Maze Path finding and Multi-Goal Search Project

Introduction

Navigating through complex mazes and finding optimal paths is a fundamental problem in artificial intelligence and robotics. In this project, we will be on a journey of exploration and problem-solving, diving into two essential aspects of path finding: basic path finding and search with multiple goals. Our objective is to develop and implement algorithms that can efficiently guide a character, represented by the iconic "Pac man," through a maze from a specified starting position to a designated goal state. To tackle this challenge, we will employ various search algorithms and adapt them for different scenarios.

Part 1: Basic Path finding

In the first part of our project, we concentrate on the problem of finding a path through a maze with a single goal, represented as a dot. The maze layout is provided in a simple text format, where '%' indicates walls, 'P' denotes the starting position, and '.' signifies the goal. All moves within the maze have equal step costs. We have the choice of employing two search algorithms: Depth-first search and Breadth-first search. Additionally, we will utilize the A* search algorithm for solving different mazes, where the Manhattan distance between the current position and the goal serves as the heuristic function.

Our task involves running each of these algorithms on various maze sizes, including small, medium, big, and open mazes. For each problem instance and each search algorithm, we will report the following key metrics:

a. The solution, graphically visualized, and its associated path cost.

b. The number of nodes expanded during the search.

c. The maximum tree depth explored.

d. The maximum size of the fringe.

Through these metrics, we aim to analyze and compare the efficiency and performance of different search algorithms in solving the basic path finding problem.

Part 2: Search with Multiple Goals

The second part of our project presents a more challenging problem: finding the shortest path through a maze while achieving multiple goals. In this scenario, the Pacman character aims to eat all the dots scattered throughout the maze. We will adapt our code from Part 1 to address this new challenge, requiring changes to the goal test and the state representation.

The modified code will then be subjected to the two search algorithms from Part 1: Depth-first search and A* search. We will evaluate their performance on different maze instances, including the tiny search, small search, and tricky search. The reported metrics will focus on solution cost and the number of nodes expanded.

To ensure efficiency in solving these complex problems, we may set an upper limit on the number of nodes to be expanded and terminate the search if this limit is reached without a solution. Additionally, we emphasize the importance of developing a strong heuristic. It is recommended to dedicate time to devising an effective heuristic that can significantly improve search efficiency. We encourage the exploration of multiple heuristics and a thorough analysis of their performance.

In summary, this project delves into the fascinating world of maze path finding, highlighting the effectiveness of different search algorithms and heuristics in solving both basic and multi-goal pathfinding scenarios. Through rigorous experimentation and analysis, we aim to gain insights into the efficiency and adaptability of these algorithms in navigating complex mazes.

Program Workflow

1. **User Selection:**

Upon launching the program, the user is prompted to select which part of the assignment to run: Part 1 (Basic Path finding) or Part 2 (Search with Multiple Goals).

2. **Maze Layout Selection:**

If Part 1 is chosen, the user is asked to select a maze layout file. This file contains the maze's layout with symbols such as '%' for walls, 'P' for the starting position, and '.' for the goal.

If Part 2 is chosen, the user can provide the maze layout with multiple goals.

3. **Algorithm Selection**:

After selecting the maze layout, the user is prompted to choose one of the available search algorithms: Depth-first search, Breadth-first search, or A* search.

4. **Pygame Initialization**:

The program initializes the Pygame environment to create a graphical representation of the maze and the chosen algorithm's execution.

5. **Visualization**:

The maze, the selected algorithm, and the character (Pacman) are displayed on the Pygame screen.

The program continuously updates the screen to show the algorithm's progress in real-time.

6. **Solution Display**:

Once the algorithm finds a solution, the program highlights the path by marking it with a specific symbol (e.g., '.').

How each Algorithm works

Depth-First Search (DFS) is employed to navigate a maze in the provided code. Starting from the initial position, the algorithm progresses by moving to unvisited neighboring cells and adding them to a stack for further exploration. The process continues until the destination is reached or all possible paths are exhausted. During this traversal, the algorithm keeps track of several statistics: the number of nodes (cells) expanded, the maximum depth reached in the search tree, and the maximum number of nodes in the stack (fringe) at any given time. If the destination is found, the algorithm returns the path taken, its length, and the recorded statistics. If no path is discovered, it returns None for all values, indicating an unsuccessful search. This way, DFS systematically explores the maze, prioritizing deep paths and only backtracking when necessary, which might not guarantee the shortest path but ensures complete coverage of the maze.

Breadth-First Search (BFS) algorithm for navigating a maze. It begins at the starting point and systematically explores the maze by examining all reachable cells level by level. It maintains a queue to track the cells to be explored, with each cell and its associated route being dequeued and expanded in a breadth-first manner. The algorithm counts the number of expanded nodes, keeps track of the maximum depth reached in the search tree, and records the maximum number of nodes in the queue (fringe) at any point during the search. If the destination is encountered, the algorithm returns the path taken, its length, and the recorded statistics. This approach ensures that BFS explores the shortest path first and guarantees completeness, making it suitable for finding the shortest path in a maze.

A* search combines the advantages of both Dijkstra's algorithm and Greedy Best-First Search. It uses a priority queue (heap) to expand nodes while keeping track of an estimated cost to reach the destination and the actual cost of the path taken so far. In each iteration, the algorithm selects the node with the lowest combined cost and explores its neighbors, prioritizing paths that appear to be more promising. It tracks statistics, such as the number of nodes expanded, the maximum depth reached, and the maximum number of nodes in the heap. When the destination is reached, the algorithm returns the path taken, its length, and the recorded statistics. If no path is found, it returns **None** for all values, indicating an unsuccessful search. This approach ensures efficient and optimal pathfinding in the maze.

main() function

1. Initialization: The code initializes a list of maze layouts (MazeListPart1 and MazeListPart2) and prompts the user to choose between two assignments: "MazeSolution" or "SearchMazeSolution."

2. LayToMatrix Function : The LayToMatrix function reads a layout file and converts it into a 2D matrix (list of lists) representing the maze.

```
def LayToMatrix(layPath):
    with open(layPath, 'r') as f:
        MazeMatrix = [list(line.strip()) for line in f.readlines()]
    return MazeMatrix
```

3. Draw Maze Functions : Two functions, drawMazePart1 and drawMazePart2 , are defined to visualize the maze using Pygame. They use various images (e.g., walls, Pac-Man, food, empty cells) to draw the maze on the screen.

```
def drawMazePart1(MazeMatrix):
    for i in range(len(MazeMatrix)):
        for j in range(len(MazeMatrix[0])):
            if MazeMatrix[i][j] == '%':
                Screen.blit(wall,(j*cell,i*cell))
            elif MazeMatrix[i][j] == 'P':
                Screen.blit(pacman,(j*cell,i*cell))
            elif MazeMatrix[i][j] == '.':
                Screen.blit(pacman_food,(j*cell,i*cell))
            else:
                Screen.blit(empty,(j*cell,i*cell))
        pygame.display.flip()

def drawMazePart2(MazeMatrix):
    for i in range(len(MazeMatrix)):
        for j in range(len(MazeMatrix[0])):
```

```
            if MazeMatrix[i][j] == '%':
                Screen.blit(wall,(j*cell,i*cell))
            elif MazeMatrix[i][j] == 'P':
                Screen.blit(pacman,(j*cell,i*cell))
            elif MazeMatrix[i][j] == '.':
                Screen.blit(pacman_food,(j*cell,i*cell))
            else:
                Screen.blit(empty,(j*cell,i*cell))
```

4. Draw Final Path Function : The drawFinalPath function takes the solution path found by the algorithms, visualizes the Pac-Man's movement, and displays information about the path (cost, depth, nodes expanded, and max fringe) on the screen.

```
def drawFinalPath(Route, Screen, Cost, MaxDepth, NoofNodesExpanded, MaxFringe):
    for currNode in Route:

                Screen.blit(pacman_track,(currNode[1]*cell,currNode[0]*cell))
        display_infoPart1(Screen, Cost, MaxDepth,NoofNodesExpanded, MaxFringe)
        pygame.display.flip()
        clock.tick(120)
```

5. Display Info Functions : Two functions, display_infoPart1 and display_infoPart2 , are used to display information about the path on the screen, including cost, max depth, nodes expanded, and max fringe.

```
def display_infoPart1(Screen, Cost, MaxDepth, nodesExpanded, MaxFringe):
    font = pygame.font.SysFont('Helvetica', 25,bold=False, italic=True)
    text1 = font.render("Route Cost: {}".format(Cost), True, (255, 255, 255))
    text2 = font.render("Maximum Depth: {}".format(MaxDepth), True, (255, 255, 255))
```

```
        text3 = font.render("No.of Nodes Expanded: {}".format(nodesExpanded),
True, (255, 255, 255))
        text4 = font.render("Maximum Fringe: {}".format(MaxFringe), True, (255,
255, 255))
        Screen.blit(text1, (LENGTH-290, 30))
        Screen.blit(text2, (LENGTH-290, 60))
        Screen.blit(text3, (LENGTH-290, 90))
        Screen.blit(text4, (LENGTH-290, 120))

    def display_infoPart2(Screen, cost, maxDepth, nodesExpanded, maxFringe):
        font = pygame.font.SysFont('Helveticas', 20, bold=False, italic=True)
        text1 = font.render("Path Cost: {}".format(cost), True, (255, 255, 255))
        text2 = font.render("Max Depth: {}".format(maxDepth), True, (255, 255,
255))
        text3 = font.render("Nodes Expanded: {}".format(nodesExpanded), True,
(255, 255, 255))
        text4 = font.render("Max Fringe: {}".format(maxFringe), True, (255, 255,
255))

        Screen.blit(text1, (LENGTH-290, 0))
        Screen.blit(text2, (LENGTH-290, 30))
        Screen.blit(text3, (LENGTH-290, 60))
        Screen.blit(text4, (LENGTH-290, 90))
```

6. User Input : The code prompts the user to choose a layout file and an algorithm (0 for original matrix, 1 for DFS, 2 for BFS, 3 for A*).

7. Pygame Initialization : Pygame is initialized, and the screen dimensions are set based on the chosen maze layout.

8. Maze Drawing Loop : The code enters a loop to draw the maze using Pygame. It listens for a user event to quit the application.

