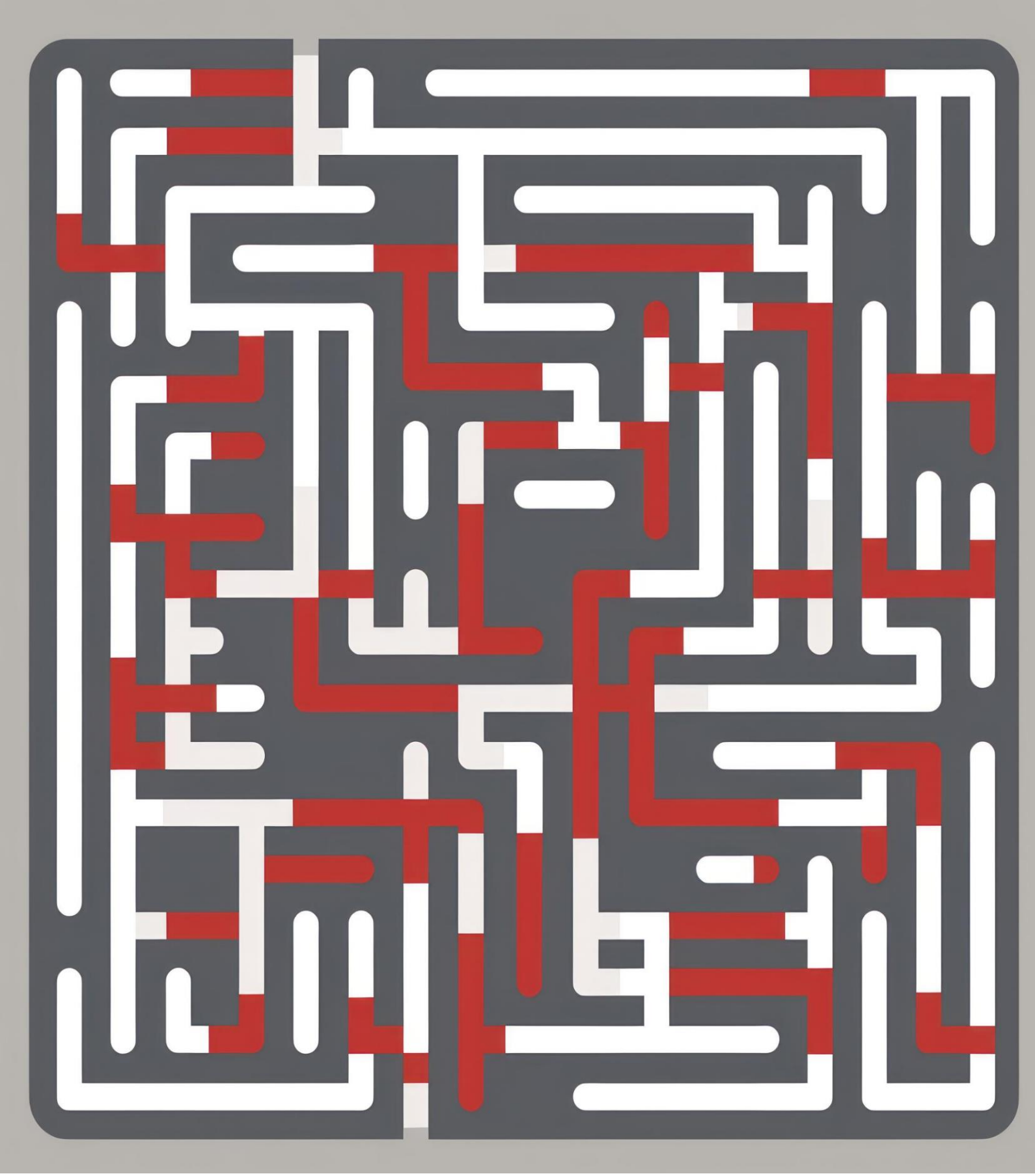


# Maze Solver

# Maze Solving Algorithms in AI

Course: CCE414(Artificial Intelligence)
Term: 242


| S# | Student Name | Student ID |
|---|---|---|
| 1 | Yousef Hossam Abdell-Fattah | 221903003 |
| 2 | Youssif Sami Hassan | 221902969 |
| 3 | Youssif Sherif Badawy | 221902970 |
| 4 | Mahmoud Magdy Mahmoud | 221902949 |
| 5 | Yousef El-Moatz Bellah | 221902967 |
| 6 | Hajer Emad Abdell-Qader | 221902962 |


| **Marks** | |
|---|---|
| Free Topic Description | **/40** |

| | | |
|---|---|---|
| Problem solving as a search problem | Problem formulation | **/40** |
| | Implementation/Evaluation /Sample output | **/60** |
| | Project interface & Poster Design | **/25** |
| | Report Style and Formatting | **/15** |
| | Presentation / response to questions | **/20** |
| | Creativity (**Bonus**) | **/10** |

| | | |
|---|---|---|
| Expert System In Prolog | Implementation/Evaluation /Sample output | **/60** |
| | Project interface & Poster Design | **/40** |

| **Total** | **/300** |
|---|---|

Submitted in partial fulfillment of the requirements of supervisor as a project in artificial intelligence study in computer and communications engineering program at Shoubra Faculty of Engineering Benha University, Cairo, Egypt

April 2024

# Disclaimer

The information and algorithms provided in this report are intended for educational purposes only. While efforts have been made to ensure the accuracy of the content, the authors make no guarantees regarding the completeness, reliability, or suitability of the information presented. The use of these algorithms for solving mazes or any other application is at the reader's own risk.

Additionally, the authors do not assume any responsibility for the consequences of using or misusing the algorithms, nor do they endorse any specific implementation or usage thereof. Users are encouraged to exercise caution and verify the correctness of the algorithms before applying them in practical scenarios.

Furthermore, this report may reference external sources or third-party materials for supplementary information. The authors do not endorse or take responsibility for the content of such external sources.

The contents of this report are subject to change without notice. The authors disclaim any liability for any loss or damage, including but not limited to indirect or consequential loss or damage, arising from the use of this report or its contents.

