Presented to the College of Computer Studies

De La Salle University - Manila

Term 1, A.Y. 2022-2023

In partial fulfillment of the course

In CSINTSY S12

MCO1 - Mazebot

Submitted by:

Angeles, Sherwynn Clarence

Bernardo, Noah Halili

Fernandez, Matthew Nathan

Robles, Donnald Miguel L.

Submitted to:

Dr. Thomas James Tiam-Lee

March 1, 2023

I. Introduction

Given an n-by-n maze with tiles that could either be a wall or an empty space and a single start and goal location in the maze, the task is to show a path from the start to the goal and the tiles explored and taken to reach the goal. Otherwise, if there is no path that leads to the goal, the program must mention that there is no solution. Additionally, the bot could only move up, down, left, and right. The bot is also not allowed to go through wall tiles.

The group used an A* Search, as this type of search ensures that the most optimal route to the goal is found since it is an informed search algorithm. This is better than other search algorithms like Breadth-First Search and Uniform-Cost Search, as A* Search does not blindly look for possible routes, and it does not only consider the past cost. Instead, it looks ahead and takes into account the possible future cost to the goal. This helps estimate the next path to choose from that could be more likely to be closer to the goal.

Euclidean Distance could have been used as the heuristic function to guide the search algorithm, but this only takes into account the direct distance between a node and the goal. Since the bot could only move vertically or horizontally, the Manhattan Distance would be a more appropriate heuristic function. As proof of the heuristic's admissibility, the Manhattan Distance returns the shortest possible path for the bot to take in the event that there are no walls. This means that it is impossible for the heuristic to overestimate the actual cost of the goal.

II. Program

Standard Run:

Go to the `source` folder and run `app.py`

Once ran, you will be prompted with an optional web-based maze animation. (This will generate files in /website)

If you choose to generate the web-based animation, an additional web animation will be spawned

Console Based Legends

- **'..' -** Road
- **'##'** - Wall
- **Numbers** - Order

Web App Legends

- **White** - Road
- **Grey** - Wall
- **Blue** - Visited Road
- **Green** - Optimal Road

FAQ/Troubleshooting:

- **Web Server Unavailable:** Another program is using the default assigned server, go to line 194 and change the server to an available one (try 8000).
- **On Web Server - animation is messing up:** Reload the web app, when using wait for the current animation to finish before prompting a new one.

- **Security prompt on animation -** python HTTP servers are safe, if you don't want to give access, build a server yourself, and run it there.
- **The web app isn't automatically opening up -** assuming nothing is broken, it should still be available at the default link http://localhost:9000/website/index.html
- **It suddenly stopped working** - make sure the console where you ran the file in is still open.
- **I want to run a different maze file -** go to line 120 and change the file path
- **The input folder or test_mazes folder and the website folder give the error: No such file or directory -** Put the said folders outside of the parent folder

Desktop GUI Run

Go to the app folder and run `gui.py`

Desktop GUI Legends

- **Black** - Road
- **White** - Wall
- **Blue** - Visited Road
- **Green** - Optimal Road

FAQ/Troubleshooting:

- **The file won't run -** You are either using an old python version or, for some reason, you do not have the Tkinter toolkit (which should have come with python)
- **Can the animation run faster? -** Go to lines 22 and 23 and change the delay to something faster.
- **I want to run a different maze file -** go to line 6 and change the file path
- **sh: python: command not found -** change python to python3

III. Algorithm

An entry represents a point(x,y) and holds a tuple containing 3 values, the priority, heuristic, and the point itself. The priority is defined by the heuristic function of the point (Manhattan Distance) added to its actual distance to the start point; when 2 points have equal priority, they are compared by their heuristic. This helps identify which neighboring point that has the least value of priority to explore next. All entries will be stored in the Priority Queue data structure. This data structure is like a normal queue, but it makes it so the lowest value inside the queue will always be retrieved first. The algorithm will be coded in a function named a_star that returns a tuple containing two lists. This includes the optimal path and the order in which states are explored.

