#1

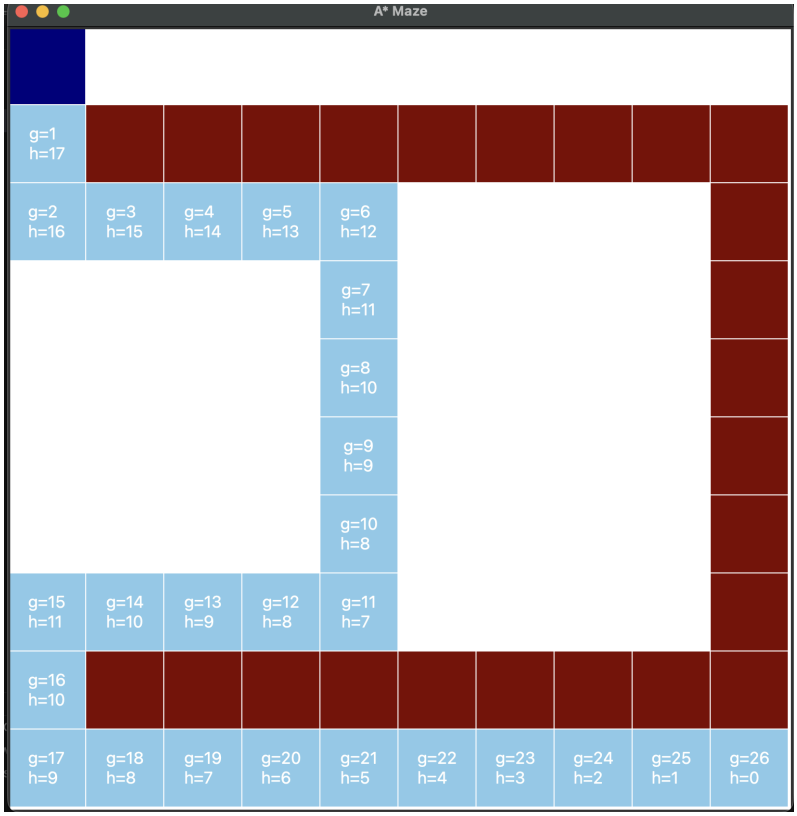
Here is the code for my matrix:

```
maze = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

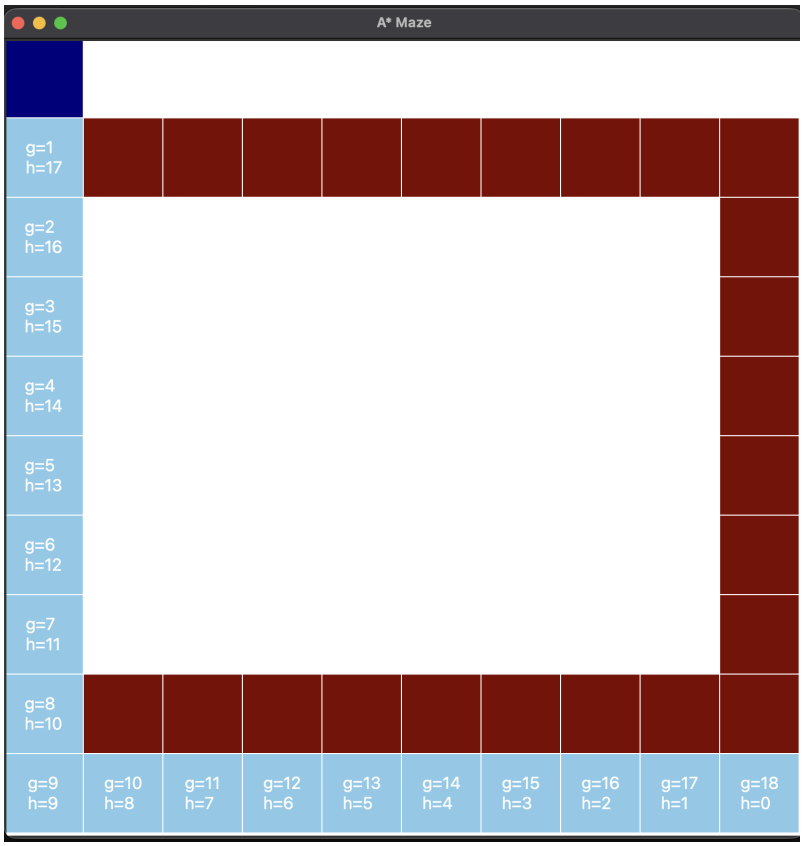
The start state is top-left (0,0) and the goal state is bottom-right (9,9). These were already set in the file.

