Searching on a Grid

Muneeb Azfar Nafees

April 2025

Contents

1 Abstract

In this project, I studied pathfinding algorithms on grid-based environments to emulate navigation challenges faced by robots and virtual agents. I implemented breadth-first search, depth-first search, and A* search on randomly generated mazes with obstacles to observe their behavior and performance. I applied core graph traversal and heuristic-search concepts to systematically compare how each algorithm balances exploration breadth and path optimality. Key findings include that A* consistently discovers the shortest paths while exploring the fewest cells, BFS also finds optimal paths but explores a substantially larger portion of the grid, and DFS yields highly variable, often much longer, paths with unpredictable exploration effort.

2 Results

2.1 Analysis 1: Relationship between Obstacle Density and Reachability

To determine how obstacle density affects the probability of finding a path, I wrote a helper method in `Simulation.java` named `analysisOne()`, which loops densities from 0.0 to 0.9 in 0.1 increments and for each density runs 10 independent trials of A* search (delay = 0, no visualization). The core of that method is shown here:

```
public void analysisOne(){
    for(int j = 0; j < 10; j++){
        double density = j/10.0;
        int count = 0;
        for(int i = 0; i < 10; i++){
            Maze maze = new Maze(20,20, density);
            MazeAStarSearch astar = new MazeAStarSearch(maze);
            if(astar.search(maze.getStart(), maze.getTarget(), false, 0) != null){
                count++;
            }
        }
        System.out.println("Density: - Paths Found:  out of 10");
        System.out.println("Probability: " + count/10.0);
    }
}
```

Running `analysisOne()` produces the following success probabilities:

Density	Success Probability
0.0	1.0
0.1	1.0
0.2	1.0
0.3	0.9
0.4	0.6
0.5	0.1
0.6	0.0
0.7	0.0
0.8	0.0
0.9	0.0

Table 1: Success probability of A* search versus obstacle density (10 trials per density).

These results show that with few or no obstacles (density ≤ 0.2), A* (or any other search algorithm for that instance) consistently finds a path in every trial. As obstacles increase, success probability declines sharply: at density 0.4, the path is found in only 60% of trials,

and beyond 0.5, it drops to zero. This indicates a threshold near 0.5, where the maze becomes too blocked for any path to exist.

2.2 Analysis 2: Path Lengths of DFS, BFS, and A*

To compare the lengths of paths produced by each algorithm, I implemented the `analysisTwo()` helper in `Simulation.java`, which iterates grid sizes from 10×10 up to 100×100 (step 10) at a fixed obstacle density of 0.2. For each grid, it runs depth-first search, breadth-first search, and A* search (with no visualization) on the same maze instance, resetting between algorithms and recording the length of the returned path (or zero if no path exists). The key portion of that method is shown below:

```
public void analysisTwo() {
    for (int i = 10; i <= 100; i += 10) {
        int size    = i;
        double density = 0.2;
        Maze maze   = new Maze(size, size, density);

        // DFS
        LinkedList<Cell> pathDfs = new MazeDepthFirstSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int lenDfs = (pathDfs == null) ? 0 : pathDfs.size();
        maze.reset();

        // BFS
        LinkedList<Cell> pathBfs = new MazeBreadthFirstSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int lenBfs = (pathBfs == null) ? 0 : pathBfs.size();
        maze.reset();

        // A*
        LinkedList<Cell> pathAstar = new MazeAStarSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int lenAstar = (pathAstar == null) ? 0 : pathAstar.size();

        System.out.println(
                "Grid Size: " + size + " x " + size +
                "\n Length of path (DFS): " + lenDfs +
                "\n Length of path (BFS): " + lenBfs +
                "\n Length of path (A*): " + lenAstar
        );
    }
}
```

Grid Size	DFS Length	BFS Length	A* Length
10×10	31	5	5
20×20	40	14	14
30×30	231	23	23
40×40	195	43	43
50×50	593	55	55
60×60	463	25	25
70×70	6	6	6
80×80	236	94	94
90×90	466	44	44
100×100	1411	49	49

Table 2: Path lengths found by DFS, BFS, and A at density 0.2 across grid sizes.