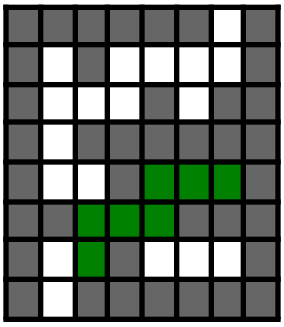
Helper Functions for a_star()

Function Name	Description
get_valid_moves(maze, point)	Given an array version of the maze and a point in the maze, return neighboring tiles (up, down, left, and right) that are not walls (meaning they're valid moves).
mahattan_distance(src, dst)	Given two points, return the manhattan distance between them.
trace_optimal_path(maze, costs, start, end)	Given the cost table, trace back the path A* took from end to start, by iteratively looking at neighbors with cost-1 starting from end and return a list of points going to the optimal path.

Pseudocode of A_Star:

```
if starting tile is equal to wall
      return goal point is unreachable
# costs is a 2D array that holds the actual distance of points from the
start point. -1 means that the point has not yet been explored.
costs = {-1 → for all points}

pq = PriorityQueue()
# each entry holds the priority, heuristic, and point
put (0, 0, start) into pq
while pq is not empty:
      get priority, heuristic, current from priority queue
      # the heuristic is consistent so the cost of the state is optimal
      # once it is explored

      # if curr has been explored
      if costs of current is not equal to -1:
            continue to next iteration
      costs of current = priority - heuristic
      if current is the goal point:
            # get and return optimal path from cost
            return trace_Optimal_Path(costs, start, end)

      # holds the distance of every child to the start
      child_cost = costs of current + 1
      for each child in get_valid_moves(curr):
            # if child point has already been explored we skip
            if costs of child not -1:
                  continue to next iteration
            heuristic = manhattan_distance(child, end)
            put (heuristic + child_cost, heuristic, child) into pq

# if we've exhausted and still no goal
return goal point is unreachable
```

How the algorithm builds the search tree for given maze:

