Team Transformers

Aashish Giri, *S02078722*, Graduate
Alson Garbuja, *S02085115*, Graduate
Kamal Batala, *S02042847*, Undergrad
Sishir Kharel, *S02087572*, Graduate

Part 1: Basic Pathfinding

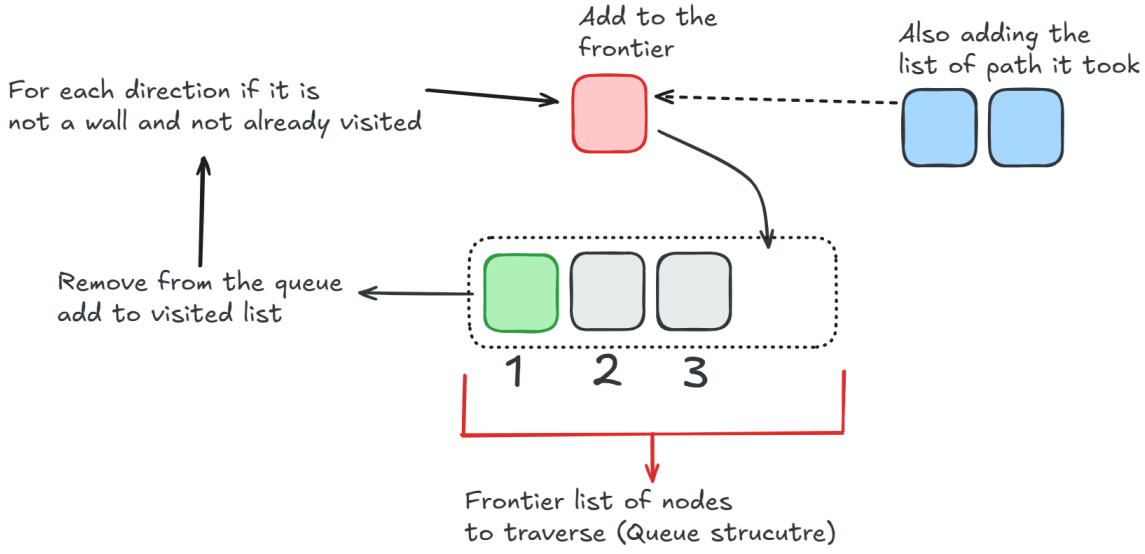
In this part we created 3 algorithms (Breadth First Search, Depth First Search and A* Search) to find the shortest path in the provided 2D maze with starting point and single goal point.

Findings for BFS (Breadth First Search) algorithm

The algorithm

The Breadth-First Search (BFS) algorithm explores all neighboring nodes at the current depth before proceeding to nodes at the next level.

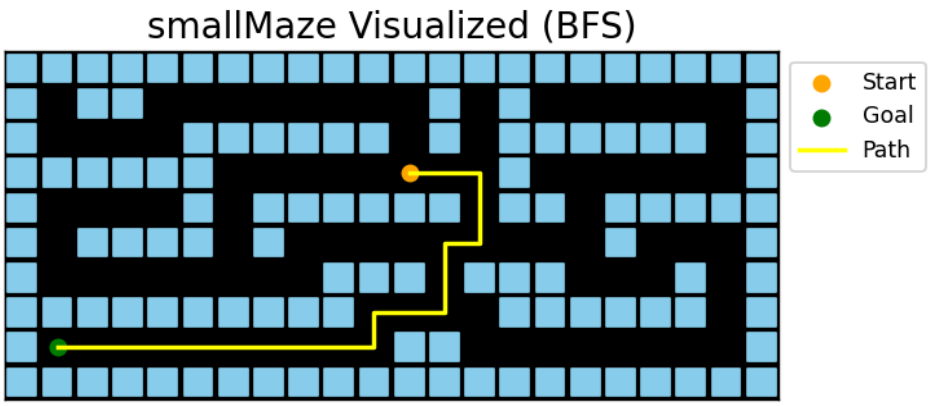
To achieve this, we used a **queue**—a FIFO (First-In, First-Out) data structure—where nodes discovered first are processed first, ensuring a level-by-level traversal of the graph or tree.
Below is a diagram showing the algorithm's pseudo code.



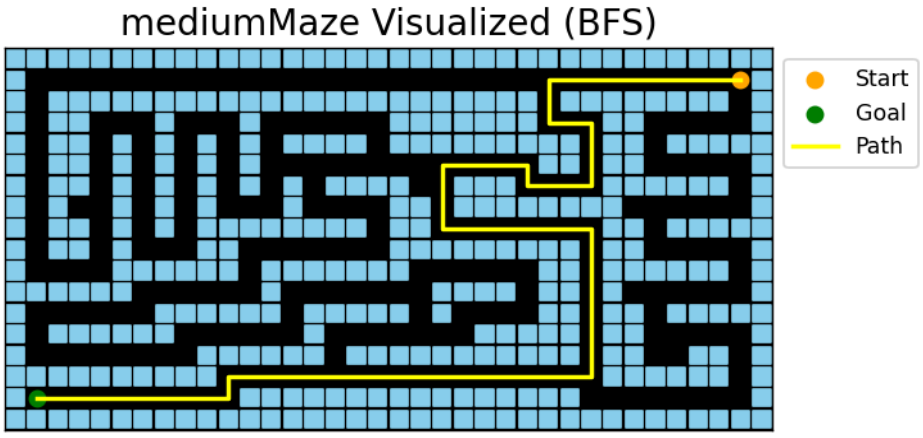
Run the loop until the either frontier is empty or the goal state is found

The result

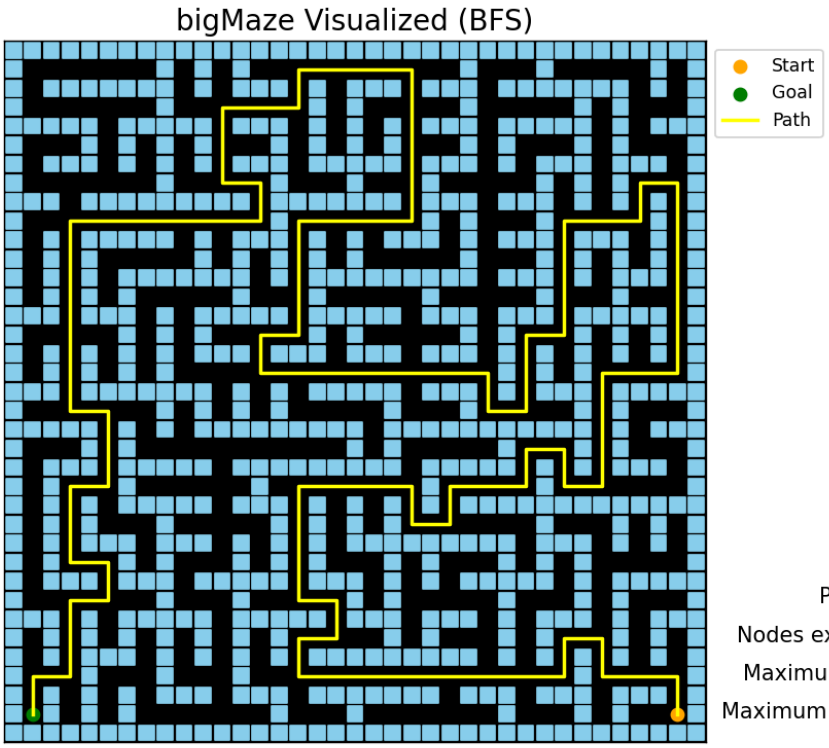
After running the algorithm for the given small, medium, big and open mazes we have the following results.



