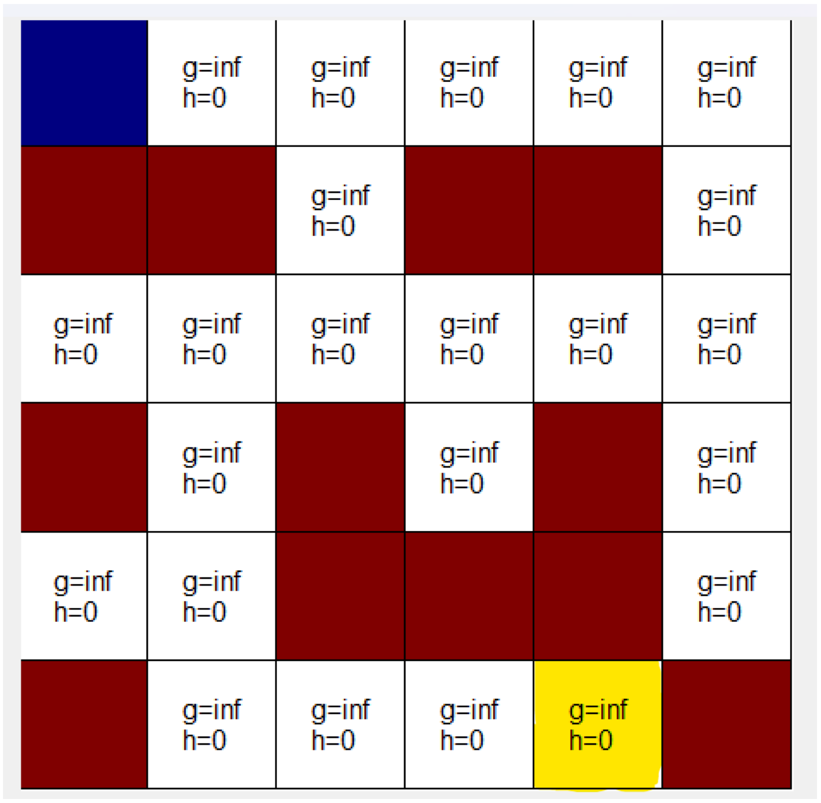
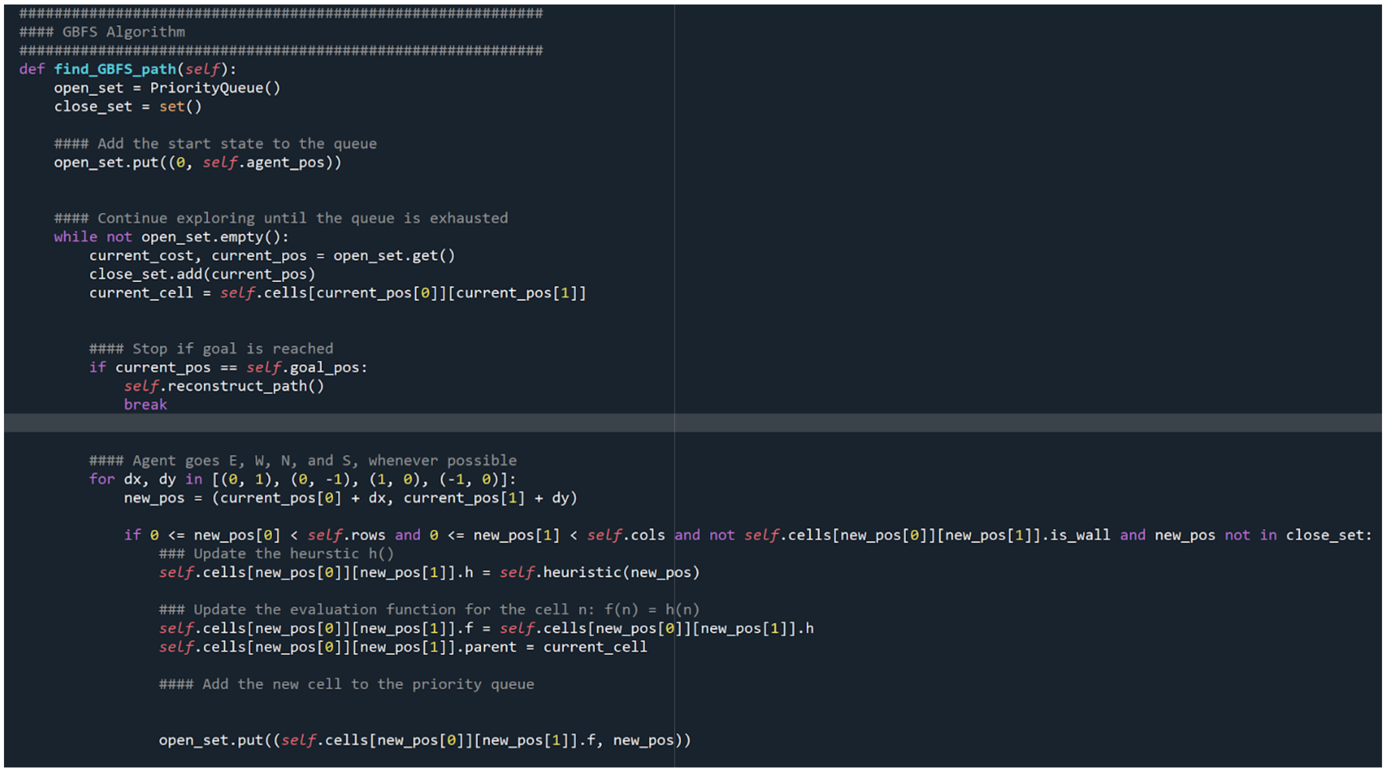
Name: Daniela Cruz Perez
Title: A.I HW 3
Date: 10/21/2025

