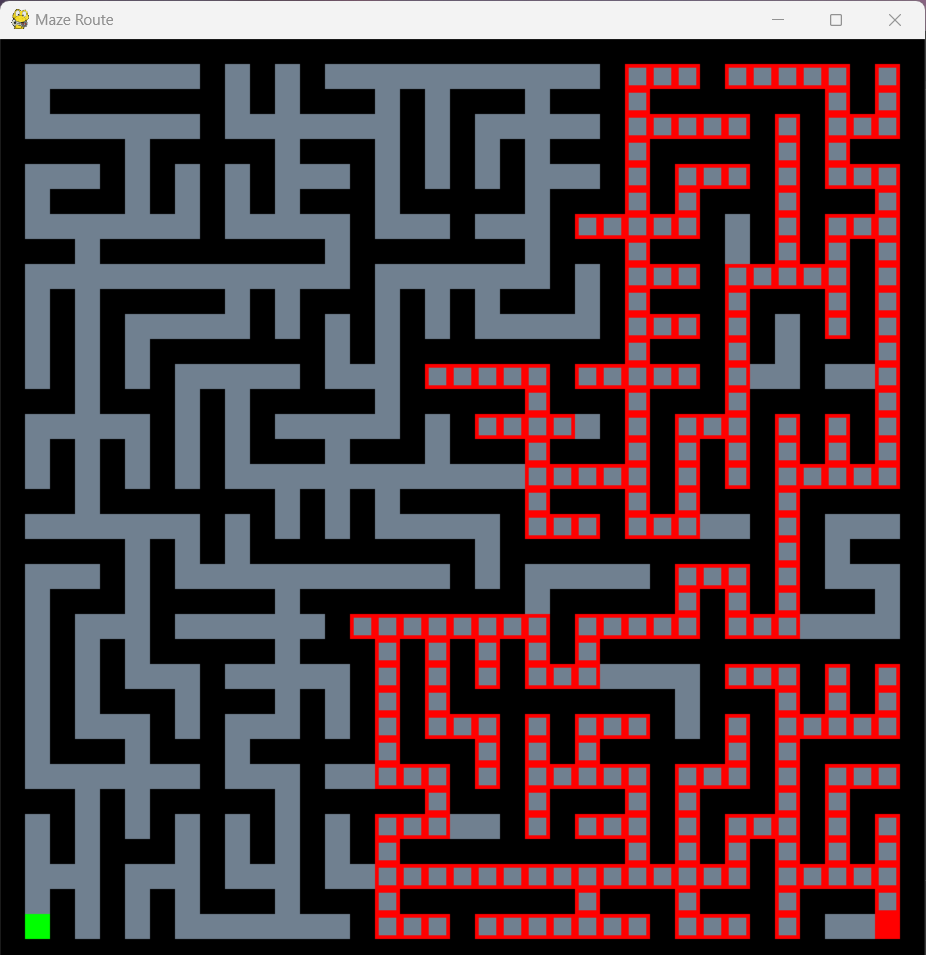
Console Path:

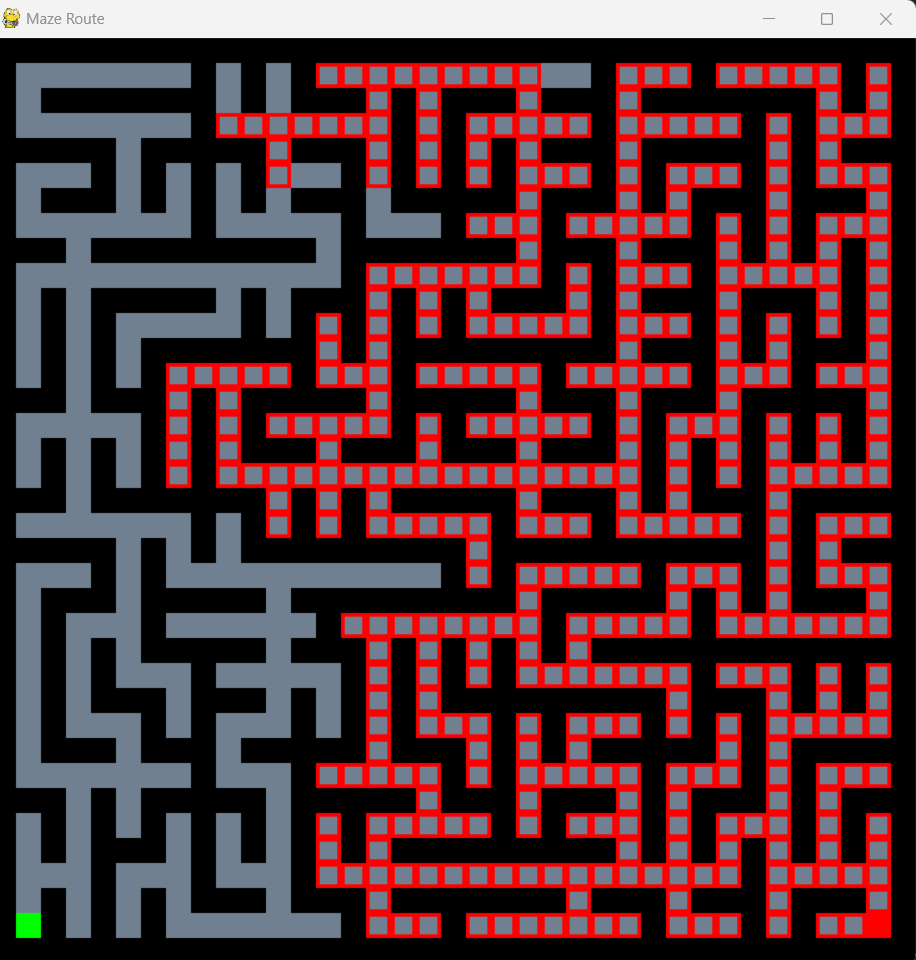
BFS ON BIGMAZE:

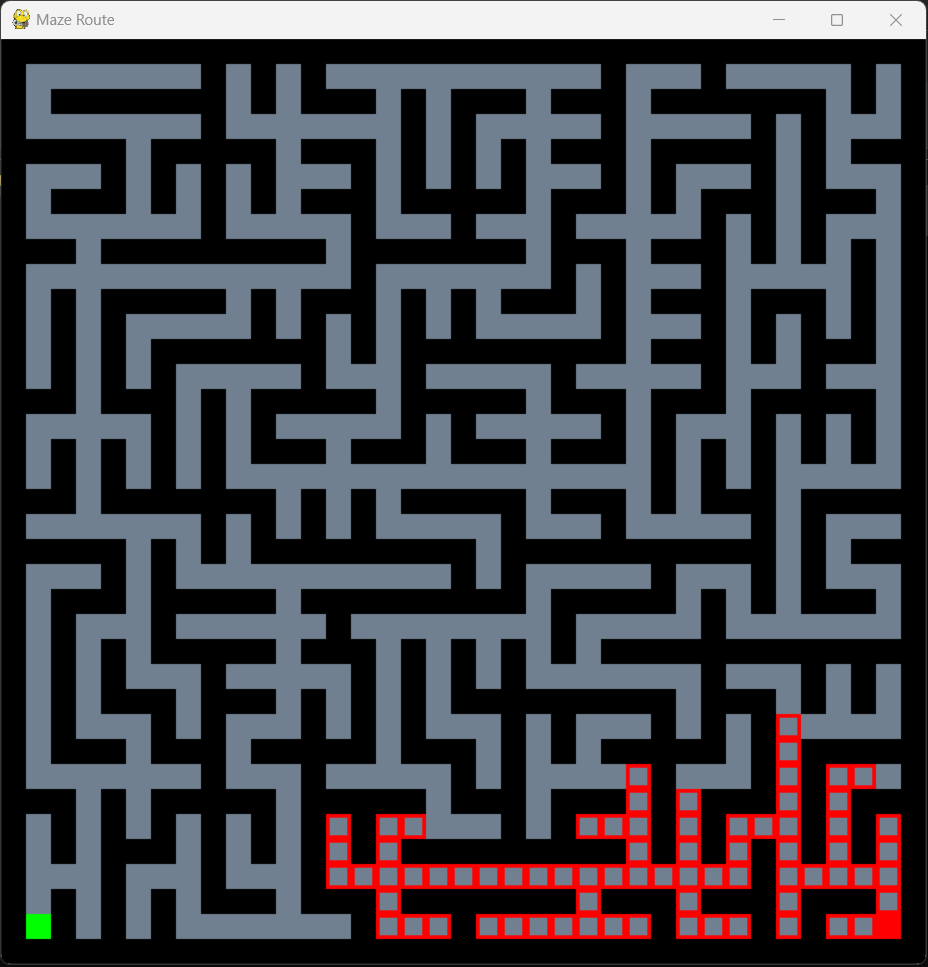
Console Path:

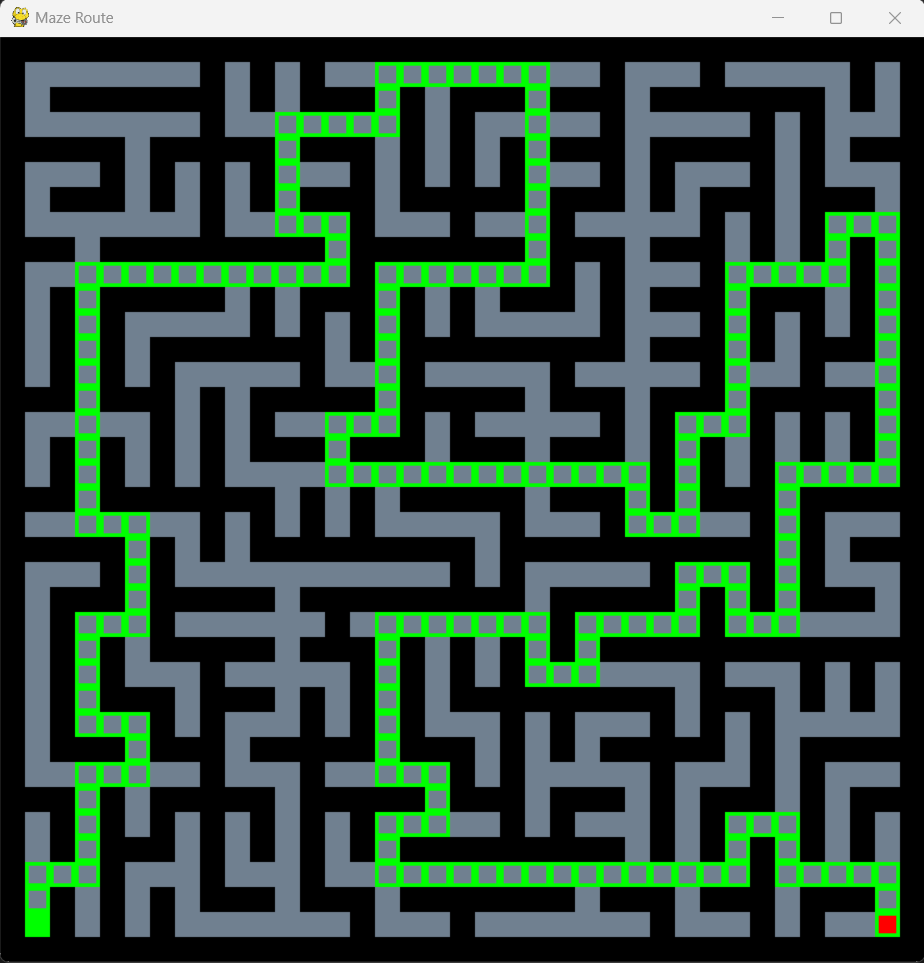
A\* ON BIGMAZE:

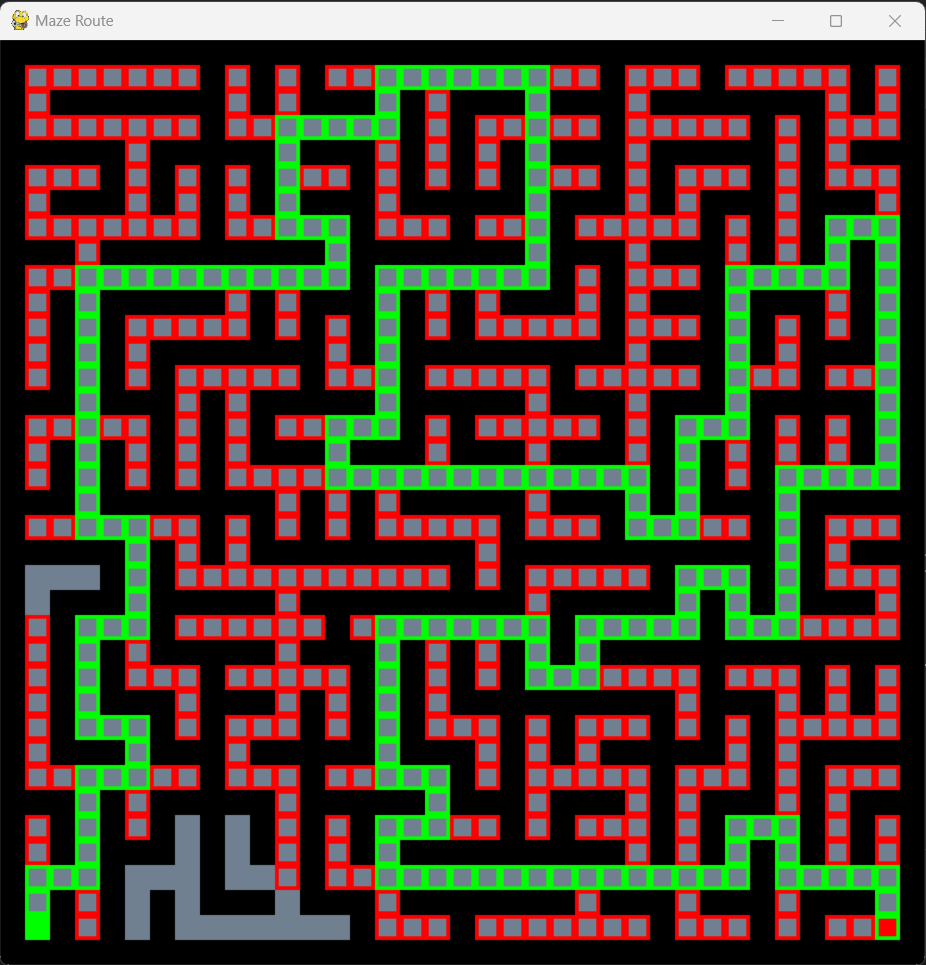
Console Path:

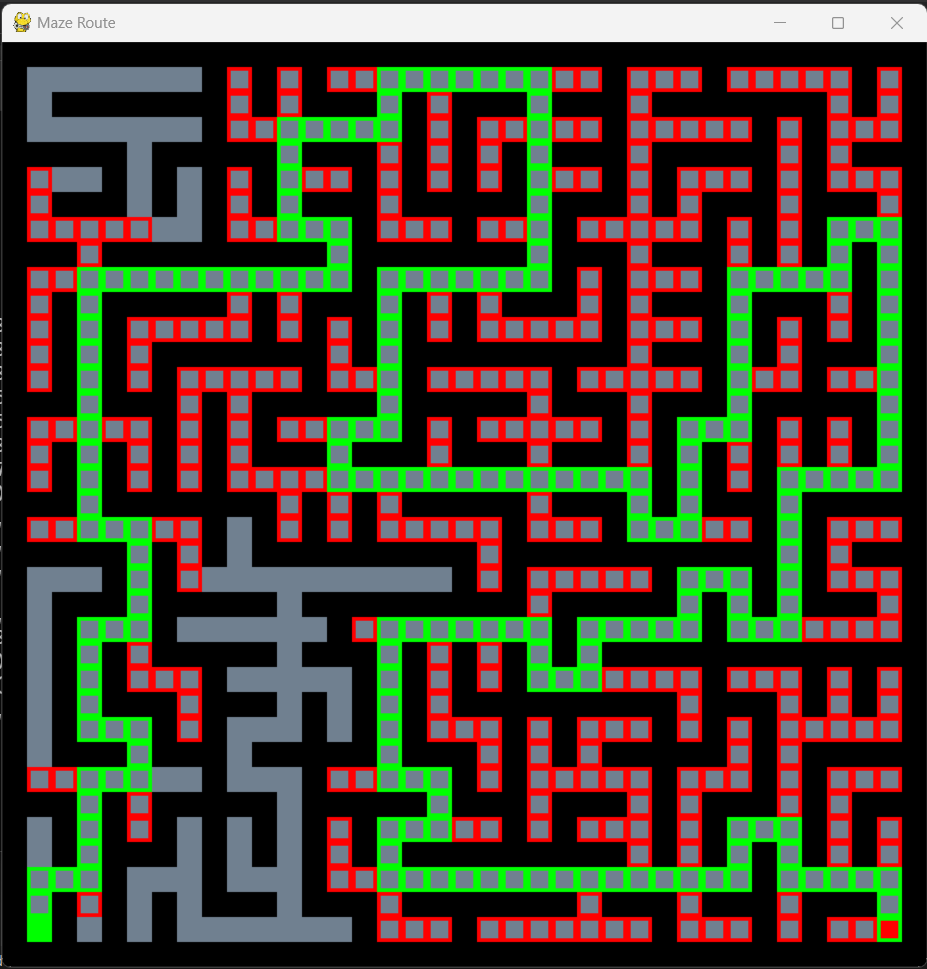
VISUALIZATION ON BIGMAZE (DFS):