```
### Update the evaluation function for the cell n: f(n) = g(n) + h(n)
# self.cells[new_pos[0]][new_pos[1]].f = new_g + self.cells[new_pos[0]][new_pos[1]].h
self.cells[new_pos[0]][new_pos[1]].f = self.cells[new_pos[0]][new_pos[1]].h
```

The commented out code was the original line. All that I changed was removing any considerations of the value of g in the f function, moving from A* (f(n) = g(n) + h(n)) to greedy best-first search (f(n) = h(n)). The following is the output:



And this is the output with regular A*:



#2

```
############################################################
#### Euclidean distance
############################################################
def heuristic(self, pos):
    return sqrt((pos[0] - self.goal_pos[0])**2 + (pos[1] - self.goal_pos[1])**2)
```

To change the heuristic from Manhattan distance to Euclidean distance, I used the built in sqrt function from the math library. The double asterisk is used for exponents.

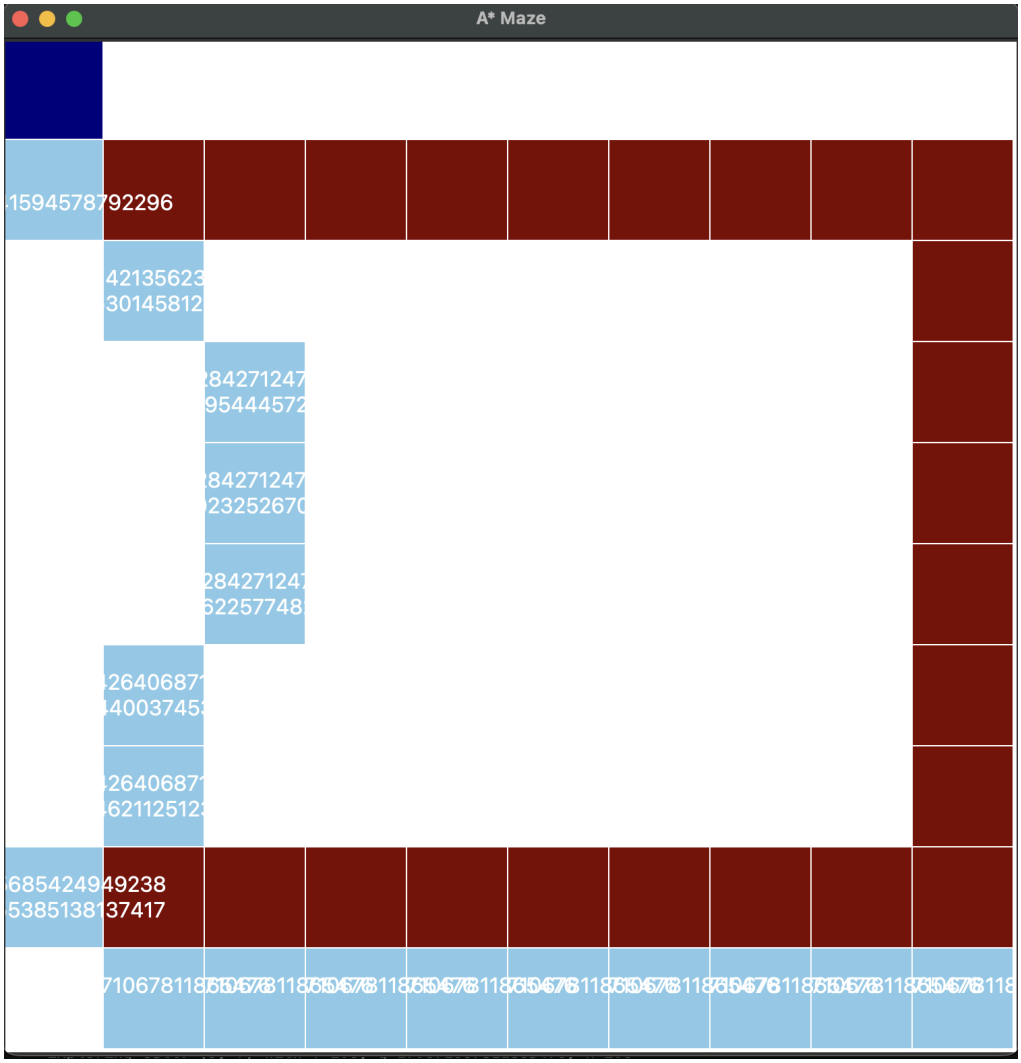
```
moves = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]

