Intro to AI Assignment 1

Theodor Novitsky (NetID: tn347, RUID: 226007608), Jad Ziab (NetID: jgz22, RUID: 209008874)

Part 0 - Setup your environments

We followed the writeup's suggested method of generating random mazes. The mazes are generated by calling a function "generateMaze()" found in the generate_maze.py file. The algorithm starts by creating a grid in which every cell is labeled "unvisited" of a given size from the GLOBAL variable "GRID_SIZE" which can be defined in utils.py. Then a random x and y are picked to be the start cell for running DFS. We change the label for this cell to "unblocked" and create our stack at first which only starts with this cell. Now we simply run DFS, and have a while loop that runs while our stack isn't empty. It'll pop the top of the stack and then find its neighbors. The code checks each neighbor and ensures that a cell can be placed on the grid (within the grid dimensions) and that the cell is labeled "unvisited". This is done with the helper function "can_generate_here(x, y, grid)". Then if there are neighbors available, we pick a random one, and then 70% of the time we label that neighbor cell as "unblocked" and add it to the stack. 30% of the time we'll label it "blocked" and not add it to the stack. If we aren't able to find any neighbors then we'll pop the top cell from the stack. This loop runs until the stack is empty, meaning each cell in the grid either has a label "unblocked" or "blocked". After the loop, we return the grid.

Additionally, we have implemented a "display_grid(grid, start = None, goal=None)" function. This function can be called on any test grid and will produce an image of the grid. You can also input a start cell(green) and a goal cell (red). Additionally, there's a function "find_unblocked_cell(grid)", which we use to help find start and goal cells when performing testing.

Demo: Region "Display_grids Demo" in main.py

Part 1 - Understanding the methods

a) When the A* algorithm is run on the graph in Figure 7 search problem, it moves east rather than north due to the f-values. Since the agent can only see the cells surrounding it (in the four cardinal directions), it won't know it is moving toward a blocked cell. The algorithm calculates f-values to determine which of the possible moves (the four directions) to take. Only valid cells will be considered, so the following types of cells will not be considered for the move: cells that are blocked, cells that are out of bounds, and cells that have already been visited (popped). The north cell from the starting node in Figure 7 (D, 2) has a heuristic value (Manhattan distance to the goal node) of 4 and a g-value (steps from the starting node) of 1. Since f = g + h, the calculated f-value of the north cell is 5. For the east cell, the

heuristic value is 2 and the g-value is still 1. Thus, the f-value of the east cell is 3, which is less than the f-value of the north cell, giving the east cell priority over the north cell. Of course, the algorithm also checks the valid west cell, but with an f-value of 5, the east cell still holds priority over it. The conclusion is that a lower calculated f-value is the reason the agent moved east.

b) The method works using a priority queue (variable name open_list in our code) to keep track of seen nodes/possible visits. Furthermore, we use a while loop with the condition that the priority queue still has cells in it. This means in a scenario where it is impossible to visit the final node, every node from the start node will definitely be visited. So by definition, the loop is finite given that the grid is finite. Since any node that is possible to visit can be visited, we will always reach the goal node if it is possible. As for the time complexity, since the worst-case scenario is where each cell is revisited n-times, the total number of moves is bounded by n^2 where n is the number of unblocked cells.

Demo: Region "Simple Visualization and Path output" in main.py

Part 2 - The Effect of Ties

How it works: To break ties, our a_star_search algorithm takes in a boolean *favorSmallG* (True by default). If *favorSmallG* is set to True, the algorithm will prioritize the cell with a smaller g-value when f-values are equal. If it is set to False, the algorithm will prioritize cells with a larger g-value when f-values are equal. This is done when we push cells onto the priority queue, where the g-value is the next important value after f. This works as is for *favorSmallG* = True, but when it is False (when we are supposed to favor large g) we flip the sign of the g-value when we push cells onto the priority queue. The reason we flip the sign is because the priority queue pops smaller numbers, so if we want it to pick a larger value, we can flip the sign so a larger value is interpreted as a smaller value. This is a simple and efficient way to change the way priority works.

What it does: As for the results, using a different tie-breaker mechanism does not affect the result in all cases. This is because we can have graphs where the tie-breaker doesn't alter the number of popped cells much or at all.