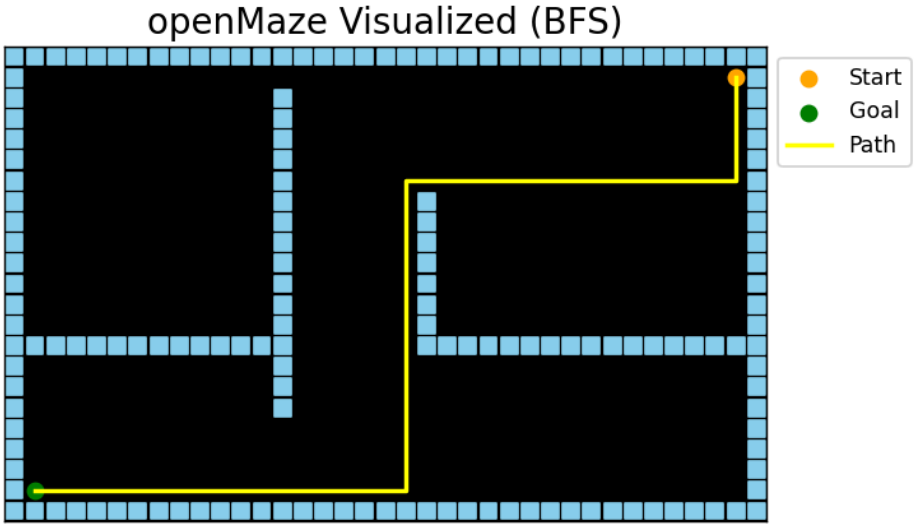
Path cost: 19
Nodes expanded: 92
Maximum depth: 18
Maximum fringe size: 8



Path cost: 68
Nodes expanded: 270
Maximum depth: 68
Maximum fringe size: 8



Path cost: 210
Nodes expanded: 621
Maximum depth: 210
Maximum fringe size: 8



Path cost: 54
Nodes expanded: 683
Maximum depth: 53
Maximum fringe size: 25

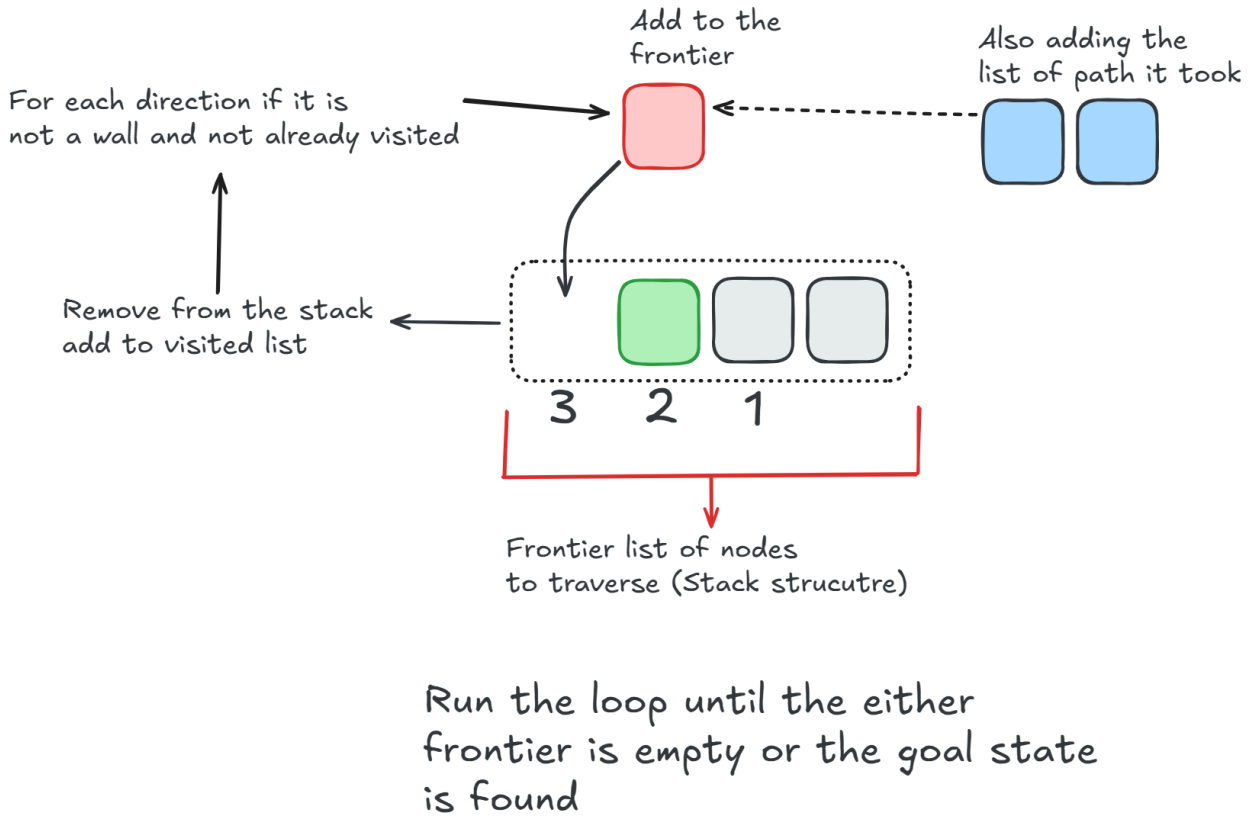
Findings for DFS (Depth First Search) algorithm

The algorithm

The Depth-First Search (DFS) algorithm explores a path as deeply as possible before backtracking to the most recent node with unvisited neighbors, continuing this process until the goal is found.

We used a **stack**—a LIFO (Last-In, First-Out) data structure—where the most recently added node is processed first, enabling deep, branch-by-branch traversal of the graph or tree.

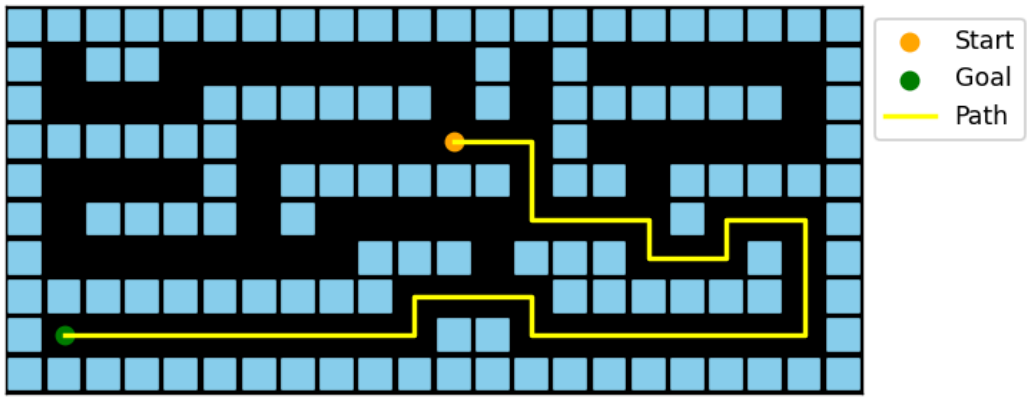
Below is a diagram showing the algorithm's pseudo code.