#### Continue exploring until the queue is exhausted
while not open_set.empty():
    current_cost, current_pos = open_set.get()
    current_cell = self.cells[current_pos[0]][current_pos[1]]

    print(f"Exploring cell: {current_pos} with f={current_cell.f}, g={current_cell.g}, h=
    {current_cell.h}")

    #### Stop if goal is reached
    if current_pos == self.goal_pos:
        self.reconstruct_path()
        break

    # Shuffle order of moves
    shuffle(moves)

    #### Agent goes E, W, N, S, SW, SE, NW, NE whenever possible
    for dx, dy in moves:
        new_pos = (current_pos[0] + dx, current_pos[1] + dy)
```

Since the directions said to make the moves randomly, I created a moves variable and shuffled the moves for each cell searched.

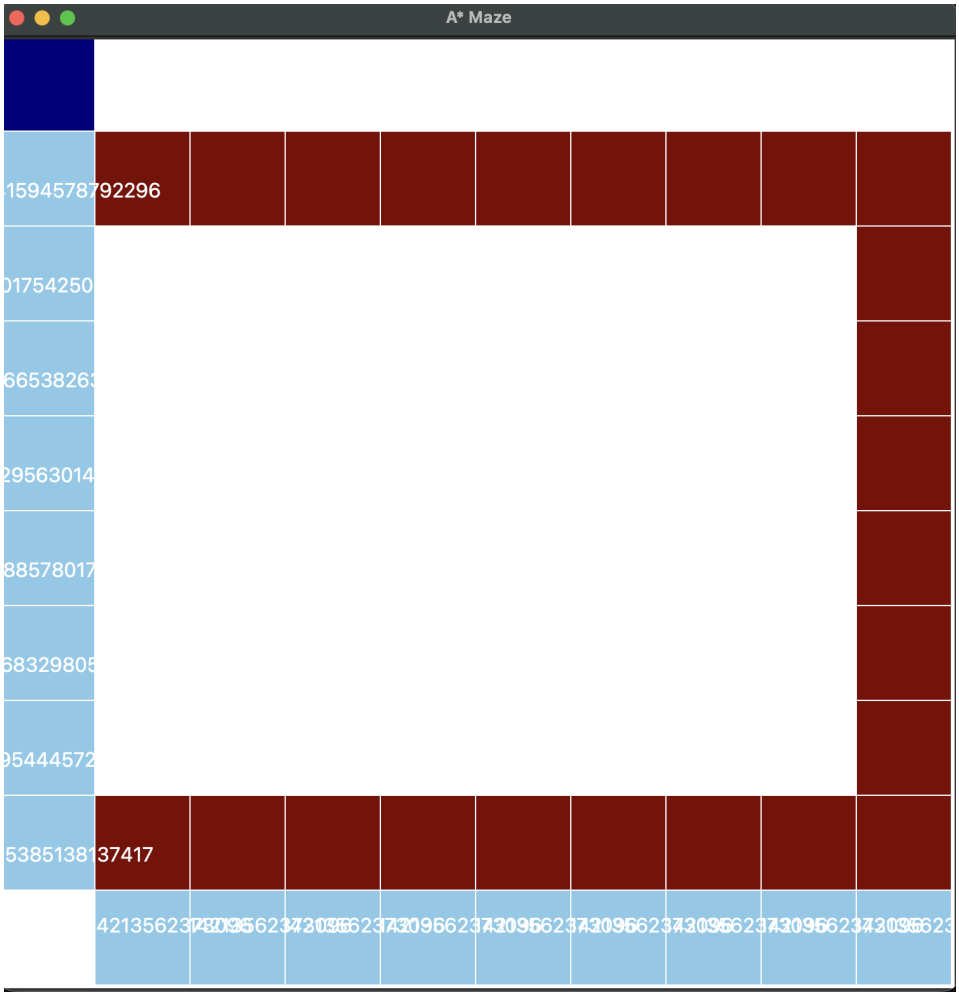
```
if 0 <= new_pos[0] < self.rows and 0 <= new_pos[1] < self.cols and not self.cells[new_pos[0]]
[new_pos[1]].is_wall:

    #### The cost of moving to a new position is 1 unit if not diagonal, sqrt(2) if diagonal
    if dx != 0 and dy != 0:
        new_g = current_cell.g + sqrt(2)
    else:
        new_g = current_cell.g + 1

    if new_g < self.cells[new_pos[0]][new_pos[1]].g:
        ### Update the path cost g()
        self.cells[new_pos[0]][new_pos[1]].g = new_g
```

I added this if-else block because the cost of moving to a new position is sqrt(2) if the move is diagonal. This is already accounted for in the heuristic, but it ensures correct implementation for A*.

Output for greedy best-first search:

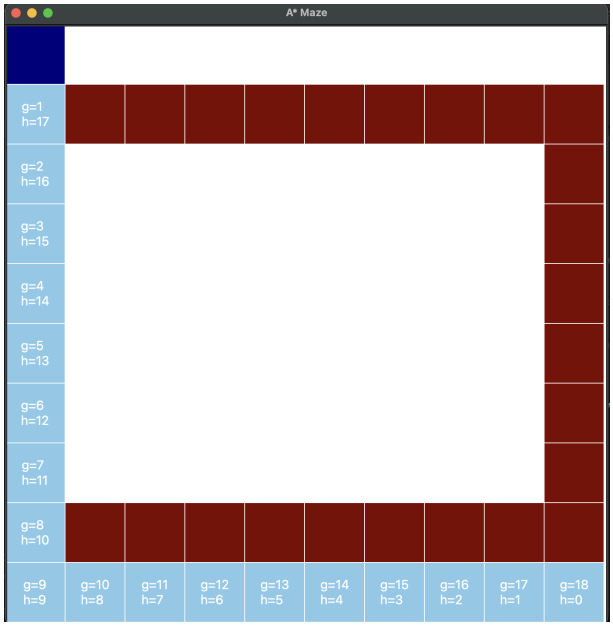


Output for A*:

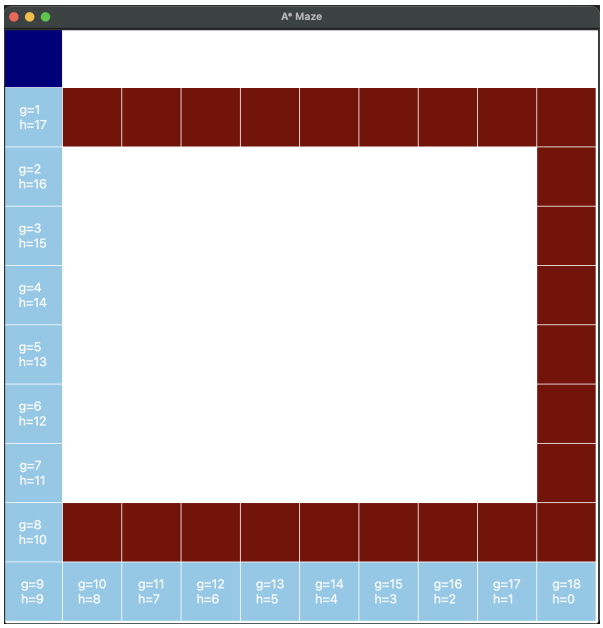


#3