VISUALIZATION ON BIGMAZE (BFS):

VISUALIZATION ON BIGMAZE (A\*):

FINALPATH ON BIGMAZE (DFS):

FINALPATH ON BIGMAZE (BFS):

FINALPATH ON BIGMAZE (A\*):

**RESULT TABLE:**

Open Maze:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Open Maze | Path Cost | No. of Nodes Expanded | Max Depth of Tree | Max Fringe |
| DepthFirstSearch | 390 | 784 | 477 | 421 |
| BreadthFirstSearch | 54 | 1260 | 55 | 43 |
| A\* | 54 | 518 | 55 | 467 |

Small Maze:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Small Maze | Path Cost | No. of Nodes Expanded | Max Depth of Tree | Max Fringe |
| DepthFirstSearch | 29 | 54 | 30 | 3 |
| BreadthFirstSearch | 19 | 95 | 20 | 8 |
| A\* | 19 | 55 | 20 | 6 |

Medium Maze:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Medium Maze | Path Cost | No. of Nodes Expanded | Max Depth of Tree | Max Fringe |
| DepthFirstSearch | 152 | 246 | 162 | 12 |
| BreadthFirstSearch | 68 | 276 | 69 | 8 |
| A\* | 38 | 224 | 69 | 7 |

Big Maze:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Big Maze | Path Cost | No. of Nodes Expanded | Max Depth of Tree | Max Fringe |
| DepthFirstSearch | 210 | 434 | 211 | 30 |
| BreadthFirstSearch | 210 | 623 | 211 | 7 |
| A\* | 210 | 549 | 211 | 12 |

**CONCLUSION:**

Based on the result of solving Mazes with algorithms on different sizes, we can conclude the following:

1. Depth first search (DFS) algorithm provides longer path solutions compared to BFS and may stuck
2. Breadth first search (BFS) algorithm provides the shortest path solutions with least path cost in all maze sizes.
3. Breath first such algorithm explored a greater number of nodes than DFS and A\* and also provides the efficient solution.
4. A\* algorithm is a good balance between “DFS” and “BFS”, as it provides the shortest part solution with a low past cost and low number of nodes expanded Compared to “DFS” and ”BFS”.
5. Both BFS and A\* algorithms have similar “maximum depth of the tree” and “maximum fringe size”. From that we can conclude that both the solutions. Explore the maze in a more organized way compared to “DFS”.
6. BFS can be useful. In small maze, where the path cost is not critical and the number of nodes explored will be less.
7. The larger the size and the larger the criticality. It will be more advantage to BFS and A\* algorithm Because they provided more efficient way to explore the maze.

BFS algorithm, which generally the best option when finding the shortest path in the objective, while DFS can be used for smaller mazes, where the path cost is not critical, whereas A\* algorithm is a good choice between DFS and BFS, providing efficient solution with a low path cost.

**REFERENCES:**

DFS 🡪 <https://en.wikipedia.org/wiki/Depth-first_search>

BFS 🡪 <https://en.wikipedia.org/wiki/Breadth-first_search>

A\* 🡪 [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

Pygame 🡪 <https://www.pygame.org/docs/>

Heap 🡪 <https://en.wikipedia.org/wiki/Heap_(data_structure)>

Stack 🡪 <https://en.wikipedia.org/wiki/Stack_(abstract_data_type)>

Fringe 🡪 <https://en.wikipedia.org/wiki/Fringe_search>