From these results, BFS and A* consistently find the same shortest path lengths, since BFS guarantees the fewest steps in an unweighted grid and A* uses an admissible heuristic, both converge on the optimal solution. In contrast, DFS produces highly variable and often much longer paths because it explores deeply along one branch without regard for overall distance. As grid size increases, optimal path lengths (BFS/A*) grow roughly in proportion to the grid dimensions, while DFS lengths can spike unpredictably. These findings highlight that while DFS may discover a connection more quickly in some cases, it is not suitable when path optimality is important.

2.3 Analysis 3: Cells Explored by DFS, BFS, and A*

To compare the search effort of each algorithm, I added an `analysisThree()` method in `Simulation.java`. This method iterates grid sizes from 10×10 up to 100×100 at a fixed obstacle density of 0.2, runs DFS, BFS, and A* in sequence on the same maze (resetting between algorithms), and uses `maze.countVisitedCells()` to measure how many cells each search marked as visited. The core of that method is:

```
public void analysisThree() {
    for (int i = 10; i <= 100; i += 10) {
        int size    = i;
        double density = 0.2;
        Maze maze   = new Maze(size, size, density);

        // DFS
        new MazeDepthFirstSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int cellsExploredDfs = maze.countVisitedCells();
        maze.reset();
```

```
        // BFS
        new MazeBreadthFirstSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int cellsExploredBfs = maze.countVisitedCells();
        maze.reset();

        // A*
        new MazeAStarSearch(maze)
            .search(maze.getStart(), maze.getTarget(), false, 0);
        int cellsExploredAstar = maze.countVisitedCells();

        System.out.println(
            "Grid Size: " + size + " x " + size +
            "\n Number of cells explored (DFS): " + cellsExploredDfs +
            "\n Number of cells explored (BFS): " + cellsExploredBfs +
            "\n Number of cells explored (A*): " + cellsExploredAstar + "\n"
        );
    }
}
```

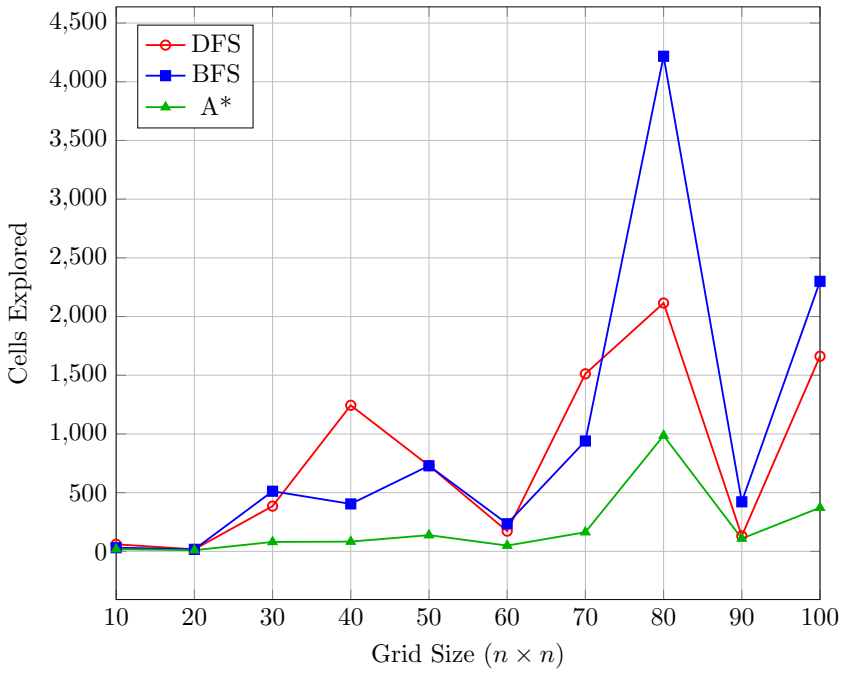

Figure 1: Number of cells explored by DFS, BFS, and A* as grid size increases at obstacle density 0.2.

