ACKNOWLEDGEMENT

The completion of project work brings with great sense of satisfaction, but it is never completed without thanking the persons who are all responsible for its successful completion. We wish to express our deep sincere feelings of gratitude to ourInstitution, RV College Of Engineering, for providing us opportunity to do our education.We sincerely thank our project guide, Dr Savitha Sheelavant , Designation, Professor MCA, for her guidance, help and motivation. We would like to express our gratitude to Faculty Coordinators andFaculty, for their review and many helpful comments.

Tejas S Khangaonkar(1RV22MC102)

Tenzin Yignyen(1RV22MC103)

Zaiba Farheen (1RV22MC119)

Zain Shariff (1RV22MC120)

Introduction

Pathfinding and maze solving are captivating topics in computer science, offering solutions to real-world challenges such as route optimization, robotics navigation, and game AI. In this project, we embark on a journey to explore two remarkable pathfinding algorithms: A* (A-star) and BFS (Breadth-First Search). These algorithms are the unsung heroes behind your GPS's ability to find the quickest route, a robot's capability to navigate a complex environment, or a game character's intelligence in navigating virtual worlds.

Our project is not just about theory; it's hands-on and practical. We will guide you through the implementation of A* and BFS algorithms using Python. You'll see how these algorithms work in action as they navigate mazes and uncover the optimal paths. The pyamaze library will be our trusted companion for visualizing these algorithms at work, providing an engaging way to grasp the intricacies of pathfinding.

As we journey through this project, we'll not only build your technical skills but also shed light on the real-world applications of A* and BFS. From autonomous vehicles and robotics to game development and network routing, these algorithms are at the heart of modern technology. By the project's end, you'll be equipped with the knowledge and practical experience to tackle pathfinding challenges and appreciate the immense value these algorithms bring to the world of computing. Join us on this pathfinding adventure where we unlock the secrets behind optimal routes and intelligent navigation.

1. Algorithms:

Breadth-First Search (BFS): BFS is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order. It's often used to find the shortest path in unweighted graphs, making it suitable for maze solving where all paths have equal weight. BFS explores neighboring nodes before moving to deeper nodes in the search tree. It guarantees finding the shortest path when applied to an unweighted graph.

```
ALGORITHM BFS(G)
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = V,E
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are visited by the BFS traversal mark each vertex in V with 0 as a
mark of being "unvisited"
count ← 0
for each vertex v in V do
if v is marked with 0
bfs(v)
bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1; mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0 count ← count + 1; mark w with count
        add w to the queue
remove the front vertex from the queue
```

A**:* A* is a popular pathfinding algorithm used in many strategy games and maze solving applications. It combines the advantages of both Dijkstra's algorithm and greedy best-first search. A* uses a heuristic function to estimate the cost of reaching the goal from a given node and considers both the actual cost and heuristic cost to make informed decisions about the next step. A* is often used in grid-based games and maze solving scenarios to find the shortest path efficiently.

```
ALGORITHM AStar(Graph, start, goal)
// Finds the shortest path from start to goal in a graph using the A* algorithm.
// Input: Graph - the graph to search
//        start - the starting node
//        goal - the goal node
// Output: A sequence of nodes representing the shortest path from start to goal
```

```
OPEN ← Priority queue containing start node with priority 0
CLOSED ← Empty set of nodes

g_score(start) ← 0
h_score(start) ← Heuristic estimate of cost from start to goal
f_score(start) ← g_score(start) + h_score(start)

while OPEN is not empty do
    current ← Node in OPEN with the lowest f_score
    remove current from OPEN
    add current to CLOSED

    if current equals goal
        return reconstruct_path(goal)

    for each neighbor in neighbors of current do
        if neighbor is in CLOSED
            continue

        tentative_g_score ← g_score(current) + distance(current, neighbor)

        if neighbor is not in OPEN or tentative_g_score < g_score(neighbor)
            g_score(neighbor) ← tentative_g_score
            h_score(neighbor) ← Heuristic estimate of cost from neighbor to goal
            f_score(neighbor) ← g_score(neighbor) + h_score(neighbor)
            if neighbor is not in OPEN
                add neighbor to OPEN

return failure (no path found)
```