Problem 1: Comparing Greedy Best-First Search and A*
Using the maze below, we are interested in observing how the two algorithms approach the optimum path. The start is marked in blue and the goal is marked in yellow.

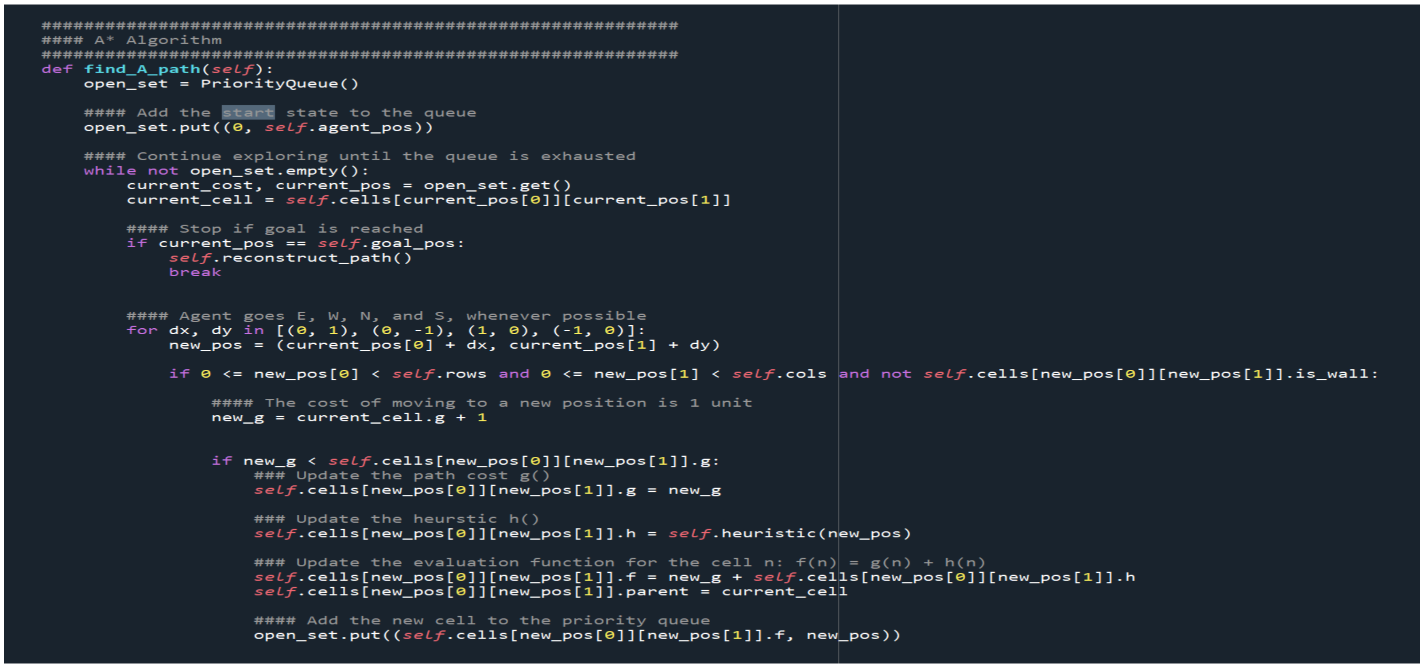


First, we implemented the GBFS algorithm: GBFS does not consider the actual cost g, and instead relies solely on the heuristic. We implemented a closed set to track the visited neighbors the path has visited/expanded. This prevents an infinite loop through the set. The heuristic is computed for each neighbor to the current node and added to the open set. The node with the lowest heuristic is then expanded and the process continues until the goal is reached.