The result

After running the algorithm for the given small, medium, big and open mazes we have the following results.

smallMaze Visualized (DFS)



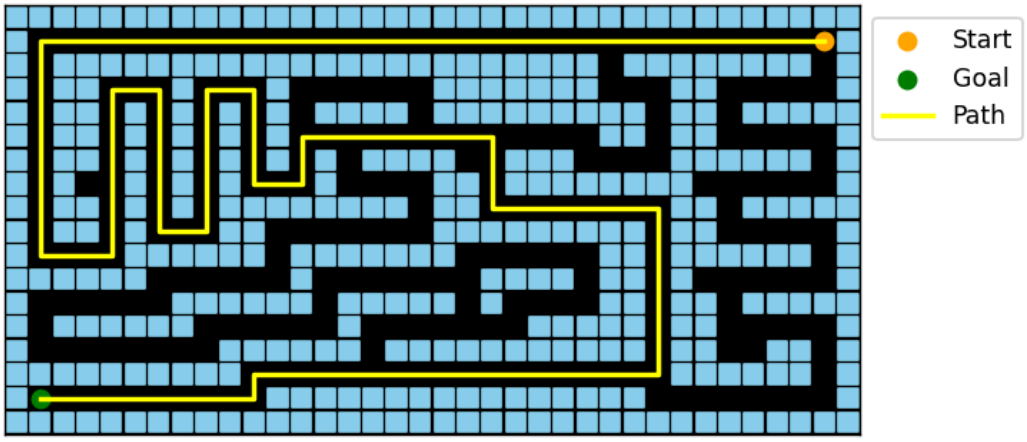
Path cost: 37

Nodes expanded: 38

Maximum depth: 37

Maximum fringe size: 7

mediumMaze Visualized (DFS)



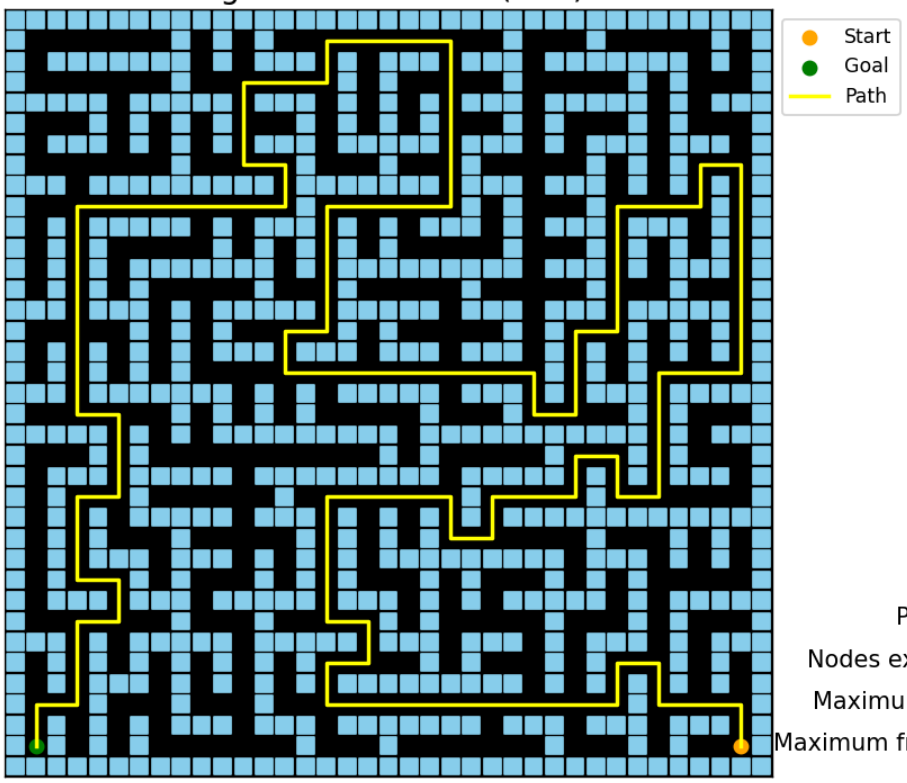
Path cost: 130

Nodes expanded: 145

Maximum depth: 130

Maximum fringe size: 9

bigMaze Visualized (DFS)



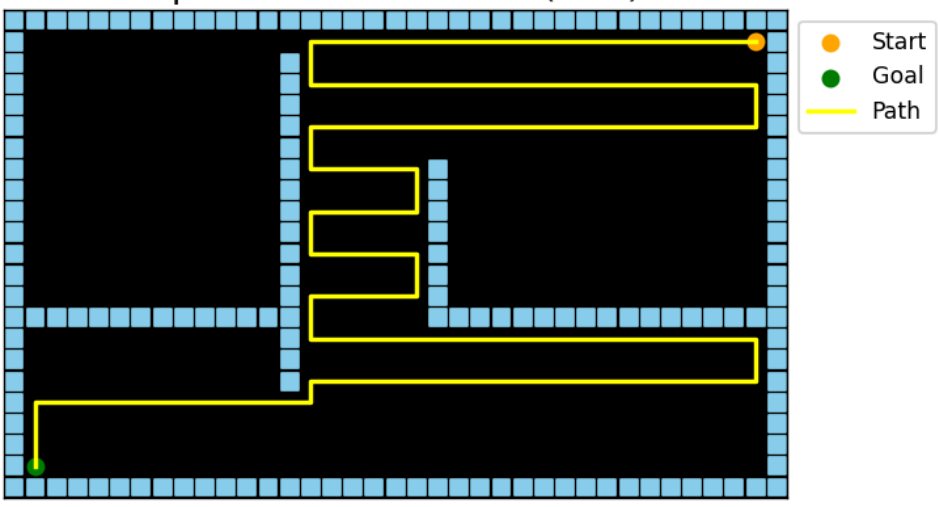
Path cost: 210

Nodes expanded: 388

Maximum depth: 223

Maximum fringe size: 39

openMaze Visualized (DFS)



Path cost: 158

Nodes expanded: 405

Maximum depth: 193

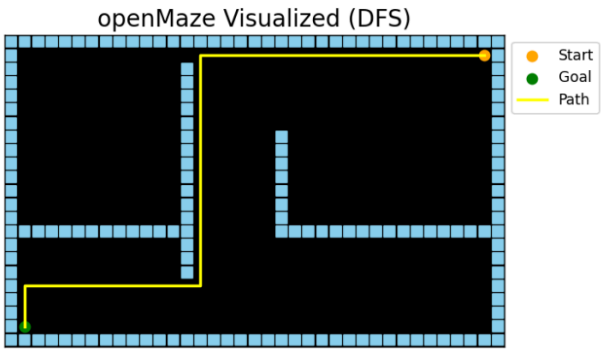
Maximum fringe size: 206

Interesting findings

While running the DFS algorithm on the given mazes, we observed several interesting findings, outlined below.

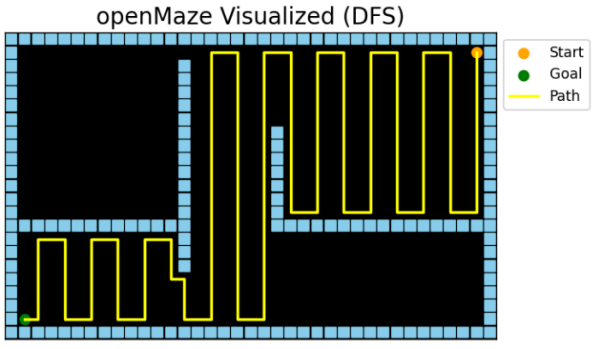
1. DFS doesn't always find the shortest path when multiple routes exist. In the big maze, ***both BFS and DFS had a path cost of 210*** since only one route was available, but in the open maze, ***BFS found a shorter path 54 compared to DFS 158***.
2. In general, ***DFS expands fewer nodes than BFS***, regardless of direction order. This happens because BFS explores all neighboring nodes before going deeper, causing it to expand more nodes when the goal lies deep or far from the starting point.
3. In contrast, ***DFS shows greater maximum depth and fringe size than BFS***. This is expected, as DFS explores deeper nodes first before backtracking to unvisited neighbors.
4. We can modify the direction order to potentially reduce the path cost and the number of nodes expanded. However, different direction orders can also increase them. For instance, when we reran the open maze with orders **[up, right, down, left]** and **[right, left, down, up]**, the results varied accordingly.