Signature: ……. Date: …….

Signature: ……. Date: …….

Signature: ……. Date: …….

Signature: ……. Date: …….

Signature: ……. Date: …….

Signature: ……. Date: …….

# Abstract

Maze solving is a classic problem in the field of artificial intelligence, requiring the development of efficient algorithms to navigate through complex structures. This report explores various search algorithms commonly employed to solve maze navigation problems. We discuss the problem formulation, the principles behind each algorithm, their implementation, and evaluate their effectiveness in finding optimal solution.

# Acknowledgment

We're over helmed in all humbleness and gratefulness to acknowledge my depth to all those who have helped me to put these ideas, well above the level of simplicity and into something concrete.

We would like to express my special thanks of gratitude to our teacher (Dr. Lamiaa El-reffai) as well as our principal who gave me the golden opportunity to do this wonderful project on the topic (Maze Solver Algorithms), which also helped me in doing a lot of research and we came to know about so many new things. We're thankful to them.

Any attempt at any level can't be satisfactorily completed without the support and guidance of my parents and friends.

We would like to thank my parents who helped me a lot in gathering different information, collecting data, and guiding me from time to time in making this project, despite their busy schedules, they gave me different ideas in making this project unique.

THANKS AGAIN TO ALL WHO HELPED US

# Contents

# 1. Introduction

Maze solving is a fundamental problem in the domain of artificial intelligence, with applications ranging from robotics to game design. The objective is to find a path from a designated start point to a goal point within a maze while navigating through a series of interconnected passages and obstacles. This report aims to explore and analyze different search algorithms used to solve maze navigation problems, highlighting their strengths, weaknesses, and implementation details.

# 2. Agent Type

The agent in the maze solving problem is typically an autonomous entity capable of perceiving its environment, making decisions based on available information, and executing actions to achieve its objective. The agent's goal is to navigate through the maze from the starting position to the goal position while avoiding obstacles, so It’s a Goal-based Agent.

# 3. Task Environment

- **Performance Measure:** The performance measure evaluates the efficiency and effectiveness of the agent's behavior in solving the maze. It may be defined in terms of factors such as path length (minimizing the number of stepstaken), computational resources utilized (minimizing time and memory requirements), and optimality of the solution (finding the shortest path).
- **Environment:** The environment consists of the maze itself, represented as a grid or graph structure, with obstacles denoted as impassable cells or edges. The maze environment is dynamic, allowing the agent to perceive its current state, execute actions, and observe the resulting state transitions.

- **Actuators:** The agent's actuators are responsible for executing the selected actions within the maze environment. These may include movement mechanisms for navigating through the maze, such as motorized wheels or simulated movement commands.
- **Sensors:** The agent's sensors gather information about its current state and surroundings within the maze environment. This includes detecting obstacles, identifying the agent's current position, and perceiving the proximity to the goal state.

# 4. Properties of Task Environment

- **Partially Observable:** The agent can't observe its' surroundings within the maze environment through its sensor.
- **Deterministic:** The effects of the agent's actions on the environment are deterministic, meaning that given a specific state and action, the resulting state transition is predictable and consistent.
- **Sequential:** The current decisions affect all the future decisions, so we consider it as a sequential environment.
- **Discrete**: The maze environment is typically represented discretely, with distinct states, actions, and transitions defined within a finite grid or graph structure.
- **Static:** The maze environment is static, meaning that it remains unchanged over the course of the agent's exploration and navigation. However, the agent's actions may dynamically affect its state within the maze.
- **Single agent:** The agent operating by itself in an environment.

# 5. Searching Algorithms Used

In this report, we focus on the following search algorithms commonly used for maze solving:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. A* Search
4. Wall Follower
5. Dijkstra's Algorithm
6. Uniform Cost Search (UCS)

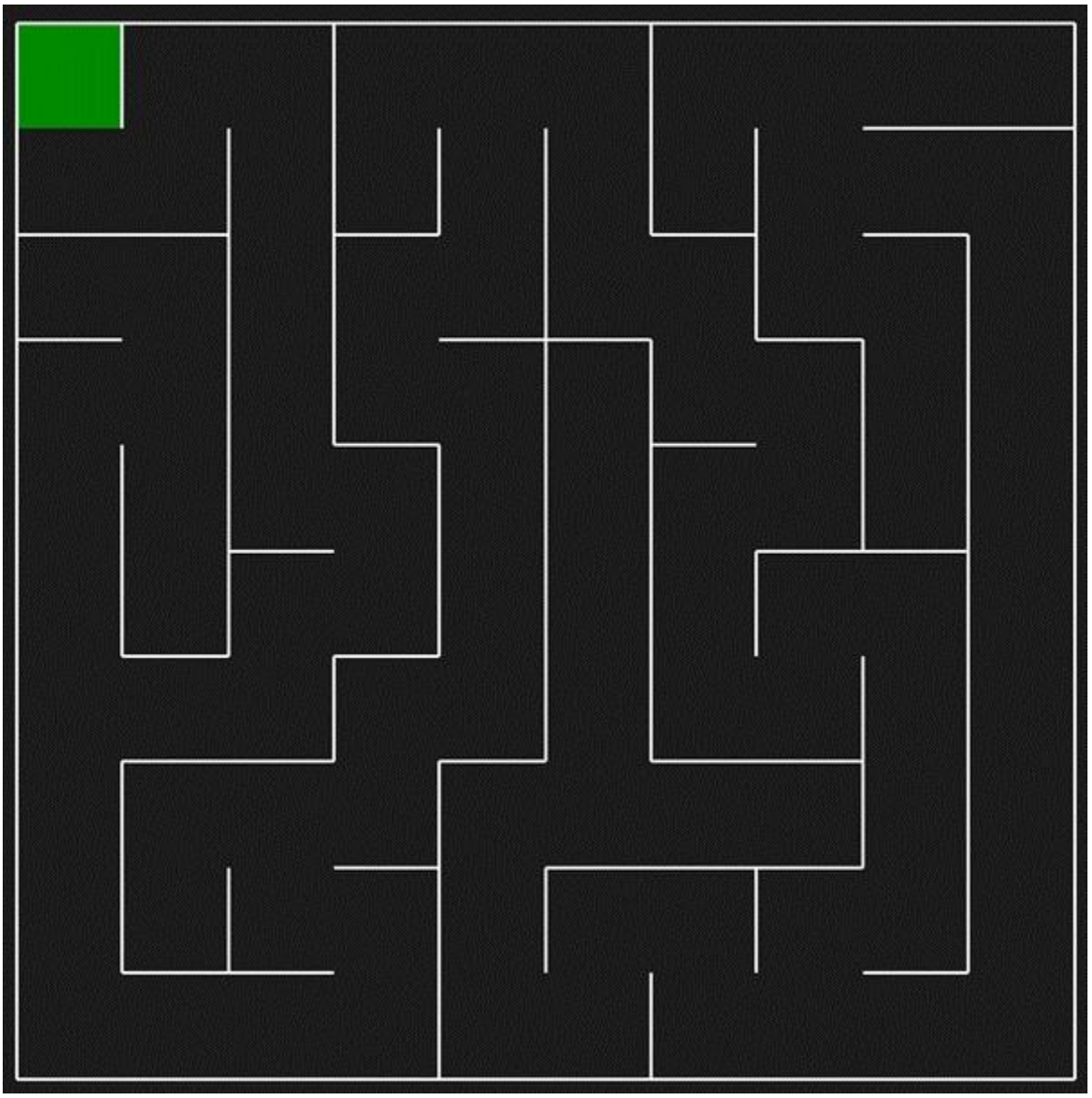
Each algorithm employs a different strategy for traversing the maze and finding the optimal path from the start to the goal state.

