Knight’s travails

ROBOT NAVIGATION PUZZLE

Table of Contents

Robot Navigation Puzzle Game

Abstract

The Robot Navigation Puzzle Game is an interactive simulation designed to explore fundamental pathfinding algorithms, including Breadth-First Search (BFS) and Depth-First Search (DFS). This project integrates algorithmic concepts with a graphical user interface (GUI) developed using Python's Tkinter library. The application enables users to set obstacles, define start and end points, and visualize the robot's traversal path, ensuring educational insights and engaging interaction.

1. Introduction

Background

Pathfinding is a critical component of computer science, used in applications ranging from robotics to video games. This project demonstrates BFS and DFS, two foundational algorithms, by simulating a robot navigating a grid-based environment.

Purpose

This project aims to create an educational tool to understand and visualize pathfinding algorithms interactively. By leveraging animations, users can comprehend algorithm behavior and compare performance in real-time.

2. Objectives

1. Implement BFS and DFS algorithms for pathfinding.
2. Create a dynamic grid-based environment allowing user-defined obstacles.
3. Visualize the robot's movement with animations and color-coded paths.
4. Ensure robustness against varying grid configurations.
5. Provide insights into algorithm performance and behavior.

3. Features

Core Functionalities

1. **Grid Customization**:

- Adjustable 10x10 grid.
- Users can set obstacles, start, and end points.

2. **Algorithm Selection**:

- Choose between BFS and DFS for pathfinding.

3.**Visual Feedback**:

- Color-coded paths for BFS (blue) and DFS (green).
- Smooth robot animation leaving visual trails.

4.**Responsive Design**:

- Intuitive mouse interactions for setting grid elements.

5,.**Obstacle Handling**:

- Accurate navigation around user-defined obstacles.

4. System Design

Architecture

1. **Frontend**:

- Tkinter for GUI elements, grid visualization, and animations.

2. **Backend**:

- BFS and DFS algorithms implemented with Python.
- State management for grid and robot movement.

Components

1. **Grid Representation**:

- 2D list initialized with zeros (empty cells).
- Obstacles marked as ones.

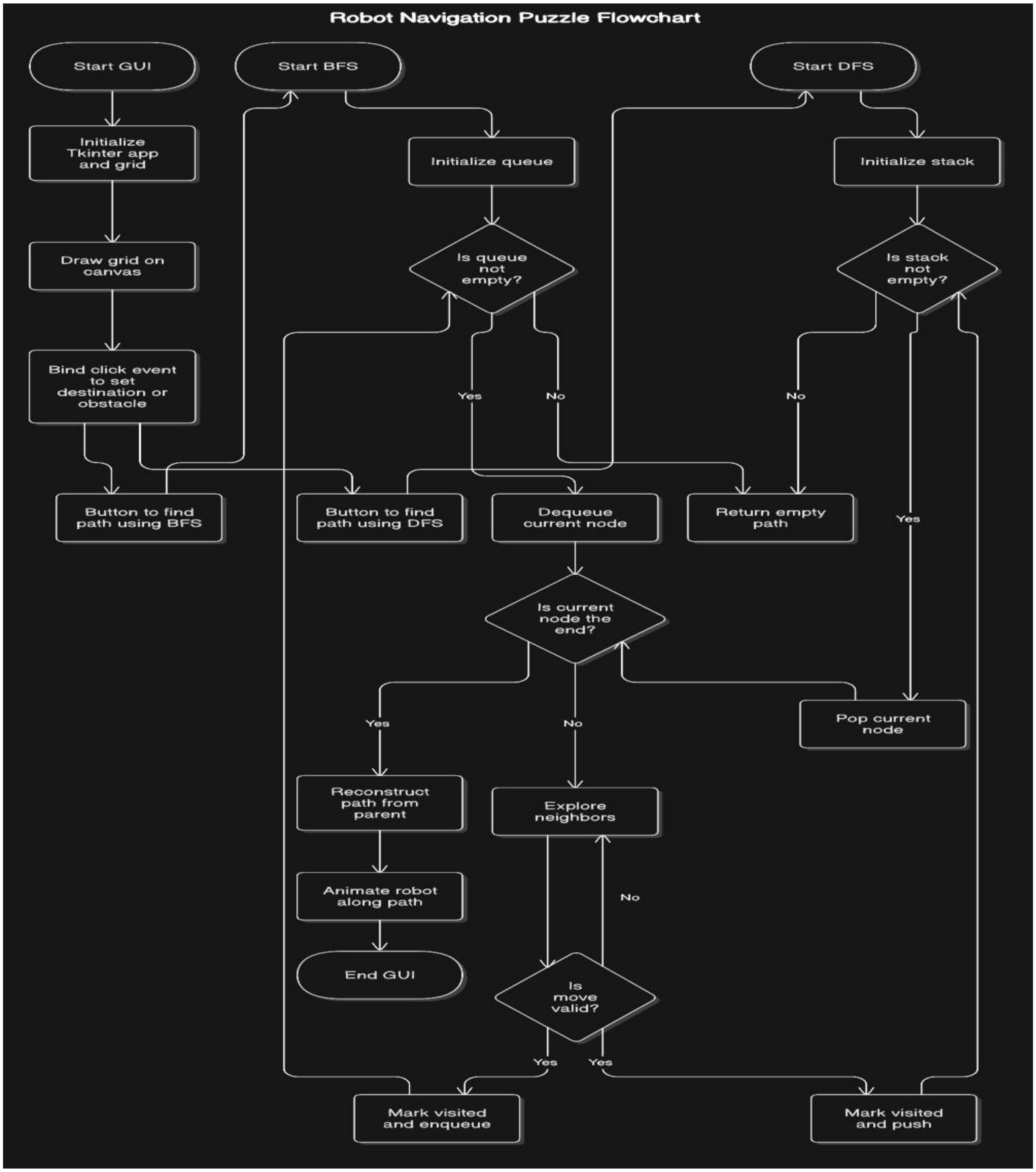
2.**Robot Movement**:

- Animated rectangle and circle combination representing the robot.

- Tracks visited cells, leaving trails.

1.**Pathfinding Algorithms**:

- BFS: Ensures shortest path.
- DFS: Explores depth-first traversal.



5. Algorithms

Breadth-First Search (BFS)

1. Approach:

- Uses a queue for level-order exploration.
- Ensures shortest path by expanding neighbors systematically.

1.Steps:

- Initialize queue with the start point.
- Mark visited cells.
- Record parent relationships for path reconstruction.
- Stop when the endpoint is reached.

Depth-First Search (DFS)

1.Approach:

- Uses a stack for depth-first exploration.
- Explores deeper paths before backtracking.

2.Steps:

- Initialize stack with the start point.
- Mark visited cells.
- Record parent relationships for path reconstruction.
- Stop when the endpoint is reached.

6. Implementation

Tools and Libraries

1. **Python**:

- Language of choice for flexibility and simplicity.

2. **Tkinter**:

- GUI development for grid and animations.

3. **Collections**:

- "deque" for BFS queue implementation.

Code Structure

1. **Main Application**:
- Initializes grid, robot, and GUI elements.
- Handles user interactions and algorithm selection.

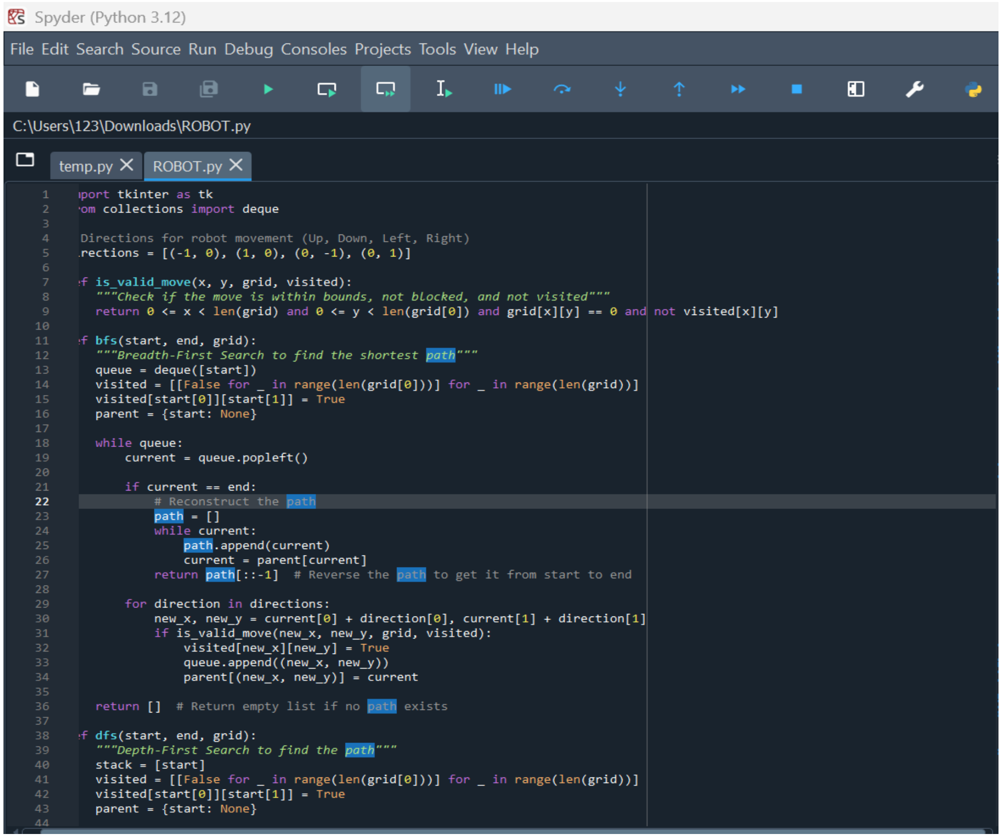
2. **Grid and Obstacles**:

- Defines cell states (empty, obstacle, visited).

3. **Animation**:

- Smoothly transitions robot between cells.
- Leaves trails to mark paths.

4. **Code:**




```
port tkinter as tk
om collections import deque

Directions for robot movement (Up, Down, Left, Right)
rections = [(-1, 0), (1, 0), (0, -1), (0, 1)]

f is_valid_move(x, y, grid, visited):
  """Check if the move is within bounds, not blocked, and not visited"""
  return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 0 and not visited[x][y]

f bfs(start, end, grid):
  """Breadth-First Search to find the shortest path"""
  queue = deque([start])
  visited = [[False for _ in range(len(grid[0]))] for _ in range(len(grid))]
  visited[start[0]][start[1]] = True
  parent = {start: None}

  while queue:
      current = queue.popleft()

      if current == end:
          # Reconstruct the path
          path = []
          while current:
              path.append(current)
              current = parent[current]
          return path[::-1]  # Reverse the path to get it from start to end

      for direction in directions:
          new_x, new_y = current[0] + direction[0], current[1] + direction[1]
          if is_valid_move(new_x, new_y, grid, visited):
              visited[new_x][new_y] = True
              queue.append((new_x, new_y))
              parent[(new_x, new_y)] = current

  return []  # Return empty list if no path exists

f dfs(start, end, grid):
  """Depth-First Search to find the path"""
  stack = [start]
  visited = [[False for _ in range(len(grid[0]))] for _ in range(len(grid))]
  visited[start[0]][start[1]] = True
  parent = {start: None}
```

