CS231 Project 7: Mazesearch

November 5th, 2023

Abstract

In this project, we implemented various searching algorithms to navigate mazes. The core purpose of the assignment was to explore and understand different search algorithms, such as Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search (implemented using Stacks, Queues, and a Heap/Priority Queue respectively). These algorithms were applied to solve maze exploration problems, where the objective was to find the shortest path from a given start point to a target point within a maze. Through the implementation of these algorithms, we aimed to compare their efficiency and suitability for different maze configurations. The project provides a practical understanding of algorithmic exploration and serves as a valuable learning experience for studying search strategies in computer science applications.

Results / Exploration

What is the relationship between the density of obstacles to probability of reaching the target? Note that this should be independent of whichever search algorithm you use, as they should all find the target if it is reachable. Specifically, around what density of obstacles does the probability of reaching the target drop to 0?

To test this, I tested each density of obstacles from 0 to 1 with 0.1 increments 100 times each using MazeDepthFirstSearch (didn't really matter which one I chose as as said the ability to find the target is independent of the algorithm) For the sake of consistency, I used a 10 x 10 Maze for all tests, and the same start and ending target, on opposite corners of the grid. Whether or not the algorithm was able to find its target was then counted out of 100. The results are as follows:

Obstacle Density (0-1)	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
# of successes (out of 100)	100	90	62	29	9	3	0	0	0	0	0

As the density of obstacles increases, the probability of reaching the target decreases. When the maze has a very low obstacle density, the probability of finding a path from the start to the target is close to 1. However, as the obstacle density increases, the available paths become limited, reducing the probability of reaching the target. Around a certain density threshold (.5 typically), the probability

of reaching the target drops to 0 because there are no viable paths left to navigate through the maze.

What is the relationship between the lengths of the paths found by DFS, BFS, and A*?

The length of the paths found by DFS, BFS, and A* depends on the specific maze configuration and the algorithms' search strategies.

To test this, I tested each algorithm 100 times and averaged the lengths of the paths it took to get to the target (if it did). For the sake of consistency, I used a 10 x 10 Maze for all tests, 0.1 probability, and the same start and ending target, on opposite corners of the grid. The results are as follows:

Algorithm	DFS	BFS	A*
Average length of the path	37.0337	19.0	19.0

DFS: DFS does not necessarily find the shortest path; it finds the first path it encounters. The path length can vary significantly based on the order in which DFS explores the maze branches. DFS can find long paths if it explores deep into one branch before considering other branches.

BFS: BFS explores all paths on a level-by-level basis. It guarantees finding the shortest path if all edges have the same weight. The path length found by BFS represents the shortest possible path from the start to the target in the given maze.

A*: A* uses heuristics to estimate the cost of reaching the target from a particular cell. If the heuristic is admissible (never overestimates the cost), A* guarantees finding the shortest path. The path length found by A* represents the shortest possible path considering both the actual cost to reach the cell and the estimated cost from that cell to the target.

What is the relationship between the average number of Cells explored by DFS, BFS, and A*?

The average number of cells explored by DFS, BFS, and A* is influenced by the maze structure and the algorithms' exploration strategies.

I used the exact same test as the previous question but instead I used numRemainingCells() and subtracted from the total number of cells, 100 (in a 10x10).

Algorithm	DFS	BFS	A*
Average # of Cells Explored	67.3295	98.1047	76.0

DFS: DFS explores deep into branches before backtracking. The number of explored cells can vary significantly based on the branching factor and the depth of the explored paths. In densely connected mazes, DFS might explore a large number of cells before finding a path. DFS's exploration efficiency is influenced by the specific maze structure. If it encounters dead-ends early, it might explore fewer cells.

BFS: BFS explores cells level by level, ensuring that it explores all possible paths before moving to the next level. This can lead to a larger number of cells explored, especially in mazes with multiple paths.

A*: A* explores cells based on both the actual cost to reach them and the estimated cost from those cells to the target. A* tends to explore fewer cells than uninformed search algorithms like BFS and DFS, especially when a good heuristic is used. A* explores cells efficiently, focusing on the most promising paths and avoiding unnecessary exploration.

Reflection

A semi-brief overview of the data structure implemented/used in this project. What was the role of the PriorityQueue / Heap in this project? How was it used in the A* algorithm?