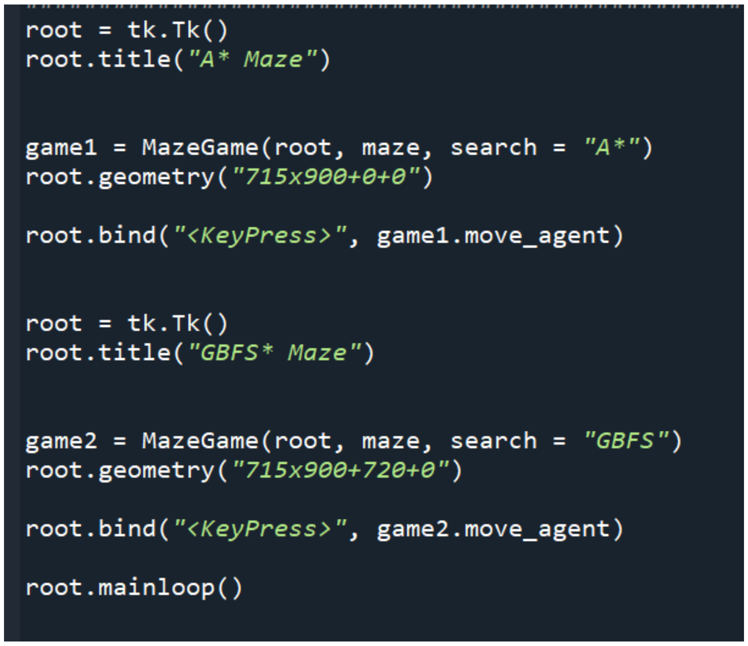
```
############################################################
#### GBFS Algorithm
############################################################
def find_GBFS_path(self):
    open_set = PriorityQueue()
    close_set = set()

    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))


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


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


        #### Agent goes E, W, N, and S, whenever possible
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_pos = (current_pos[0] + dx, current_pos[1] + dy)

            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 and new_pos not in close_set:
                ### Update the heurstic h()
                self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

                ### Update the evaluation function for the cell n: f(n) = h(n)
                self.cells[new_pos[0]][new_pos[1]].f = self.cells[new_pos[0]][new_pos[1]].h
                self.cells[new_pos[0]][new_pos[1]].parent = current_cell

                #### Add the new cell to the priority queue


                open_set.put((self.cells[new_pos[0]][new_pos[1]].f, new_pos))
```

Compared to the given A*:



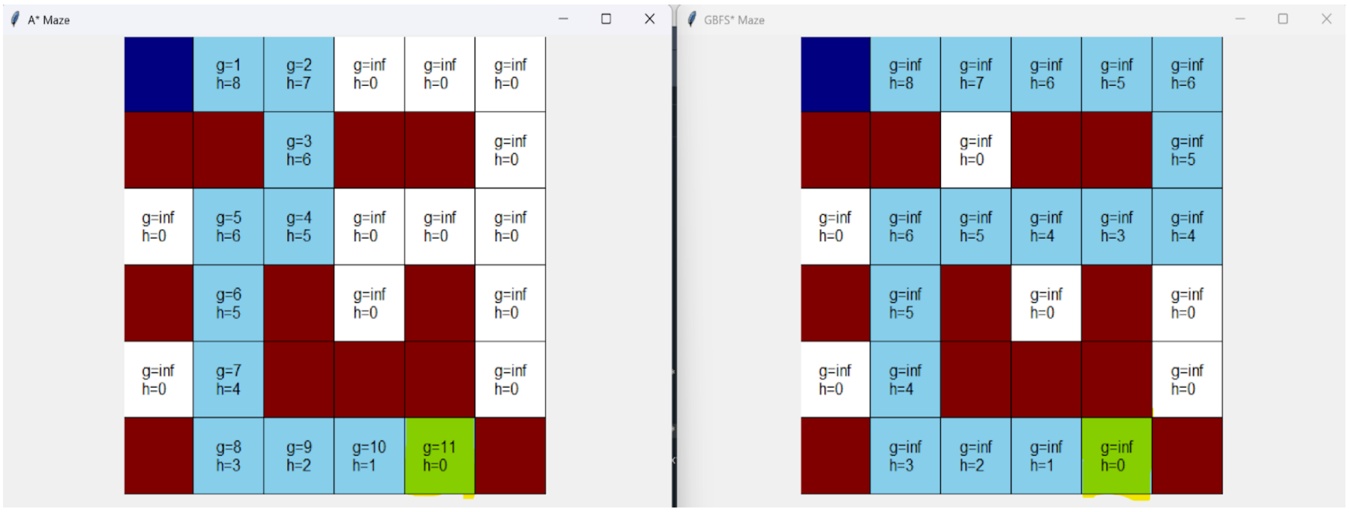
```
############################################################
#### A* Algorithm
############################################################
def find_A_path(self):
    open_set = PriorityQueue()

    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))

    #### 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]]

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


        #### Agent goes E, W, N, and S, whenever possible
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_pos = (current_pos[0] + dx, current_pos[1] + dy)

            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
                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

                    ### Update the heurstic h()
                    self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

                    ### 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]].parent = current_cell

                    #### Add the new cell to the priority queue
                    open_set.put((self.cells[new_pos[0]][new_pos[1]].f, new_pos))
```