# 6. Implementation

## 1. Graphical User Interface (GUI)

a. **Implementation:** Developed a user-friendly interface facilitating interactive exploration of maze-solving algorithms.

b. **Output:** Offers seamless application and comparison of six distinct algorithms on mazes of varying complexities.

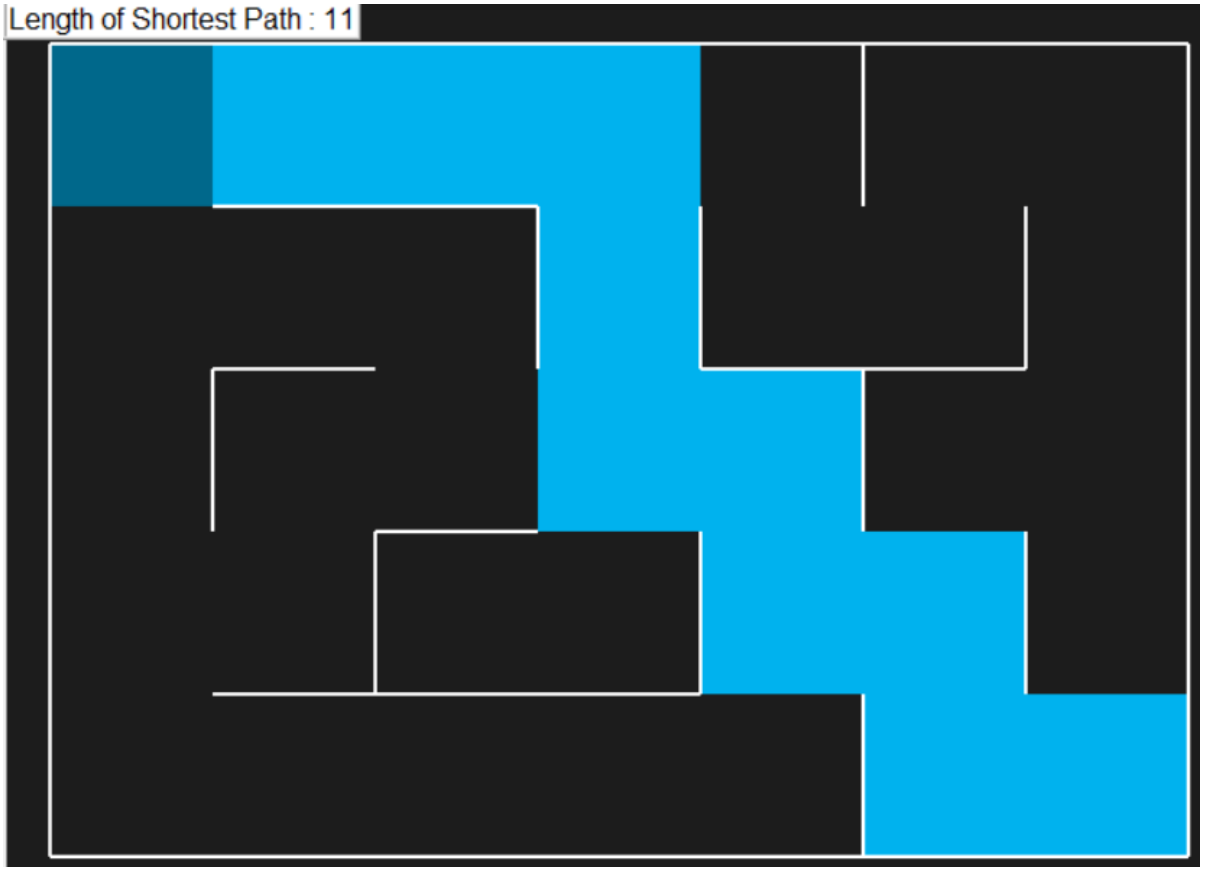


## 2. Breadth First Search (BFS)

a. **Implementation:** Executed BFS algorithm, ensuring systematic exploration of maze paths layer by layer.

```python
from pyamaze import maze,agent,COLOR,textLabel
def BFS(m):
    start=(m.rows,m.cols)
    frontier=[start]
    explored=[start]
    bfsPath={}
    while len(frontier)>0:
        currCell=frontier.pop(0)
        if currCell==(1,1):
            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])
                if childCell in explored:
                    continue
                frontier.append(childCell)
                explored.append(childCell)
                bfsPath[childCell]=currCell
    fwdPath={}
    cell=(1,1)
    while cell!=start:
        fwdPath[bfsPath[cell]]=cell
        cell=bfsPath[cell]
    return fwdPath

if __name__=='__main__':
    m=maze(5,7)
    m.CreateMaze(loopPercent=40)
    path=BFS(m)

    a=agent(m,footprints=True,filled=True)
    m.tracePath({a:path})
    l=textLabel(m,'Length of Shortest Path',len(path)+1)

    m.run()
```

b. **Output:** Provides optimal pathfinding solutions with an emphasis on completeness and minimal steps.