9. Algorithm Execution : If the user selects an algorithm (1 for DFS, 2 for BFS, 3 for A*), the corresponding pathfinding algorithm is executed, and the solution path, along with path information, is displayed on the screen. The path is visualized by moving Pac-Man through the maze. Afterward, the user can choose to run other algorithms.

10. Endless Loop : The loop continues, allowing the user to experiment with different algorithms on the selected maze layout.

assignmentMazeSolvingPart1()

1. **Depth-First Search (DFS):** When algo is 1, the function performs a depth-first search to find a path through the maze. It initializes a stack and explores the maze by pushing and popping nodes in a last-in, first-out order. It keeps track of statistics such as nodes expanded, maximum depth, and maximum fringe size. If the goal is reached, it returns the path taken, its length, and the recorded statistics.

2. **Breadth-First Search (BFS):** When algo is 2, the function conducts a breadth-first search to find a path through the maze. It uses a queue (implemented as a deque) to explore the maze level by level, ensuring the shortest path is found. Similar to DFS, it tracks statistics and returns the path, its length, and the statistics when the goal is reached.

3. **A* Search:** When algo is 3, the function utilizes the A* search algorithm, which is informed and combines the best aspects of both BFS and DFS. It uses a priority queue (heap) to explore the maze nodes based on the combined cost to reach the destination and the actual cost of the path taken so far. This ensures optimality and efficiency. It tracks statistics and returns the path, its length, and the recorded statistics when the destination is found.

markAsVisited():
it is responsible for visually marking a visited node in the maze. When a node is visited during a search algorithm, it is marked with a "visited" symbol (represented by the **circle** image) on the screen. The function takes two parameters: **currNode**, which specifies the coordinates of the node to mark, and **Screen**, which represents the Pygame window or display.

```
def markAsVisited(currNode, Screen):
    Screen.blit(circle, (currNode[1] * cell, currNode[0] * cell))
    pygame.display.flip()
    clock.tick(60)
```

closeFood():
It is designed to find the position of the closest food item (represented as a dot '.') in a maze-like environment. It uses a breadth-first search approach to efficiently locate the nearest food item from a given starting position.

```
def closeFood(matrix, start):
    queue = deque([(start, 0)])
    visited = set()
    FoodPositions = []
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == '.':
                FoodPositions.append((i, j))
    while queue:
        currPos, distance = queue.popleft()
        if currPos in FoodPositions:
            return currPos
        visited.add(currPos)
        neighbourNodes = []
        for dr, dc in [(1, 0), (-1, 0),(0, 1), (0, -1)]:
            nr, nc = currPos[0] + dr, currPos[1] + dc
            if 0 <= nr < len(matrix) and 0 <= nc < len(matrix[0]) and matrix[nr][nc] !=
'%' and (nr, nc) not in visited:
                neighbourNodes.append((nr, nc))
        for neighbor in neighbourNodes:
            queue.append((neighbor, distance + 1))
    return None
```

neighbourNodes()
this function is designed to find and return the neighboring nodes (cells) that are accessible from a given current node within a maze, represented by a 2D matrix (**MazeMatrix**). This function is typically used in pathfinding and search algorithms to determine which adjacent cells can be visited from the current position.

```
def neighbourNodes(MazeMatrix, currNode):
    nodesList = []
    dirs = [(1, 0), (-1, 0),(0, 1), (0, -1)]
    for x, y in dirs:
```

```
        xNode = currNode[0]+x
        yNode = currNode[1]+y
        if 0 <= xNode < len(MazeMatrix) and 0 <= yNode < len(MazeMatrix[0]) and
MazeMatrix[xNode][yNode] != '%':
            nodesList.append((xNode, yNode))
    return nodesList
```

StartandEndPositions

This function is designed to identify and return the positions of the starting point (indicated by 'P') and the ending point (indicated by '.') within a given 2D matrix (**mat**). This function is commonly used in maze-solving and pathfinding tasks to determine the coordinates of the initial and target locations in a maze.

```
def StartandEndPositions(mat):
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            if mat[i][j] == 'P':
                StartPoint = (i, j)
            elif mat[i][j] == '.':
                EndingPoint = (i, j)
    return StartPoint, EndingPoint
```

