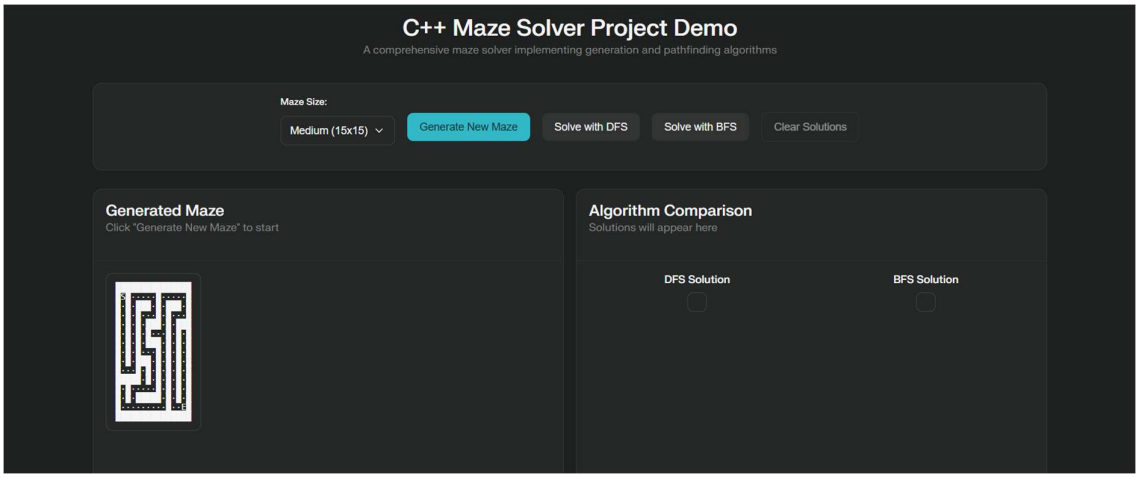
Maze Solver



The project features a modular design with three main components: random maze generation using recursive backtracking, maze solving with Depth-First Search (DFS), and optimal pathfinding with Breadth-First Search (BFS). Each algorithm serves a specific purpose and showcases different approaches to problem-solving in computer science.

