Pathfinding and Maze Traversal Engine: An Object-Oriented Design

OOPS Phase-1 Project Report by
CS24I1029 YASHVANTH S
CS24I1035 JEEVANANDHAM T
CS24I1039 P Y NITHILAKRISHI
CS24B2054 SRI HARI S

November 12, 2025

1. Introduction

This report describes the design and implementation of the "Path Analyzer and Maze Solver" project. The goal is to build a console-based maze generator, solver, visualizer, and analyzer that demonstrates object-oriented programming principles, modular design, and practical algorithm implementations (BFS and Dijkstra). The objectives include:

- Implement a maze representation and generation utilities.
- Provide solver modules (BFS for unweighted shortest path, Dijkstra for weighted terrains).
- Offer a renderer and CLI utilities for human-friendly visualization and interaction.
- Analyze and compare paths using configurable metrics.
- Demonstrate OOP concepts in a small, maintainable C++ codebase.

2. OOP Features Used

Classes and Objects

The project is organized into classes representing the main domain entities: `Maze`, `MazeGenerator`, `Point`, `Path`, `BFSSolver`, `DijkstraSolver`, `Renderer`, `CLIUtils`, and `PathAnalyzer`. Each class encapsulates state and behavior relevant to a single responsibility.

Encapsulation

Data members are private where appropriate (e.g., grid storage in `Maze`, dynamic arrays in `Path`) and exposed via well-defined getters/setters or methods. This prevents external modules from manipulating internals directly and allows internal changes without affecting API users.

Abstraction

High-level concepts such as "generate maze" or "solve maze" are presented through simple methods (e.g., `MazeGenerator::generateEasy()`, `BFSSolver::solve()`). Implementation details (randomized backtracking, priority queue operations) are hidden inside classes.

Inheritance and Polymorphism

The core design favors composition over inheritance. However, where appropriate, polymorphism is used to allow interchangeable solver strategies. For example, an abstract solver interface (or common method signatures) allows code that requests a solution to accept either BFS or Dijkstra solver objects. If an abstract base class `ISolver` were added, both solvers could inherit from it to enable runtime polymorphism.

3. System Design

Below is a high-level class diagram expressed in text and listing of classes with their main members.

Class: Point

- Members: `int x_, y_`
- Methods: constructors, getters (`getX()`, `getY()`), operators (+, -, ==, !=, ¡), `manhattanDistance`

Class: Maze

- Members: `char** grid_`, `int width_, height_`, `Point start_, goal_`
- Methods: constructors/destructor, `getCellAt(x,y)`, `setCellAt(x,y,c)`, `loadFromFile(filename)`, `saveToFile(filename)`, `isWalkable(x,y)`, `getNeighbors(Point)`

Class: MazeGenerator

- Members: RNG seed, width, height
- Methods: `generatePerfect()`, `generateWithLoops(n)`, `generateWithTerrain()`, preset generators (`generateEasy/Medium/Hard`), setters

Class: Path

- Members: dynamic array of `Point` (`Point* points_`), `int size_, capacity_`, `double cost_`
- Methods: add point, remove point, `operator[]`, `size()`, `clear()`, `toString()`

Class: BFSSolver

- Members: internal queue, visited map, parent map
- Methods: `solve(const Maze&)` returning `Path`, utility helpers for path reconstruction

Class: DijkstraSolver

- Members: min-heap priority queue, distance array, parent map
- Methods: `solve(const Maze&)` returning `Path`, cost function for terrain

Class: Renderer

- Members: display buffer, width, height, color mode flag, `CLIUtils` reference
- Methods: `render(const Maze&)` , `render(const Maze&, const Path&)`, `renderAnimated(...)`

Class: CLIUtils

- Members: options for colors, spinner state
- Methods: terminal helpers (clear, moveCursor, drawBox, drawProgressBar, input helpers, printColored)

Class: PathAnalyzer

- Members: cached metrics, configuration thresholds
- Methods: computeMetrics(const Path&, const Maze&), compare(const Path&, const Path&), computeTurns, computeStraightSegments, computeExposure

4. Sample Source Code

Below are concise, relevant snippets illustrating key classes and how they interact. These are intentionally short and focused on clarity.

Point.h (snippet)

```
class Point {
private:
  int x_, y_;
public:
  Point(int x = 0, int y = 0): x_(x), y_(y) {}
  int getX() const { return x_; }
  int getY() const { return y_; }
  bool operator==(const Point& o) const { return x_ == o.x_ && y_ == o.
     y_; }
};
```

Rationale: small value object, immutable accessors, simple equality for container operations.