Across all grid sizes, A* explores the fewest cells, benefiting from its heuristic to focus the search toward the goal. BFS explores significantly more cells than A*, since it expands uniformly outward until the target is reached. DFS shows highly variable exploration counts, sometimes very low when it happens upon the goal quickly, but often much higher than both BFS and A*, because it follows one branch deeply before backtracking. These results demonstrate that A* achieves the best balance of completeness and efficiency in grid-based pathfinding.

3 Extensions

3.1 Graphical User Interface Extension

3.1.1 What I did and why

I created a Swing-based GUI application, `MazeSearchApp`, to allow users to specify maze dimensions, obstacle density, and the search algorithm at runtime. This interface replaces the hard-coded simulation parameters with input fields, making the tool more accessible and interactive for exploration or demonstration purposes.

3.1.2 Outcome

When launched, `MazeSearchApp` prompts for rows, columns, obstacle density, and algorithm choice, then animates the chosen search (DFS, BFS, or A*) on the generated maze. After the path is found, the GUI displays three key metrics:

- the total number of cells explored
- the length of the found path
- the elapsed time in milliseconds

3.1.3 How to replicate this extension in the code base

The GUI class is located in the `extension/` directory alongside the core maze and search classes. To run:

```
cd extension
javac *.java
java MazeSearchApp
```

3.2 Wall-Follower (Right-Hand Rule) Extension

3.2.1 What I did and why

I implemented a "wall-follower" strategy, inspired by how a human might navigate a maze, by always moving forward when possible, turning right on walls or obstacles, and backtracking when no unvisited cell remains. This logic is encapsulated in `MazeWallFollowerSearch.java`, which extends `AbstractMazeSearch` and maintains orientation to simulate step-by-step walking rather than teleporting between frontier cells. I also updated the simulation class (in the `extension/` directory) to include this new searcher alongside DFS, BFS, and A* for direct comparisons.

3.2.2 Results

Analysis 1: Reachability vs. Obstacle Density The results show that the wall-follower nearly matches A* in reachability at low densities but loses effectiveness slightly earlier (around density 0.1 vs. 0.3 for A*/DFS/BFS), reflecting its local, non-heuristic nature.

Density	A*/DFS/BFS Success	Wall-Follower Success
0.0	1.0	1.0
0.1	1.0	0.9
0.2	1.0	0.9
0.3	0.8	0.8
0.4	0.4	0.3
0.5	0.2	0.2
0.6	0.0	0.0
0.7	0.0	0.0
0.8	0.0	0.0
0.9	0.0	0.0

Table 3: Success probability of A* vs. Wall-Follower across densities (10 trials each).

Analysis 2: Path Length Comparison The wall-follower's path lengths are consistently larger than A* and BFS and often larger than DFS. Whereas BFS and A* produce the optimal, shortest routes (identical lengths), the wall-follower must trace the maze perimeter and backtrack extensively, leading to much longer traversals, especially on larger grids. In some cases (e.g., 70×70 and 90×90), its path length exceeds the grid's perimeter, reflecting loops and multiple backtracking. This underscores that although wall-following can guarantee eventual exit in simply-connected mazes, it is highly inefficient compared to heuristic or breadth-first methods.

Grid	DFS	BFS	A*	Wall-Follower
10×10	15	7	7	23
20×20	84	26	26	60
30×30	91	27	27	143
40×40	261	7	7	249
50×50	400	50	50	410
60×60	366	44	44	938
70×70	351	53	53	1225
80×80	792	90	90	600
90×90	258	66	66	1690
100×100	651	45	45	871

Table 4: Path lengths (steps) found on 10×10–100×100 grids at density 0.2, comparing DFS, BFS, A*, and Wall-Follower.