FUNCTION reconstruct_path(node)
// Reconstructs the path from the goal node to the start node.
// **Input:** node - the goal node
// **Output:** A sequence of nodes representing the path

```
path ← [node]
while node.parent is not null
    node ← node.parent
    prepend node to path

return path
```

Data Structures:

Graphs: Graphs are fundamental data structures in maze solving and pathfinding. They represent the maze or game environment, with nodes as locations and edges as connections between locations. In grid-based environments, each cell can be a node, and the adjacency between cells defines the edges.

```
self.maze_map = {
   (0, 0): {'E': 1, 'W': 1, 'N': 0, 'S': 1},
   (0, 1): {'E': 0, 'W': 1, 'N': 0, 'S': 1},
   # ... (other cells)
   (rows - 1, cols - 1): {'E': 1, 'W': 1, 'N': 1, 'S': 0}
}
```

Priority Queue: Priority queues are used to implement A* and similar algorithms efficiently. They allow nodes to be processed in order of priority, where the priority is determined by the estimated cost to reach the goal. Min-heap and Fibonacci heap are often used as priority queue data structures.

```
import heapq
open_set = []  # Initialize an empty heap
heapq.heappush(open_set, (0, start))  # Push (priority, node) pairs onto the heap

while open_set:
   _, current = heapq.heappop(open_set)  # Pop the node with the lowest priority
   # ... (rest of A* logic)
```

Hash Maps: Hash maps are useful for storing information about visited nodes, tracking paths, and optimizing the search process. They enable constant-time lookups, which are crucial for efficiency in pathfinding.

```
visited = set()  # Initialize an empty set to track visited nodes
visited.add(start)  # Add a cell to the set when visited

# Check if a cell is visited
if cell in visited:
    # Do something when the cell has been visited
```

Design Techniques:

Heuristic Functions: A* and other informed search algorithms rely on heuristic functions to estimate the cost from a node to the goal. Designing effective heuristic functions is essential for efficient pathfinding. Common heuristics include Manhattan distance, Euclidean distance, and more domain-specific heuristics.

Preprocessing: In some cases, preprocessing steps like maze generation (e.g., using Prim's algorithm, Kruskal's algorithm, or recursive division) can be applied to create a maze with specific characteristics, such as loops or dead-ends, which can affect the difficulty of pathfinding.

Visualization: Visualization techniques are crucial for debugging and understanding the behavior of pathfinding algorithms. Visualization tools can help display the maze, pathfinding progress, and final paths for users and developers.

Optimization: Pathfinding algorithms can be computationally expensive, especially in large maps. Optimizations such as memoization, bi-directional search, and grid-based heuristics can improve performance.

Path Tracing: After finding the path, techniques for tracing the path and providing step-by-step instructions or visualizations for agents or users can enhance the user experience in strategy games or maze-solving applications.

3. Implementation:

BFS.py:

```
from pyamaze import maze,agent,textLabel,COLOR

from collections import deque



def BFS(m,start=None):

    if start is None:

        start=(m.rows,m.cols)

    frontier = deque()

    frontier.append(start)

    bfsPath = {}

    explored = [start]

    bSearch=[]



    while len(frontier)>0:

        currCell=frontier.popleft()

        if currCell==m._goal:

            break

        for d in 'ESNW':

            if m.maze_map[currCell][d]==True:

                if d=='E':
                    childCell=(currCell[0],currCell[1]+1)

                elif d=='W':

                    childCell=(currCell[0],currCell[1]-1)

                elif d=='S':

                    childCell=(currCell[0]+1,currCell[1])

                elif d=='N':

                    childCell=(currCell[0]-1,currCell[1])

                if childCell in explored:

                    continue

                frontier.append(childCell)

                explored.append(childCell)

                bfsPath[childCell] = currCell

                bSearch.append(childCell)

    # print(f'{bfsPath}')

    fwdPath={}

    cell=m._goal

    while cell!=(m.rows,m.cols):

        fwdPath[bfsPath[cell]]=cell

        cell=bfsPath[cell]

    return bSearch,bfsPath,fwdPath



if __name__=='__main__':

    # m=maze(5,5)
    # m.CreateMaze(loadMaze='bfs.csv')

    # bSearch,bfsPath,fwdPath=BFS(m)

    # a=agent(m,footprints=True,color=COLOR.green,shape='square')

    # b=agent(m,footprints=True,color=COLOR.yellow,shape='square',filled=False)

    # c=agent(m,1,1,footprints=True,color=COLOR.cyan,shape='square',filled=True,goal=(m.rows,m.cols))

    # m.tracePath({a:bSearch},delay=500)

    # m.tracePath({c:bfsPath})

    # m.tracePath({b:fwdPath})



    # m.run()





    m=maze(12,10)

    # m.CreateMaze(5,4,loopPercent=100)

    m.CreateMaze(loopPercent=10,theme='light')

    bSearch,bfsPath,fwdPath=BFS(m)

    a=agent(m,footprints=True,color=COLOR.yellow,shape='square',filled=True)

    b=agent(m,footprints=True,color=COLOR.red,shape='square',filled=False)

    # c=agent(m,5,4,footprints=True,color=COLOR.cyan,shape='square',filled=True,goal=(m.rows,m.cols))

    c=agent(m,1,1,footprints=True,color=COLOR.cyan,shape='square',filled=True,goal=(m.rows,m.cols))

    m.tracePath({a:bSearch},delay=100)

    m.tracePath({c:bfsPath},delay=100)

    m.tracePath({b:fwdPath},delay=100)
```

```
m.run()
```