\Users\123\Downloads\ROBOT.py

temp.py ✕ ROBOT.py ✕

```python
    visited = [[False for _ in range(len(grid[0]))] for _ in range(len(grid))]
    visited[start[0]][start[1]] = True
    parent = {start: None}

    while stack:
        current = stack.pop()

        if current == end:
            # Reconstruct the path
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]  # Reverse the path to get it from start to end

        for direction in directions:
            new_x, new_y = current[0] + direction[0], current[1] + direction[1]
            if is_valid_move(new_x, new_y, grid, visited):
                visited[new_x][new_y] = True
                stack.append((new_x, new_y))
                parent[(new_x, new_y)] = current

    return []  # Return empty list if no path exists

f create_grid(rows, cols):
    """Create a grid initialized with 0s (empty cells)"""
    return [[0 for _ in range(cols)] for _ in range(rows)]

ass RobotPuzzleApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Robot Navigation Puzzle Game")

        self.grid_size = 10  # Set grid size to 10x10 for complexity
        self.grid = create_grid(self.grid_size, self.grid_size)
        self.start = (0, 0)  # Fixed starting point
        self.end = None  # Initially no destination
        self.cell_size = 50  # Size of each cell in pixels

        # Create the Tkinter canvas for grid
        self.canvas = tk.Canvas(self.root, width=self.cell_size * self.grid_size, height=self.cell_size * self.grid_size)
        self.canvas.pack()

        self.draw_grid()
```