To run side-by-side the two mazes are stored in one main.loop:



```
root = tk.Tk()
root.title("A* Maze")


game1 = MazeGame(root, maze, search = "A*")
root.geometry("715x900+0+0")

root.bind("<KeyPress>", game1.move_agent)


root = tk.Tk()
root.title("GBFS* Maze")


game2 = MazeGame(root, maze, search = "GBFS")
root.geometry("715x900+720+0")

root.bind("<KeyPress>", game2.move_agent)

root.mainloop()
```

Result:



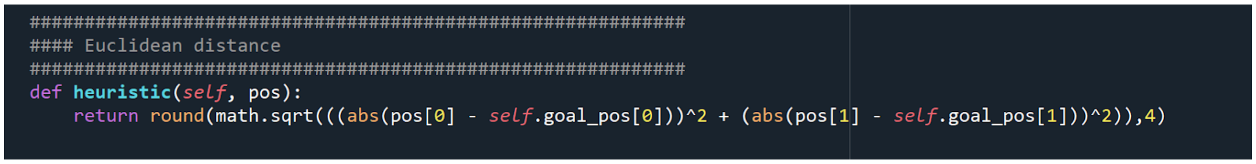
🟦	g=1 h=8	g=2 h=7	g=inf h=0	g=inf h=0	g=inf h=0
🟥	🟥	g=3 h=6	🟥	🟥	g=inf h=0
g=inf h=0	g=5 h=6	g=4 h=5	g=inf h=0	g=inf h=0	g=inf h=0
🟥	g=6 h=5	🟥	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=7 h=4	🟥	🟥	🟥	g=inf h=0
🟥	g=8 h=3	g=9 h=2	g=10 h=1	g=11 h=0	🟥

🟦	g=inf h=8	g=inf h=7	g=inf h=6	g=inf h=5	g=inf h=6
🟥	🟥	g=inf h=0	🟥	🟥	g=inf h=5
g=inf h=0	g=inf h=6	g=inf h=5	g=inf h=4	g=inf h=3	g=inf h=4
🟥	g=inf h=5	🟥	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=4	🟥	🟥	🟥	g=inf h=0
🟥	g=inf h=3	g=inf h=2	g=inf h=1	g=inf h=0	🟥

Comments: The A* algorithm achieved the optimal path, and the GBFS failed. Both the positions at (0,3) and (1,2) have a heuristic of 6 to the goal position. However, since GBFS does not implement g and moves to the smallest coordinate when there is a tie in heuristics by default, it fails to return the optimal path.