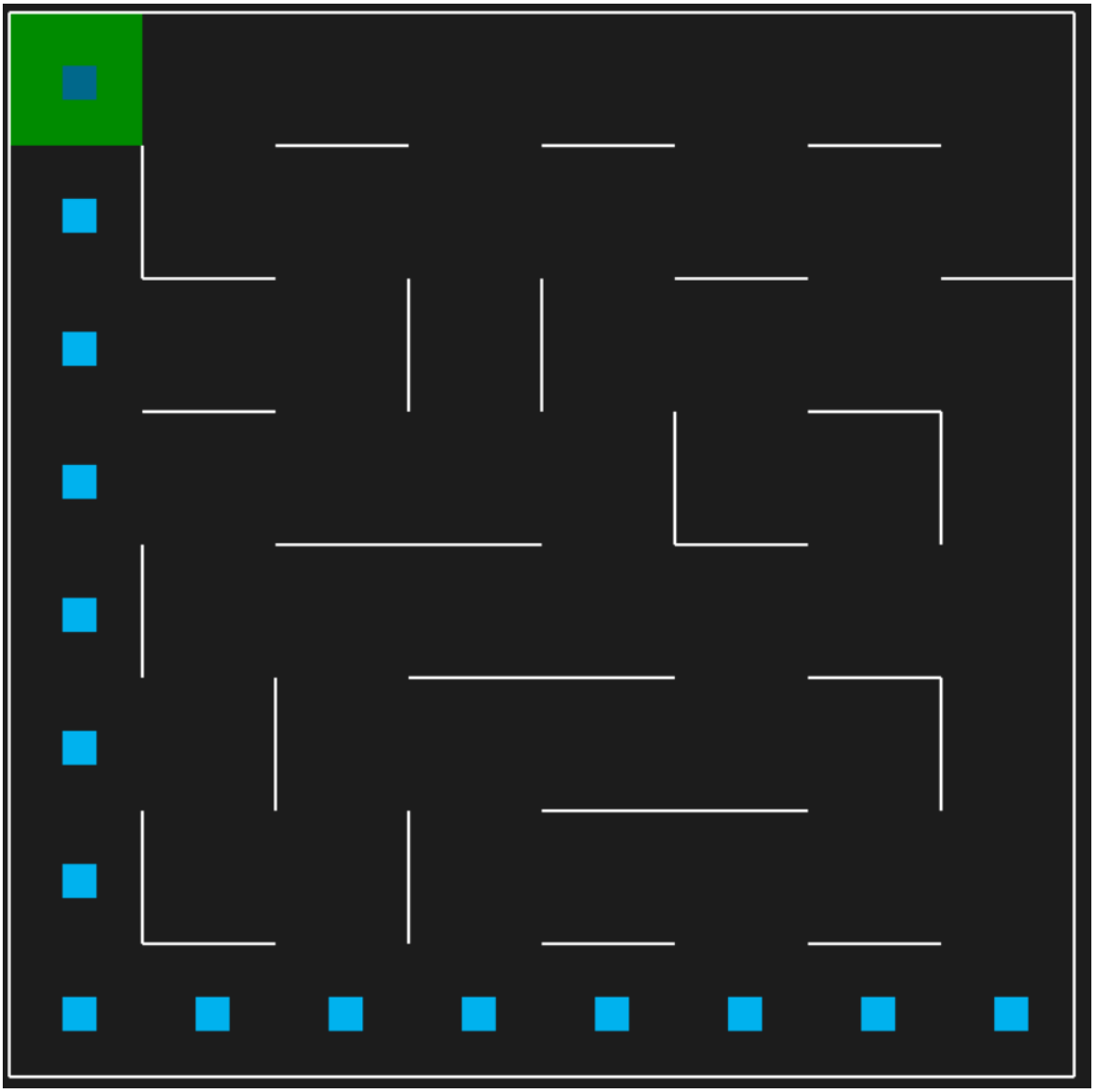


## 3. Depth First Search (DFS)

a. **Implementation:** Integrated DFS algorithm for maze traversal, prioritizing exploration depth before breadth.

```python
from pyamaze import maze,agent,COLOR
def DFS(m):
    start=(m.rows,m.cols)
    explored=[start]
    frontier=[start]
    dfsPath={}
    while len(frontier)>0:
        currCell=frontier.pop()
        if currCell==(1,1):
            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
                explored.append(childCell)
                frontier.append(childCell)
                dfsPath[childCell]=currCell
    fwdPath={}
    cell=(1,1)
    while cell!=start:
        fwdPath[dfsPath[cell]]=cell
        cell=dfsPath[cell]
    return fwdPath


if __name__=='__main__':
    m=maze(15,10)
    m.CreateMaze(loopPercent=100)
    path=DFS(m)
    a=agent(m,footprints=True)
    m.tracePath({a:path})


    m.run()
```

b. **Output:** Offers solutions emphasizing depth-first exploration, often useful for maze generation and analysis.