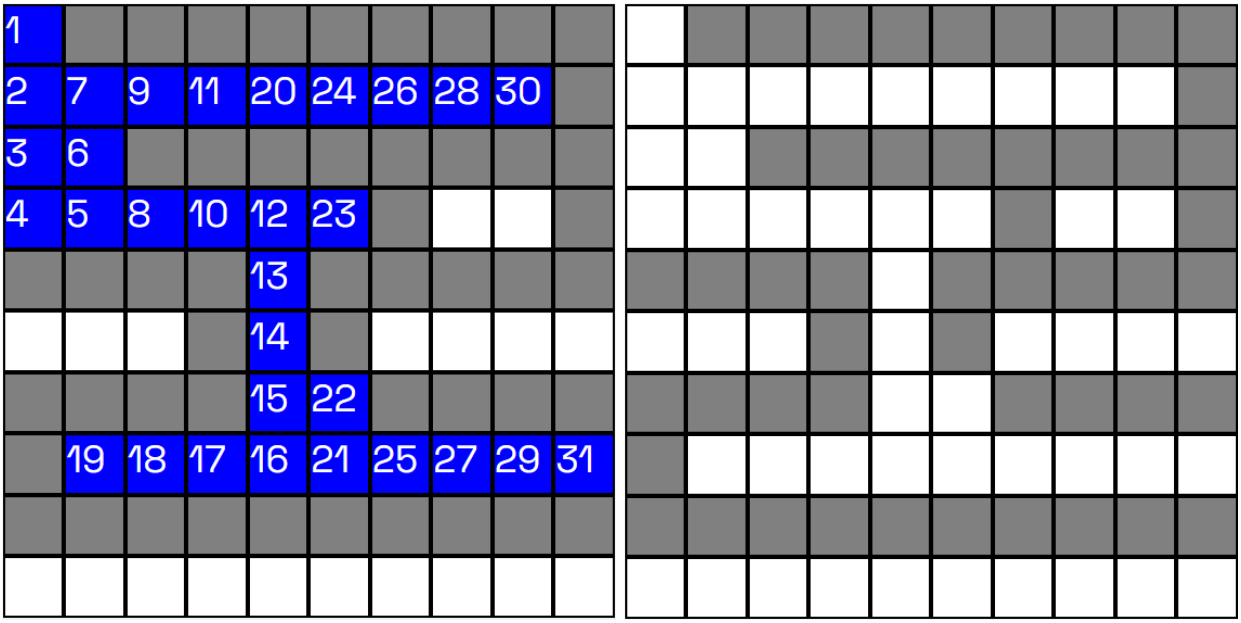


#	Frontier	Explored
1	Point(2,6): {cost: 0, prio: 0}	
2	Point(2,5): {cost: 1, prio: 6} Point(1,6): {cost: 1, prio: 8}	Point(2,6): {cost: 0, prio: 0)
3	Point(3,5): {cost: 2, prio: 6} Point(1,6): {cost: 1, prio: 8} Point(2,4): {cost: 2, prio: 6}	Point(2,6): {cost: 0, prio: 0), Point(2,5): {cost: 1, prio: 6)
4	Point(4,5): {cost: 3, prio: 6} Point(1,6): {cost: 1, prio: 8} Point(2,4): {cost: 2, prio: 6}	Point(2, 6): {cost: 0, prio: 0} Point(2, 5): {cost: 1, prio: 6} Point(3, 5): {cost: 2, prio: 6}
5	Point(4,4): {cost: 4, prio: 6} Point(2,4): {cost: 2, prio: 6} Point(4,6): {cost: 4, prio: 8} Point(1,6): {cost: 1, prio: 8}	Point(2, 6): {cost: 0, prio: 0} Point(2, 5): {cost: 1, prio: 6} Point(3, 5): {cost: 2, prio: 6} Point(4, 5): {cost: 3, prio: 6}
6	Point(5,4): {cost: 5, prio: 6} Point(2,4): {cost: 2, prio: 6} Point(4,6): {cost: 4, prio: 8} Point(1,6): {cost: 1, prio: 8}	Point(2, 6): {cost: 0, prio: 0} Point(2, 5): {cost: 1, prio: 6} Point(3, 5): {cost: 2, prio: 6} Point(4, 5): {cost: 3, prio: 6} Point(4, 4): {cost: 4, prio: 6}
7	Point(6,4): {cost: 6, prio: 6} Point(2,4): {cost: 2, prio: 6} Point(4,6): {cost: 4, prio: 8} Point(1,6): {cost: 1, prio: 8}	Point(2, 6): {cost: 0, prio: 0} Point(2, 5): {cost: 1, prio: 6} Point(3, 5): {cost: 2, prio: 6} Point(4, 5): {cost: 3, prio: 6} Point(4, 4): {cost: 4, prio: 6} Point(5, 4): {cost: 5, prio: 6}
8	Point(2,4): {cost: 2, prio: 6} Point(1,6): {cost: 1, prio: 8} Point(4,6): {cost: 4, prio: 8}	Point(2, 6): {cost: 0, prio: 0} Point(2, 5): {cost: 1, prio: 6} Point(3, 5): {cost: 2, prio: 6} Point(4, 5): {cost: 3, prio: 6} Point(4, 4): {cost: 4, prio: 6} Point(5, 4): {cost: 5, prio: 6} Point(6, 4): {cost: 6, prio: 6}