Using (up, right, down and left)



Path cost: 54
Nodes expanded: 212
Maximum depth: 113
Maximum fringe size: 89

Using (right, left, down and up)



Path cost: 226
Nodes expanded: 227
Maximum depth: 226
Maximum fringe size: 196

Findings for A* Search (A Star Search) algorithm

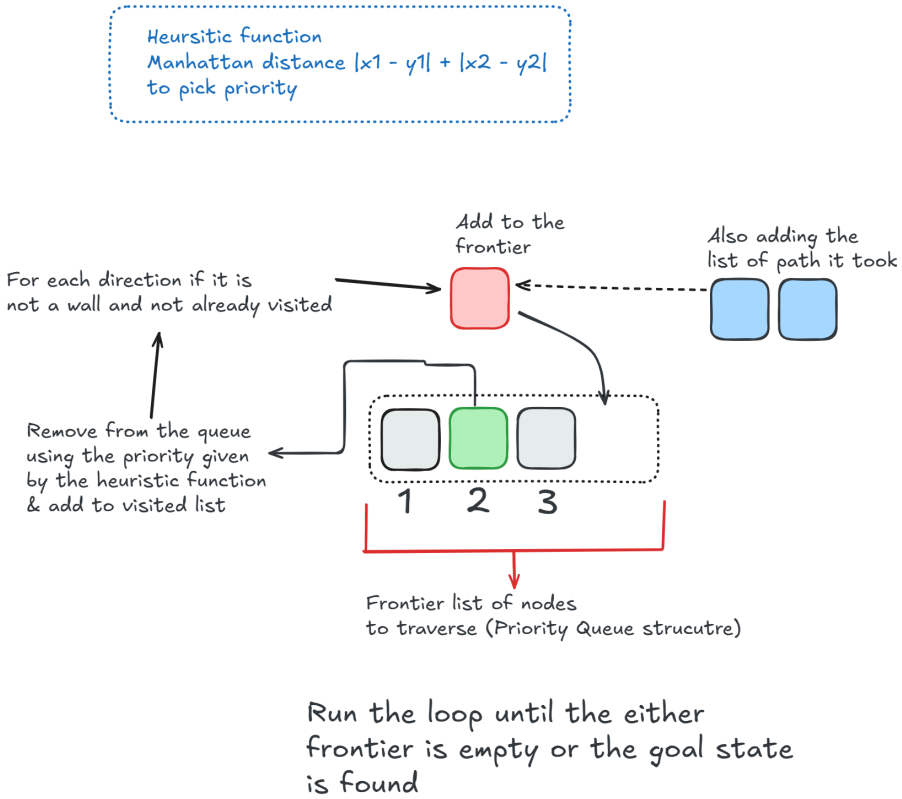
The algorithm

The A* search algorithm is an **informed search algorithm**, meaning it estimates how close each node is to the goal using a **heuristic function**. In our case, we used the **Manhattan Distance** as the heuristic. A* leverages this heuristic to prioritize which node to explore next.

Therefore, we used a **Priority Queue**—a queue where elements with higher priority are processed first (and in case of ties, **FIFO** is applied)—to manage the **frontier** (list of nodes to be explored).

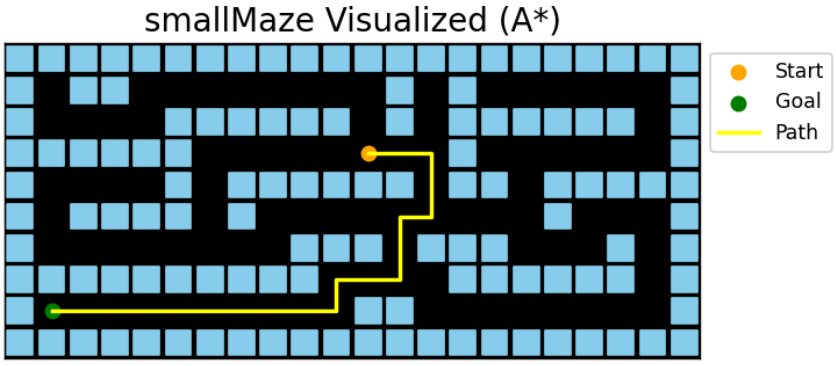
Note: *We implemented this using Python's built-in* ***heapq*** *module, which provides an efficient priority queue.*

Below is a diagram showing the algorithm's pseudo code.