PYTHON SCRIPT

```
from collections import deque
import heapq
import os
import sys
import pygame

cell = 20
clock = pygame.time.Clock()

program_directory = os.path.dirname(sys.argv[0])
circle =
pygame.transform.scale(pygame.image.load(os.path.join(program_directory,
"circle.png")),(15,15))
```

```
pacman
=pygame.transform.scale(pygame.image.load(os.path.join(program_directory,
"pacman.jpg")),(20,20))
pacman_food
=pygame.transform.scale(pygame.image.load(os.path.join(program_directory,
"pacman_food.png")),(20,20))
pacman_track=pygame.transform.scale(pygame.image.load(os.path.join(program_
directory, "pacman_track.png")),(15,15))
wall =pygame.transform.scale(pygame.image.load(os.path.join(program_directory,
"wall.png")),(20,20))
empty=pygame.transform.scale(pygame.image.load(os.path.join(program_director
y, "empty.jpg")),(20,20))

def markAsVisited(currNode, Screen):
    Screen.blit(circle, (currNode[1] * cell, currNode[0] * cell))
    pygame.display.flip()
    clock.tick(60)

def closeFood(matrix, start):
    queue = deque([(start, 0)])
    visited = set()
    FoodPositions = []
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == '.':
                FoodPositions.append((i, j))
    while queue:
        currPos, distance = queue.popleft()
        if currPos in FoodPositions:
            return currPos
        visited.add(currPos)
        neighbourNodes = []
        for dr, dc in [(1, 0), (-1, 0),(0, 1), (0, -1)]:
            nr, nc = currPos[0] + dr, currPos[1] + dc
            if 0 <= nr < len(matrix) and 0 <= nc < len(matrix[0]) and matrix[nr][nc] !=
'%' and (nr, nc) not in visited:
                neighbourNodes.append((nr, nc))
        for neighbor in neighbourNodes:
            queue.append((neighbor, distance + 1))
```

```
    return None

def neighbourNodes(MazeMatrix, currNode):
    nodesList = []
    dirs = [(1, 0), (-1, 0),(0, 1), (0, -1)]
    for x, y in dirs:
        xNode = currNode[0]+x
        yNode = currNode[1]+y
        if 0 <= xNode < len(MazeMatrix) and 0 <= yNode < len(MazeMatrix[0]) and
MazeMatrix[xNode][yNode] != '%':
            nodesList.append((xNode, yNode))
    return nodesList

def StartandEndPositions(mat):
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            if mat[i][j] == 'P':
                StartPoint = (i, j)
            elif mat[i][j] == '.':
                EndingPoint = (i, j)
    return StartPoint, EndingPoint

def assignmentMazeSolvingPart1(MazeMatrix,Screen,algo):
    if algo==1:
        nodesExpanded = 0
        MaxDepth = 0
        MaxFringe = 0
        StartPoint, EndingPoint = StartandEndPositions(MazeMatrix)
        stack = [(StartPoint, [StartPoint])]
        visited = set()
        while stack:
            currNode, Route = stack.pop()
            nodesExpanded += 1
            MaxDepth = max(MaxDepth, len(Route))
            MaxFringe = max(MaxFringe, len(stack))


            if currNode == EndingPoint:
                return Route, len(Route)-1, nodesExpanded, MaxDepth, MaxFringe
```

```
        if currNode not in visited:
            markAsVisited(currNode, Screen)
            visited.add(currNode)
            for _ in neighbourNodes(MazeMatrix, currNode):
                if _ not in visited:
                    stack.append((_, Route+[_]))

    return None, None, None, None, None
elif algo==2:
    nodesExpanded = 0
    MaxDepth = 0
    MaxFringe = 0
    StartPoint, EndingPoint = StartandEndPositions(MazeMatrix)
    queue = deque([(StartPoint, [StartPoint])])
    VisitedSet = set()
    while queue:
        currNode, Route = queue.popleft()
        nodesExpanded += 1
        MaxDepth = max(MaxDepth, len(Route))
        MaxFringe = max(MaxFringe, len(queue))
        if currNode == EndingPoint:
            return Route, len(Route)-1, nodesExpanded, MaxDepth, MaxFringe
        if currNode not in VisitedSet:
            markAsVisited(currNode, Screen)
            VisitedSet.add(currNode)
            for _ in neighbourNodes(MazeMatrix, currNode):
                if _ not in VisitedSet:
                    queue.append((_, Route+[_]))
    return None, None, None, None, None
elif algo==3:
    nodesExpanded = 0
    MaxDepth = 0
    MaxFringe = 0
    StartPoint, EndingPoint = StartandEndPositions(MazeMatrix)
    heap = [(0, StartPoint, [StartPoint])]
    VisitedSet = set()
    while heap:
        k, currNode, Route = heapq.heappop(heap)
```

```
            nodesExpanded += 1
            MaxDepth = max(MaxDepth, len(Route))
            MaxFringe = max(MaxFringe, len(heap))
            if currNode == EndingPoint:
                return Route, len(Route)-1, nodesExpanded, MaxDepth, MaxFringe
            if currNode not in VisitedSet:
                markAsVisited(currNode, Screen)
                VisitedSet.add(currNode)
                for _ in neighbourNodes(MazeMatrix, currNode):
                    if _ not in VisitedSet:
                        Cost = abs(_[0]-EndingPoint[0]) +abs(_[1]-EndingPoint[1])
                        heapq.heappush(heap, (Cost, _, Route+[_]))
        return None, None, None, None, None

def assignmentMazeSolvingPart2(MazeMatrix,algo):

    direction = [(1, 0), (-1, 0),(0, 1), (0, -1)]
    if algo==1:
        nodesExpanded = 0
        MaxDepth = 0
        MaxFringe = 0
        matrix = MazeMatrix

        FoodPositions = []
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == 'P':
                    start = (i, j)
                elif matrix[i][j] == '.':
                    FoodPositions.append((i, j))

        Route = []
        while FoodPositions:
            closestFood = closeFood(matrix, start)
            visitedSet = set()
            stack = [(start, 0, [])]

            while stack:
                pos, currdist, path = stack.pop(0)
```

```
            nodesExpanded += 1
            MaxDepth = max(MaxDepth, len(path))
            MaxFringe = max(MaxFringe, len(stack))

            if pos == closestFood:
                Route.append(path + [pos])
                matrix[pos[0]] = list(matrix[pos[0]])
                matrix[pos[0]][pos[1]] = 'P'
                matrix[pos[0]] = ''.join(matrix[pos[0]])
                start = pos
                FoodPositions.remove(pos)
                break

            if pos not in visitedSet:
                visitedSet.add(pos)
                for i, j in direction:
                    nextRow = pos[0] + i
                    nextCol = pos[1] + j
                    if (
                        0 <= nextRow < len(matrix)
                        and 0 <= nextCol < len(matrix[0])
                        and matrix[nextRow][nextCol] != '%'
                        and (nextRow, nextCol) not in visitedSet
                    ):
                        stack.append(
                            ((nextRow, nextCol), currdist + 1, path + [pos]))

    return Route, len(Route)  1, nodesExpanded, MaxDepth, MaxFringe
elif algo==2:
    nodesExpanded = 0
    MaxDepth = 0
    MaxFringe = 0
    matrix = MazeMatrix
    FoodPositions = []
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 'P':
                start = (i, j)
            elif matrix[i][j] == '.':
```

```
                FoodPositions.append((i, j))

        Route = []
        while FoodPositions:
            closetFood = closeFood(matrix, start)
            vistedSet = set()
            queue = deque([(start, 0, [])])
            while queue:
                pos, currdist, path = queue.popleft()
                nodesExpanded += 1
                MaxDepth = max(MaxDepth, len(path))
                MaxFringe = max(MaxFringe, len(queue))

                if (pos == closetFood):
                    Route.append(path+[pos])
                    matrix[pos[0]] = list(matrix[pos[0]])
                    matrix[pos[0]][pos[1]] = 'P'
                    matrix[pos[0]] = ''.join(matrix[pos[0]])
                    start = pos
                    FoodPositions.remove(pos)
                    break
                if pos not in vistedSet:
                    vistedSet.add(pos)
                    for i, j in direction:
                        nextRow = pos[0]+i
                        nextCol = pos[1]+j
                        if 0 <= nextRow < len(matrix) and 0 <= nextCol < len(matrix[0])
and matrix[nextRow][nextCol] != '%' and (nextRow, nextCol) not in vistedSet:
                            queue.append(
                                ((nextRow, nextCol), currdist+1, path+[pos]))

        return Route,len(Route)-1, nodesExpanded, MaxDepth, MaxFringe
    elif algo==3:
        nodesExpanded = 0
        MaxDepth = 0
        MaxFringe = 0
        matrix = MazeMatrix

        FoodPositions = []
```

```
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 'P':
                start = (i, j)
            elif matrix[i][j] == '.':
                FoodPositions.append((i, j))

    Route=[]
    while FoodPositions:

        closetFood = closeFood(matrix, start)
        vistedSet = set()
        heap = [(start, 0, [])]
        while heap:
            pos, currdist, path = heapq.heappop(heap)
            nodesExpanded += 1
            MaxDepth = max(MaxDepth, len(path))
            MaxFringe = max(MaxFringe, len(heap))

            if (pos == closetFood):
                Route.append(path+[pos])
                matrix[pos[0]] = list(matrix[pos[0]])
                matrix[pos[0]][pos[1]] = 'P'
                matrix[pos[0]] = ''.join(matrix[pos[0]])
                start = pos
                FoodPositions.remove(pos)
                break
            if pos not in vistedSet:
                vistedSet.add(pos)
                for i, j in direction:
                    nextRow = pos[0]+i
                    nextCol = pos[1]+j
                    if 0 <= nextRow < len(matrix) and 0 <= nextCol < len(matrix[0])
and matrix[nextRow][nextCol] != '%' and (nextRow, nextCol) not in vistedSet:
                        heapq.heappush(heap, (((nextRow, nextCol),
currdist+1+abs(nextRow-closetFood[0])+abs(nextCol-closetFood[1]),
path+[pos])))

    return Route,len(Route)-1, nodesExpanded, MaxDepth, MaxFringe
```

```
def main():
    MazeListPart1 = ['bigMaze.lay', 'mediumMaze.lay',
'smallMaze.lay','openMaze.lay']
    MazeListPart2 =['trickySearch.lay', 'tinySearch.lay', 'smallSearch.lay']
    print("Choose an Assignment to Run:\n1 for MazeSolution \n2 for
SearchMazeSolution ")

    nameList=["","DFS","BFS","ASTAR"]

    def LayToMatrix(layPath):
        with open(layPath, 'r') as f:
            MazeMatrix = [list(line.strip()) for line in f.readlines()]
        return MazeMatrix

    def drawMazePart1(MazeMatrix):
        for i in range(len(MazeMatrix)):
            for j in range(len(MazeMatrix[0])):
                if MazeMatrix[i][j] == '%':
                    Screen.blit(wall,(j*cell,i*cell))
                elif MazeMatrix[i][j] == 'P':
                    Screen.blit(pacman,(j*cell,i*cell))
                elif MazeMatrix[i][j] == '.':
                    Screen.blit(pacman_food,(j*cell,i*cell))
                else:
                    Screen.blit(empty,(j*cell,i*cell))
            pygame.display.flip()

    def drawMazePart2(MazeMatrix):
        for i in range(len(MazeMatrix)):
            for j in range(len(MazeMatrix[0])):
                if MazeMatrix[i][j] == '%':
                    Screen.blit(wall,(j*cell,i*cell))
                elif MazeMatrix[i][j] == 'P':
                    Screen.blit(pacman,(j*cell,i*cell))
                elif MazeMatrix[i][j] == '.':
                    Screen.blit(pacman_food,(j*cell,i*cell))
                else:
                    Screen.blit(empty,(j*cell,i*cell))
```

```
    def drawFinalPath(Route, Screen, Cost, MaxDepth, NoofNodesExpanded,
MaxFringe):
        for currNode in Route:
            Screen.blit(pacman_track,(currNode[1]*cell,currNode[0]*cell))
            display_infoPart1(Screen, Cost, MaxDepth,NoofNodesExpanded,
MaxFringe)
            pygame.display.flip()
            clock.tick(120)

    def display_infoPart1(Screen, Cost, MaxDepth, nodesExpanded, MaxFringe):
        font = pygame.font.SysFont('Helvetica', 25,bold=False, italic=True)
        text1 = font.render("Route Cost: {}".format(Cost), True, (255, 255, 255))
        text2 = font.render("Maximum Depth: {}".format(MaxDepth), True, (255,
255, 255))
        text3 = font.render("No.of Nodes Expanded: {}".format(nodesExpanded),
True, (255, 255, 255))
        text4 = font.render("Maximum Fringe: {}".format(MaxFringe), True, (255,
255, 255))
        Screen.blit(text1, (LENGTH-290, 30))
        Screen.blit(text2, (LENGTH-290, 60))
        Screen.blit(text3, (LENGTH-290, 90))
        Screen.blit(text4, (LENGTH-290, 120))

    def display_infoPart2(Screen, cost, maxDepth, nodesExpanded, maxFringe):
        font = pygame.font.SysFont('Helveticas', 20, bold=False, italic=True)
        text1 = font.render("Path Cost: {}".format(cost), True, (255, 255, 255))
        text2 = font.render("Max Depth: {}".format(maxDepth), True, (255, 255,
255))
        text3 = font.render("Nodes Expanded: {}".format(nodesExpanded), True,
(255, 255, 255))
        text4 = font.render("Max Fringe: {}".format(maxFringe), True, (255, 255,
255))

        Screen.blit(text1, (LENGTH-290, 0))
        Screen.blit(text2, (LENGTH-290, 30))
        Screen.blit(text3, (LENGTH-290, 60))
        Screen.blit(text4, (LENGTH-290, 90))
```