Maze.h (snippet)

```
class Maze {
private:
  char **grid_;
  int width_, height_;
public:
  Maze(int w = 0, int h = 0);
  ~Maze();
  char getCellAt(int x, int y) const;
  void setCellAt(int x, int y, char c);
};
```

Rationale: encapsulate grid memory and provide safe accessors.

BFSSolver.cpp (snippet)

```
Path BFSSolver::solve(const Maze &maze) {
  // Classic BFS on grid
  enqueue(start);
  markVisited(start);
  while (!queueEmpty()) {
    Point cur = dequeue();
    if (cur == maze.getGoal()) break;
    for (Point n : maze.getNeighbors(cur)) {
      if (!visited(n)) { markVisited(n); setParent(n, cur); enqueue(n);
           }
    }
  }
  return reconstructPath();
}
```

Rationale: BFS yields shortest path in unweighted mazes and is simple and robust.

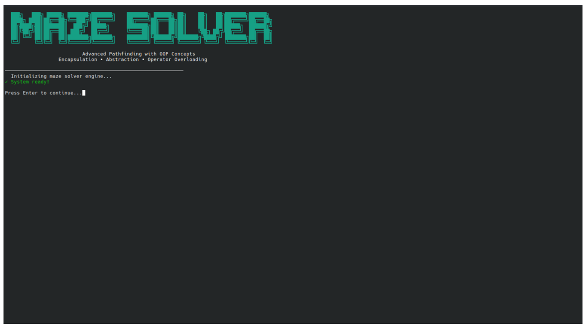
DijkstraSolver.cpp (snippet)

```
Path DijkstraSolver::solve(const Maze &maze) {
  // Min-heap by distance
  push(start, 0.0);
  while (!pqEmpty()) {
    Node n = pop();
    if (n.p == maze.getGoal()) break;
    for (auto nb : maze.getNeighbors(n.p)) {
      double nd = distance[n.p] + cost(nb);
      if (nd < distance[nb]) { distance[nb] = nd; setParent(nb, n.p);
          push(nb, nd); }
    }
  }
  return reconstructPath();
}
```

Rationale: supports variable terrain costs and produces least-cost paths.

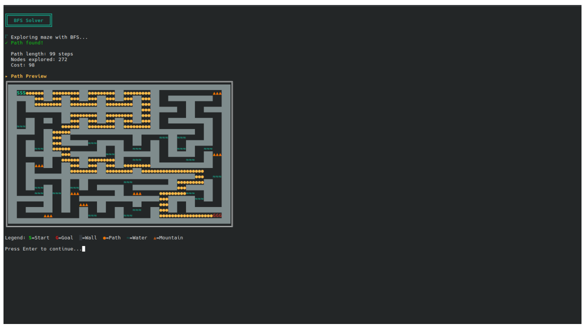
5. Conclusion

Here's our key takeaway: Object-oriented design isn't just about classes and syntax—it's a practical way to manage complexity, make code reusable, and enable safe evolution. In this project (Path Analyzer & Maze Solver) we used encapsulation to protect internal state, abstraction to expose simple solver and renderer APIs, and polymorphism-friendly design so different algorithms (BFS, Dijkstra, A*) can be swapped without rewriting the UI. The result is code that's easier to reason about, test, and extend — for example adding A* or a new terrain type becomes a local change rather than a cross-cutting rewrite.



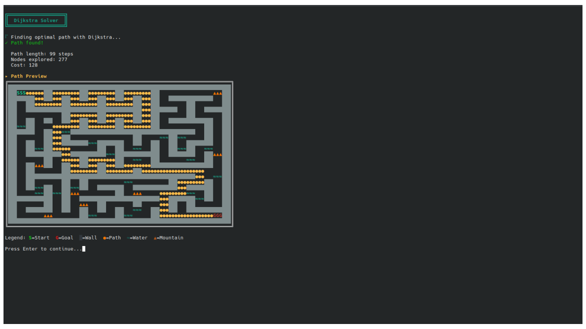
```
MAZE SOLVER

         Advanced Pathfinding with OOP Concepts
    Encapsulation • Abstraction • Operator Overloading
─────────────────────────────────────────────
  Initializing maze solver engine...
✓ System ready!

Press Enter to continue...
```

