9/1/2025

Knight's travails

ROBOT NAVIGATION PUZZLE

DSA PROJECT REPORT

Amna Pervez 2023- BCS-010
Muneeba Jamil 2023-BCS-066
Insa Saleem 2023- BCS-043
Maria Yasmeen 2023- BCS-059

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:

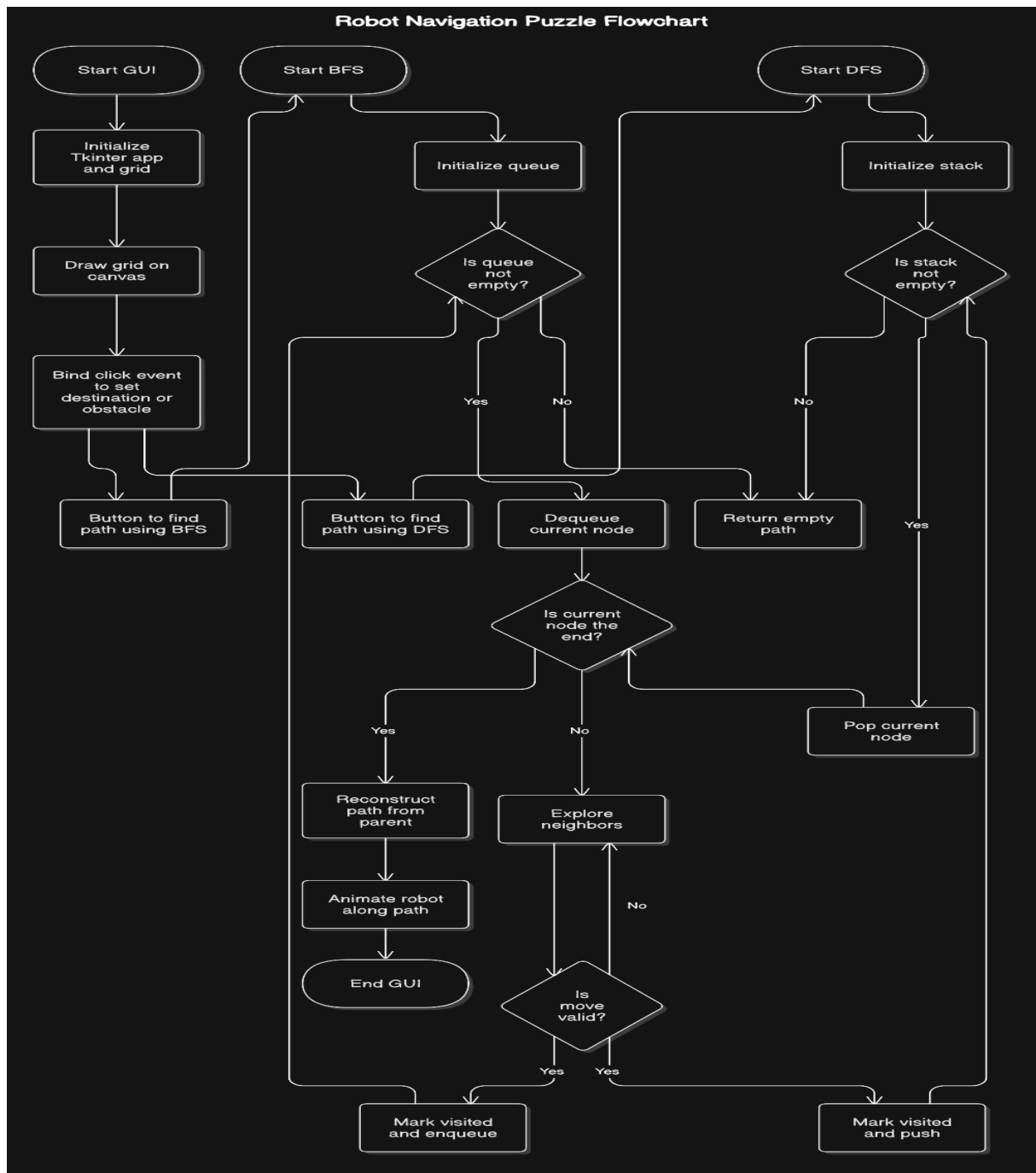- 2D list initialized with zeros (empty cells).
- Obstacles marked as ones.