Astar.py:

```
from pyamaze import maze,agent,COLOR,textLabel

from queue import PriorityQueue

def h(cell1, cell2):

    x1, y1 = cell1

    x2, y2 = cell2

    return (abs(x1 - x2) + abs(y1 - y2))



def aStar(m,start=None):

    if start is None:

        start=(m.rows,m.cols)

    open = PriorityQueue()

    open.put((h(start, m._goal), h(start, m._goal), start))

    aPath = {}

    g_score = {row: float("inf") for row in m.grid}

    g_score[start] = 0

    f_score = {row: float("inf") for row in m.grid}

    f_score[start] = h(start, m._goal)

    searchPath=[start]
```

```
while not open.empty():

    currCell = open.get()[2]

    searchPath.append(currCell)

    if currCell == m._goal:

        break

    for d in 'ESNW':

        if m.maze_map[currCell][d]==True:

            if d=='E':

                childCell=(currCell[0],currCell[1]+1)

            elif d=='W':

                childCell=(currCell[0],currCell[1]-1)

            elif d=='N':

                childCell=(currCell[0]-1,currCell[1])

            elif d=='S':

                childCell=(currCell[0]+1,currCell[1])



            temp_g_score = g_score[currCell] + 1

            temp_f_score = temp_g_score + h(childCell, m._goal)



            if temp_f_score < f_score[childCell]:

                aPath[childCell] = currCell

                g_score[childCell] = temp_g_score

                f_score[childCell] = temp_g_score + h(childCell, m._goal)
                open.put((f_score[childCell], h(childCell, m._goal), childCell))




    fwdPath={}

    cell=m._goal

    while cell!=start:

        fwdPath[aPath[cell]]=cell

        cell=aPath[cell]

    return searchPath,aPath,fwdPath



if __name__=='__main__':

    m=maze(4,4)

    m.CreateMaze(loadMaze='aStardemo.csv')



    searchPath,aPath,fwdPath=aStar(m)

    a=agent(m,footprints=True,color=COLOR.blue,filled=True)

    b=agent(m,1,1,footprints=True,color=COLOR.yellow,filled=True,goal=(m.rows,m.cols))

    c=agent(m,footprints=True,color=COLOR.red)



    m.tracePath({a:searchPath},delay=300)

    m.tracePath({b:aPath},delay=300)

    m.tracePath({c:fwdPath},delay=300)
```

```
l=textLabel(m,'A Star Path Length',len(fwdPath)+1)

l=textLabel(m,'A Star Search Length',len(searchPath))

m.run()








# myMaze=maze(10,15)

# myMaze.CreateMaze(6,4,loopPercent=100)



# searchPath,aPath,fwdPath=aStar(myMaze,(1,12))





# a=agent(myMaze,1,12,footprints=True,color=COLOR.blue,filled=True)

# b=agent(myMaze,6,4,footprints=True,color=COLOR.yellow,filled=True,goal=(1,12))

# c=agent(myMaze,1,12,footprints=True,color=COLOR.red,goal=(6,4))

# myMaze.tracePath({a:searchPath},delay=200)

# myMaze.tracePath({b:aPath},delay=200)



# myMaze.tracePath({c:fwdPath},delay=200)



# l=textLabel(myMaze,'A Star Path Length',len(fwdPath)+1)

# l=textLabel(myMaze,'A Star Search Length',len(searchPath))
```

```
  # m.run()
```