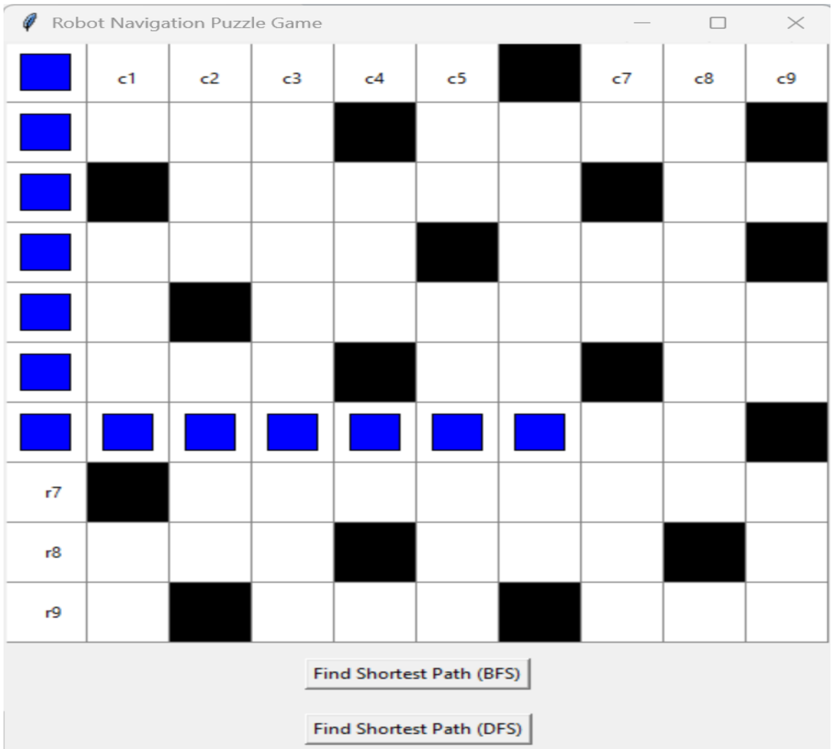
temp.py ✕ ROBOT.py ✕

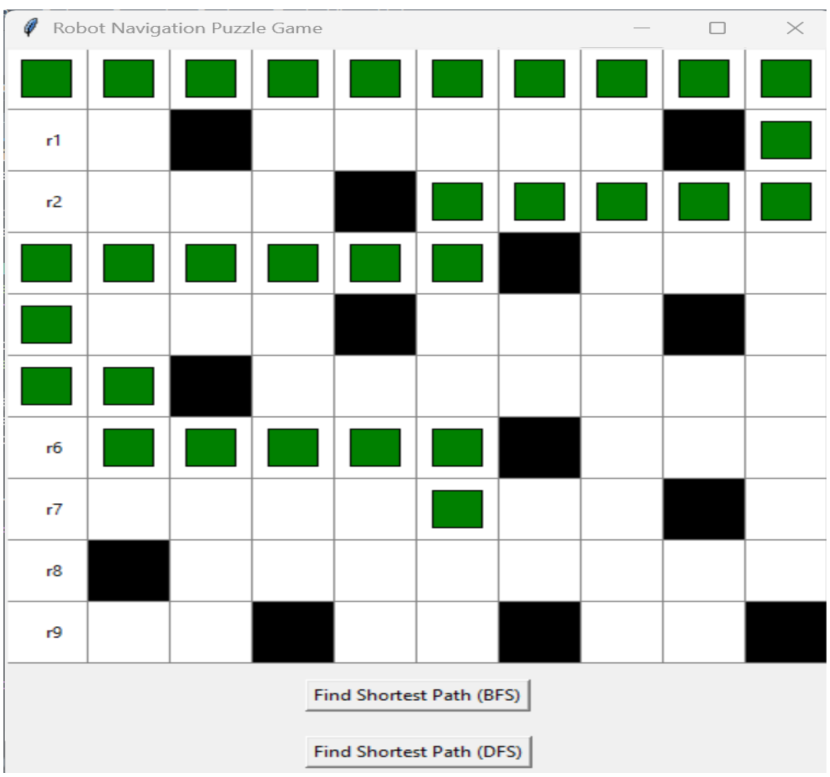
```python
        self.canvas = tk.Canvas(self.root, width=self.cell_size * self.grid_size, height=self.cell_size * self.grid_size)
        self.canvas.pack()

        self.draw_grid()

        # Event to set destination or obstacles by clicking
        self.canvas.bind("<Button-1>", self.set_destination_or_obstacle)

        # Button to find shortest path using BFS
        self.find_button = tk.Button(self.root, text="Find Shortest Path (BFS)", command=self.find_path_bfs)
        self.find_button.pack(pady=10)

        # Button to find shortest path using DFS
        self.dfs_button = tk.Button(self.root, text="Find Shortest Path (DFS)", command=self.find_path_dfs)
        self.dfs_button.pack(pady=10)

        # Initialize robot icon
        self.robot = None
        self.visited_path = set()  # To keep track of the visited cells for marking

    def draw_grid(self):
        """Draw the grid on the canvas"""
        self.canvas.delete("all")  # Clear the canvas before redrawing

        # Draw grid cells
        for i in range(self.grid_size):
            for j in range(self.grid_size):
                x1 = j * self.cell_size
                y1 = i * self.cell_size
                x2 = (j + 1) * self.cell_size
                y2 = (i + 1) * self.cell_size
                color = "white" if self.grid[i][j] == 0 else "black"
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="gray")

        # Draw row and column names
        for i in range(self.grid_size):
            self.canvas.create_text(30, i * self.cell_size + self.cell_size // 2, text=f"r{i}", fill="black")
            self.canvas.create_text(i * self.cell_size + self.cell_size // 2, 30, text=f"c{i}", fill="black")

        # Draw start point (green)
        start_x, start_y = self.start
        self.canvas.create_oval(start_y * self.cell_size + 20, start_x * self.cell_size + 20,
                                start_y * self.cell_size + 40, start_x * self.cell_size + 40, fill="green")
```

```python
        self.canvas.create_oval(start_y * self.cell_size + 20, start_x * self.cell_size + 20,
                                start_y * self.cell_size + 40, start_x * self.cell_size + 40, fill="green")

        # Draw end point (red)
        if self.end:
            end_x, end_y = self.end
            self.canvas.create_oval(end_y * self.cell_size + 20, end_x * self.cell_size + 20,
                                    end_y * self.cell_size + 40, end_x * self.cell_size + 40, fill="red")

    def set_destination_or_obstacle(self, event):
        """Set destination or obstacle based on click position"""
        x = event.y // self.cell_size
        y = event.x // self.cell_size

        if (x, y) == self.start:
            return  # Do nothing if clicked on the start point

        if self.grid[x][y] == 0:
            if self.end is None:
                self.end = (x, y)  # Set destination if not set
                self.draw_grid()
            else:
                self.grid[x][y] = 1  # Add obstacle (mark as 1)
                self.draw_grid()
        else:
            self.grid[x][y] = 0  # Remove obstacle if clicked again
            self.draw_grid()

    def find_path_bfs(self):
        """Find the path using BFS"""
        if not self.end:
            print("Please select a destination.")
            return

        path = bfs(self.start, self.end, self.grid)
        if path:
            self.animate_robot(path, color="blue")  # Assign unique color for BFS
        else:
            print("No path found")

    def find_path_dfs(self):
        """Find the path using DFS"""
        if not self.end:
            print("Please select a destination.")
```

```python
    def find_path_dfs(self):
        """Find the path using DFS"""
        if not self.end:
            print("Please select a destination.")
            return

        path = dfs(self.start, self.end, self.grid)
        if path:
            self.animate_robot(path, color="green")  # Assign unique color for DFS
        else:
            print("No path found")

    def animate_robot(self, path, color):
        """Animate the robot moving along the path, leaving marks"""
        # Create a simple representation of the robot (a rectangle with a head)
        if not self.robot:
            self.robot = self.canvas.create_rectangle(self.start[1] * self.cell_size + 10, self.start[0] * self.cell_size + 10,
                                                      self.start[1] * self.cell_size + 40, self.start[0] * self.cell_size + 40,
                                                      fill="blue")
            # Robot's "head"
            self.canvas.create_oval(self.start[1] * self.cell_size + 20, self.start[0] * self.cell_size + 20,
                                    self.start[1] * self.cell_size + 30, self.start[0] * self.cell_size + 30,
                                    fill="blue")

        def move_robot(i):
            if i < len(path):
                x, y = path[i]
                # Leave a mark (colored square) where the robot has been
                self.canvas.create_rectangle(y * self.cell_size + 10, x * self.cell_size + 10,
                                             y * self.cell_size + 40, x * self.cell_size + 40,
                                             fill=color)
                # Move the robot
                self.canvas.coords(self.robot, y * self.cell_size + 10, x * self.cell_size + 10,
                                   y * self.cell_size + 40, x * self.cell_size + 40)
                self.root.after(200, move_robot, i + 1)  # Move robot every 200ms

        move_robot(0)

 Running the application
 __name__ == "__main__":
    root = tk.Tk()
    app = RobotPuzzleApp(root)
    root.mainloop()
```