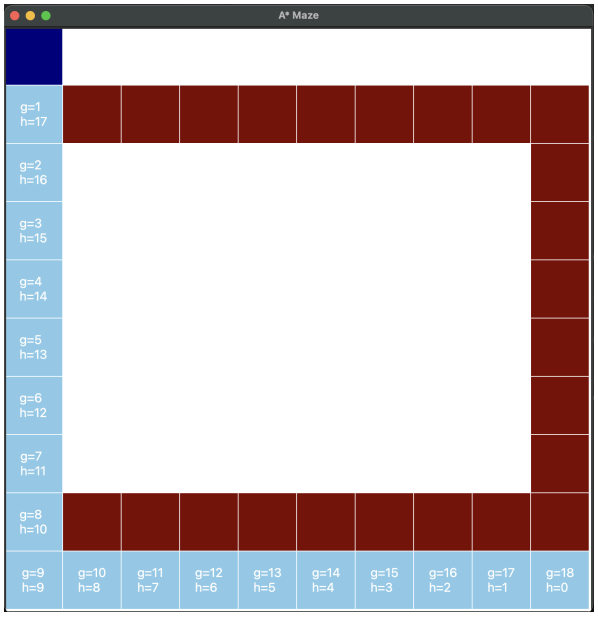
α = 0.9, β = 0.1:



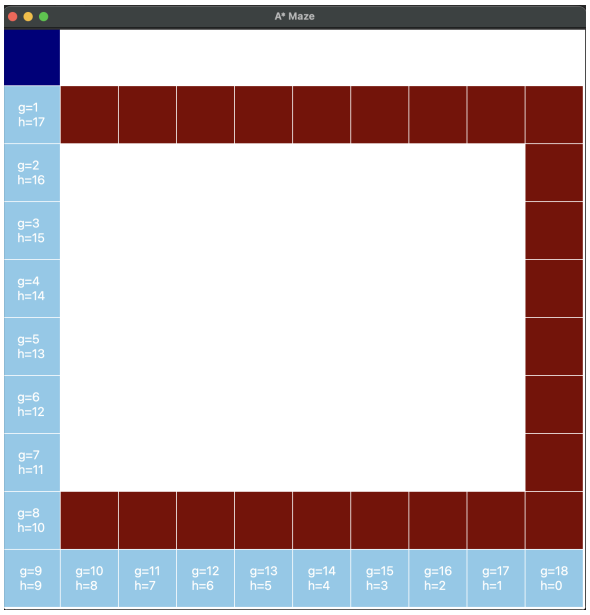
α = 0.7, β = 0.3:



α = 0.5, β = 0.5:

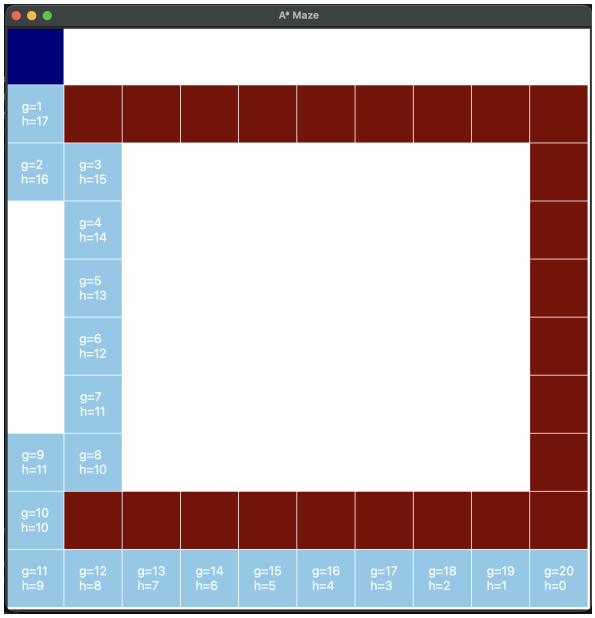


α = 0.4, β = 0.6:



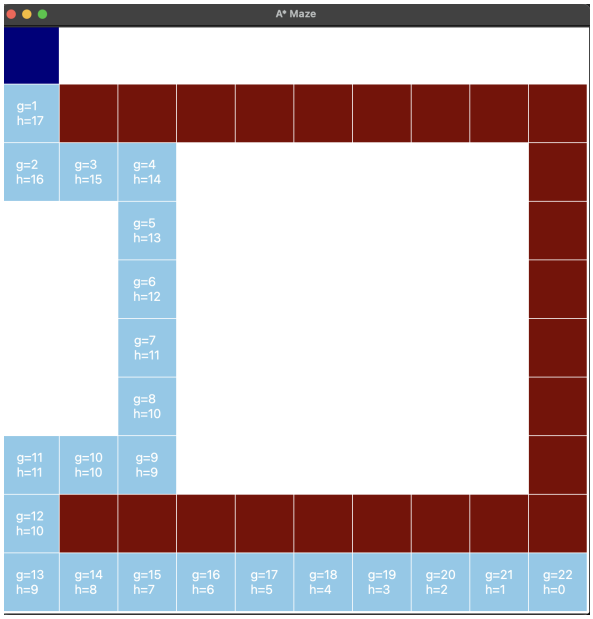
Weighted in favor of h has no impact.

α = 0.3, β = 0.7:



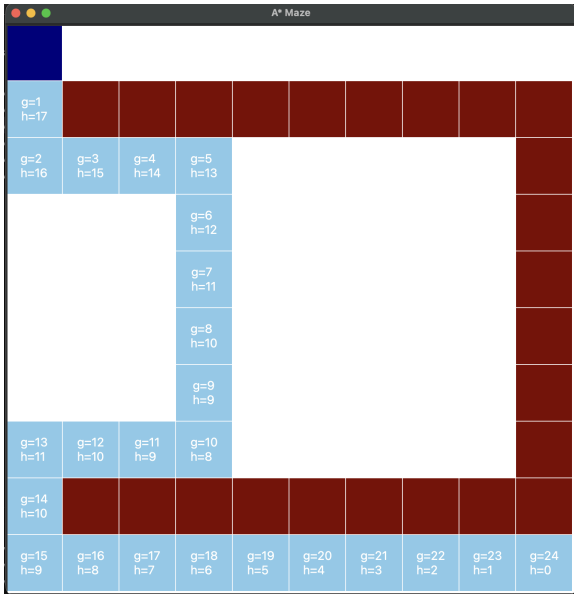
Makes it to (2, 1) before traversing down.

α = 0.2, β = 0.8:



Makes it to (2, 2) before traversing down.

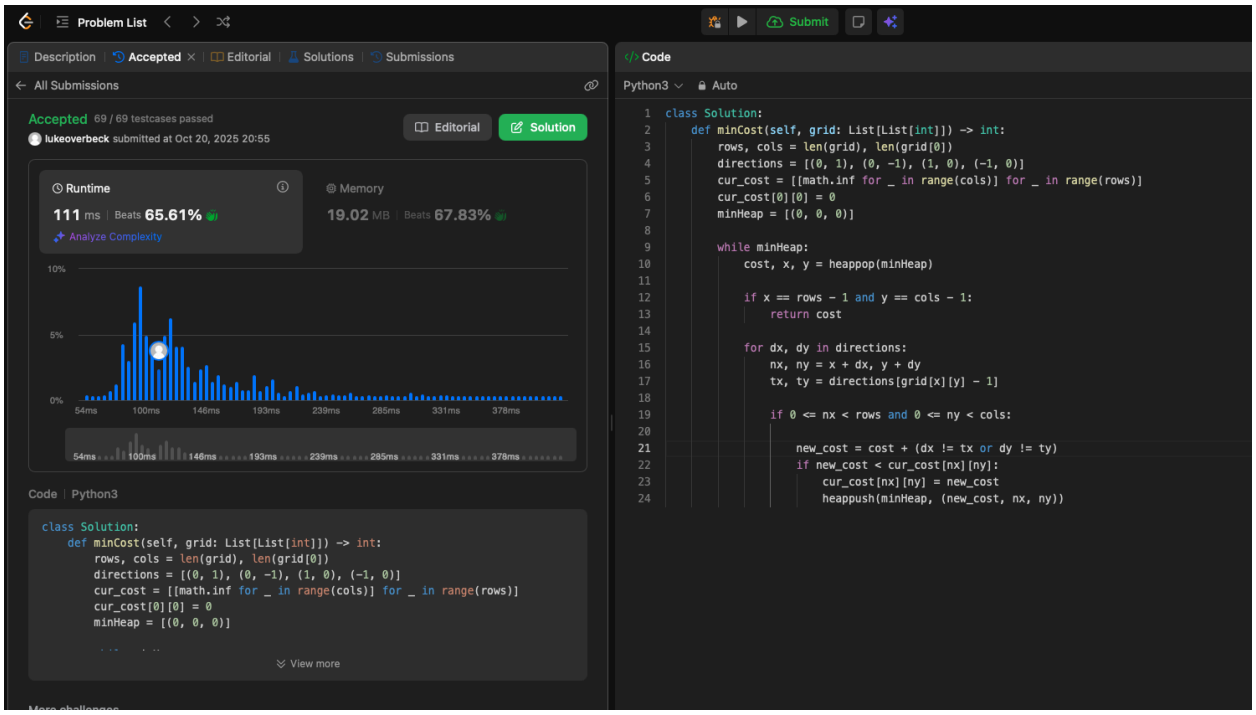
α = 0.1, β = 0.9:



Makes it to (2, 3) before traversing down.

As β increases and α decreases past β = 0.6 and α = 0.4, the algorithm moves further along the top of the maze before traversing down. This makes sense because moves closer