IV. Results and Analysis

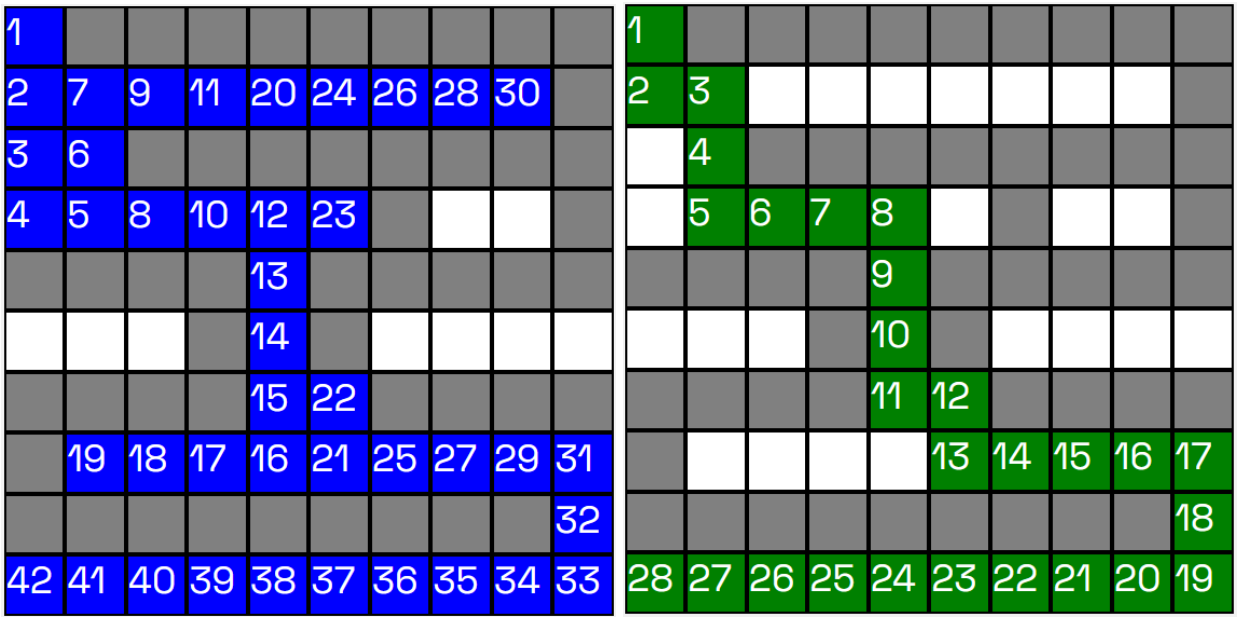
Overall, the bot worked as expected when compared to versions found online. After rigorous testing, we've found that it can handle all cases within the given specifications. It is found that while it can handle all cases in the specification, there are cases where it performs poorly, and something like depth-first will outperform it. Displayed below are specific test cases.

Test Case #1: Bot is Unable to Reach Goal (Search Order/Optimal)



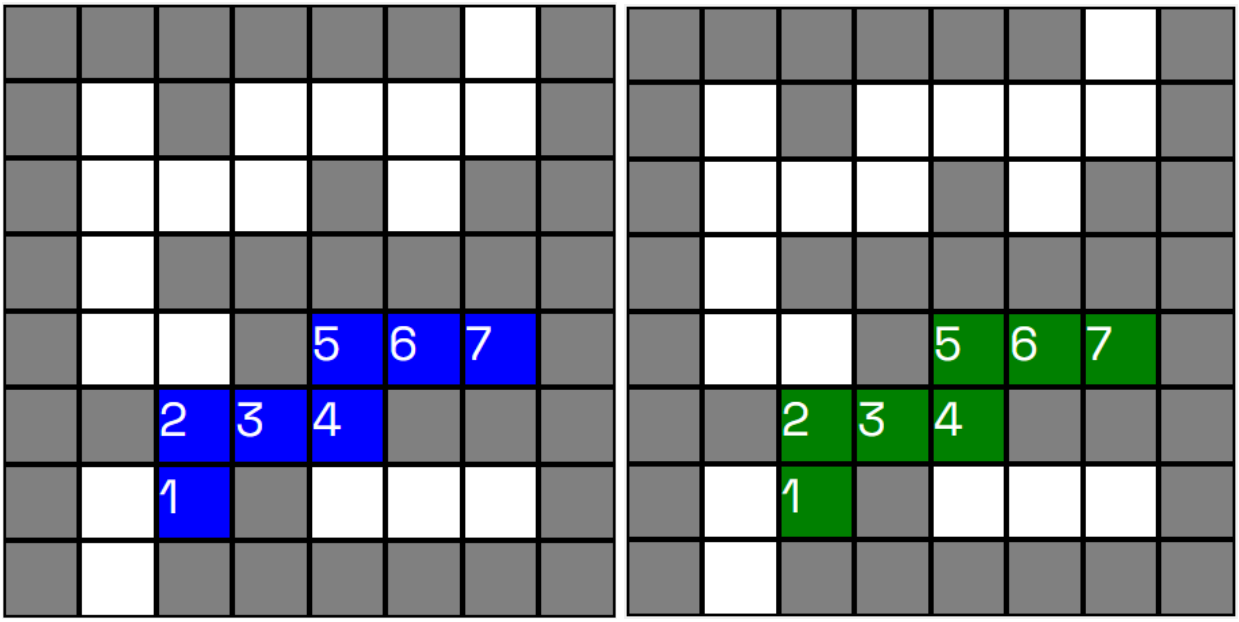
Will simply display search order, not optimal path, as goal is impossible to reach.