Analysis 3: Cells Explored A* and the wall-follower visit the same cells when they succeed (since the wall-follower backtracks similarly), but BFS and especially DFS often explore far more of the grid. The wall-follower's exploration cost is close to A*'s but without optimal guarantees.

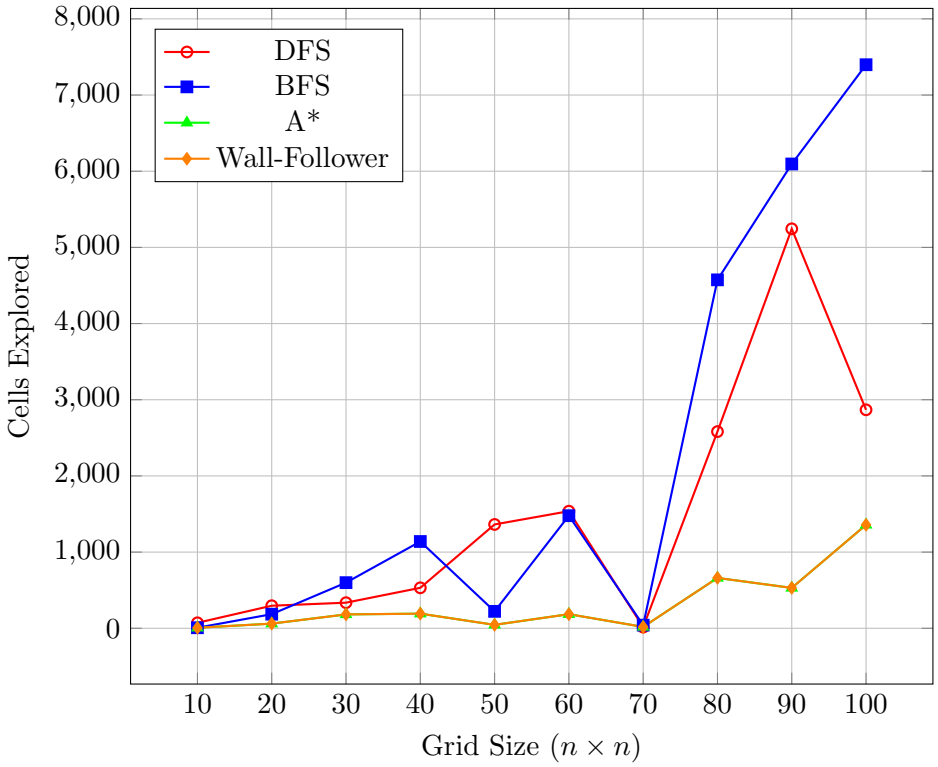

Figure 2: Cells explored by each algorithm on density 0.2 mazes.

3.2.3 How to replicate this extension in the code base

In the `extension/` directory:

```
javac *.java
java Simulation                 # runs all analyses including MazeWallFollowerSearch
java MazeWallFollowerSearch     # launches the wall-follower searcher
```

This compiles and runs the wall-follower extension alongside the original algorithms. To just visualize the wall-follower searcher use the main method for the wMazeWallFollowerSearch class.

4 Acknowledgments

Acknowledgments

1. Trail: Creating a GUI With Swing – Oracle Java™ Tutorials
2. How to Use BorderLayout – Oracle Java™ Tutorials
3. How to Use Combo Boxes – Oracle Java™ Tutorials
4. How to Use Text Fields – Oracle Java™ Tutorials
5. What does SwingUtilities.invokeLater do? – Stack Overflow
6. Enum (Java Platform SE 8) – Oracle Help Center
7. Pausing Execution with Sleep – Oracle Java™ Tutorials
8. Maze-solving algorithm – Wikipedia