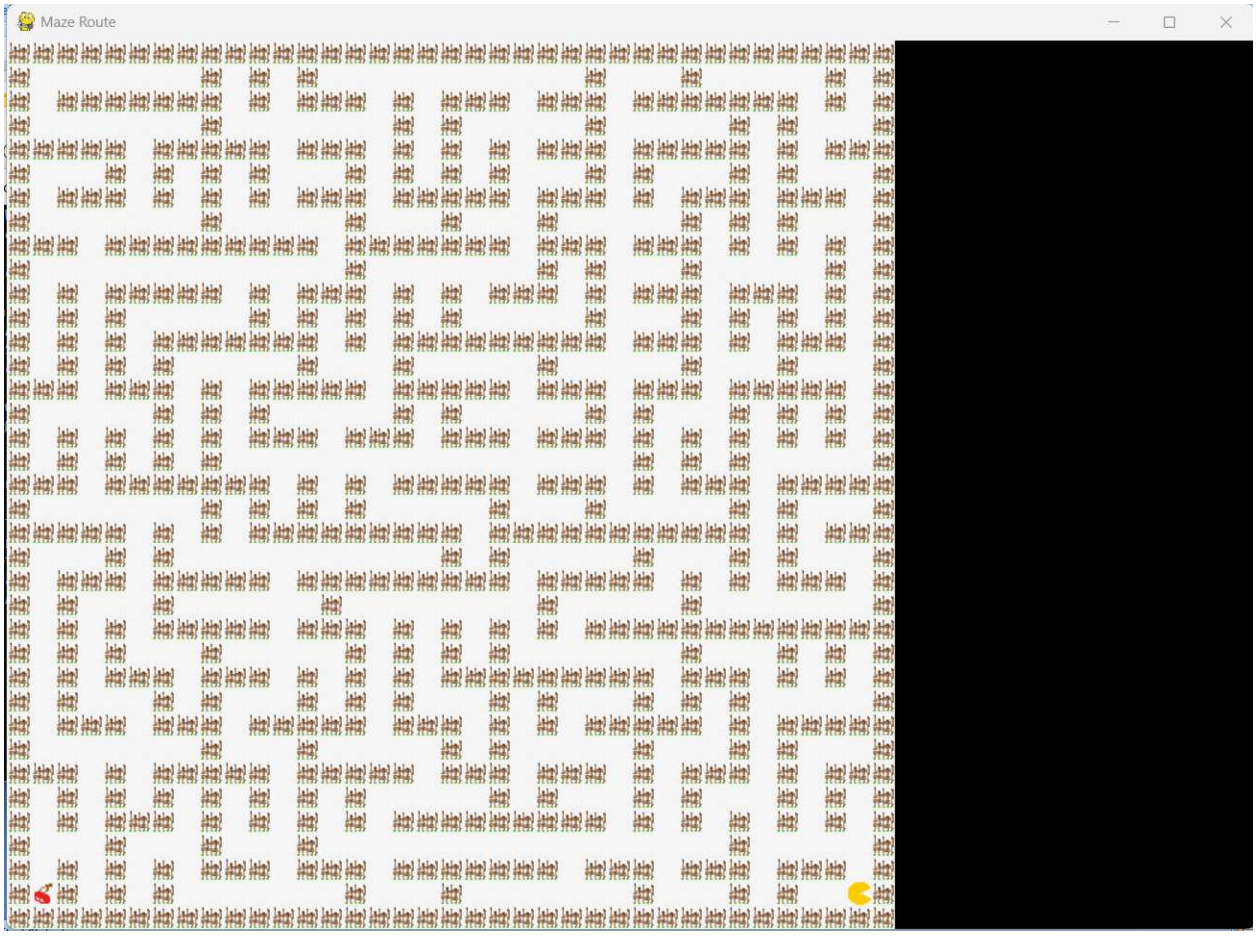
```
    ChoiceOfAssignment = int(input())
    if ChoiceOfAssignment == 1:

        print("Choose LayOut File : \n 0 for bigMaze \n 1 for mediunMaze \n 2 for
smallMaze \n 3 for openMaze")

        ChoiceOfLayout = int(input())

        MazeMatrix = LayToMatrix(MazeListPart1[ChoiceOfLayout])

        print("Enter Input of Algorithum Choice:\n 0 for original matrix \n 1 for
DepthFirstSearch Algo Route \n 2 for BreathFirstSearch Algo Route \n 3 for A*
Algo Route")

        ChoiceOfAlgo = int(input())


        cell = 20
        clock = pygame.time.Clock()
        pygame.init()
        BREATH = (len(MazeMatrix)*cell)
        LENGTH = (len(MazeMatrix[0])*cell)+300
        Screen = pygame.display.set_mode((LENGTH, BREATH),
pygame.RESIZABLE)

        flag = True

        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
            if flag:
                drawMazePart1(MazeMatrix)

            if ChoiceOfAlgo == 0:
                pygame.display.set_caption("Maze Route")
                pygame.display.flip()

            if ChoiceOfAlgo in [1,2,3]:
```

```
            pygame.display.set_caption("Maze Route"+nameList[ChoiceOfAlgo])
            pygame.display.flip()
            Route, Cost, NoofNodesExpanded, MaxDepth, MaxFringe =
assignmentMazeSolvingPart1(MazeMatrix, Screen, ChoiceOfAlgo)
            print(nameList[ChoiceOfAlgo]+" Route:", Route)
            drawFinalPath(Route, Screen, Cost,MaxDepth, NoofNodesExpanded,
MaxFringe)
            pygame.time.wait(10)
            pygame.display.flip()
            ChoiceOfAlgo = 0
            flag = False

    elif ChoiceOfAssignment == 2:
        print("Choose a Layout File : \n0 for trickySearch\n1 for tinySearch\n2 for
smallSearch")

        ChoiceOfLayout = int(input())

        print("Choose an algo to run : \n0 for original matrix\n1 for dfs Algo path\n2
for bfs Algo path\n3 for A* Algo path")

        ChoiceOfAlgo = int(input())

        MazeMatrix = LayToMatrix(MazeListPart2[ChoiceOfLayout])

        pygame.init()
        cell = 20
        BREATH = (len(MazeMatrix)*cell)
        LENGTH = len(MazeMatrix[0])*cell+300

        if ChoiceOfLayout != 1:
            Screen = pygame.display.set_mode(