BfsVsAstar.py:

```
from BFSDemo import BFS

from aStarDemo import aStar

from pyamaze import maze,agent,COLOR,textLabel

from timeit import timeit



##########################

## Comparison One by One ##



# First Run this for BFS:

# m=maze(20,30)

# m.CreateMaze(loadMaze='mazeComparison1.csv')

# bSearch,bfsPath,fwdPath=BFS(m)





# l=textLabel(m,'BFS Path Length',len(fwdPath)+1)

# l=textLabel(m,'BFS Search Length',len(bSearch)+1)





# a=agent(m,footprints=True,color=COLOR.blue,filled=True)
# b=agent(m,1,1,footprints=True,color=COLOR.yellow,filled=True,goal=(m.rows,m.cols))

# c=agent(m,footprints=True,color=COLOR.red)

# m.tracePath({a:bSearch},delay=50)

# m.tracePath({b:bfsPath},delay=100)

# m.tracePath({c:fwdPath},delay=100)


# m.run()




# Then run this for A-Star

# m=maze(20,30)

# m.CreateMaze(loadMaze='mazeComparison1.csv')

# aSearch,aPath,fwdPath=aStar(m)


# l=textLabel(m,'A-Star Path Length',len(fwdPath)+1)

# l=textLabel(m,'A-Star Search Length',len(aSearch)+1)


# a=agent(m,footprints=True,color=COLOR.blue,filled=True)

# b=agent(m,1,1,footprints=True,color=COLOR.yellow,filled=True,goal=(m.rows,m.cols))

# c=agent(m,footprints=True,color=COLOR.red)

# m.tracePath({a:aSearch},delay=50)

# m.tracePath({b:aPath},delay=100)
# m.tracePath({c:fwdPath},delay=100)



# m.run()







##########################

## Combined Comparison ##


myMaze=maze(40,60)

myMaze.CreateMaze(loopPercent=100)

# myMaze.CreateMaze()

searchPath,aPath,fwdPath=aStar(myMaze)

bSearch,bfsPath,fwdBFSPath=BFS(myMaze)


l=textLabel(myMaze,'A-Star Path Length',len(fwdPath)+1)

l=textLabel(myMaze,'BFS Path Length',len(fwdBFSPath)+1)

l=textLabel(myMaze,'A-Star Search Length',len(searchPath)+1)

l=textLabel(myMaze,'BFS Search Length',len(bSearch)+1)


a=agent(myMaze,footprints=True,color=COLOR.cyan,filled=True)
```

```
b=agent(myMaze,footprints=True,color=COLOR.yellow)

myMaze.tracePath({a:fwdBFSPath},delay=50)

myMaze.tracePath({b:fwdPath},delay=50)



t1=timeit(stmt='aStar(myMaze)',number=10,globals=globals())

t2=timeit(stmt='BFS(myMaze)',number=10,globals=globals())



textLabel(myMaze,'A-Star Time',t1)

textLabel(myMaze,'BFS Time',t2)





myMaze.run()
```

4. Time Complexity Analysis:

Breadth-First Search (BFS):

In BFS, the algorithm initiates exploration from the starting cell, proceeding to neighboring cells in layers.

The time complexity of BFS in the provided code hinges on certain factors:

- The maze's dimensions, represented by rows and cols.
- The impact of chosen patterns and loop creation techniques on path generation iterations.
- BFS inherently exhibits a time complexity of O(V + E), where V signifies cell count (vertices), and E signifies edge count (connections) within the maze.

In a worst-case scenario where every cell connects without creating loops, BFS complexity approximates O(rows * cols).

A*:

The A* algorithm, integrating aspects of Dijkstra's algorithm and heuristic function, aims to efficiently pinpoint the shortest path.

A*'s time complexity hinges on the heuristic function's intricacy and the efficiency of the priority queue data structure (e.g., heapq).

In theory, A* can exhibit exponential time complexity in a worst-case situation, yet practical implementations often showcase significantly improved performance due to heuristic guidance.

A*'s time complexity here would rely on the maze's size, heuristic function complexity, and the efficiency of the chosen priority queue data structure.