Test Case #2: Unblocked Maze 1 (Search Order/Optimal)



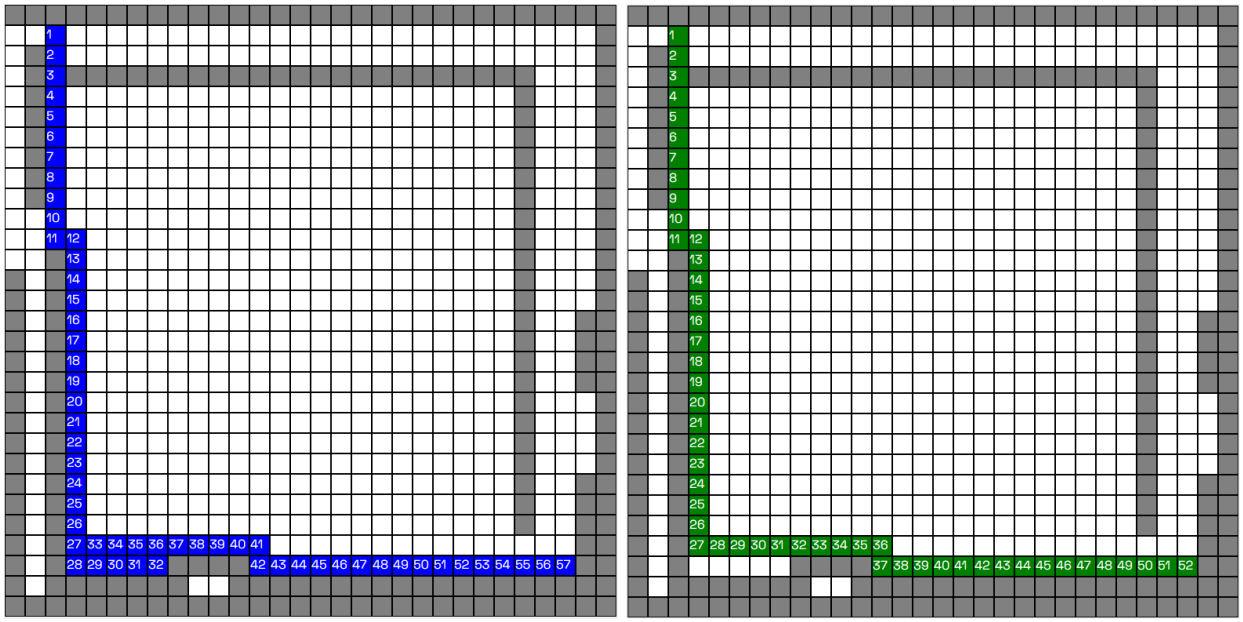
When unblocked, the search finds the goal.

Test Case #3: Squares Explored is Optimal Already (Search Order/Optimal)