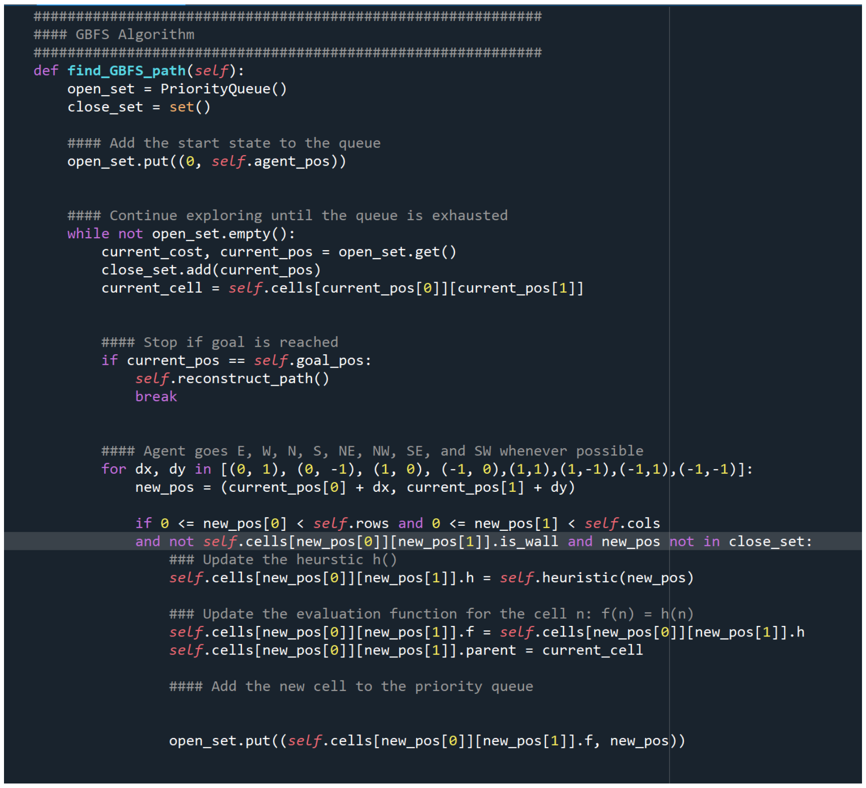
Problem 2: Repeat the above experiment using the Euclidean Distance heuristic

First we compute the euclidean distance:

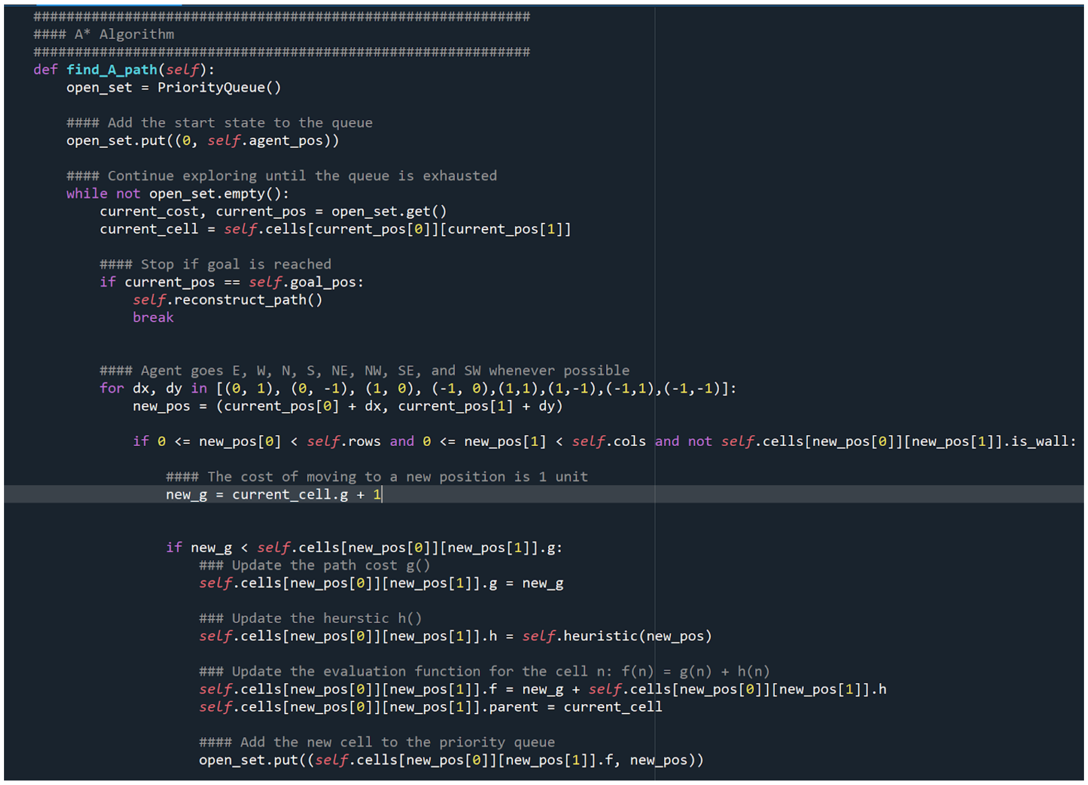


```
############################################################
#### Euclidean distance
############################################################
def heuristic(self, pos):
    return round(math.sqrt(((abs(pos[0] - self.goal_pos[0]))^2 + (abs(pos[1] - self.goal_pos[1]))^2)),4)
```