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.

Robot Navigation Puzzle Flowchart

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.
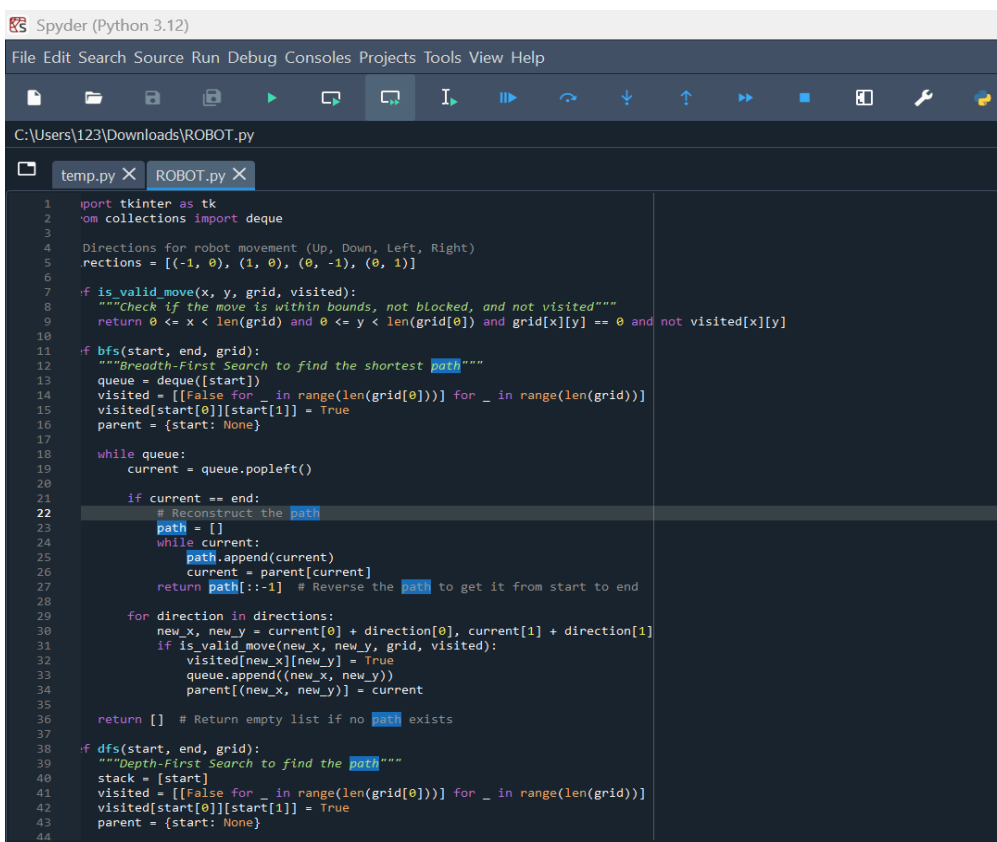- Handles user interactions and algorithm selection.

2. **Grid and Obstacles**:

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

3. **Animation**:

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

4. **Code:**

temp.py ✕ | ROBOT.py ✕

```python
41      visited = [[False for _ in range(len(grid[0]))] for _ in range(len(grid))]
42      visited[start[0]][start[1]] = True
43      parent = {start: None}
44
45      while stack:
46          current = stack.pop()
47
48          if current == end:
49              # Reconstruct the path
50              path = []
51              while current:
52                  path.append(current)
53                  current = parent[current]
54              return path[::-1]  # Reverse the path to get it from start to end
55
56          for direction in directions:
57              new_x, new_y = current[0] + direction[0], current[1] + direction[1]
58              if is_valid_move(new_x, new_y, grid, visited):
59                  visited[new_x][new_y] = True
60                  stack.append((new_x, new_y))
61                  parent[(new_x, new_y)] = current
62
63      return []  # Return empty list if no path exists
64
65  def create_grid(rows, cols):
66      """Create a grid initialized with 0s (empty cells)"""
67      return [[0 for _ in range(cols)] for _ in range(rows)]
68
69  class RobotPuzzleApp:
70      def __init__(self, root):
71          self.root = root
72          self.root.title("Robot Navigation Puzzle Game")
73
74          self.grid_size = 10  # Set grid size to 10x10 for complexity
75          self.grid = create_grid(self.grid_size, self.grid_size)
76          self.start = (0, 0)  # Fixed starting point
77          self.end = None  # Initially no destination
78          self.cell_size = 50  # Size of each cell in pixels
79
80          # Create the Tkinter canvas for grid
81          self.canvas = tk.Canvas(self.root, width=self.cell_size * self.grid_size, height=self.cell_size * self.grid_size)
82          self.canvas.pack()
83
84          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")
```