OUTPUT:


Robot Navigation Puzzle Game
c1
c2
c3
c4
c5
c7
c8
c9
r7
r8
r9
Find Shortest Path (BFS)
Find Shortest Path (DFS)


Robot Navigation Puzzle Game
r1
r2
r6
r7
r8
r9
Find Shortest Path (BFS)
Find Shortest Path (DFS)

7. Testing and Validation

Obstacle Handling

1. Accurate navigation around obstacles.
2. Multiple test cases validate handling of dense and sparse layouts.

Algorithm Validation

1. BFS consistently identifies the shortest path.
2. DFS demonstrates depth-first traversal with reconstructed paths.

User Interaction

1. Responsive grid interactions.
2. Smooth animations enhance clarity.

Additional Testing

1. Stress-tested with grids of varying complexities.
2. Verified edge cases, such as blocked endpoints.

Performance Metrics

1. BFS:

- Time Complexity: O(V + E).
- Space Complexity: O(V).

2. DFS:

- Time Complexity: O(V + E).
- Space Complexity: O(V).

8. Results

1. BFS and DFS successfully implemented and visualized.
2. Intuitive GUI enhances user experience.
3. Clear differentiation between BFS and DFS behaviors.
4. Robust performance with diverse grid configurations.

9. Future Improvements

1. **Advanced Algorithms**:

- Include A* and Dijkstra’s algorithms for comparison.

2. **Scalability**:

- Support larger grid sizes.

3. **User Features**:

- Save and load grid configurations.
- Adjustable robot speed.

4. **Performance Insights**:

- Real-time metrics display (e.g., nodes visited).

5. **Obstacle Types**:

- Introduce dynamic obstacles and weights.

10. Conclusion

The Robot Navigation Puzzle Game combines algorithm visualization with an interactive user experience. By integrating BFS and DFS, this project offers educational value, showcasing pathfinding behaviors in real-world scenarios. The project’s modular design and robust implementation provide a foundation for future enhancements, ensuring its utility as a teaching and learning tool in computer science and robotics.

Appendices

1. **Code Snippets**:

- BFS and DFS implementations.

2. **Screenshots**:

- Annotated images of grid states and robot navigation.

3. **References**:

- Documentation on BFS, DFS, and Tkinter.