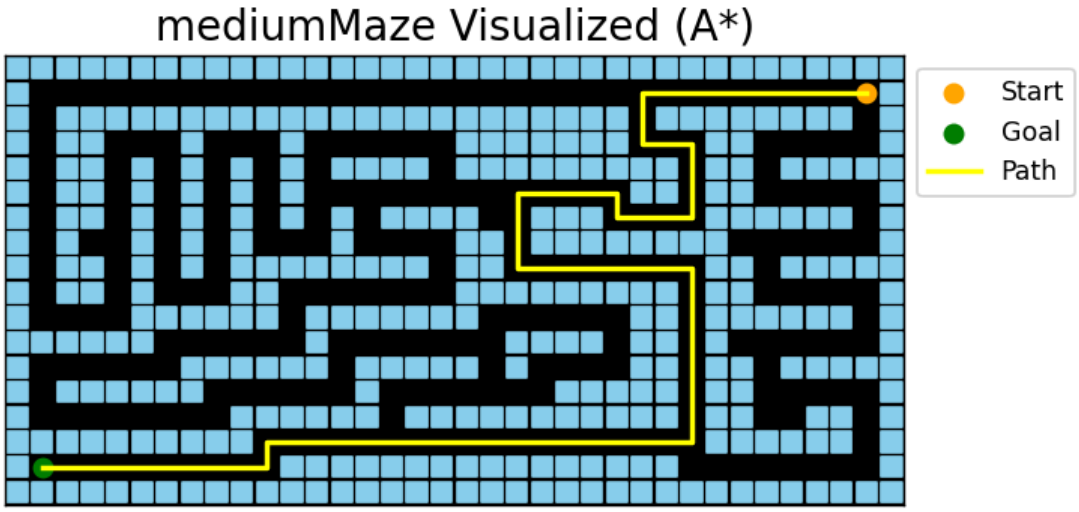


The result

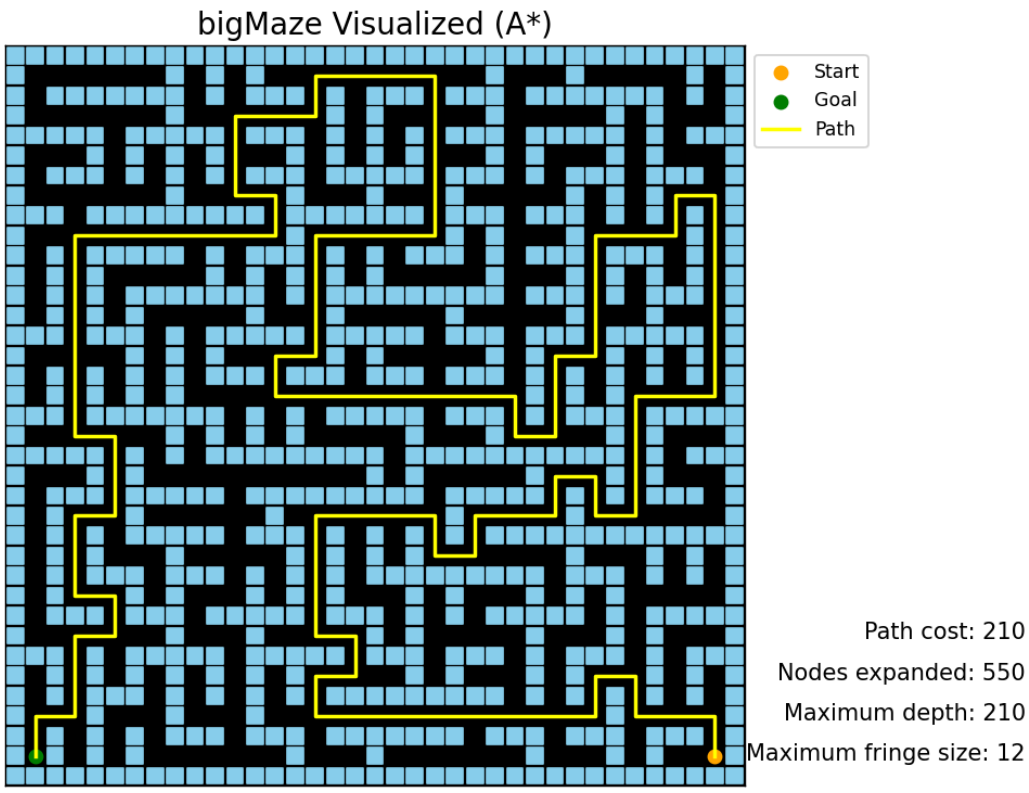
After running the algorithm for the given small, medium, big and open mazes we have the following results.

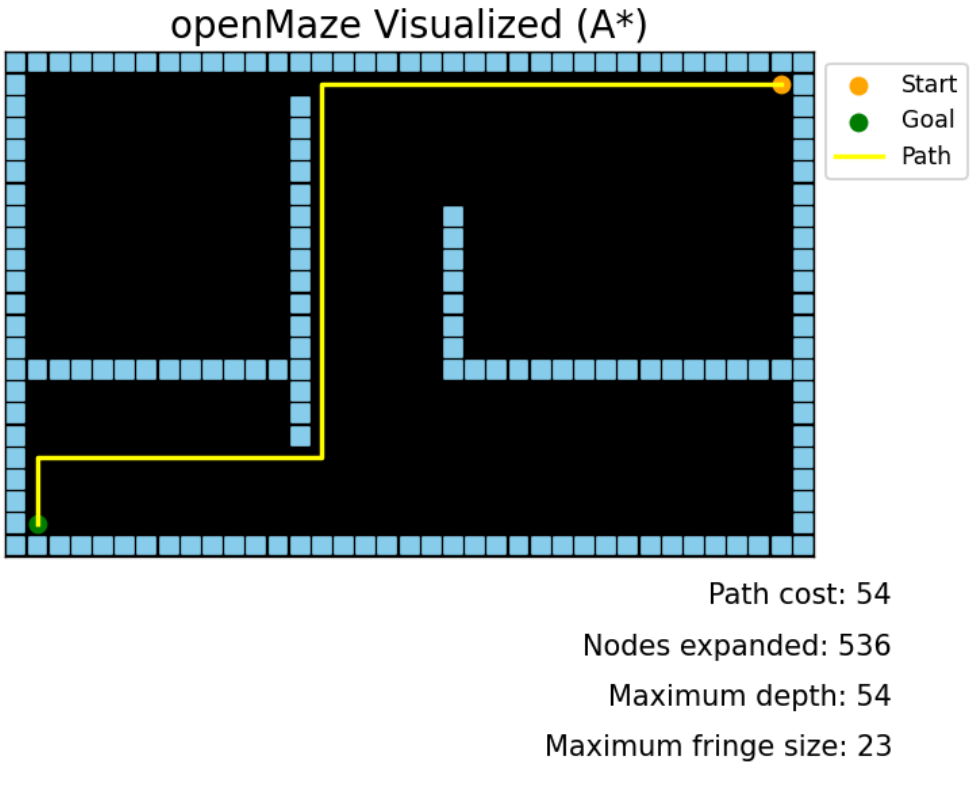


Path cost: 19
Nodes expanded: 54
Maximum depth: 19
Maximum fringe size: 8



Path cost: 68
Nodes expanded: 222
Maximum depth: 68
Maximum fringe size: 8


bigMaze Visualized (A*)
Start
Goal
Path
Path cost: 210
Nodes expanded: 550
Maximum depth: 210
Maximum fringe size: 12

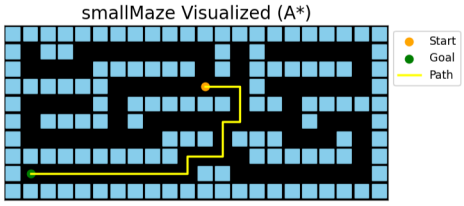

openMaze Visualized (A*)
Start
Goal
Path
Path cost: 54
Nodes expanded: 536
Maximum depth: 54
Maximum fringe size: 23

Interesting findings

Our key observations from running the A* search algorithm are as follows:

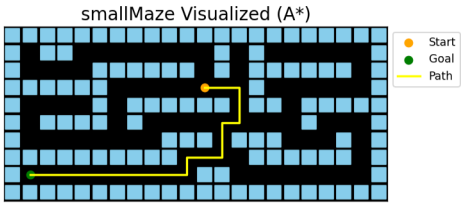
1. Since A* is an informed algorithm, the direction order of node expansion did not affect the number of nodes expanded or the maximum depth.
2. The **Manhattan distance** heuristic is **admissible** in our 2D maze as it never overestimates the cost to the goal.
3. The choice of heuristic had minimal impact in our case because each step cost was uniform (**1**)..

Manhattan Distance



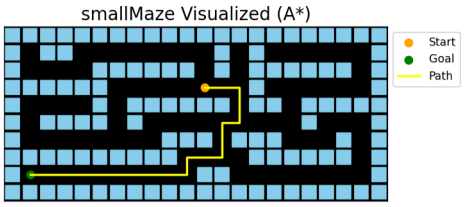
Path cost: 19
Nodes expanded: 54
Maximum depth: 19
Maximum fringe size: 8

Euclidean Function



Path cost: 19
Nodes expanded: 57
Maximum depth: 19
Maximum fringe size: 7

Chebyshev Function

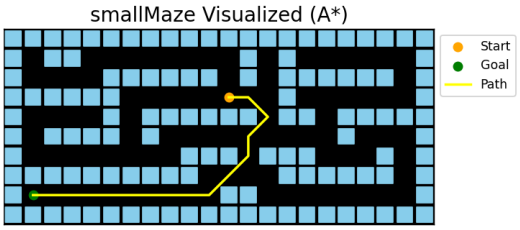


Path cost: 19
Nodes expanded: 57
Maximum depth: 19
Maximum fringe size: 6

We can see each heuristic function has a path cost of **19** while having very minimal increase in nodes expanded.