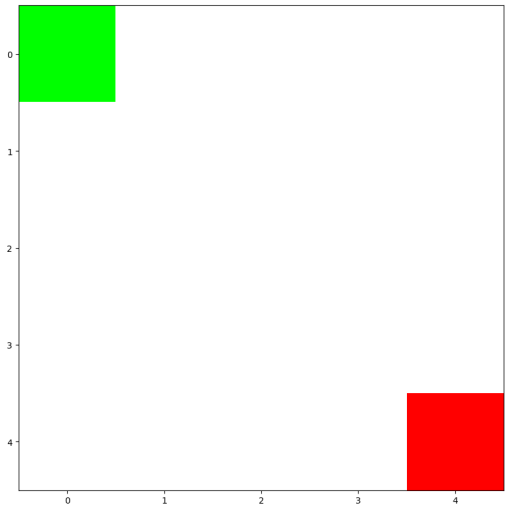
Figure 8 Search Problem: In Figure 8, we are met with a graph where every cell is unblocked. With that property and the start and goal being on opposite diagonals of the graph, every cell has the same f-value (8). While both algorithms will find the same path, they differ greatly in efficiency. Using our first approach of prioritizing the cells with a smaller g-value, we immediately run into a major optimization issue. Since the algorithm is picking the lower g-values and every lower g-value cell is in the opposite direction as the goal cell, we hit every other cell in the graph before reaching the goal cell. Specifically, we check each diagonal

of equivalent cells from small to large, and in this case the larger the g-value of the cell, the closer we get to the goal. This is the worst possible scenario for the A* algorithm where we favor small g. The opposite is true for the approach of selecting the largest g-values in this scenario; we end up taking the most efficient path to the goal (technically there are other most efficient paths in this case, but it will take one of them). While it is the same path that small g finds, large g does so without popping any unnecessary cells here.

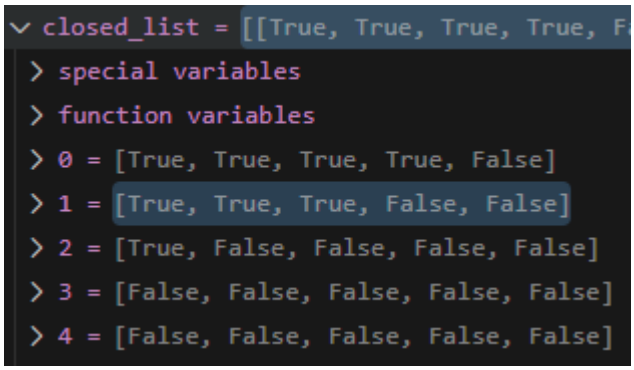
Conclusion: Since a smaller g-value means you are prioritizing exploring cells near the start cell, it is better in cases where we want to search a lot closer to the start cell. The opposite is true for a larger g-value, where if we want to search far away from the start cell, it ends up being the more efficient algorithm. These properties make small g vs large g quite similar to BFS vs DFS. That being said, it is more like a weighted BFS vs DFS, since the A* algorithm is still at play. This makes a DFS-like approach much more viable since we are exploring deeply toward a goal rather than randomly. Weighing the pros and the cons, prioritizing small g-values vs prioritizing large g-values can both be good in different scenarios and in general cases, doesn't make a big difference.

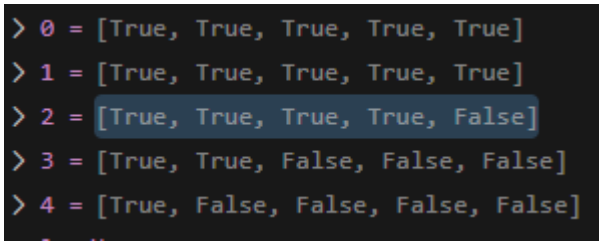
Attached below are graphs we drew to make this explanation and our understanding of these tie-breaker methods more clear.

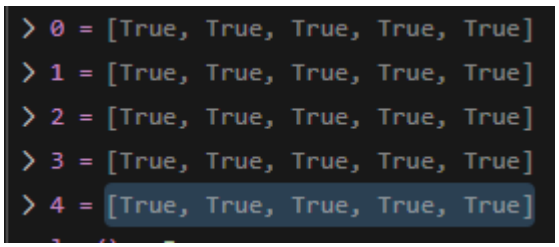
For the Case of a blank 5x5:



Favor smallG closed_list(list of explored cells)