                (LENGTH, BREATH), pygame.RESIZABLE)
        else:
            Screen = pygame.display.set_mode(
                (LENGTH*1.5, BREATH), pygame.RESIZABLE)
        pygame.display.set_caption("Maze Route with Multiple Targets")
        clock = pygame.time.Clock()
```

```
        flag = 0
        while True:

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

            if flag == 0:
                drawMazePart2(MazeMatrix)

            if ChoiceOfAlgo == 0:
                pygame.display.set_caption("Maze Route ")
                pygame.display.flip()

            if ChoiceOfAlgo in [1,2,3]:
                path, cost, node_expanded, max_depth, max_fringe =
assignmentMazeSolvingPart2(
                    MazeMatrix,ChoiceOfAlgo)
                result_list = [tup for sublist in path for tup in sublist]
                prev = path[0][0]
                for node in result_list:
                    pygame.display.set_caption("Maze Route part2
"+nameList[ChoiceOfAlgo])
                    Screen.blit(empty,(prev[1]*cell,prev[0]*cell))
                    clock.tick(10)
                    Screen.blit(pacman,(node[1]*cell,node[0]*cell))
                    prev = node
                    pygame.display.flip()
                display_infoPart2(Screen, cost, max_depth,node_expanded, max_fringe)
                print(nameList[ChoiceOfAlgo]+" Route :", path)

                ChoiceOfAlgo = 0
            pygame.display.flip()
            flag = flag+1

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

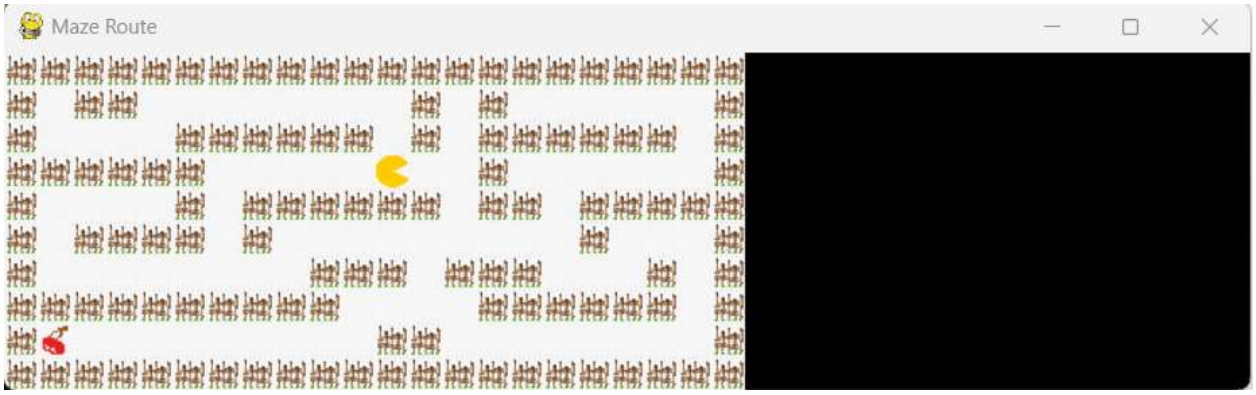
Visual Representation of Part 1 Mazes:

Big Maze Visual Representation:

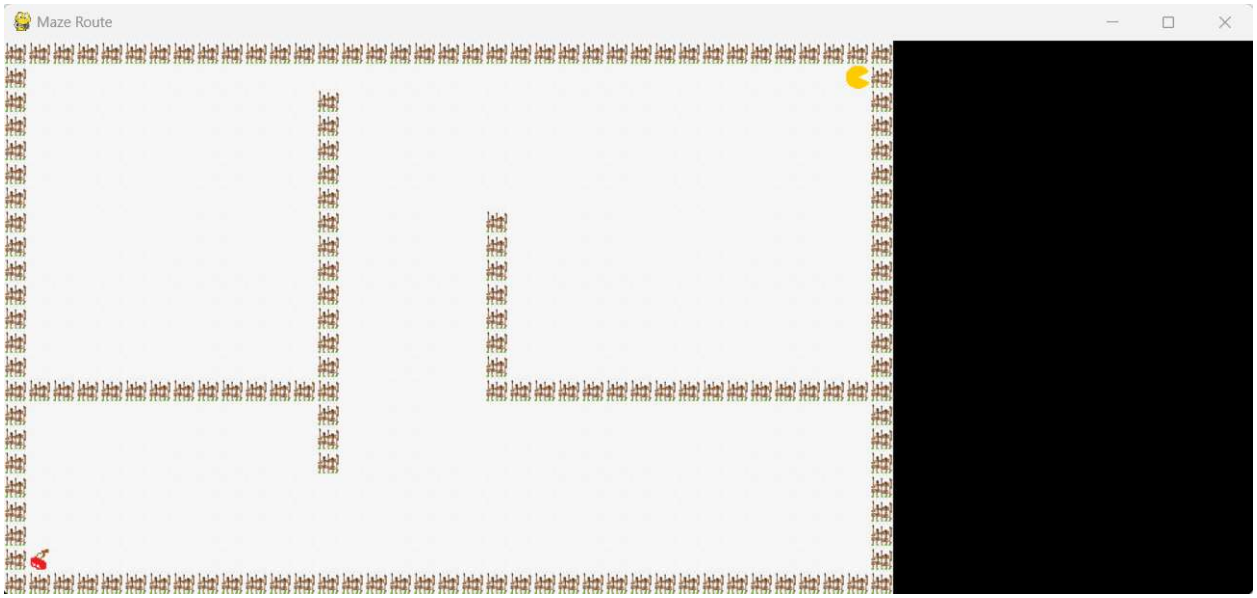


Medium Maze Visual Representation:

Small Maze Visual Representation:

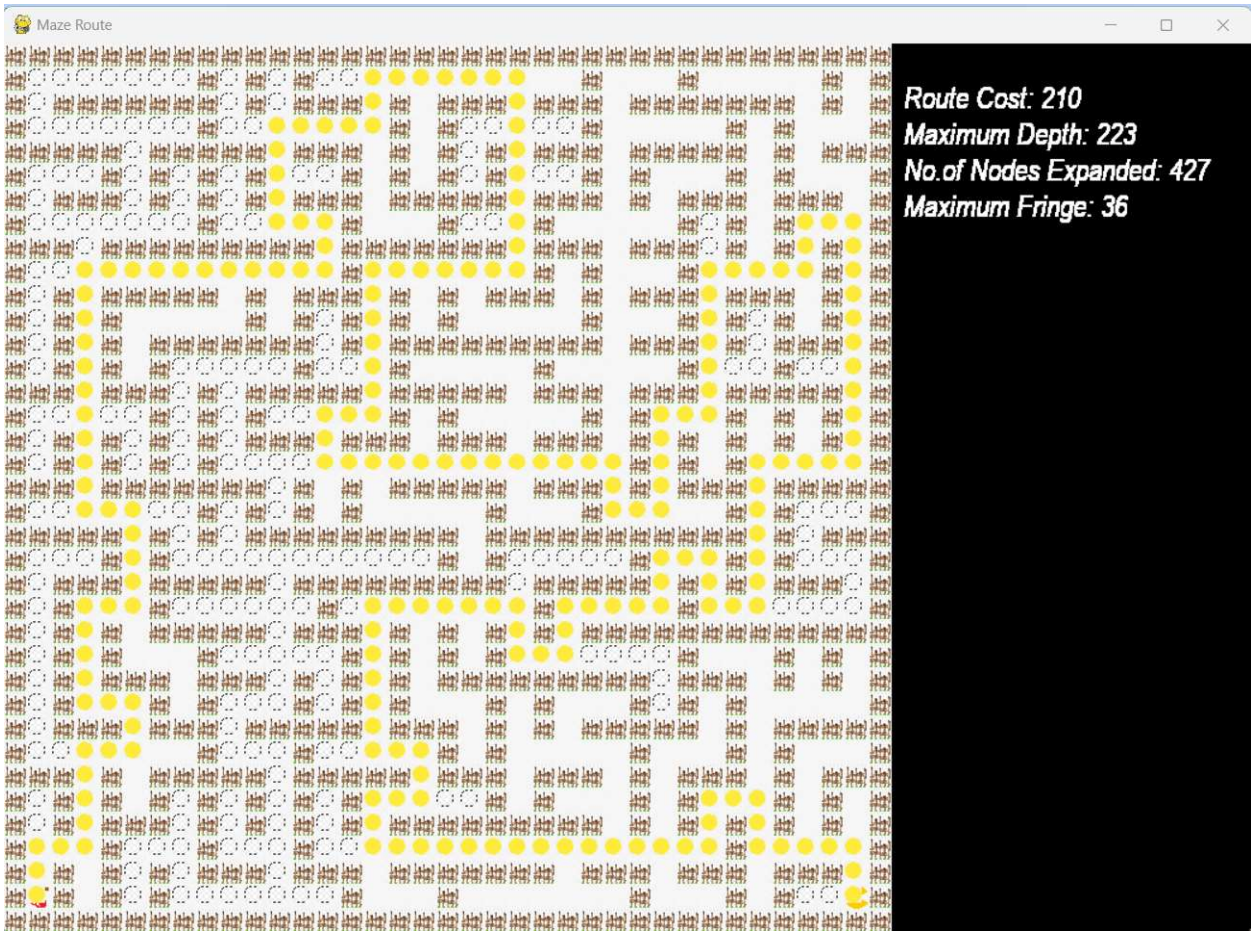


Open Maze Visual Representation:



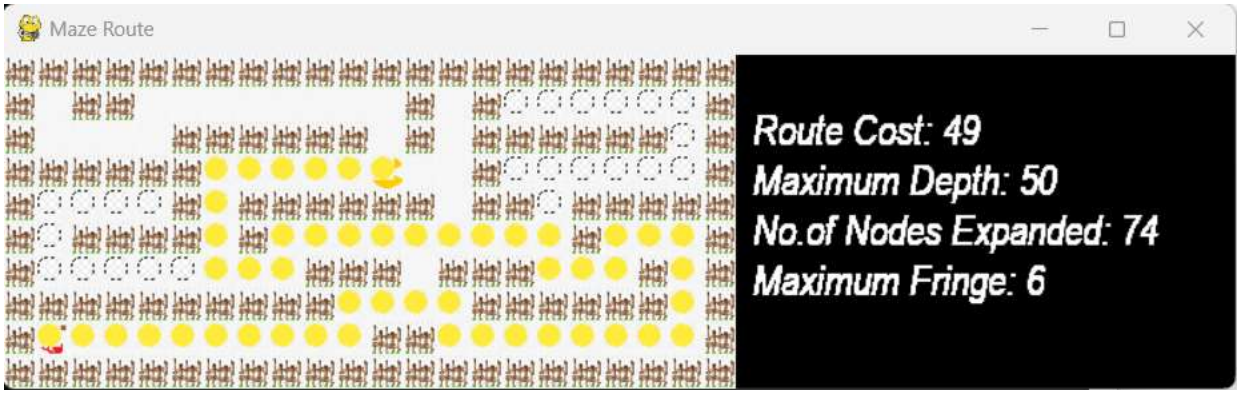
DFS Visual Representation of Maze:

Big Maze Visual Representation:

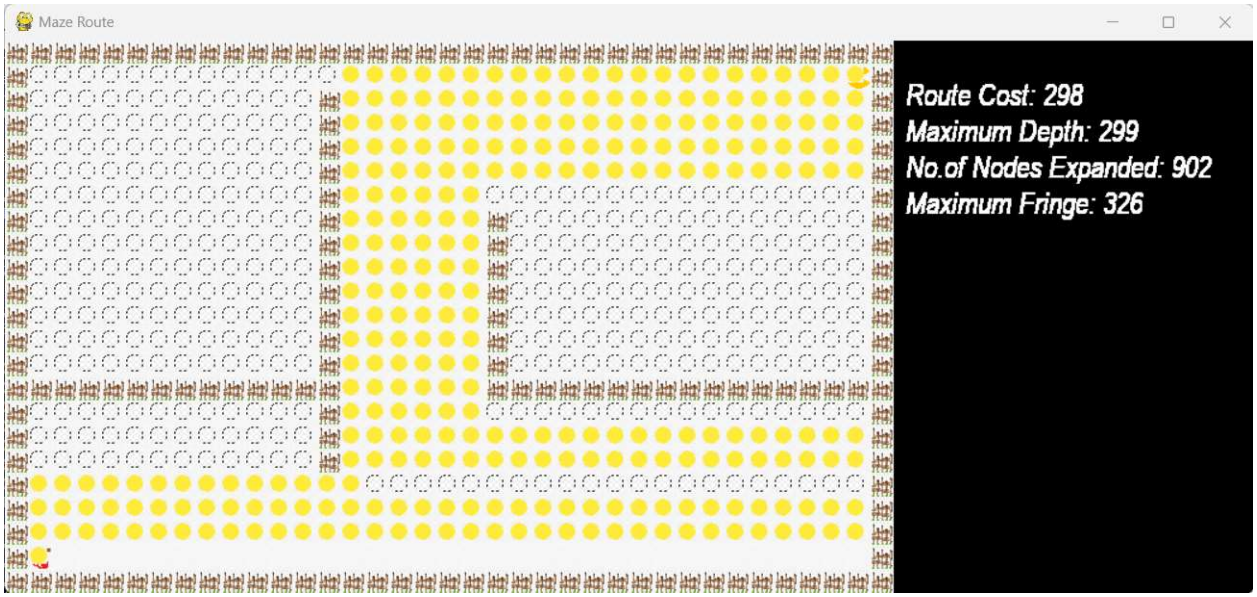


Medium Maze Visual Representation:

Small Maze Visual Representation:

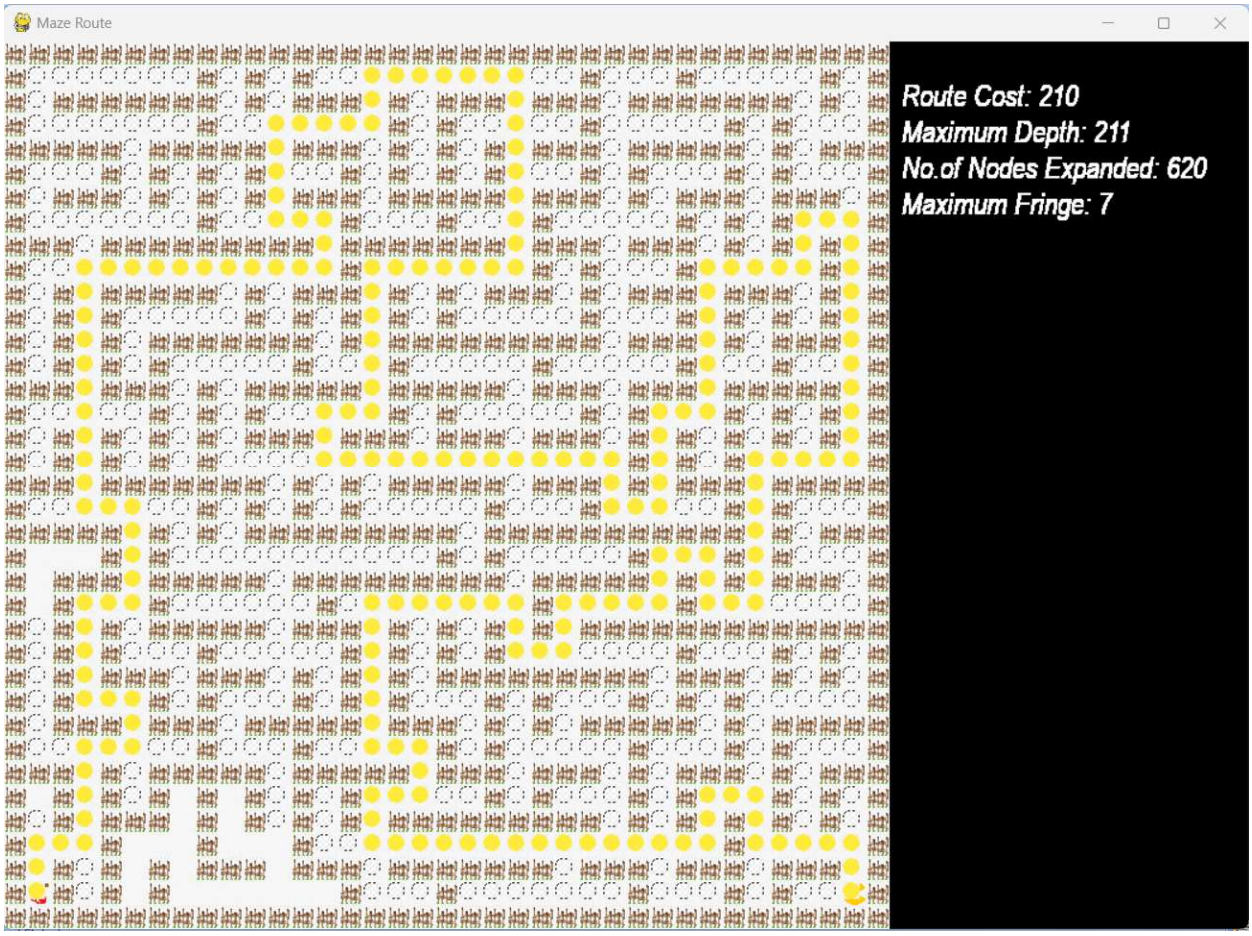


Open Maze Visual Representation:



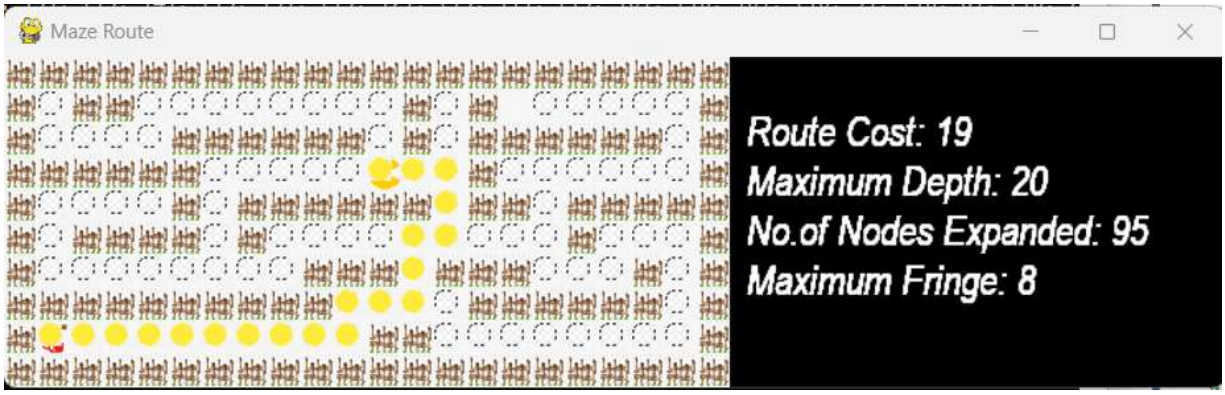
BFS Visual Representation of Maze:

Big Maze Visual Representation:

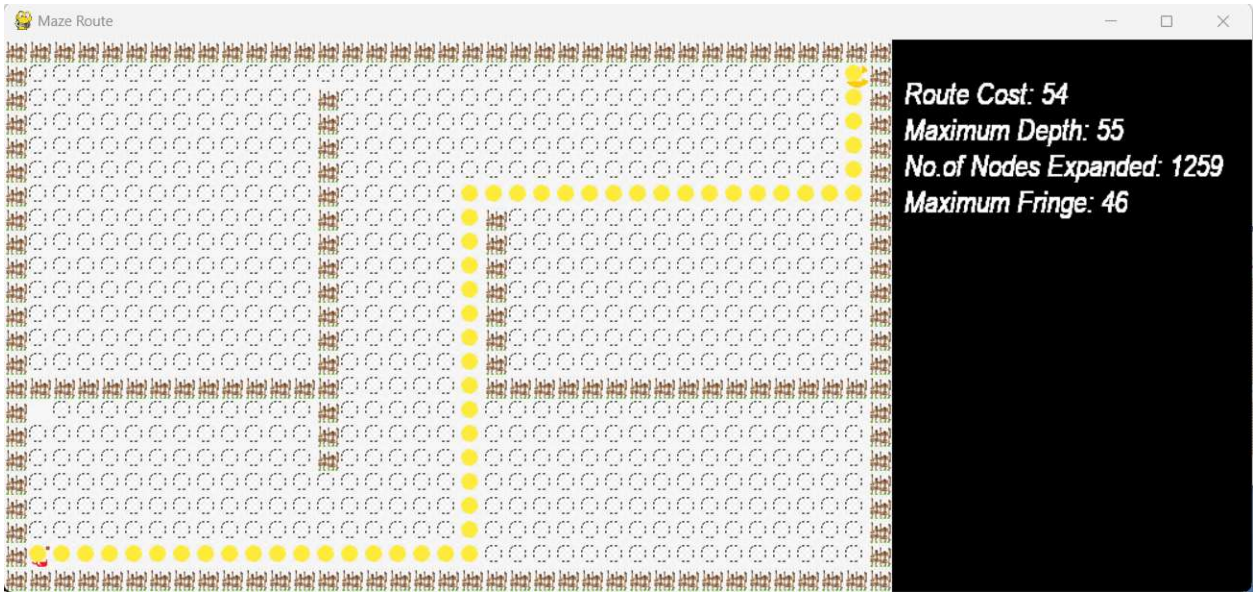


Medium Maze Visual Representation:

Small Maze Visual Representation:

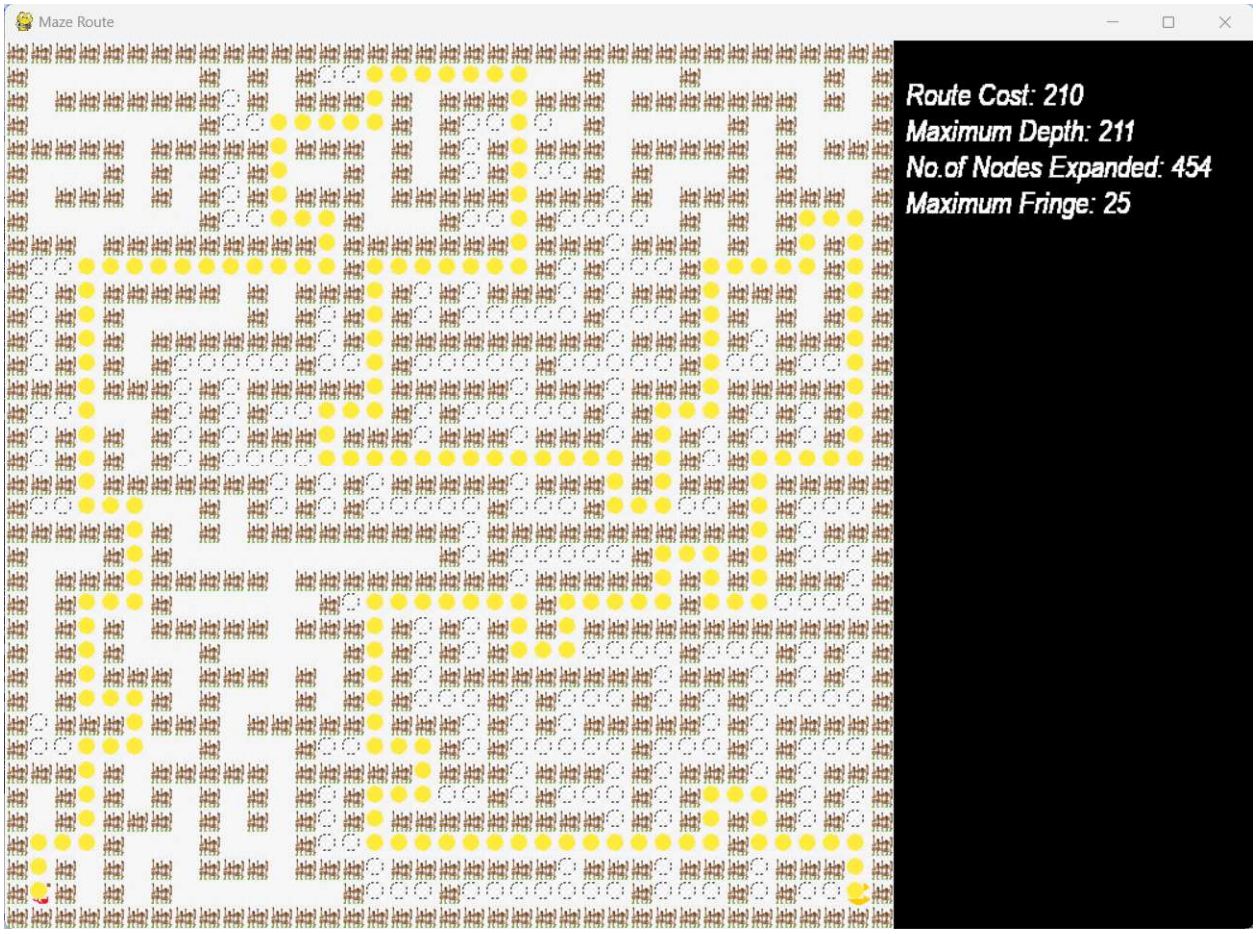


Open Maze Visual Representation:



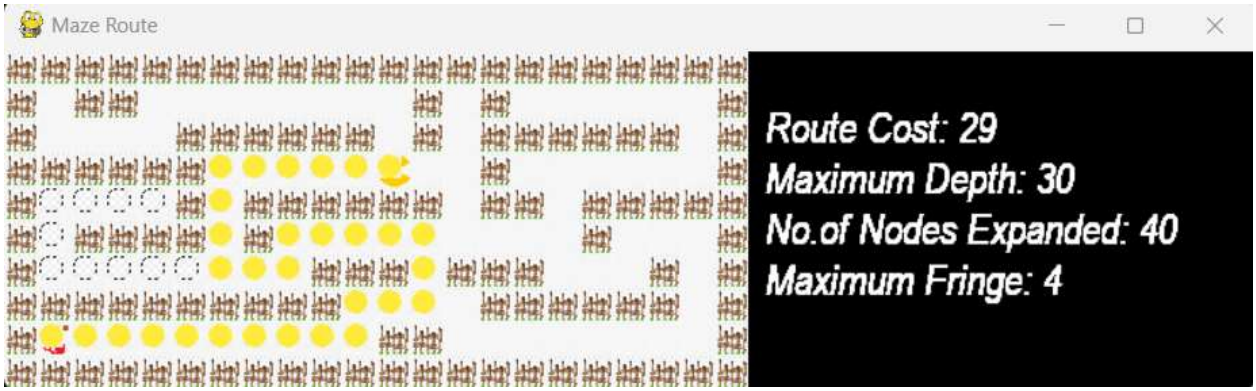
A* Visual Representation of Maze:

Big Maze Visual Representation:

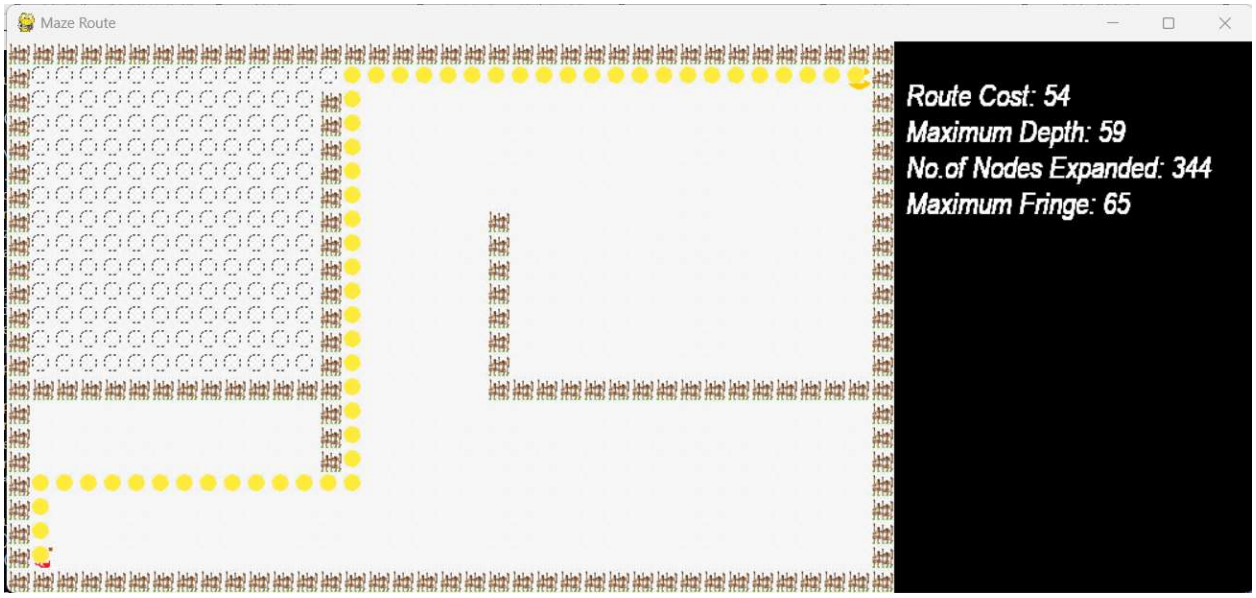


Medium Maze Visual Representation:

Small Maze Visual Representation:



Open Maze Visual Representation:



Detailed of each algorithm on each Maze

DFS Traverse Details

	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Big Maze	210	427	223	36
Medium Maze	130	147	131	8
Small Maze	49	74	50	6
Open Maze	298	902	299	326

BFS Traverse Details

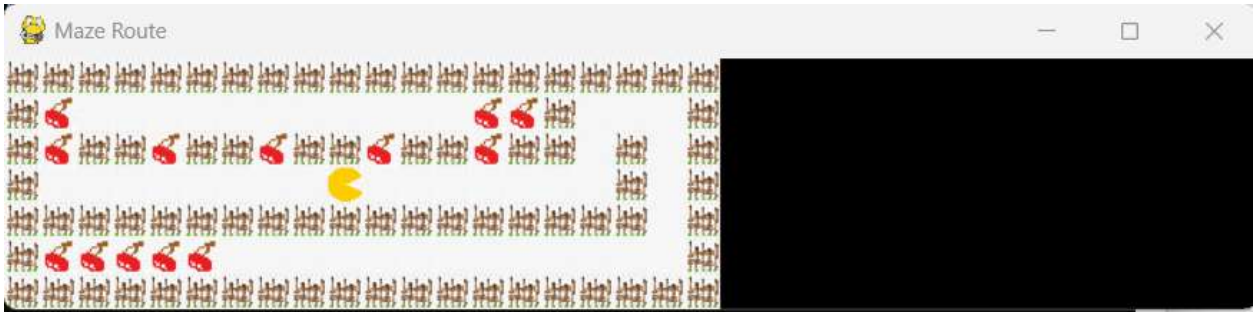
	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Big Maze	210	620	211	7
Medium Maze	68	276	69	8
Small Maze	19	95	20	8
Open Maze	54	1259	55	48

ASTAR Traverse Details

	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Big Maze	210	454	211	25
Medium Maze	152	158	153	7
Small Maze	29	0	30	4
Open Maze	54	344	59	65

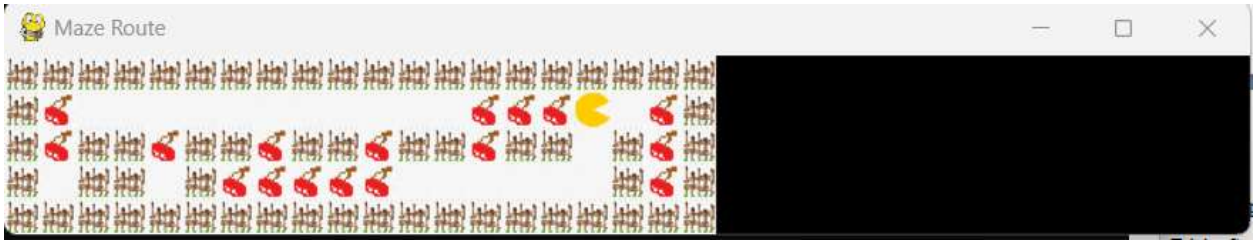
Visual Representation of part 2 Maze:

Tricky Search Maze visual representation



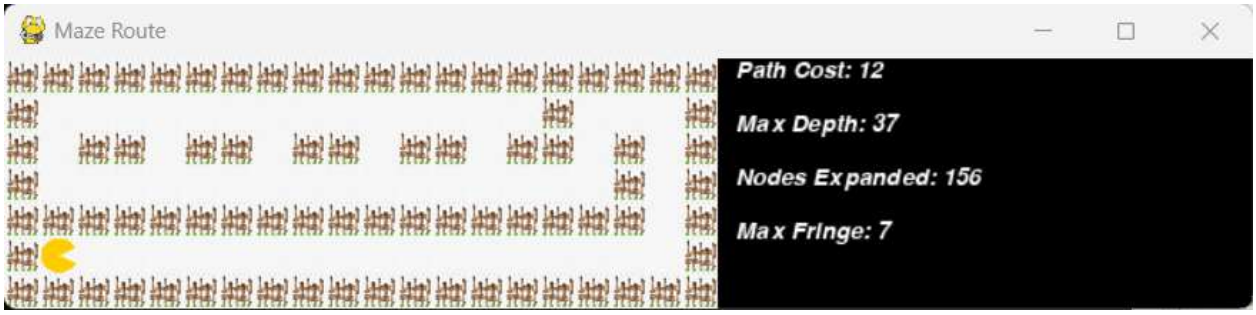
Tiny Search maze visual representation

Small Search maze visual representation

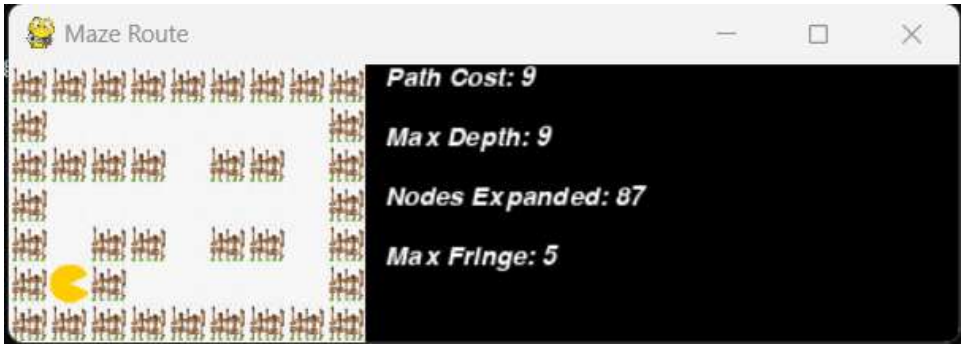


DFS Visual Representation of part 2 Maze:

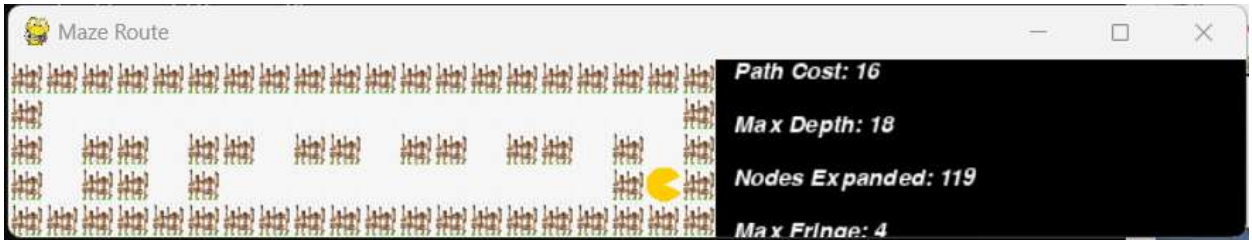
Tricky Search Maze visual representation



Tiny Search maze visual representation

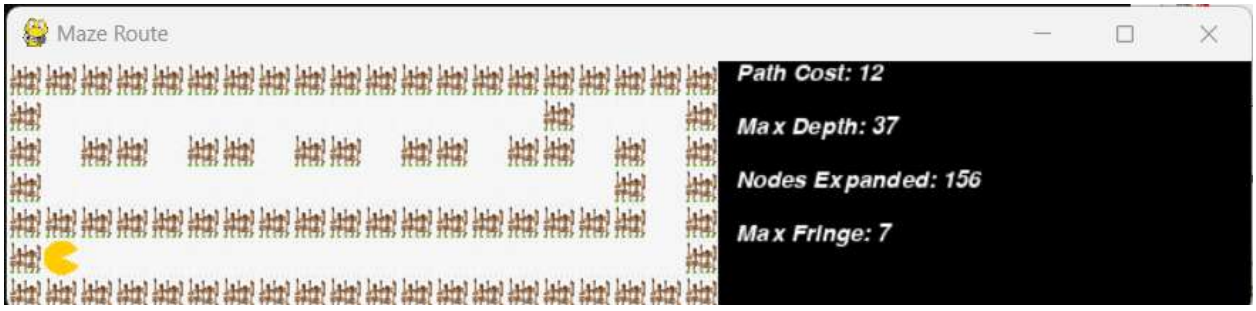


Small Search maze visual representation

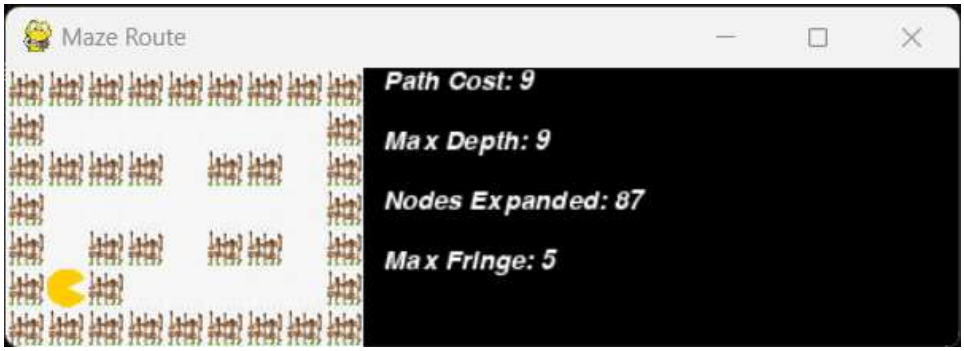


BFS Visual Representation of part 2 Maze:

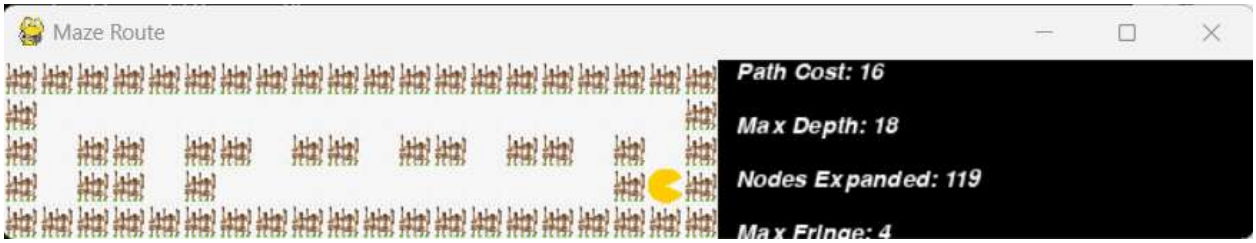
Tricky Search Maze visual representation



Tiny Search maze visual representation

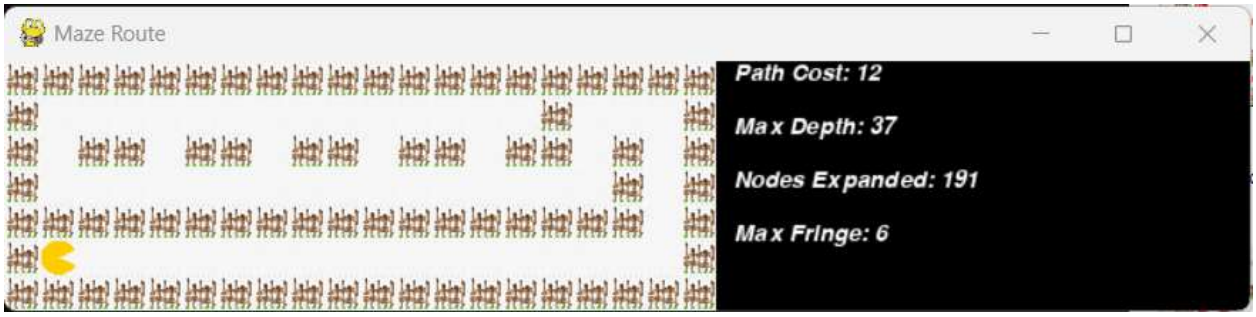


Small Search maze visual representation

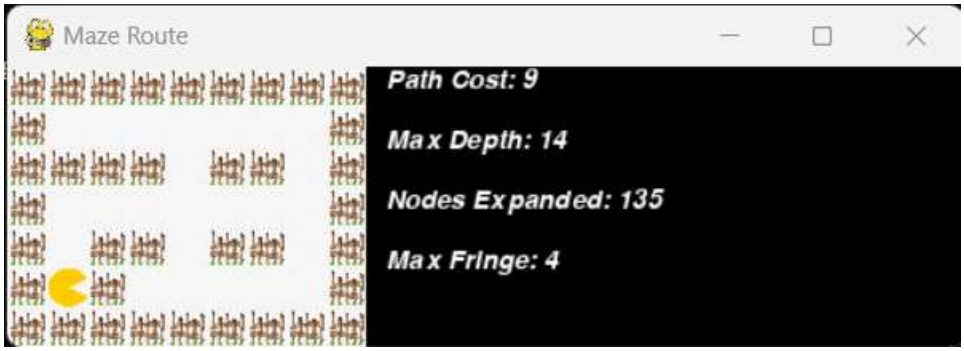


ASTAR Visual Representation of part 2 Maze:

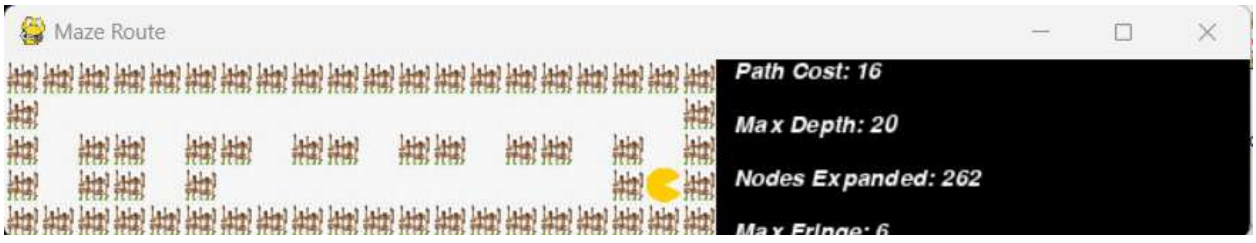
Tricky Search Maze visual representation



Tiny Search maze visual representation



Small Search maze visual representation



Detailed of each algorithm on each Maze of part 2