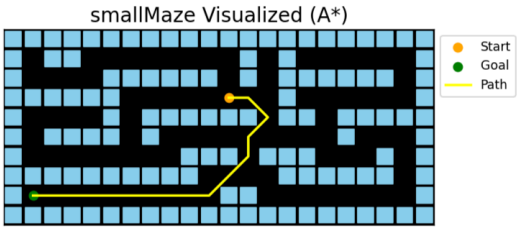
In order to decrease the path cost we could allow diagonal movement as seen below.

Manhattan Distance



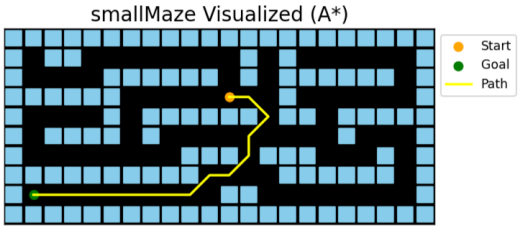
Path cost: 15
Nodes expanded: 37
Maximum depth: 15
Maximum fringe size: 14

Euclidean Function



Path cost: 15
Nodes expanded: 46
Maximum depth: 15
Maximum fringe size: 13

Chebyshev Function

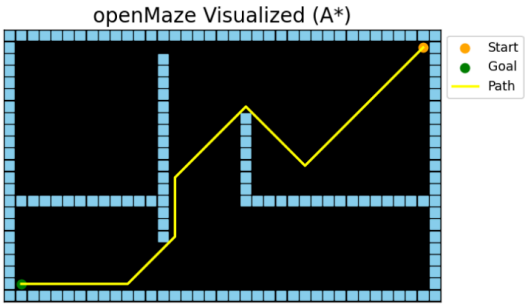


Path cost: 15
Nodes expanded: 54
Maximum depth: 15
Maximum fringe size: 11

Here the path cost **decreased by 4**. But we can notice the Manhattan distance function is more efficient than the other two (euclidean and chebyshev) heuristic functions due to the fact the cost for traverse is uniform (1).

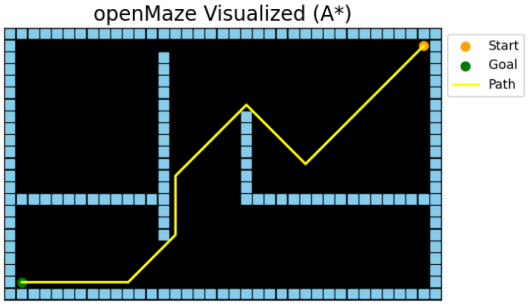
But using the different heuristic functions in the open maze while allowing diagonal movement we can see quite an interesting result.

Manhattan Distance



Path cost: 39
Nodes expanded: 110
Maximum depth: 39
Maximum fringe size: 85

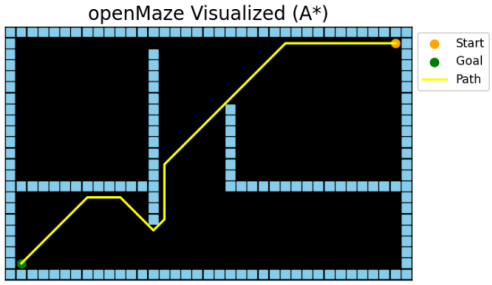
Euclidean Function



Path cost: 39
Nodes expanded: 162
Maximum depth: 39
Maximum fringe size: 82

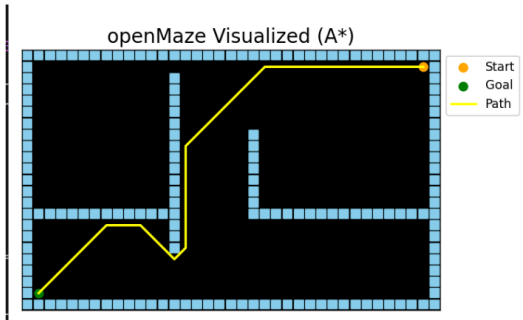
Chebyshev Function

Regular tie breaker



Path cost: 39
Nodes expanded: 315
Maximum depth: 39
Maximum fringe size: 50

Prefer deeper nodes



Path cost: 43
Nodes expanded: 44
Maximum depth: 43
Maximum fringe size: 115

The Chebyshev heuristic, with the **standard tie-breaker** (choosing the lower estimated cost when actual costs are equal), finds the shortest path but **expands nearly three times more nodes** than the other heuristics. By instead favoring deeper nodes (choosing a larger estimated cost), we can drastically reduce nodes expanded (to 44 in our case) at the cost of a slightly longer path.

Thus, we conclude that the **Manhattan distance** is the most suitable and admissible heuristic for our 2D mazes, where each step costs 1 and the goal lies deep within the maze.

Part 2: Search With multiple goals

In part 2 we modified the previous 3 algorithms such that they are able to find the shortest path in the provided 2D maze which consists of multiple goals.

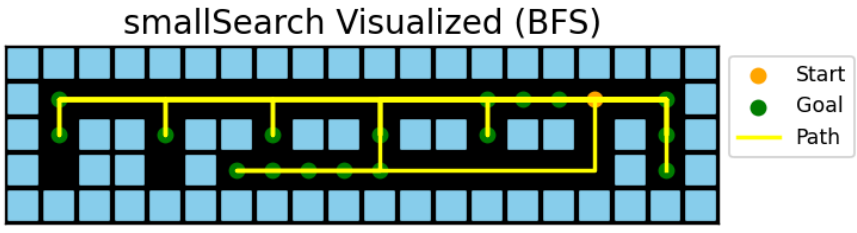
Findings for BFS (Breadth First Search) algorithm

The algorithm

We reused the BFS algorithm from the single-goal case, looping through each goal by updating the start and end points for each iteration.

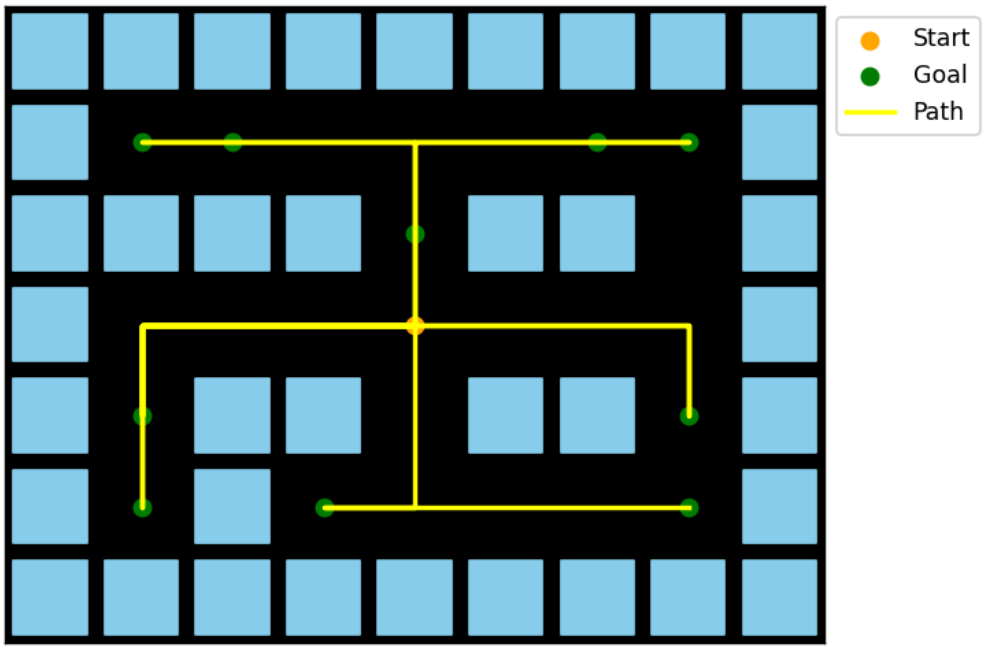
The result

Following are the results we got after running the modified BFS algorithm to the given small, tiny and tricky mazes.