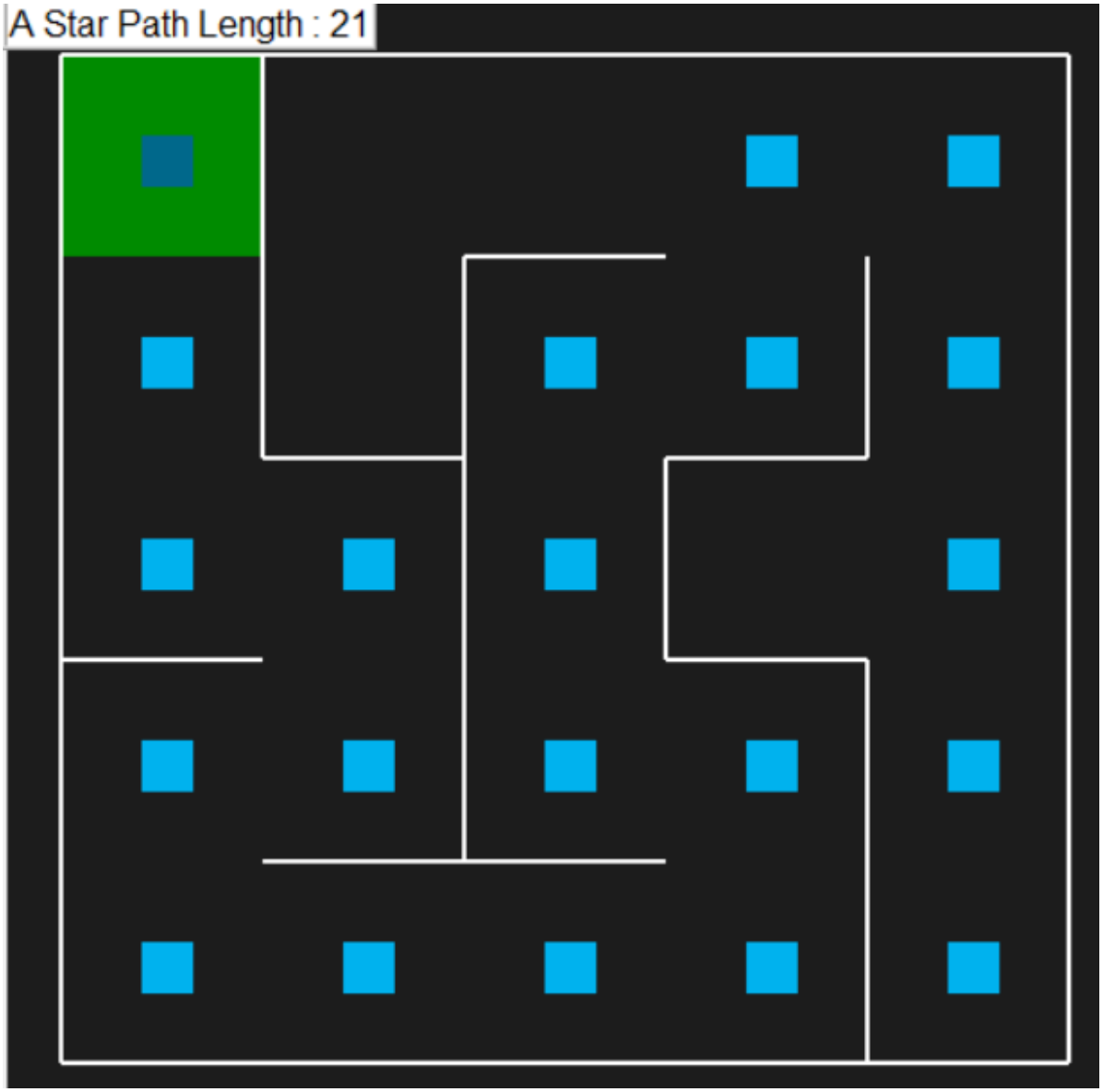


## 4. A Search*

a. **Implementation:** Implemented A* algorithm, incorporating heuristic estimates for efficient pathfinding.

```python
from pyamaze import maze,agent,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=(m.rows,m.cols)
    g_score={cell:float('inf') for cell in m.grid}
    g_score[start]=0
    f_score={cell:float('inf') for cell in m.grid}
    f_score[start]=h(start,(1,1))

    open=PriorityQueue()
    open.put((h(start,(1,1)),h(start,(1,1)),start))
    aPath={}
    while not open.empty():
        currCell=open.get()[2]
        if currCell==(1,1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d]==True:
                if d=='E':
                    childCell=(currCell[0],currCell[1]+1)
                if d=='W':
                    childCell=(currCell[0],currCell[1]-1)
                if d=='N':
                    childCell=(currCell[0]-1,currCell[1])
                if d=='S':
                    childCell=(currCell[0]+1,currCell[1])

                temp_g_score=g_score[currCell]+1
                temp_f_score=temp_g_score+h(childCell,(1,1))

                if temp_f_score < f_score[childCell]:
                    g_score[childCell]= temp_g_score
                    f_score[childCell]= temp_f_score
                    open.put((temp_f_score,h(childCell,(1,1)),childCell))
                    aPath[childCell]=currCell
    fwdPath={}
    cell=(1,1)
    while cell!=start:
        fwdPath[aPath[cell]]=cell
        cell=aPath[cell]
    return fwdPath

if __name__=='__main__':
    m=maze(5,5)
    m.CreateMaze()
    path=aStar(m)

    a=agent(m,footprints=True)
    m.tracePath({a:path})
    l=textLabel(m,'A Star Path Length',len(path)+1)

    m.run()
```

b. **Output:** Delivers optimal solutions by balancing the cost of the path and the estimated remaining cost to the goal.