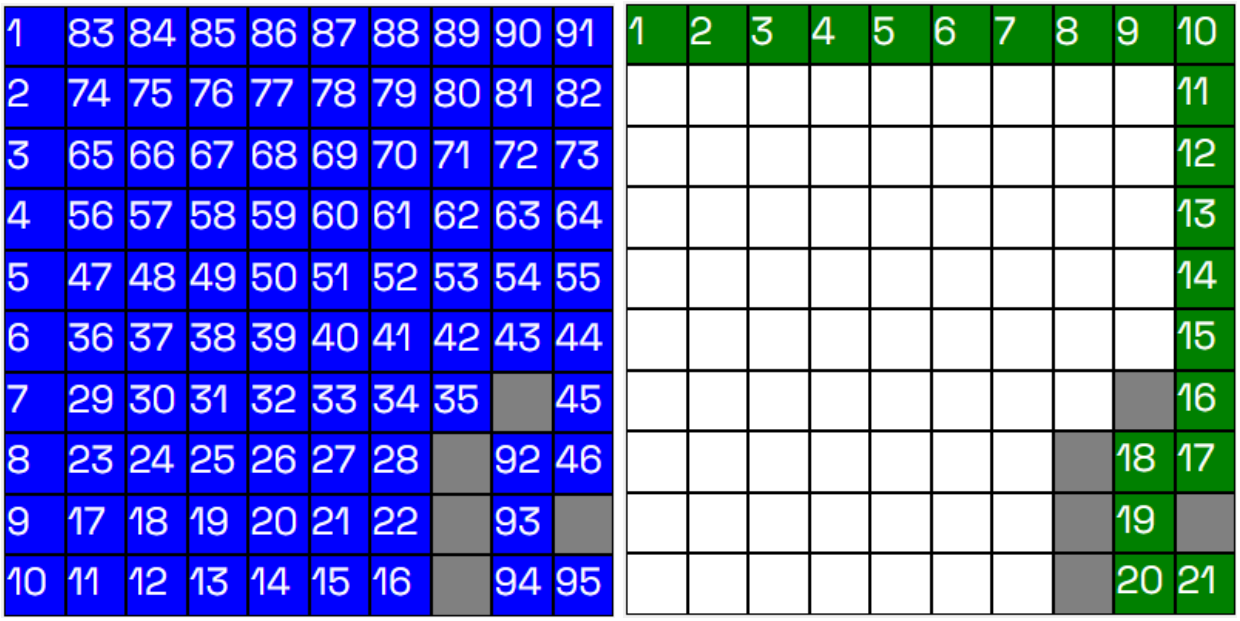
Search finds with no error.

Test Case #4: Large, Open-Space Maze (Search Order/Optimal)



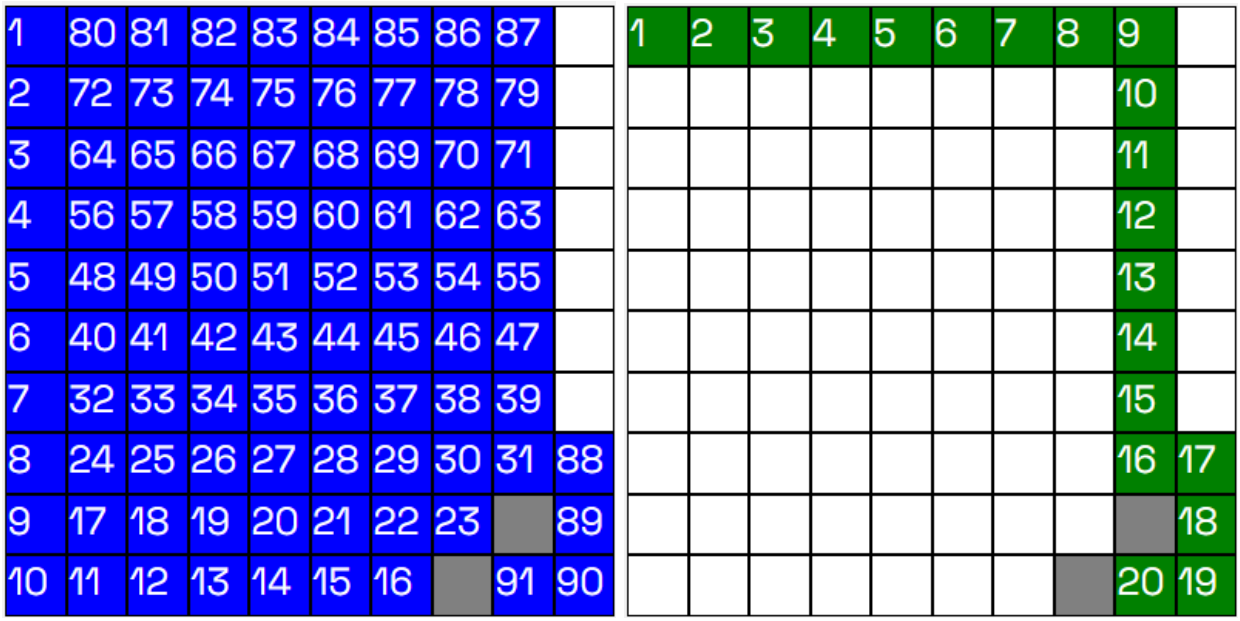
Search finds as expected.