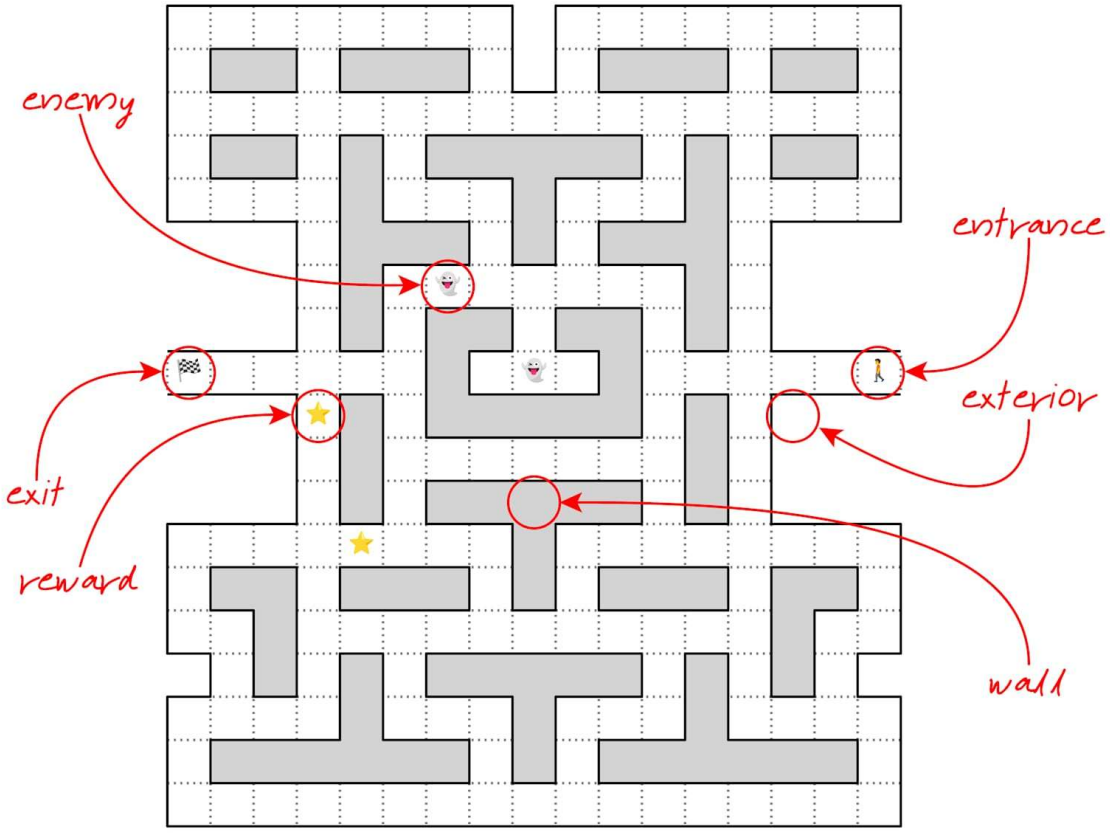
Algorithms Implemented

1. Recursive Backtracking for Maze Generation

The maze generation algorithm creates perfect mazes (mazes with exactly one path between any two points) using a recursive backtracking approach. This algorithm ensures that every cell in the maze is reachable and there are no loops or isolated areas

Process:

- Start with a grid completely filled with walls
- Choose a random starting cell and mark it as a path
- Randomly select an unvisited neighboring cell
- Remove the wall between the current and chosen cell
- Recursively continue until all cells are visited
- Backtrack when no unvisited neighbors are available



2. Depth-First Search (DFS) Solving

DFS explores the maze by going as far as possible along each branch before backtracking. While it doesn't guarantee the shortest path, it's memory-efficient and finds a solution quickly if one exists.

Characteristics:

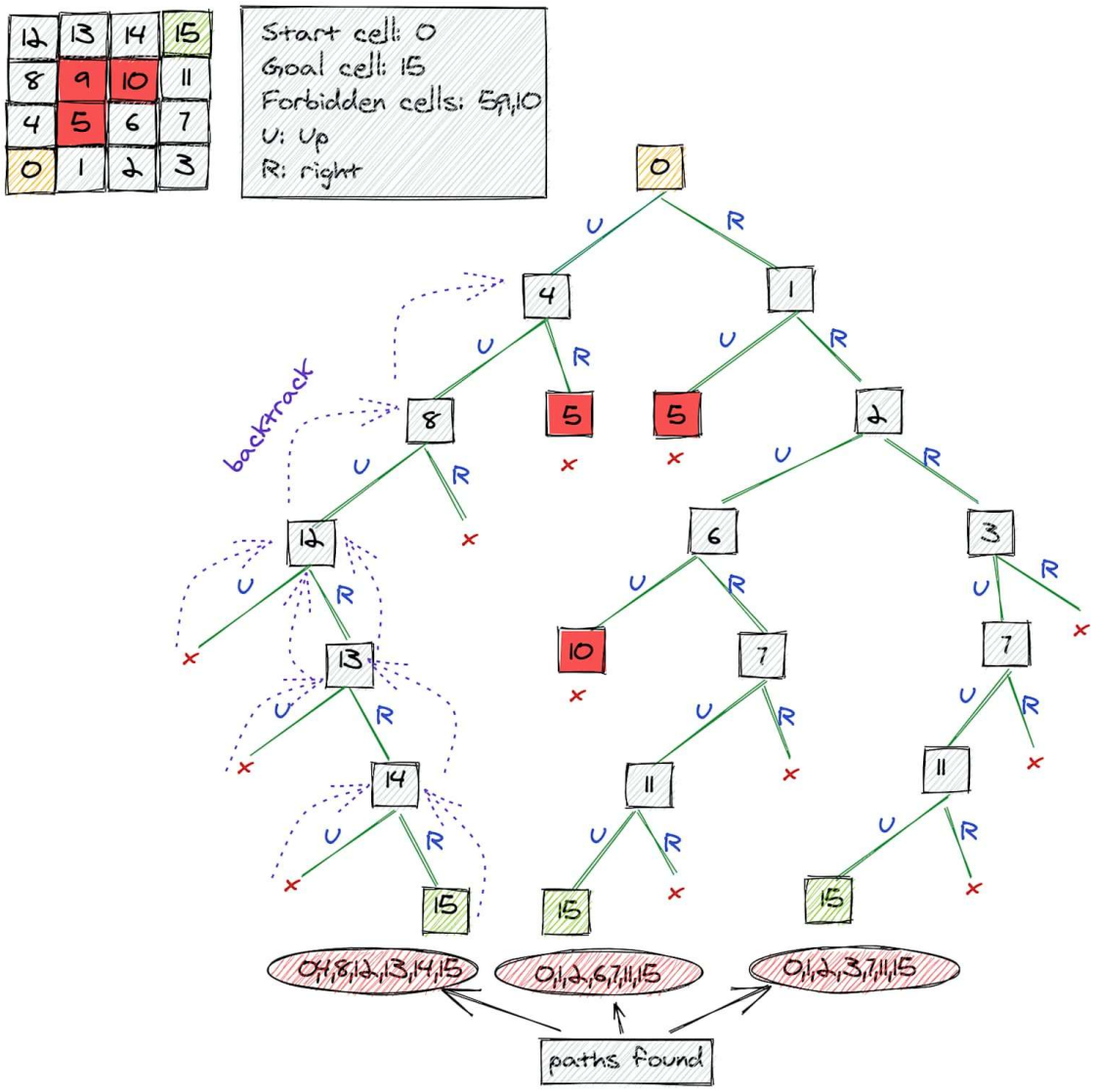
- **Time Complexity:** O(V + E) where V = vertices, E = edges
- **Space Complexity:** O(V) for the recursion stack
- **Finds:** Any valid path from start to end
- **Advantage:** Low memory usage, simple implementation