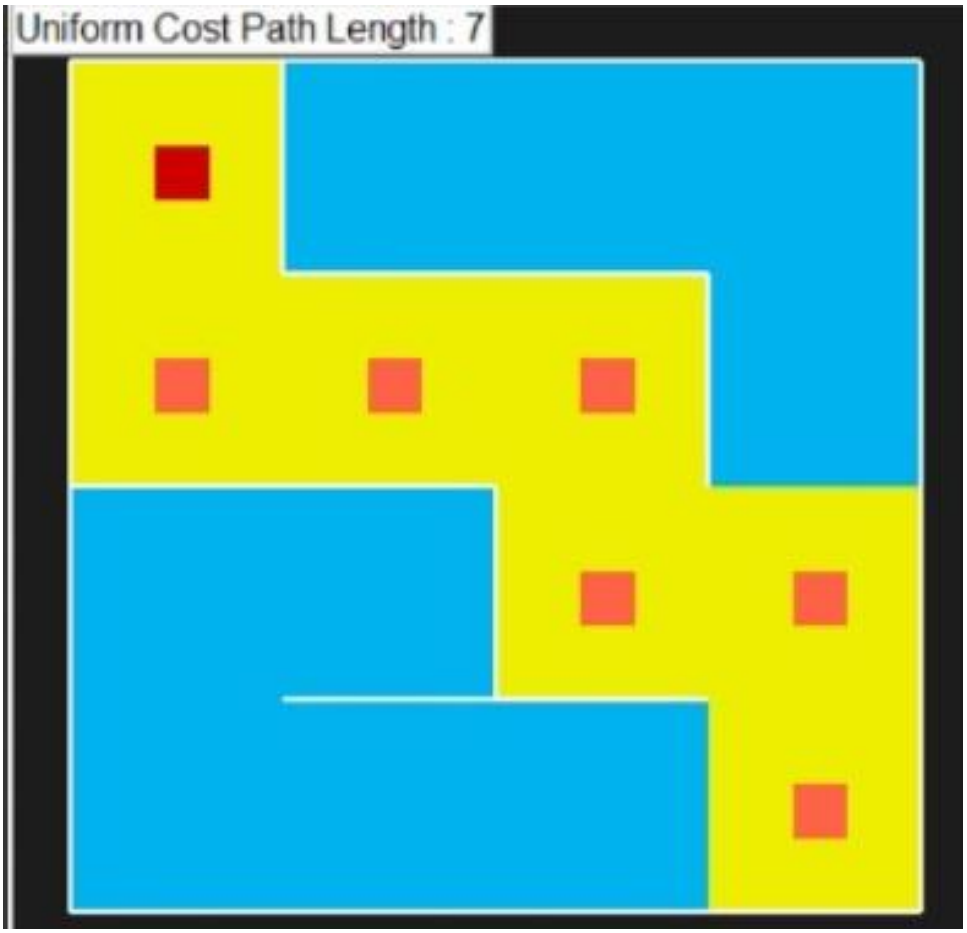


## 5. Uniform Cost Search (UCS)

a. **Implementation:** Utilized UCS algorithm, prioritizing paths with the least cumulative cost.

```python
from pyMaze import maze, agent, COLOR, textLabel
from queue import PriorityQueue

def cost(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return abs(x1 - x2) + abs(y1 - y2)

def uniform_cost(m, start=None):
    if start is None:
        start = (m.rows, m.cols)
    open = PriorityQueue()
    open.put((0, start))
    aPath = {}
    searchPath = [start]
    while not open.empty():
        _, currCell = open.get()
        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])

                new_cost = cost(start, currCell) + 1
                if childCell not in aPath or new_cost < cost(start, aPath[childCell]):
                    aPath[childCell] = currCell
                    open.put((new_cost, 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()

    # Set the goal manually when creating the agent
    a_goal = (m.rows, m.cols)

    searchPath, aPath, fwdPath = uniform_cost(m)
    a = agent(m, footprints=True, color=COLOR.blue, filled=True)
    b = agent(m, 1, 1, footprints=True, color=COLOR.yellow, filled=True, goal=a_goal)
    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, 'Uniform Cost Path Length', len(fwdPath) + 1)
    l = textLabel(m, 'Uniform Cost Search Length', len(searchPath))
    m.run()
```

b. **Output:** Provides optimal solutions by systematically exploring paths with incremental costs.