Path cost: 108
Nodes expanded: 275
Maximum depth: 17
Maximum fringe size: 6

tinySearch Visualized (BFS)



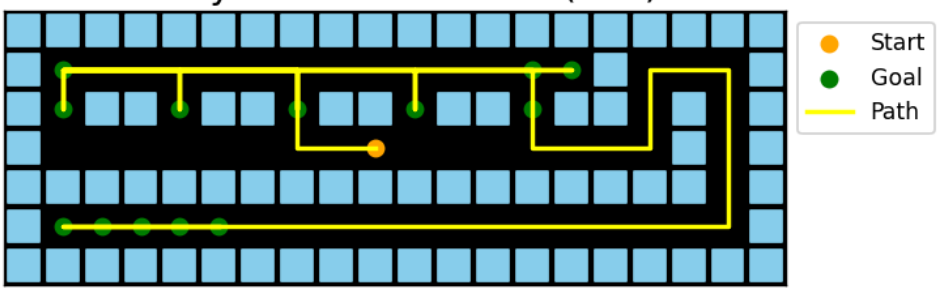
Path cost: 49

Nodes expanded: 140

Maximum depth: 9

Maximum fringe size: 6

trickySearch Visualized (BFS)



Path cost: 90

Nodes expanded: 245

Maximum depth: 28

Maximum fringe size: 6

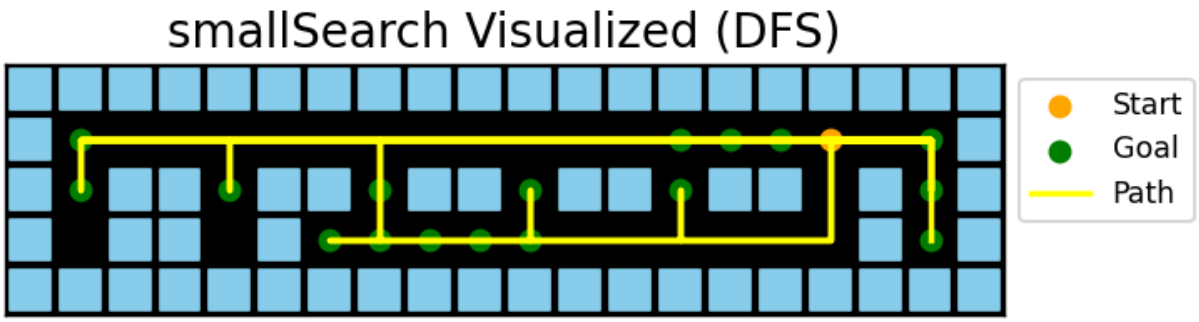
Findings for DFS (Depth First Search) algorithm

The algorithm

We reused the DFS algorithm from the single-goal case, looping through each goal by updating the start and end points for each iteration.

The result

Following are the results we got after running the modified DFS algorithm to the given small, tiny and tricky mazes.

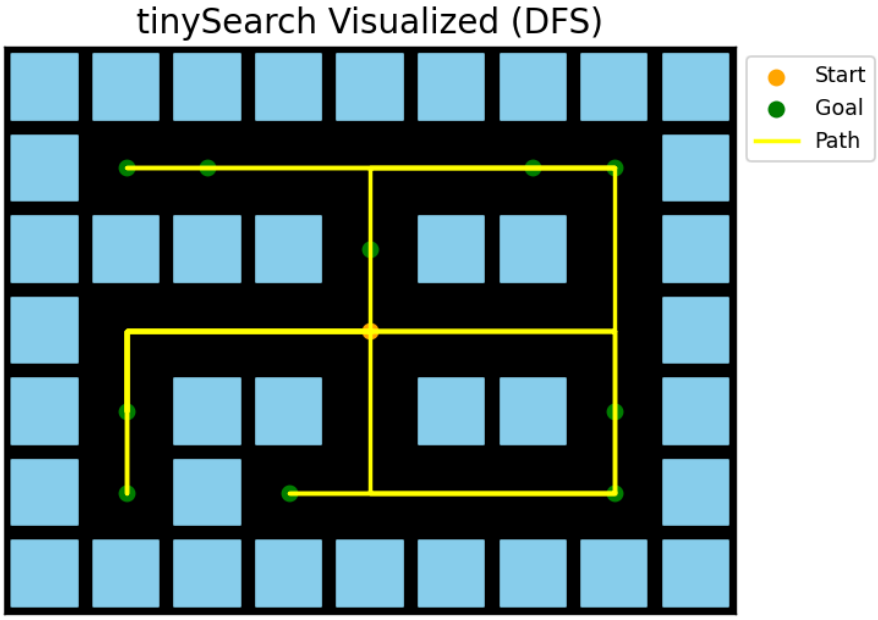


Path cost: 110

Nodes expanded: 246

Maximum depth: 29

Maximum fringe size: 7

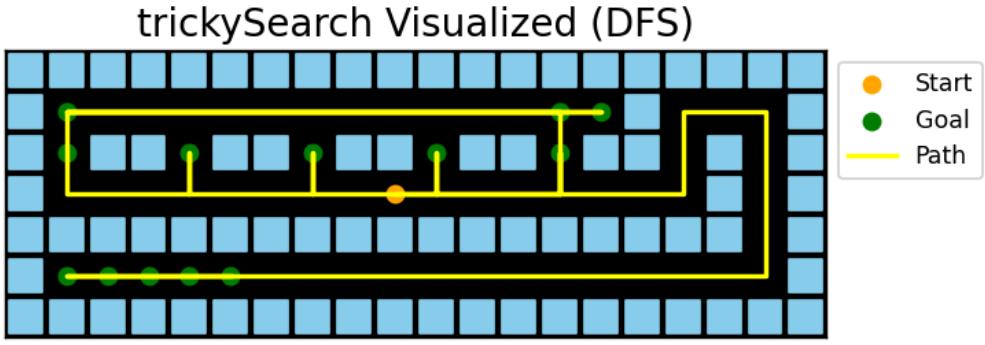


Path cost: 63

Nodes expanded: 99

Maximum depth: 14

Maximum fringe size: 5



Path cost: 98

Nodes expanded: 306

Maximum depth: 42

Maximum fringe size: 6

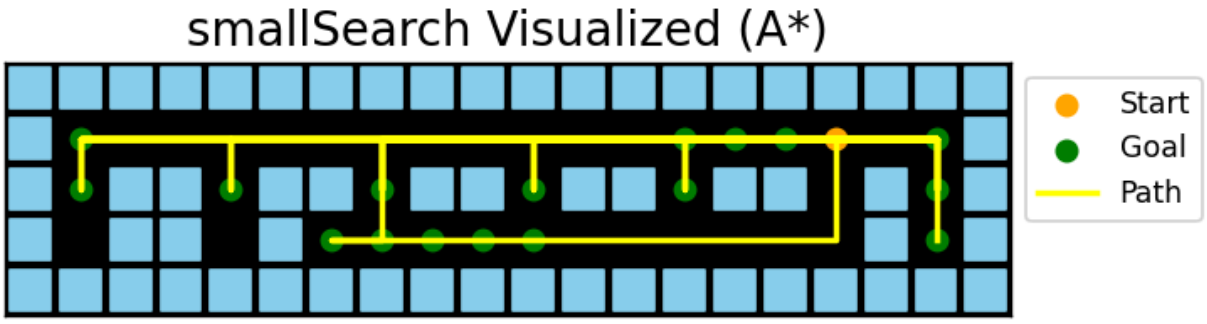
Findings for A* (A star Search) algorithm

The algorithm

We reused the A* algorithm from the single-goal case, looping through each goal by updating the start and end points for each iteration.

