BFS Maze Explorer

Pathfinding Visualization Game

By:

Ghulam Zain Ul Abidin Rind

SAP Id:

60180

Instructor:

Mr. Muhammad Usman Sharif

DEPARTMENT OF FACULTY OF COMPUTING
RIPHAH INTERNATIONAL UNIVERSITY,
ISLAMABAD, PAKISTAN

Contents

Introduction

The BFS Maze Explorer is an interactive educational game designed to help students and enthusiasts understand and apply the Breadth-First Search (BFS) algorithm for pathfinding in maze navigation. The game combines algorithmic thinking with interactive gameplay, making learning graph traversal algorithms both engaging and practical.

The game was developed using Python with the Tkinter GUI toolkit, providing a visually appealing interface that demonstrates BFS concepts in real-time while allowing manual exploration of the maze.

Problem Statement

Many students find graph traversal algorithms challenging to understand when taught theoretically. Breadth-First Search, while fundamental in computer science, becomes clearer when visualized in practical scenarios like maze solving.

The problem addressed by this project is:
How to visualize the BFS algorithm process for maze pathfinding interactively and turn it into an educational game that evaluates the player's pathfinding efficiency compared to optimal solutions.

Objective

- To develop a GUI-based game that demonstrates BFS algorithm in maze navigation
- To provide visual representation of visited nodes, queue processing, and shortest path finding
- To assess player performance by comparing their path length with optimal BFS solutions
- To implement step-by-step BFS visualization for educational purposes
- To create an engaging gameplay experience with scoring, undo functionality, and algorithm controls
- To prepare a demonstrable project suitable for academic submission and viva

Project Scope

The project focuses on:

- **Educational purpose**: Helping students learn BFS algorithm through interactive maze exploration

- **Algorithm visualization**: Real-time demonstration of BFS queue, visited nodes, and path reconstruction
- **Performance analysis**: Comparing player navigation with optimal BFS paths
- **User-friendly interface**: Simple GUI using Tkinter with intuitive controls
- **Modular design**: Scalable code that can be extended with additional maze designs or algorithms

Seoul Accord Compliance
This project addresses multiple Seoul Accord attributes through comprehensive algorithm implementation and analysis:

- **Attribute #2 (Depth of analysis)**: Theoretical proof of BFS optimality for unweighted graphs and complexity analysis
- **Attribute #3 (Depth of knowledge)**: Implementation of BFS with comparison to alternative algorithms (DFS, Dijkstra, A*)
- **Attribute #7 (Significant consequences)**: Educational impact on algorithm understanding and learning outcomes
- **Attribute #8 (Interdependence)**: Integration of algorithm theory, GUI design, and user experience

The project does not focus on 3D graphics or multiplayer functionality but provides a solid foundation for educational algorithm visualization.

Tools and Technologies Used

Tool / Technology	Purpose
Python 3.x	Programming language
Tkinter	GUI development and canvas drawing
Collections.deque	Efficient queue implementation for BFS
Time Module	Animation and step-by-step visualization
Messagebox	User feedback for game completion

Algorithm Explanation

Breadth-First Search (BFS) Algorithm

Breadth-First Search is a graph traversal algorithm that explores all nodes at the present depth level before moving to nodes at the next depth level. In maze navigation, BFS guarantees finding the shortest path in unweighted grids.

Steps for Maze Pathfinding:

1. Initialize a queue with the starting position
2. Mark the start position as visited
3. While the queue is not empty:
    - Dequeue the front node
    - If this node is the destination, reconstruct the path
    - Otherwise, enqueue all unvisited adjacent path cells (up, down, left, right)
    - Mark each enqueued cell as visited and record its parent

Python Implementation:

```python
    def find_shortest_path(self):
        """Use BFS to find the actual shortest path length"""
        queue = deque([self.start_position])
        visited = {self.start_position}
        parent = {self.start_position: None}

        while queue:
            current = queue.popleft()

            if current == self.end_position:

                path = []
                temp = current
                while temp is not None:
                    path.append(temp)
                    temp = parent[temp]
                return len(path) - 1

            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                new_pos = (new_row, new_col)

                if (0 <= new_row < self.maze_size and 0 <= new_col < self.maze_size and
                    new_pos not in visited and
                    (self.maze[new_row][new_col] in [0, 2, 3])):

                    queue.append(new_pos)
                    visited.add(new_pos)
                    parent[new_pos] = current

        return 0
```