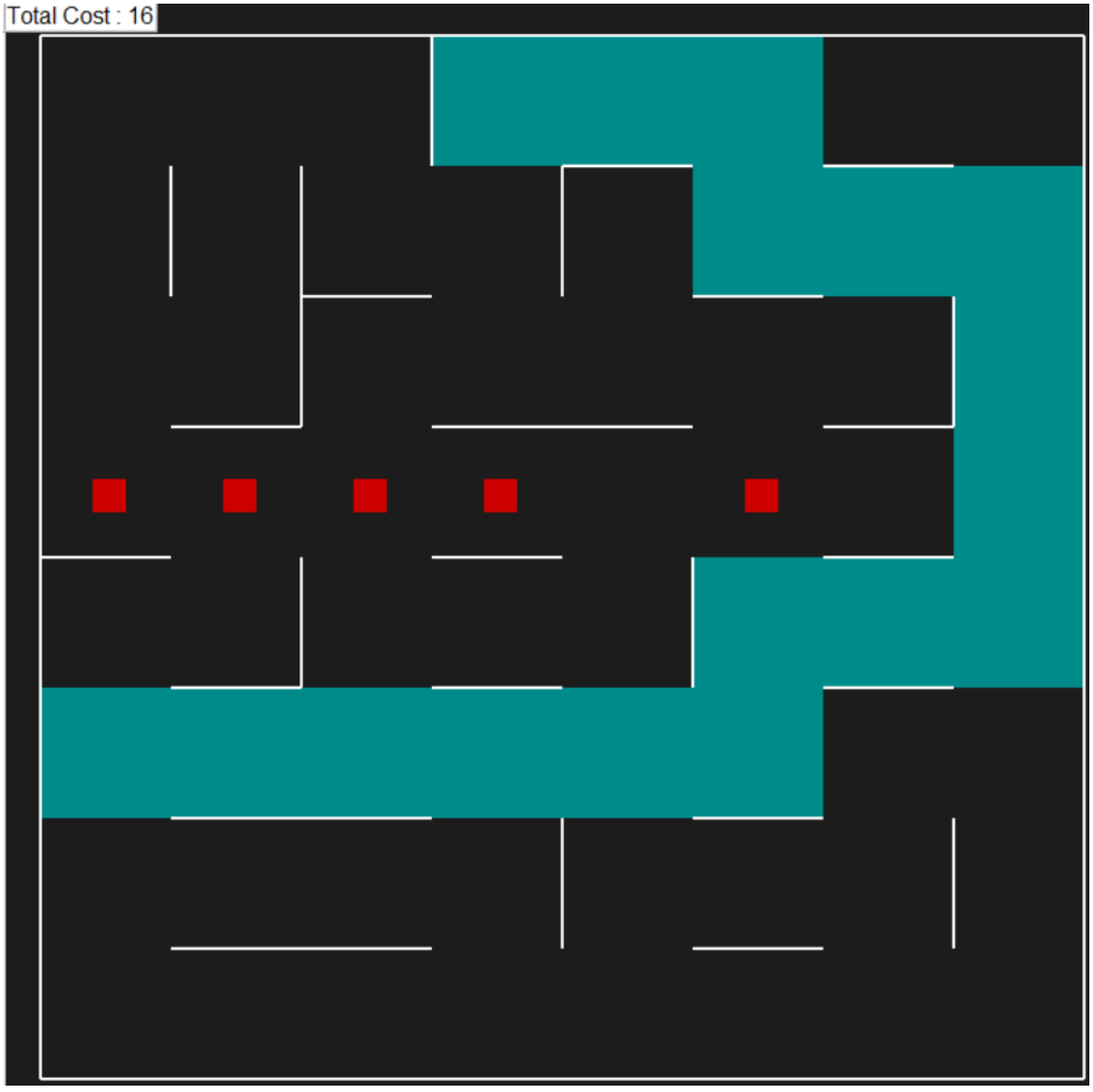


## 6. Dijkstra's Algorithm

a. **Implementation:** Employed Dijkstra's algorithm for finding the shortest paths from a single source node.

```python
from pyamaze import maze,agent,COLOR,textLabel
def dijkstra(m,*h,start=None):
    if start is None:
        start=(m.rows,m.cols)
    hurdles=[(i.position,i.cost) for i in h]
    unvisited={n:float('inf') for n in m.grid}
    unvisited[start]=0
    visited={}
    revPath={}
    while unvisited:
        currCell=min(unvisited,key=unvisited.get)
        visited[currCell]=unvisited[currCell]
        if currCell==m._goal:
            break
        for d in 'EWNS':
            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 visited:
                    continue
                tempDist= unvisited[currCell]+1
                for hurdle in hurdles:
                    if hurdle[0]==currCell:
                        tempDist+=hurdle[1]
                if tempDist < unvisited[childCell]:
                    unvisited[childCell]=tempDist
                    revPath[childCell]=currCell
        unvisited.pop(currCell)
    fwdPath={}
    cell=m._goal
    while cell!=start:
        fwdPath[revPath[cell]]=cell
        cell=revPath[cell]
    return fwdPath,visited[m._goal]
if __name__=='__main__':
    myMaze=maze(8,8)
    myMaze.CreateMaze(1,4,loopPercent=100)
    # myMaze.CreateMaze(loadMaze='dijkMaze.csv')
    h1=agent(myMaze,4,4,color=COLOR.red)
    h2=agent(myMaze,4,6,color=COLOR.red)
    h3=agent(myMaze,4,1,color=COLOR.red)
    h4=agent(myMaze,4,2,color=COLOR.red)
    h5=agent(myMaze,4,3,color=COLOR.red)
    h1.cost=100
    h2.cost=100
    h3.cost=100
    h4.cost=100
    h5.cost=100
    # path,c=dijstra(myMaze,h1,h2,h2,h3,h4,h5)
    path,c=dijkstra(myMaze,h1,h2,h3,h4,h5,start=(6,1))
    textLabel(myMaze,'Total Cost',c)
    # a=agent(myMaze,color=COLOR.cyan,filled=True,footprints=True)
    a=agent(myMaze,6,1,color=COLOR.cyan,filled=True,footprints=True)
    myMaze.tracePath({a:path})
    myMaze.run()
```

b. **Output:** Outputs optimal paths by exploring all possible routes and selecting the shortest path.