DFS Traverse Details

	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Tricky Search	12	156	37	7
Tiny Search	9	87	9	5
Small Search	16	119	18	4

BFS Traverse Details

	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Tricky Search	12	156	37	7
Tiny Search	9	87	9	5
Small Search	16	119	18	4

ASTAR Traverse Details

	Path cost	Number of nodes expanded	Maximum depth	Maximum fringe
Tricky Search	12	191	37	6
Tiny Search	9	135	14	4
Small Search	16	262	20	6

Conclusion of Part 1 of assignment:

Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search, we have gathered detailed traversal information for various maze sizes and complexities.

DFS (Depth-First Search):

- In the case of the big maze, DFS achieved a path cost of 210, expanded 427 nodes, reached a maximum depth of 223, and had a maximum fringe size of 36 nodes.
- For the medium and small mazes, DFS performed efficiently, with low node expansion counts and reasonably deep searches.
- However, in the open maze, DFS encountered a significant number of nodes to expand, indicating its vulnerability to maze complexity.

BFS (Breadth-First Search):

- BFS, in general, yielded path costs that were quite competitive with the other algorithms, demonstrating its ability to find shorter paths.
- In all mazes, BFS expanded a larger number of nodes compared to DFS, showcasing its thorough exploration of the search space.
- The maximum depth and fringe size remained relatively low, indicating BFS's effectiveness in maintaining breadth-first exploration.

A* Search:

- A* Search consistently found paths with minimal cost in all mazes, making it a strong performer in terms of path quality.
- The number of nodes expanded was typically lower than in BFS, demonstrating A*'s ability to combine both breadth and depth for efficient searching.
- In the small maze, A* Search reached a maximum depth of 30, indicating deeper exploration compared to other algorithms.

In conclusion, each algorithm has its own strengths and weaknesses. DFS might be suitable for simpler mazes but struggles with complex ones due to its depth-first nature. BFS is thorough and effective in finding shorter paths, but it can consume more memory. A* Search balances path quality and efficiency by considering both cost and heuristic.