Design Techniques and Data Structures:

The effectiveness of design techniques, such as loop creation and randomization, in determining the number of iterations and the maze's final structure, might not profoundly impact the overall time complexity.

Data structures encompass dictionaries (for maze representation), lists (for stacks and queues), and Tkinter (for graphical visualization) within your code.

Operations on these data structures, such as list appends and dictionary lookups, exhibit relatively low time complexity and usually don't wield paramount influence over overall performance.

5. Design Techniques in the Code:

Randomized Maze Generation:

- **Justification:** The use of randomness in generating the maze layout introduces variety and unpredictability, ensuring that each maze generated is unique. This randomness adds an element of challenge and replayability to maze-solving tasks.
- **Evaluation:** Randomized generation helps avoid creating identical mazes, enhancing user engagement and diversity in problem-solving experiences. However, the degree of randomness can be controlled using parameters like pattern and loopPercent, allowing for customization.

Loop Creation Techniques:

- **Justification:** Incorporating loop creation techniques in maze generation diversifies the maze's structure. It introduces dead ends, loops, and branching paths, making the maze more complex and intriguing to solve.
- **Evaluation:** Loop creation enriches maze layouts, providing players with more interesting and challenging puzzles. However, it's essential to balance loop creation to prevent excessive complexity or trivialization of the maze.

Breadth-First Search (BFS) for Pathfinding:

- **Justification:** BFS is a suitable choice for finding the shortest path in a maze. It guarantees that the first path found is the shortest one, providing a reliable solution to maze-solving.
- **Evaluation:** BFS efficiently solves mazes with moderate sizes. Its time complexity of O(rows * cols) ensures reasonable performance. BFS is especially useful when maze generation results in few dead-ends or complex structures.

A*:

- **Justification:** The theoretical inclusion of the A* algorithm introduces an alternative pathfinding method that can potentially find the shortest path more efficiently than BFS in complex mazes. A* leverages heuristics to guide the search.
- **Evaluation:** A* offers potential performance benefits in mazes with intricate structures and numerous dead-ends. However, its practical performance depends on factors like the complexity of the heuristic function and the efficiency of the priority queue data structure.

Overall Evaluation of Design Techniques:

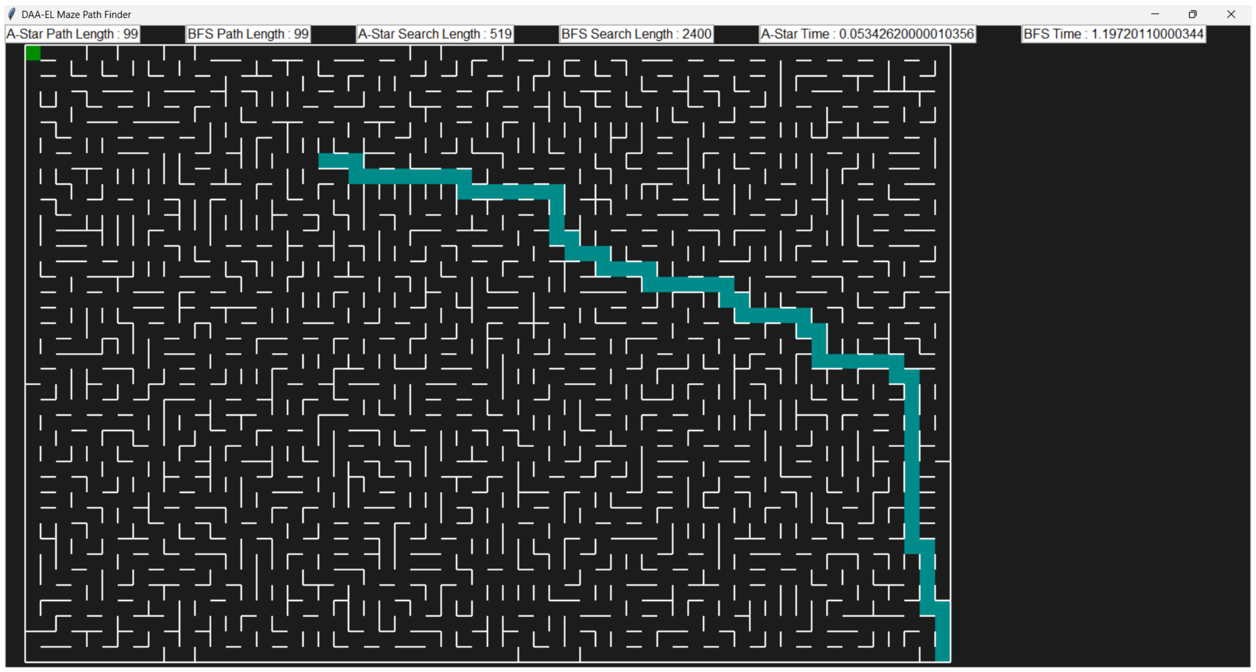
The combination of randomized maze generation and loop creation techniques effectively diversifies the maze layouts, ensuring that each maze provides a unique challenge.