In this project, a binary heap-based PriorityQueue data structure was implemented and utilized, specifically in the A* algorithm, to efficiently manage cells during pathfinding in a maze. A PriorityQueue is a type of queue where each element has a priority, and elements are dequeued based on their priorities. A binary heap is an efficient implementation of a PriorityQueue using a binary tree structure. The PriorityQueue allowed the algorithm to handle cells based on their priorities, where the priority was determined by a combination of the actual cost to reach a cell (from the starting cell) and a heuristic estimate of the cost to reach the target cell. This combination ensured that cells with lower total estimated costs were explored first, guiding the search towards the most promising paths.

The binary heap structure of the PriorityQueue facilitated fast insertion and removal operations, crucial for the efficiency of algorithms like A*. Inserting a cell with its priority and removing the cell with the highest priority (lowest total estimated cost) could be done in logarithmic time, making the overall pathfinding process more efficient.

In the A* algorithm, the PriorityQueue was employed to keep track of the cells that needed to be explored. As the algorithm progressed, cells were added to the PriorityQueue with their priority values calculated based on the sum of the cost to reach the cell from the start and the heuristic estimate of the remaining cost to reach the target. The binary heap structure ensured that the cell with the lowest total estimated cost (priority) was always at the front of the PriorityQueue. This data structure significantly improved the algorithm's speed and accuracy in finding optimal paths in the maze.

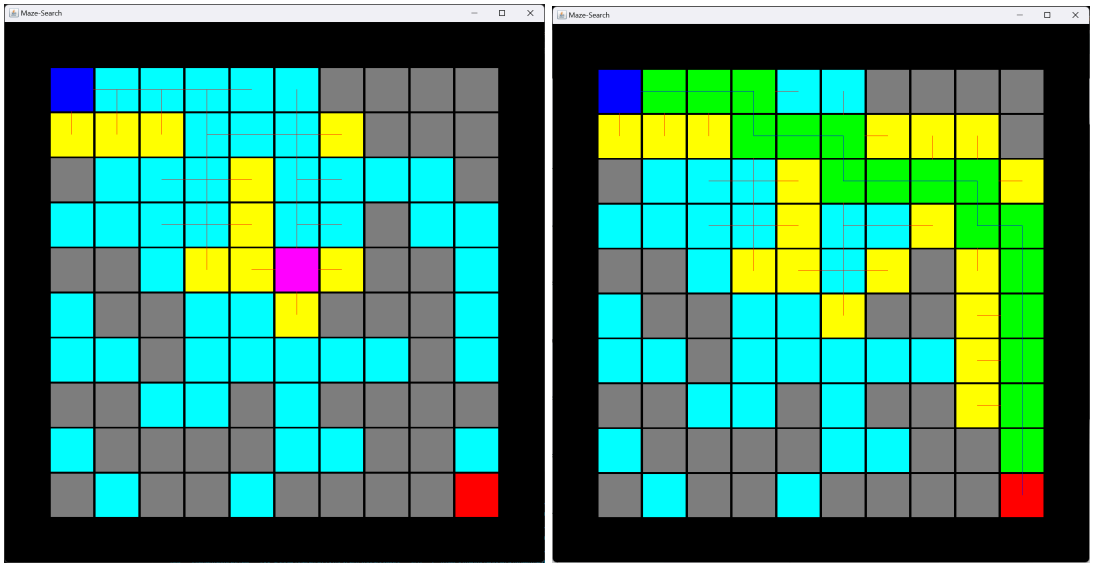
Extensions

Implement different types of Cells, like mud (where it's slow to walk through) or ice (where it's difficult to turn, but going straight takes less time). How do your algorithms do in this changed environment?