(a) A



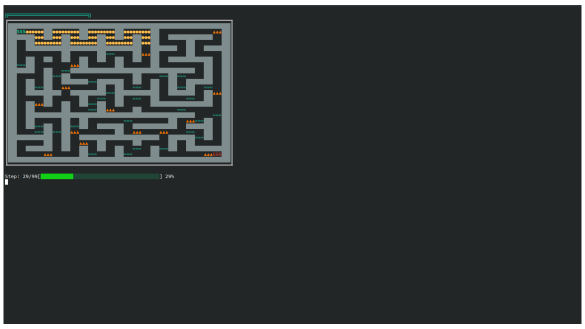
```
BFS Solver

  Exploring maze with BFS...
✓ Path found!

  Path length: 99 steps
  Nodes explored: 272
  Cost: 98

▸ Path Preview

Legend: S=Start  G=Goal  █=Wall  ●=Path  ≈=Water  ▲=Mountain

Press Enter to continue...
```

(b) B



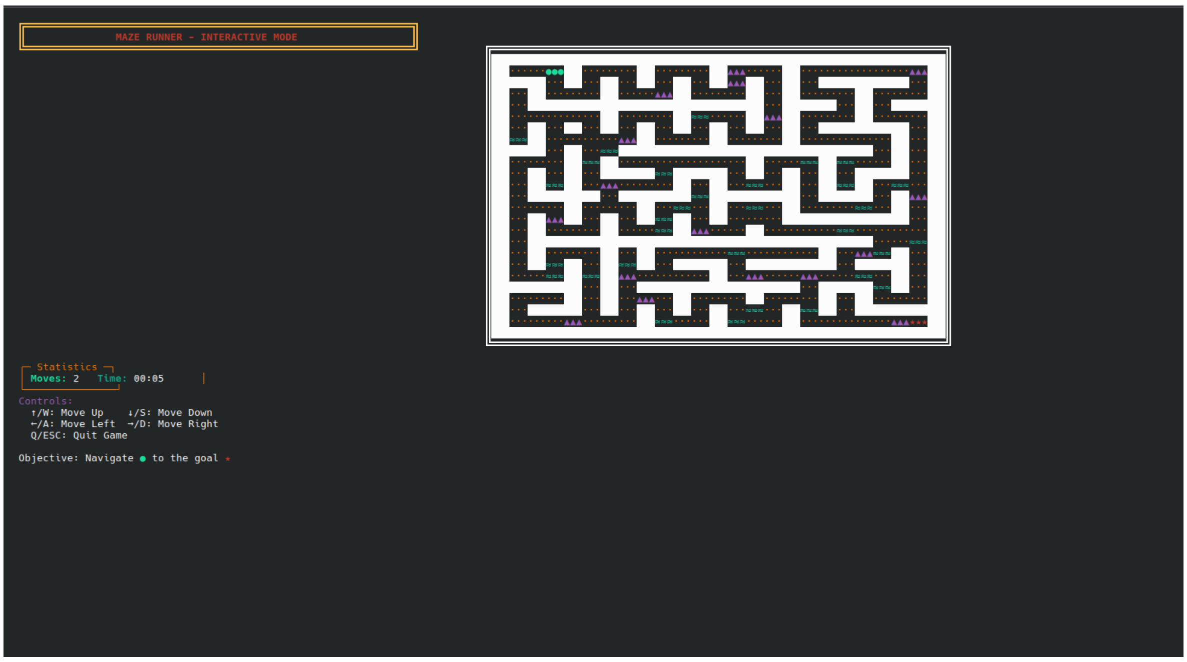
```
Dijkstra Solver

  Finding optimal path with Dijkstra...
✓ Path found!

  Path length: 99 steps
  Nodes explored: 277
  Cost: 128

▸ Path Preview

Legend: S=Start  G=Goal  █=Wall  ●=Path  ≈=Water  ▲=Mountain

Press Enter to continue...
```

(c) C



```
Step: 29/99[██████                    ] 29%
```

(d) D



```
MAZE RUNNER - INTERACTIVE MODE

┌ Statistics ┐
 Moves: 2   Time: 00:05       │
└──────────────┘
Controls:
  ↑/W: Move Up     ↓/S: Move Down
  ←/A: Move Left  →/D: Move Right
  Q/ESC: Quit Game

Objective: Navigate ● to the goal ★
```

(e) E

Figure 1: Project Screenshots