temp.py ✕ **ROBOT.py** ✕

```python
22          self.canvas.create_oval(start_y * self.cell_size + 20, start_x * self.cell_size + 20,
23                                  start_y * self.cell_size + 40, start_x * self.cell_size + 40, fill="green")
24
25          # Draw end point (red)
26          if self.end:
27              end_x, end_y = self.end
28              self.canvas.create_oval(end_y * self.cell_size + 20, end_x * self.cell_size + 20,
29                                      end_y * self.cell_size + 40, end_x * self.cell_size + 40, fill="red")
30
31      def set_destination_or_obstacle(self, event):
32          """Set destination or obstacle based on click position"""
33          x = event.y // self.cell_size
34          y = event.x // self.cell_size
35
36          if (x, y) == self.start:
37              return  # Do nothing if clicked on the start point
38
39          if self.grid[x][y] == 0:
40              if self.end is None:
41                  self.end = (x, y)  # Set destination if not set
42                  self.draw_grid()
43              else:
44                  self.grid[x][y] = 1  # Add obstacle (mark as 1)
45                  self.draw_grid()
46          else:
47              self.grid[x][y] = 0  # Remove obstacle if clicked again
48              self.draw_grid()
49
50      def find_path_bfs(self):
51          """Find the path using BFS"""
52          if not self.end:
53              print("Please select a destination.")
54              return
55
56          path = bfs(self.start, self.end, self.grid)
57          if path:
58              self.animate_robot(path, color="blue")  # Assign unique color for BFS
59          else:
60              print("No path found")
61
62      def find_path_dfs(self):
63          """Find the path using DFS"""
64          if not self.end:
65              print("Please select a destination.")
```
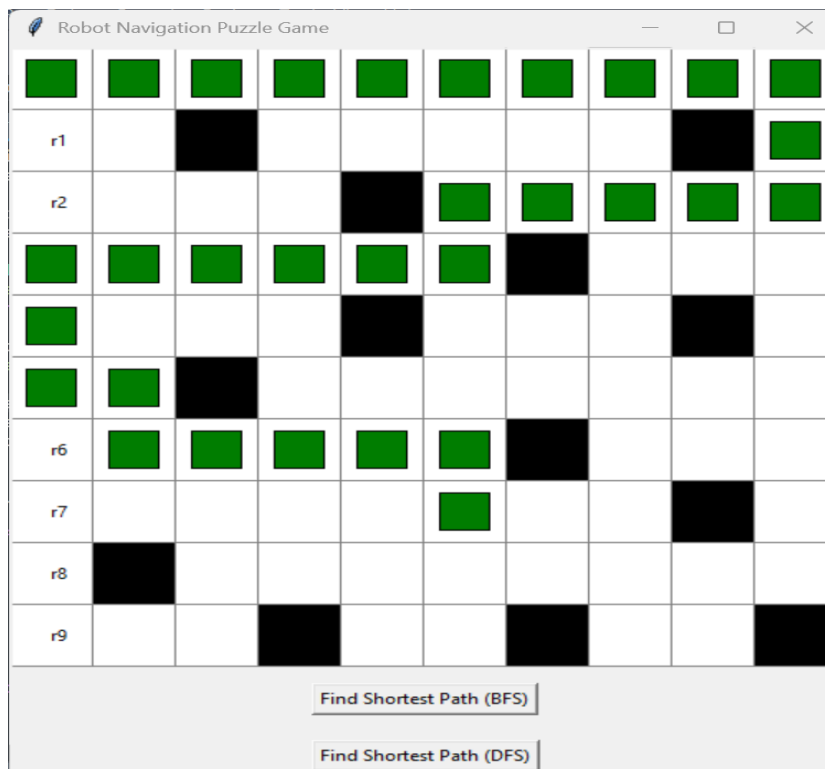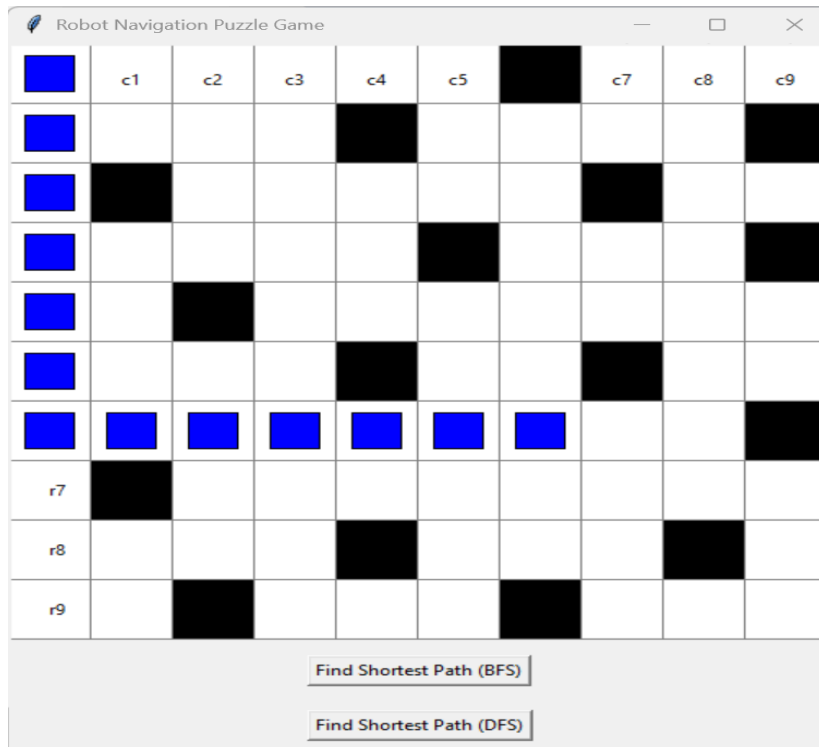
temp.py ✕ **ROBOT.py** ✕

```python
51
52      def find_path_dfs(self):
53          """Find the path using DFS"""
54          if not self.end:
55              print("Please select a destination.")