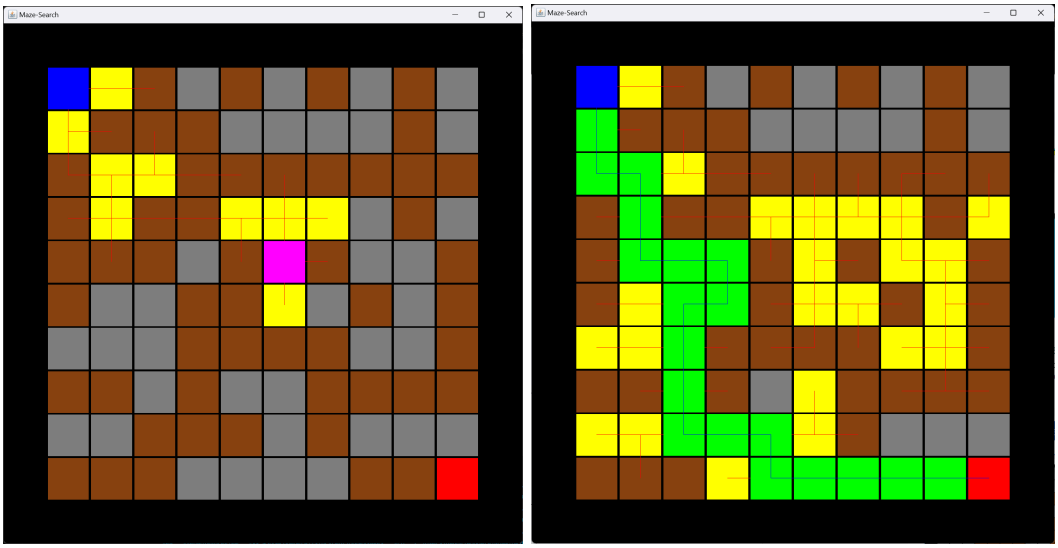
To implement different types of Cells, such as mud (slower to walk through) or ice (faster to go through), I extended the existing CellType enumeration to include these new types.

```
public enum CellType {

	FREE, OBSTACLE, MUD, ICE;

}
```

Both these cells typically only affect the A* Search that prioritizes exploring whichever path has the shortest distance to the target, by adding or subtracting one from the distance. In the case of a MUD Cell, it counts as walking through 2 cells, and with the case of an ICE cell, it counts as not walking through anything at all.



When evaluating which is the best path to take, the line will mostly prioritize ones that have the most amount of ice Cells, because they don't count as many traversals.



When evaluating which is the best path to take, the line will mostly avoid MUD Cells, because they count as double the traversals.

MUD Cells will also affect BFS in the way that since they count as double the Depth, it would not go in the same progression as it would if every cell was a normal cell.

Realistically, some of these algorithms don't make a lot of sense from a human's perspective. How are you, a human, going to teleport around the maze to get to the next Cell your search wants to move to? Analyze the number of steps required in a grid to 'walk to' the next Cell while finding the target and see how your search algorithms fare.

When analyzing the number of steps required to walk to the next cell in a grid-based maze, we can consider two important factors: distance in terms of grid cells and actual path length in terms of steps taken.

Distance in Grid Cells:

- BFS explores the shortest paths first, so it generally finds the shortest distance in terms of grid cells. It guarantees the shortest path if all edges have the same weight.
- A* also finds the shortest path when using an admissible heuristic function. It explores paths based on a combination of actual cost and heuristic estimate.
- DFS does not guarantee the shortest path; it may explore longer paths before finding the target. Therefore, it might take more grid cells to reach the target compared to BFS and A*.

Actual Path Length (Steps Taken):

- BFS and A* tend to find shorter paths in terms of steps taken because they explore the maze systematically, ensuring that they find the target along the shortest possible route. A* particularly excels when using a good heuristic as it guides the search efficiently.
- DFS, while not guaranteeing the shortest path, might find shorter paths in terms of steps taken if it stumbles upon the target early during its exploration. However, it's more likely to explore longer paths due to its nature of deep diving into the search space.

In summary, BFS and A* are more likely to find the shortest path both in terms of grid cells and actual steps taken, provided A* uses an admissible heuristic. DFS might find shorter paths in terms of steps taken if it gets lucky, but it's not reliable for finding the shortest path. When comparing these algorithms in a grid-based maze, BFS and A* are generally better choices if you want to find the shortest path in both grid cells and actual steps taken. The choice between BFS and A* depends on the specific requirements and whether an admissible heuristic can be applied effectively in your problem domain.

References/Acknowledgements

A lot of troubleshooting was done in order to make this work, the TAs, Professor Bender and Professor Lage were both consulted as there were many, many errors. Although most errors occur because I missed very small things with my code.