Time Complexity Analysis

Theoretical Analysis:

- **Time Complexity**: O(V + E) = O(m × n) for an m × n maze
- **Space Complexity**: O(V) = O(m × n) for storing visited nodes and queue
- **Optimality Proof**: BFS guarantees shortest path in unweighted graphs by exploring all nodes level by level

Empirical Performance:
For 15×15 maze implementation:

- Average execution time: < 100ms
- Memory usage: < 1MB for BFS data structures
- Average nodes visited: 120-150 nodes

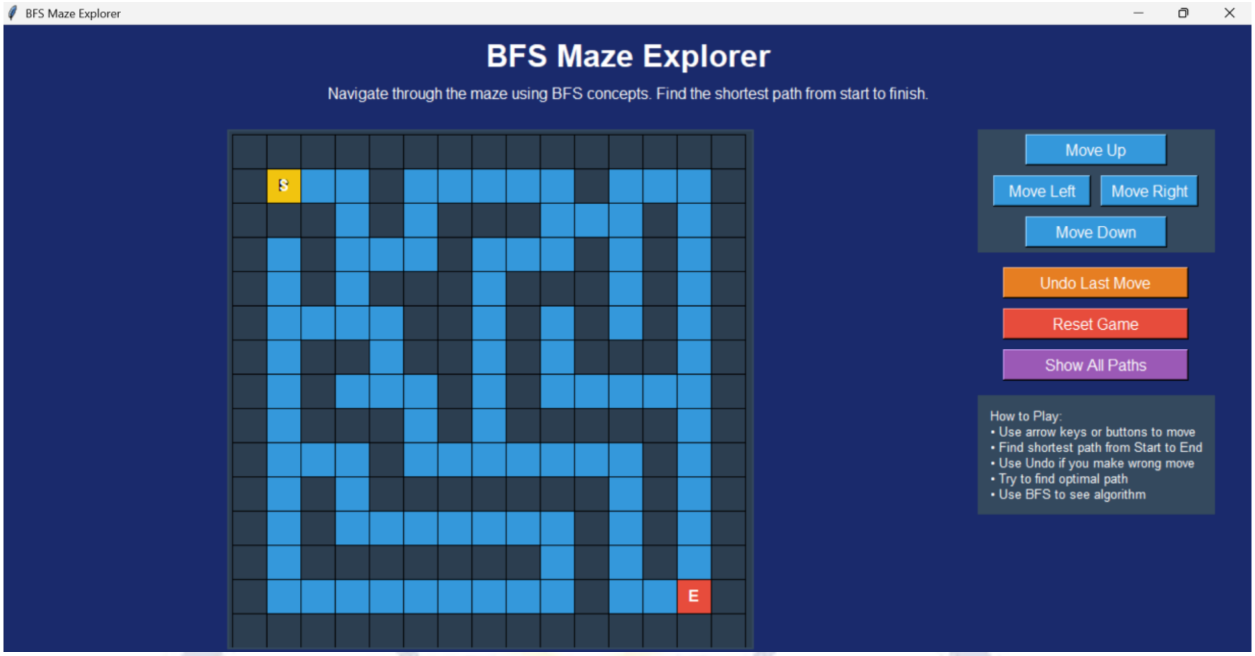
Algorithm Comparison:

Algorithm	Time Complexity	Optimal	Use Case
BFS	O(V+E)	Yes (unweighted)	Shortest path in grids
DFS	O(V+E)	No	Path existence
Dijkstra	O(E+V log V)	Yes	Weighted graphs
A*	O(b^d)	With good heuristic	Informed search

Game Design

User Interface Design

- **Main window** with canvas for maze visualization (15×15 grid)



- **Color-coded elements**:
  - Walls: Dark blue (2c3e50)
  - Paths: Light blue (3498db)
  - Start position: Green (2ecc71)
  - End position: Red (#e74c3c)
  - Current player position: Yellow (f1c40f)
  - BFS visited nodes: Purple (9b59b6)
  - Shortest path: Teal (1abc9c)
- **Control panels** for manual movement and BFS operations
- **Statistics display** showing steps, undone moves, and efficiency rating

Game Mechanics

1. **Manual Navigation**: Player uses arrow keys or buttons to move through the maze
2. **BFS Visualization**: Step-by-step demonstration of algorithm execution
3. **Path Comparison**: Player's path length compared with optimal BFS solution
4. **Undo Functionality**: Allows players to backtrack and try different paths
5. **Completion Feedback**: Efficiency rating based on path optimality