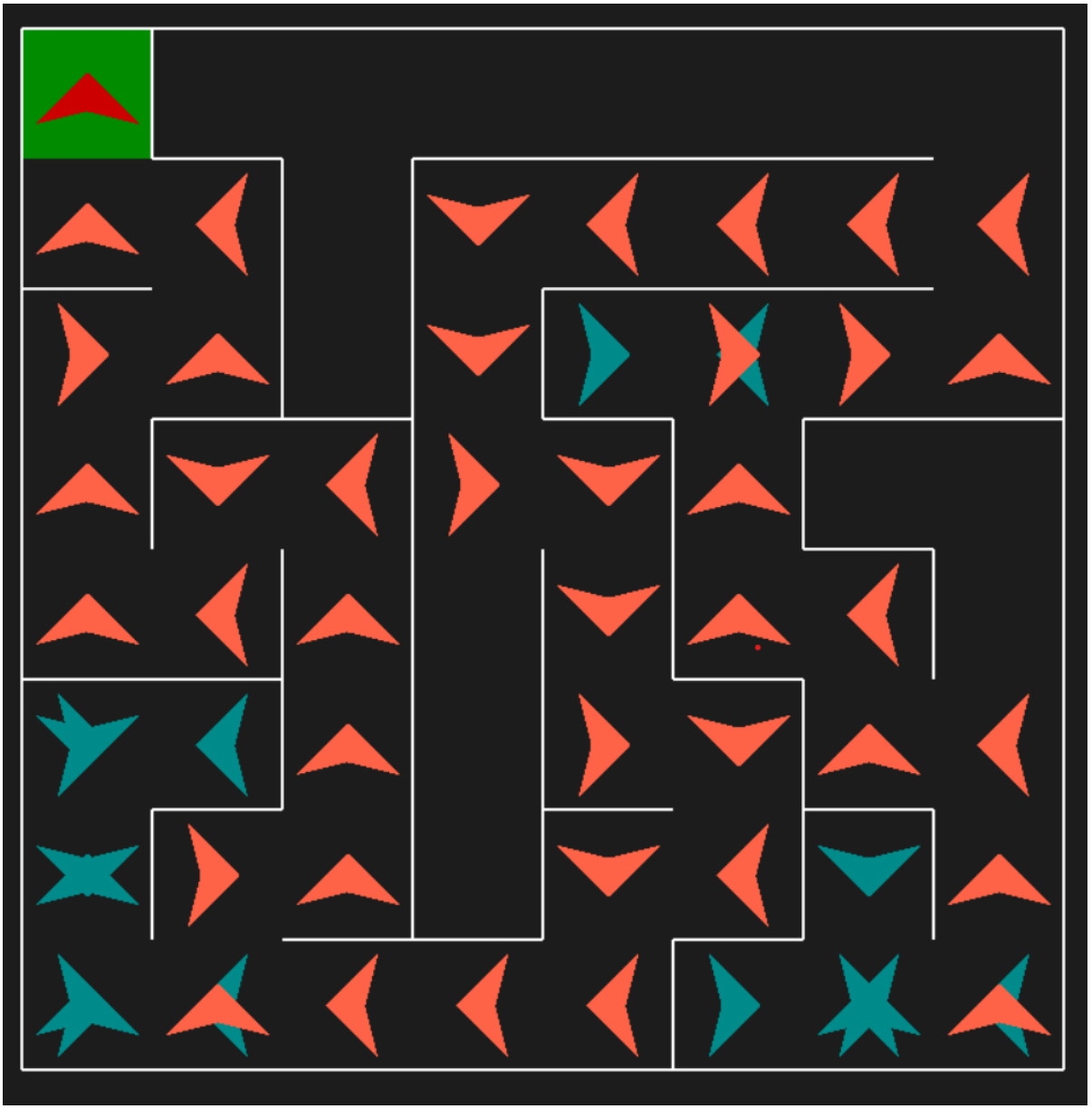


## 7. Wall Follower

a. **Implementation:** Developed Wall Follower algorithm for maze traversal, relying on following the walls of the maze.

```python
from pyMaze import maze, agent, COLOR


def RCW():
    global direction
    k = list(direction.keys())
    v = list(direction.values())
    v_rotated = [v[-1]] + v[:-1]
    direction = dict(zip(k, v_rotated))


def RCCW():
    global direction
    k = list(direction.keys())
    v = list(direction.values())
    v_rotated = v[1:] + [v[0]]
    direction = dict(zip(k, v_rotated))


def moveForward(cell):
    if direction['forward'] == 'E':
        return (cell[0], cell[1] + 1), 'E'
    if direction['forward'] == 'W':
        return (cell[0], cell[1] - 1), 'W'
    if direction['forward'] == 'N':
        return (cell[0] - 1, cell[1]), 'N'
    if direction['forward'] == 'S':
        return (cell[0] + 1, cell[1]), 'S'


def wallFollower(m):
    global direction
    direction = {'forward': 'N', 'left': 'W', 'back': 'S', 'right': 'E'}
    currCell = (m.rows, m.cols)
    path = ''
    while True:

        if currCell == (1, 1):
            break
        if m.maze_map[currCell][direction['left']] == 0:
            if m.maze_map[currCell][direction['forward']] == 0:
                RCW()
            else:
                currCell, d = moveForward(currCell)
                path += d
        else:
            RCCW()
            currCell, d = moveForward(currCell)
            path += d
    path2 = path
    while 'EW' in path2 or 'WE' in path2 or 'NS' in path2 or 'SN' in path2:
        path2 = path2.replace('EW', '')
        path2 = path2.replace('WE', '')
        path2 = path2.replace('NS', '')
        path2 = path2.replace('SN', '')
    return path, path2


if __name__ == '__main__':
    myMaze = maze(8, 8)
    myMaze.CreateMaze()


    a=agent(myMaze,shape='arrow',footprints=True , color=COLOR.cyan)
    b = agent(myMaze, shape='arrow', footprints=True,color=COLOR.red)
    path, path2 = wallFollower(myMaze)
    myMaze.tracePath({a:path})
    myMaze.tracePath({b: path2})

    print(path)
    myMaze.run()
```

b. **Output:** Offers a unique approach to maze-solving by continuously tracing the walls, often effective in certain maze configurations.



# 7. Future Enhancements

To further improve the system, future enhancements may include:

- **Custom Maze Design:** Allow users to create their own mazes easily by adding walls and paths with a simple drag-and-drop interface.
- **Algorithm Comparison:** Include a feature that lets users select different algorithms and see how they perform on the same maze, helping them understand which algorithm works best in different situations.
- **Interactive Learning:** Provide tooltips and explanations throughout the GUI to help users understand each algorithm's behavior and how they can tweak arameters for better results.

# Conclusion

In conclusion, maze solving algorithms play a vital role in artificial intelligence applications requiring pathfinding and navigation through complex environments. By implementing and evaluating various search algorithms, we gain insights into their performance characteristics, computational complexity, and suitability for different maze structures. Understanding the strengths and weaknesses of each algorithm allows us to choose the most appropriate approach based on the specific requirements of the problem at hand. Further research may explore enhancements and optimizations to existing algorithms or develop novel approaches for more efficient and effective maze solving.

# Task Assignment

| S# | Student Name | Student Email | Student ID | Tasks | % |
|---|---|---|---|---|---|
| **1** | Yousef Hossam Abd-Elfattah | Youssef20306@feng.bu.edu.eg | 221903003 | Breadth First | 16.67 |
| **2** | Yousef Samy Hassan | Youssef20966@feng.bu.edu.eg | 221902969 | A* | 16.67 |
| **3** | Yousef Sherif Badawy | Youssef20967@feng.bu.edu.eg | 221902970 | Dijkstra's Algorithm | 16.67 |
| **4** | Mahmoud Magdy Mahmoud | Mahmoud20162@feng.bu.edu.eg | 221902949 | Depth First | 16.67 |
| **5** | Yousef El-Moatz Bellah | Yusef20302@feng.bu.edu.eg | 221902967 | Uniform Cost | 16.67 |
| **6** | Hajer Emad Abd-ElQader | Hajar20264@feng.bu.edu.eg | 221902962 | Wall Follower | 16.67 |

# References


- https://github.com/MAN1986/pyamaze
- https://www.youtube.com/watch?v=pNWCCHFw7og
- https://www.youtube.com/watch?v=vAvPkrLbczg
- https://www.youtube.com/watch?v=rUkj_Wiurd