Test Case #5: Worst case (Search Order/Optimal)



While this works as expected, it shows that the implemented search isn't perfect and struggles with open mazes with closed endings.

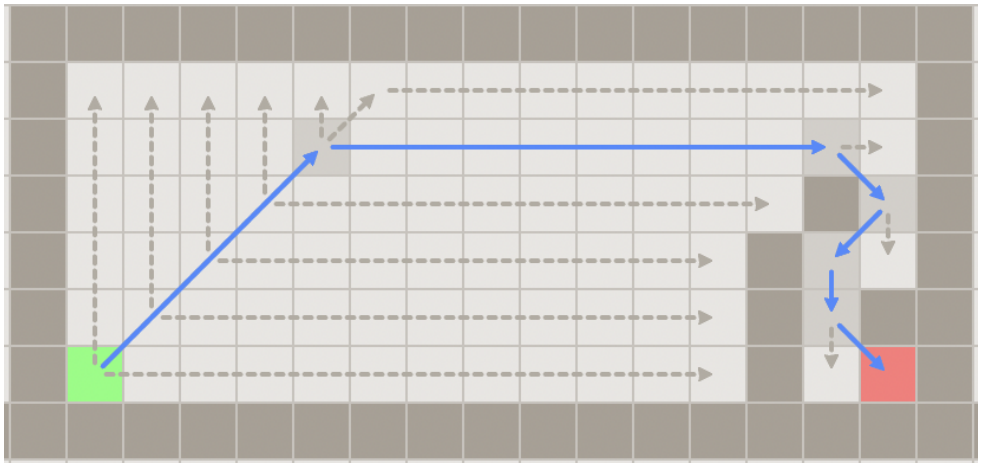
Test Case #6: Poor performance on large, open space maze (Search Order/Optimal)



Similar to the worst case, the search will prioritize states with the smallest priority (the large empty space). In both cases, a depth-first search would have done a better job.

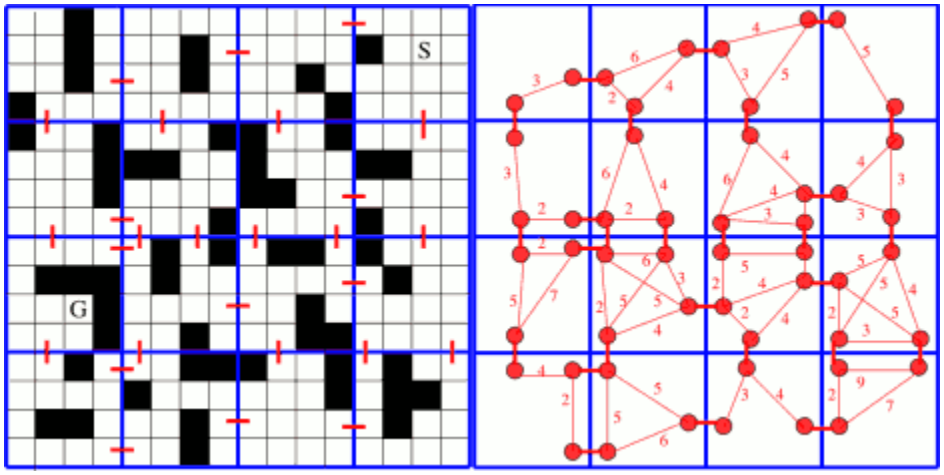
V. Recommendations

When it comes to solving mazes, using specialized algorithms can often yield better results than more general approaches like A* with Manhattan distance. One such specialized algorithm is jump-point search, which is able to reduce the number of nodes that need to be expanded by "jumping" over certain parts of the grid based on the layout of obstacles or walls. This approach allows the algorithm to skip large portions of the maze and reach the goal faster than other methods. For example,



*Jump Point Search visualization from zerowidth on the worst case of A**

In A* with Manhattan distance, the heuristic function only knows about the goal state and can only give hints on the general direction the algorithm should take, which may lead to the algorithm hinting in the wrong direction; In JPS the efficiency gained by skipping over large portions of the maze generally outweighs this drawback. Additionally, since jump point search takes obstacles into account in a more efficient way than A*, it can be a better choice for more complex mazes.