56              return
57
58          path = dfs(self.start, self.end, self.grid)
59          if path:
60              self.animate_robot(path, color="green")  # Assign unique color for DFS
71          else:
72              print("No path found")
73
74      def animate_robot(self, path, color):
75          """Animate the robot moving along the path, leaving marks"""
76          # Create a simple representation of the robot (a rectangle with a head)
77          if not self.robot:
78              self.robot = self.canvas.create_rectangle(self.start[1] * self.cell_size + 10, self.start[0] * self.cell_size + 10,
79                                                         self.start[1] * self.cell_size + 40, self.start[0] * self.cell_size + 40,
80                                                         fill="blue")
81              # Robot's "head"
82              self.canvas.create_oval(self.start[1] * self.cell_size + 20, self.start[0] * self.cell_size + 20,
83                                      self.start[1] * self.cell_size + 30, self.start[0] * self.cell_size + 30,
84                                      fill="blue")
85
86          def move_robot(i):
87              if i < len(path):
88                  x, y = path[i]
89                  # Leave a mark (colored square) where the robot has been
90                  self.canvas.create_rectangle(y * self.cell_size + 10, x * self.cell_size + 10,
91                                               y * self.cell_size + 40, x * self.cell_size + 40,
92                                               fill=color)
93                  # Move the robot
94                  self.canvas.coords(self.robot, y * self.cell_size + 10, x * self.cell_size + 10,
95                                     y * self.cell_size + 40, x * self.cell_size + 40)
96                  self.root.after(200, move_robot, i + 1)  # Move robot every 200ms
97
98          move_robot(0)
99
00  Running the application
01  if __name__ == "__main__":
02      root = tk.Tk()
03      app = RobotPuzzleApp(root)
04      root.mainloop()
```

OUTPUT:

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.