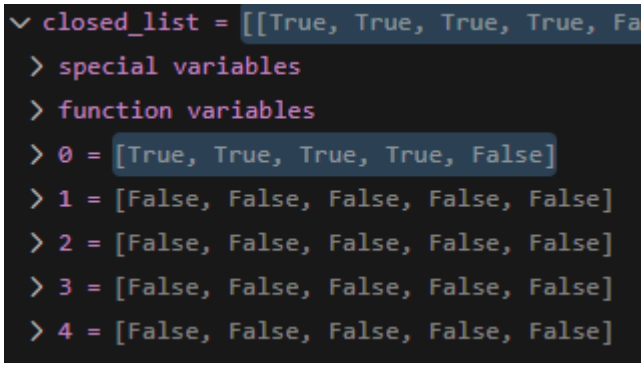


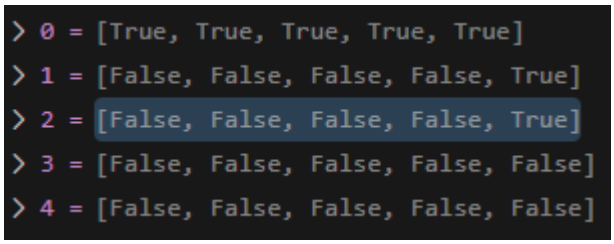


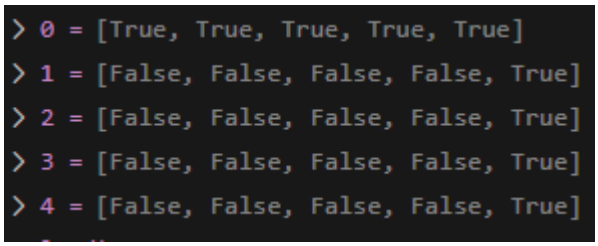


If the algorithm favors small G, in this example, then we eventually pop every single cell in the grid. Working our way diagonally (0,0),(0,1),(1,0),(0,2),(1,1),(2,0) etc... This is much much less efficient when comparing with favor large G.

Favor largeG closed_list(list of explored cells)







Favor large G is much more efficient in this case because we are popping much less cells from our heap to find an optimal solution to the maze. For example, after cell (0,1) has been popped, there are 3 possible choices for the next cell to pop. (0,2),(1,0) and (1,1). All the cells have the same f value of 8. But only cells (0,2) and (1,1) have g values of 2. So we won't explore cell (1,0) with g value 1. There is no other tiebreaker, so we pop the cell with lower row value. Which ends up becoming (0,2). This process repeats and we find an optimal path to the goal cell much faster this way.

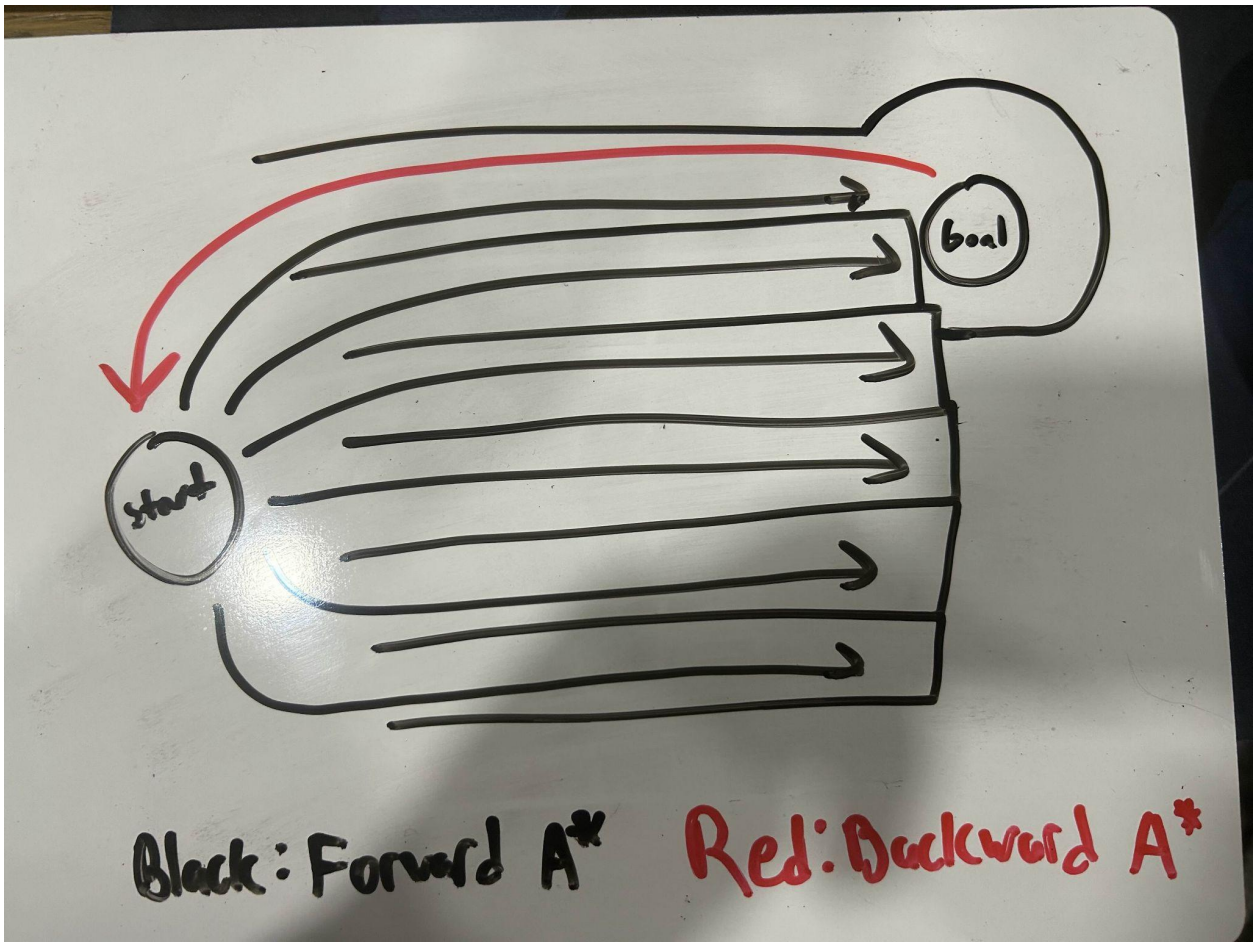
Demos:

-Region “favorSmallG vs favorLargeG” in main.py

-Region “Effect of Ties Demo” in main.py

Part 3 - Forward vs Backward

With some variation (due to the randomness of generating random mazes and running the algorithm many times), neither forward nor backward is better than the other in general cases. This is to be expected, as all we are doing is running the same algorithm with reversed start and goal cells. However, this doesn’t mean there is no value in this concept. Take this example: an agent has 4 paths to reach the goal cell. Every one of these paths is a tunnel of a similar length. This means for every tunnel picked incorrectly, about 1/4 of the graph will be explored going forward. However, from the position of the goal cell, there is only one path to the start cell. So while the start cell’s minimum amount of steps to the goal is 1/4, the goal’s amount of steps to the start cell will always be 1/4. Thus, it can be concluded that if the goal cell has a less complex environment than the start cell (and there is a path between the start and the goal), that is to say, there are fewer “wrong” options for it, then Backward A* has its use. However, in most general cases, Forward A* and Backward A* will yield similar results.The image below helps illustrate our thought process.



Demo: Region "forward vs backward A*" in main.py

Part 4 - Heuristics in the Adaptive A*

Proof 1: To prove the Manhattan distance is consistent in grid worlds where the agent can only move in the four cardinal directions, we must satisfy the consistency condition: h(s) h(s') + c(s, s') where s and s' are neighboring cells and c(s, s') is the cost between the two cells. Since our cost is 1 between every cell, this can be simplified to h(s) h(s') + 1. Since Manhattan distance decreases by 1 when moving to a neighboring state, the consistency condition is satisfied.

Proof 2: To know if the heuristic is admissible, first we must know what it means to be admissible. Let's use the previous condition h(s) h(s') + c(s, s') and substitute some values. We will consider an s that is one cell away from the goal cell, where the goal cell is s'. Since the heuristic at the goal is always 0, we can substitute h(s') for 0. This leaves us with: h(s) c(s, s'), and since c(s, s') is the cost from s to the goal, we can write it as h*(s). Thus, h(s) h*(s). What this

means is that the heuristic h will never overestimate the true cost of reaching the goal.

Conclusion: With this, we have proved that the Manhattan distance is consistent and admissible in our environment. It is worth noting that consistency also implies admissibility, which is clear in our proof since we use the consistency condition to prove it as admissible.

Part 5 - Heuristics in the Adaptive A*

How it works: Mathematically, using the Adaptive A* algorithm is very simple. We will loop A* n times, and every time we find the optimal path to the goal cell (popping the goal cell from the priority queue) we update a newly implemented value h* (h_star). H* is calculated like this: g(goal) - g(cell) = h*(cell). This is used to update the h value in cell_details, which is referenced in every run in the loop. What this does is effectively increase the f-value of "bad cells".

Implementation: To see the effects of the Adaptive A* algorithm, we must compare it to our original A* algorithm. An easy implementation of this is to run Adaptive A* n times, running the same start node for the first and last run of each graph. The rest of the runs and the graphs are both random. We compare the runtime in run 0 (the first run, which is effectively the regular A* algorithm) and the n-1th run (the final run). Seeing the improvement in efficiency between the first and last runs shows us the effect of the Adaptive A* algorithm. We also run this loop on 50 different randomly generated graphs, and take the average improvement time. The Adaptive A* algorithm does improve the efficiency of over 50 graphs, proving it to be an effective way to update heuristics to more quickly find an optimal path. You can also adjust the amount of times you want to run adaptive A* on each grid, the more times you run it on a single grid, the more efficient finding the goal cell from any random start cell is.

Demo:"Adaptive A*" in main.py (You can see the improvement on each grid my comparing Runtime 0 and Runtime 49 for that specific grid)