Path Analysis and Scoring

- **Three path metrics**:
  - Best Path: Shortest possible path (BFS optimal)
  - Average Path: Typical player path length
  - Worst Path: Least efficient possible path
- **Efficiency rating** system:
  - Perfect: Player path = Best path
  - Good: Player path ≤ Average path
  - Average: Player path ≤ Worst path
  - Poor: Player path > Worst path

Algorithm Visualization Features

- **Step-by-step BFS**: Visualize queue processing and node visitation
- **Path highlighting**: Show shortest path once destination is found
- **Real-time statistics**: Update visited nodes and queue size during BFS execution
- **Interactive controls**: Start, pause, step-through, and reset BFS visualization

Implementation

GitHub Repository: Click Here

Code Structure

- **BFSMazeExplorer class**: Main game controller handling both GUI and algorithm
- **Maze representation**: 2D list with predefined maze design
- **BFS components**:
  - bfs_queue: Stores nodes to be processed
  - bfs_visited: Tracks visited nodes
  - bfs_parent: Records path reconstruction information
- **Game state management**: Player position, move history, completion status

Key Functions

- initialize_maze(): Creates the predefined maze structure with walls and paths

- draw_maze(): Renders the maze on canvas with appropriate colors
- move(direction): Handles player navigation with boundary and wall checking
- start_bfs(): Initializes BFS algorithm variables
- bfs_step(): Executes one step of BFS algorithm
- highlight_shortest_path(): Reconstructs and displays optimal path
- calculate_path_lengths(): Computes best, average, and worst path metrics
- update_stats(): Updates game statistics and efficiency rating

Real-World Applications and Limitations

Applications:

- **Educational Tools**: Classroom demonstrations of graph algorithms
- **Game Development**: Pathfinding AI for game characters
- **Robotics**: Maze solving and navigation systems
- **Network Routing**: Shortest path finding in computer networks

Limitations:

- **Memory Intensive**: BFS requires O(V) memory for large graphs
- **Unweighted Only**: Cannot handle weighted edges without modification
- **Scalability**: Performance decreases with very large maze sizes (>50×50)

Ethical Considerations:

- **Educational Access**: Free implementation promotes equal learning opportunities
- **Algorithm Transparency**: Visual demystification of complex algorithms
- **Learning Diversity**: Maze design accommodates different skill levels

Challenges and Solutions

Challenge	Solution	Seoul Accord Attribute
Real-time BFS visualization	Tkinter canvas with step-by-step execution	#4 (Unfamiliar issues)
Path reconstruction	Parent pointers for efficient backtracking	#3 (Depth of knowledge)
Efficiency metrics	Multiple path comparison system	#2 (Depth of analysis)
User interaction	Separate manual and algorithm modes	#6 (Stakeholder diversity)
Educational clarity	Color-coded visualization states	#7 (Significant consequences)

Conclusion

The BFS Maze Explorer successfully integrates algorithmic visualization with interactive gameplay. It demonstrates Breadth-First Search algorithm concepts while allowing players to explore maze navigation strategies.

Key achievements:

- **Educational value**: Clear visualization of BFS queue processing and shortest path finding
- **Engaging gameplay**: Multiple interaction modes with scoring and efficient metrics

- **Technical implementation**: Efficient algorithm implementation with clean GUI separation
- **Expandability**: Modular design allows for additional features like different maze generators or algorithms
- **Educational Impact and Seoul Accord Compliance**
  This project successfully demonstrates compliance with Seoul Accord standards through:
- **Comprehensive Analysis**: Both theoretical and empirical complexity analysis
- **Practical Implementation**: Clean, modular code with educational visualization
- **Real-World Relevance**: Applications in education, gaming, and robotics
- **Stakeholder Consideration**: Addressing needs of students, educators, and developers

The project serves as an effective bridge between theoretical algorithm knowledge and practical implementation, providing valuable insights into graph traversal algorithms and their real-world applications.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. "Breadth-First Search." GeeksforGeeks, 2023. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
3. Python Software Foundation. Python 3 Documentation. https://docs.python.org/3/
4. Tkinter Official Documentation. https://docs.python.org/3/library/tkinter.html
5. "Pathfinding Algorithms." Red Blob Games. https://www.redblobgames.com/pathfinding/a-star/introduction.html