Conclusion of Part 2 of assignment:

Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search offer valuable insights into the performance of these search algorithms across different maze layouts

Depth-First Search (DFS):

- For all three maze layouts (Tricky Search, Tiny Search, and Small Search), DFS consistently finds paths with the same path cost as the other algorithms.
- However, it expands a significantly larger number of nodes, resulting in higher computational complexity.
- In terms of depth, DFS reaches the same maximum depth as the other algorithms for each maze.
- The maximum fringe size is generally larger than that of BFS and A*, indicating that DFS keeps a larger number of potential paths in memory during traversal.

Breadth-First Search (BFS):

- BFS performs efficiently in terms of path cost, often finding optimal paths with lower node expansions.
- It offers relatively low computational complexity by expanding fewer nodes compared to DFS and A*.
- The maximum depth reached by BFS is relatively shallow, demonstrating its preference for exploring all possible paths at the current depth level before moving deeper into the search tree.
- The maximum fringe size is moderate, indicating a balanced exploration of the search space.

A* Search:

A* Search consistently finds optimal paths with the lowest path cost, making it the most reliable algorithm in terms of finding the shortest path.

It expands a moderate number of nodes, which is slightly higher than BFS but significantly lower than DFS.

A* reaches the same maximum depth as DFS and BFS in all cases.

The maximum fringe size is often smaller than that of DFS, indicating efficient exploration of the search space.

In conclusion, If finding the shortest path is critical, A* Search is the best option. However, if computational resources are limited, BFS offers a good balance between optimality and efficiency. DFS, while often finding optimal paths, may not be the best choice when computational resources are a concern due to its high node expansion count.

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

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

.