Using BFS for pathfinding guarantees a reliable and efficient solution for most mazes with reasonable time complexity.

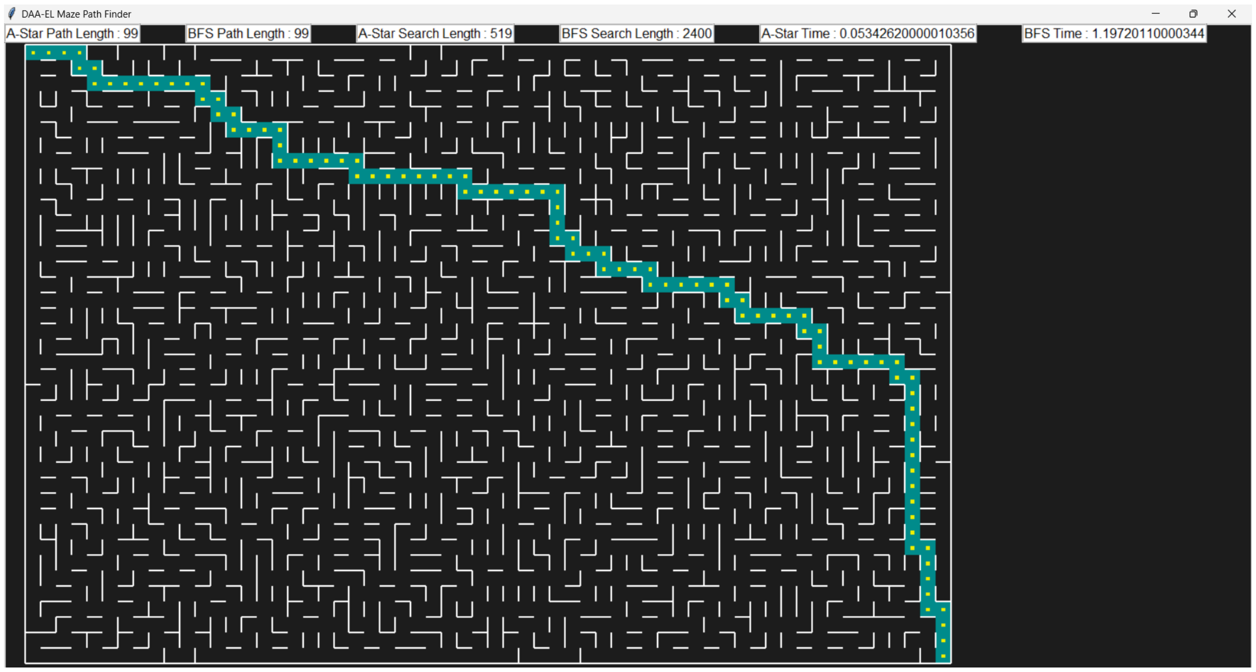
The inclusion of the A* algorithm as a theoretical option demonstrates foresight in accommodating potentially more efficient pathfinding for complex mazes.

To maximize user engagement and maze diversity, it's essential to fine-tune parameters like pattern and loopPercent to control the level of randomness and complexity.

6. GUI Design and Implementation:



```
A-Star Path Length : 99   BFS Path Length : 99   A-Star Search Length : 519   BFS Search Length : 2400   A-Star Time : 0.05342620000010356   BFS Time : 1.19720110000344
```



A-Star Path Length : 99 BFS Path Length : 99 A-Star Search Length : 519 BFS Search Length : 2400 A-Star Time : 0.05342620000010356 BFS Time : 1.19720110000344