HPS Chunking and Pathway Finding

There are also other algorithms that generally do better than A* search aside from JPS, but can't always guarantee optimality. One such algorithm is depth-first search in some very niche cases. Another one to consider is Hierarchical Pathfinding (HPA*), an informed search that generally runs faster than A* Search but only guarantees near optimality. The way HPS*

works is very complex, but generally, in HPS*, mazes are broken down into n chunks, usually of equal size, and HPA* then finds pathways between these maps (from chunk a to chunk b, for example), and then the paths and distances are stored for comparison, eventually returning the best path. This divide-and-conquer-like approach is what makes it run fast.

Type	Algorithm	Execution time(ms)	Traversd Nodes	Length
Uninformed Search	Dijkstra	1.89	496	23.36
	IDDFS	9.64	423	23.36
	BIDDFS	3.67	231	23.36
	BFS(Breadth)	7.33	993	23.36
Informed Search	Greedy Best First Search	2.2	53	29.31
	Ida*	5.232	312	28.54
	A*	1.96	46	23.36
	Jump point search	1.54	312	23.36
	HPA*	1.11	36	23.36

*Execution time (ms), Traversed Nodes and Length of path with 10% blocked node in grid map (Grid size : 64*64 blocked node : 10%) from a paper by Noori, Azad & Moradi, Farzad. (2015).*

In conclusion, while both A* with Manhattan distance and Jump Point Search are effective algorithms for pathfinding in mazes, the specialized nature of Jump Point Search makes it more efficient. There are other search algorithms that generally run faster like HPA*, but do not guarantee the most optimal paths that we can also consider when tackling solving mazes. By utilizing the presence of obstacles or walls in the maze, Jump Point Search can "jump" over large portions of the maze and reduce the number of nodes that need to be expanded, ultimately reaching the goal faster. Meanwhile, HPA* breaks a maze into chunks and identifies pathways through them, storing optimal ones. However, it's important to note that the heuristic function using manhattan distance only knows the goal state and may hint at the wrong direction in some cases. Ultimately, the choice of algorithm will depend on the specific needs and constraints of the task at hand.

VI. References

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. J. Game Dev., 1(1), 1-30.

MAN1986. (2022). *Man1986/Pyamaze*. GitHub. Retrieved February 25, 2023, from https://github.com/MAN1986/pyamaze

Noori, Azad & Moradi, Farzad. (2015). Simulation and Comparison of Efficency in Pathfinding algorithms in Games. Ciência e Natura. 37. 230. 10.5902/2179460X20778.

Tahir, A. (2021). *GBFS-A_Star-maze-slover-in-python*. GitHub. Retrieved February 25, 2023, from https://github.com/Areesha-Tahir/GBFS-A_Star-Maze-Slover-In-Python

Witmer, N. (2013, May 5). zerowidth positive lookahead | A Visual Explanation of Jump Point Search. https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html

VII. Contributions of Each Group Member

Member	**Contributions**
Angeles, Sherwynn Clarence	A* Search Code Desktop GUI Implementation Code Bugs Fixing Collaborated on **III. Algorithm** Collaborated on **V. Recommendations** Revisions in: **I. Introduction** **III. Algorithm** **IV. Results and Analysis**
Bernardo, Noah Halili	Wrote **I. Introduction** Collaborated **III. Algorithm** Wrote pseudocode for **III. Algorithm** Collaborated on **IV. Results and Analysis**
Fernandez, Matthew Nathan	Code Bugs Fixing Collaborated on **III. Algorithm** Collaborated on **IV. Results and Analysis** Collaborated on **V. Recommendations**
Robles, Donnald Miguel L.	A* Search Code Web App Implementation Code Bugs Fixing Wrote **II. Program** Collaborated on **III. Algorithm** Collaborated on **V. Recommendations** Whole Docs Review Revisions in: **I. Introduction** **III. Algorithm** **IV. Results and Analysis**