to the goal get favored more and more. The path for greedy best-first search (α = 0, β = 1) reaches (2,4), so the pattern continues past β = 0.9.

#4 - Minimum Cost to Make at Least One Valid Path in a Grid



class Solution:

    def minCost(self, grid: List[List[int]]) -> int:

```
rows, cols = len(grid), len(grid[0])
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
cur_cost = [[math.inf for _ in range(cols)] for _ in range(rows)]
cur_cost[0][0] = 0
minHeap = [(0, 0, 0)]

while minHeap:
    cost, x, y = heappop(minHeap)

    if x == rows - 1 and y == cols - 1:
        return cost

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        tx, ty = directions[grid[x][y] - 1]

        if 0 <= nx < rows and 0 <= ny < cols:

            new_cost = cost + (dx != tx or dy != ty)
            if new_cost < cur_cost[nx][ny]:
                cur_cost[nx][ny] = new_cost
                heappush(minHeap, (new_cost, nx, ny))
```

I found the general formula for Djikstra's algorithm online. I edited it to work for this prompt. I used a second array to keep track of the shortest cost to each node. The core of the algorithm was the comparison between the current point and the next point. I used two sets of variables and compared them. If either of the x or y values of the next point are different from the current point, add 1 to the current cost for that point because that means the direction of the arrow must be changed.