The result

Following are the results we got after running the modified A* algorithm to the given small, tiny and tricky mazes.



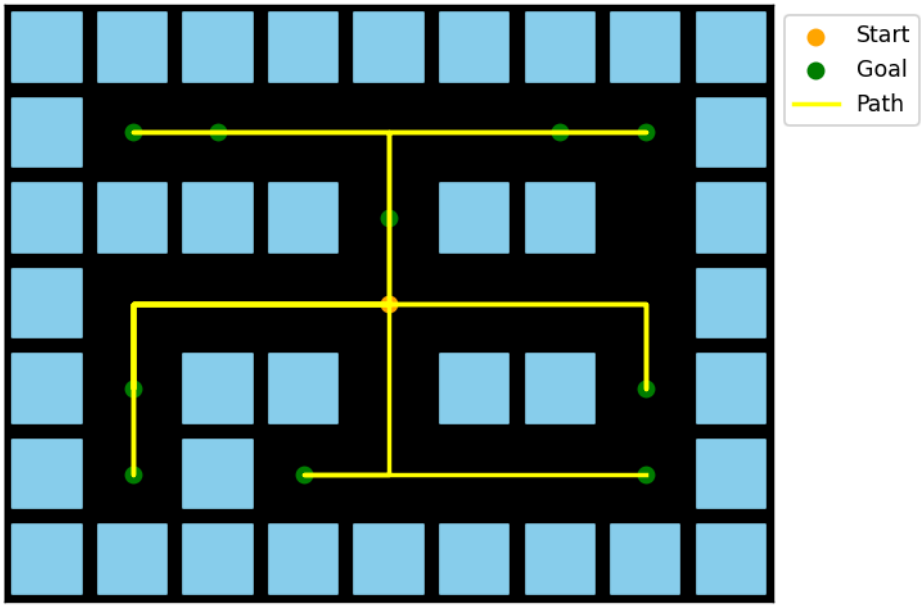
Path cost: 108

Nodes expanded: 170

Maximum depth: 18

Maximum fringe size: 7

tinySearch Visualized (A*)



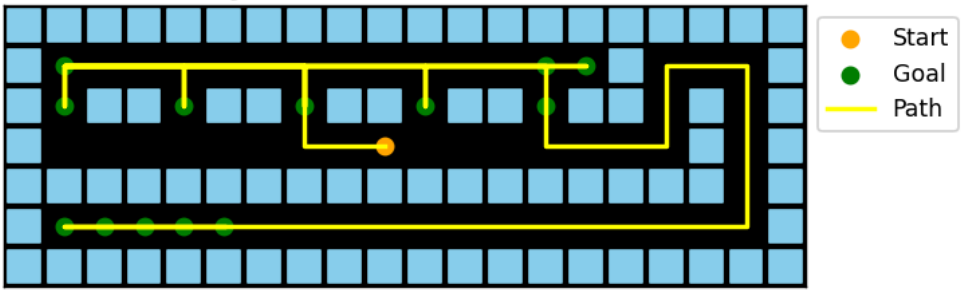
Path cost: 49

Nodes expanded: 72

Maximum depth: 9

Maximum fringe size: 5

trickySearch Visualized (A*)



Path cost: 90

Nodes expanded: 161

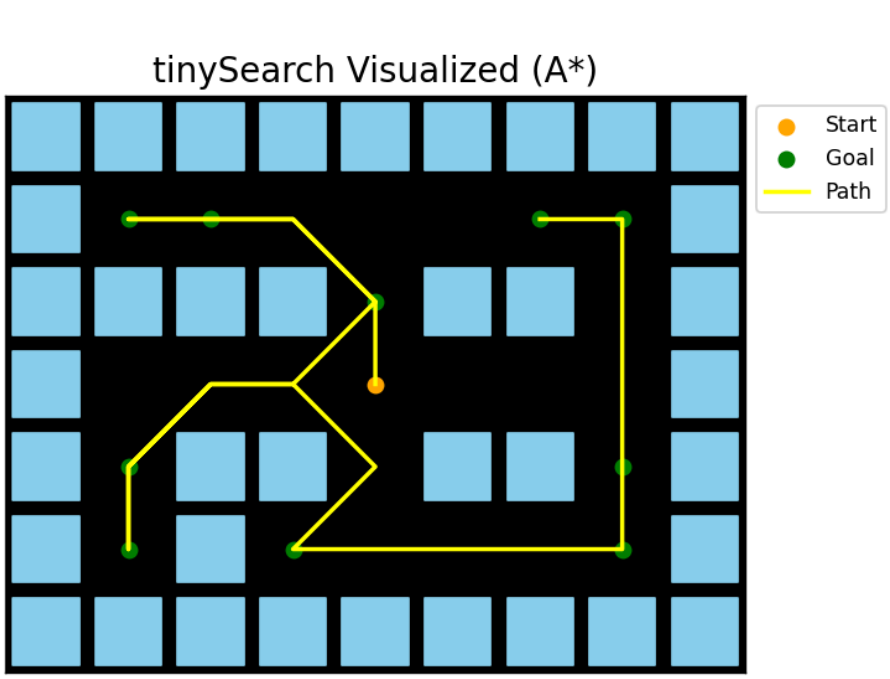
Maximum depth: 29

Maximum fringe size: 7

Findings

Key Findings for Multiple Goals:

- Even in small mazes, having multiple goals reduced the efficiency of **BFS** and **DFS**, as seen in the number of nodes expanded (e.g., **BFS: 140**, **DFS: 99**) compared to **A*** (**72**) in the tiny maze.
- To improve efficiency, we first allowed **diagonal movement** and then used a **Manhattan distance-based approach** to select the next goal with the least cost instead of simply looping through all goals.



The path cost has decreased to **25 from 49,** while the nodes expanded drastically decreased from **72 to 36.**

Resources

Following are the resources we used to improve our algorithms and findings.

- Matplotlib library for visualizing the maze solution.
- ChatGPT for refining some algorithms like greedy goals sorting.
- Geekforgeeks and wikipedia websites for definition and internal working of the algorithms used.

Instructions to run the code

In order to properly run the codes follow this simple steps:

1. After unzipping the source code folder, cd into the project-1 folder `` `cd project-1` ``
2. Run each algorithm using python. (for example: `` `python a_star_single_goal.py` ``)
3. Select the maze, heuristic function (for the A star search to use), allowing diagonal movement or not and greedy sorting of goals for multiple goals or not as per needed.

For the A star search algorithm **Chebyshev heuristic** option if you want to see the result for non-standard tie-breaking then ***comment the line 209 and uncomment line 211 in search_algorithm.py file.***