3. Breadth-First Search (BFS) Solving

BFS systematically explores all possible paths level by level, guaranteeing the shortest path from start to end. It uses a queue-based approach to ensure optimal solutions.

Characteristics:

- **Time Complexity:** O(V + E)
- **Space Complexity:** O(V) for queue storage
- **Finds:** Shortest path from start to end
- **Advantage:** Guaranteed optimal soluti



Key Features and Functionality

User Interface Options

The program provides three distinct modes of operation:

1. **Random Maze Generation**
    - Generates mazes of user-specified dimensions
    - Solves the generated maze using both DFS and BFS
    - Compares the different solution paths visually
2. **Custom Maze Input**

- Allows users to input their own maze configurations
- Supports any maze size and layout
- Solves custom mazes using DFS algorithm

3. **Quick Demo Mode**
    - Provides a predefined 7x7 maze for immediate testing
    - Demonstrates the algorithms without user input
    - Shows the complete solution process

Visual Output System

The program uses clear ASCII symbols for maze visualization:

- # represents walls
- . represents open paths
- S marks the starting position
- E marks the ending position
- * shows the solution path

Technical Implementation Details

Data Structures

The maze is represented as a 2D vector of integers where:

- 0 represents passable cells (paths)
- 1 represents impassable cells (walls)

This binary representation simplifies boundary checking and pathfinding logic while maintaining memory efficiency.

Memory Management

The implementation follows modern C++ best practices:

- RAII (Resource Acquisition Is Initialization) principles
- Automatic memory management with STL containers
- No manual memory allocation or deallocation
- Stack-based data structures for optimal performance

Educational Value and Applications

This project serves as an excellent educational tool for understanding:

- **Graph Theory:** Mazes as graph representations with nodes and edges
- **Algorithm Design:** Different approaches to search and optimization problems
- **Data Structures:** Practical use of stacks, queues, and 2D arrays
- **Recursion:** Backtracking implementation and call stack management
- **Object-Oriented Programming:** Class design and encapsulation principles

The algorithms demonstrated have real-world applications in:

- Robotics path planning
- Game AI development
- Network routing protocols
- Circuit board design
- Social network analysis