Next, we reconfigure the A* and GBFS algorithms to allow diagonals:

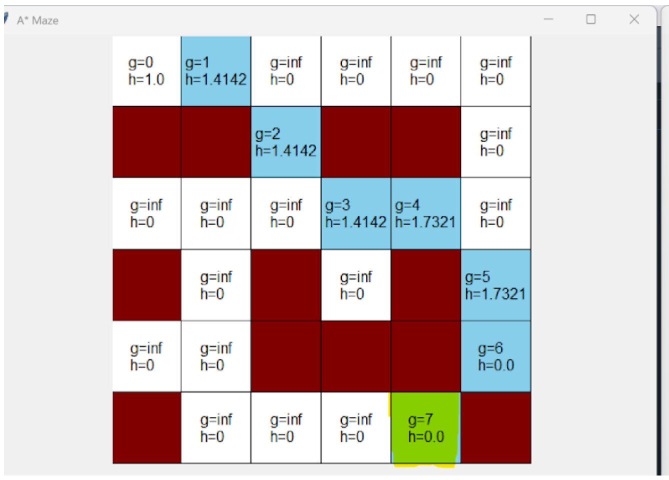


```
############################################################
#### GBFS Algorithm
############################################################
def find_GBFS_path(self):
    open_set = PriorityQueue()
    close_set = set()

    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))


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


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


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

            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 and new_pos not in close_set:
                ### Update the heurstic h()
                self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

                ### Update the evaluation function for the cell n: f(n) = h(n)
                self.cells[new_pos[0]][new_pos[1]].f = self.cells[new_pos[0]][new_pos[1]].h
                self.cells[new_pos[0]][new_pos[1]].parent = current_cell

                #### Add the new cell to the priority queue


                open_set.put((self.cells[new_pos[0]][new_pos[1]].f, new_pos))
```

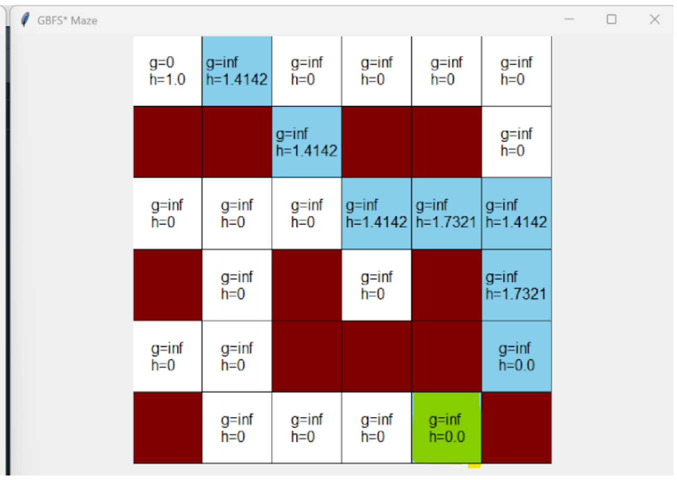


```python
############################################################
#### A* Algorithm
############################################################
def find_A_path(self):
    open_set = PriorityQueue()

    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))

    #### 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]]

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


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

            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
                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

                    ### Update the heurstic h()
                    self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

                    ### 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]].parent = current_cell

                    #### Add the new cell to the priority queue
                    open_set.put((self.cells[new_pos[0]][new_pos[1]].f, new_pos))
```

Result:



g=0 h=1.0	g=1 h=1.4142	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0
🟥	🟥	g=2 h=1.4142	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=3 h=1.4142	g=4 h=1.7321	g=inf h=0
🟥	g=inf h=0	🟥	g=inf h=0	🟥	g=5 h=1.7321
g=inf h=0	g=inf h=0	🟥	🟥	🟥	g=6 h=0.0
🟥	g=inf h=0	g=inf h=0	g=inf h=0	g=7 h=0.0	🟥



g=0 h=1.0	g=inf h=1.4142	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0
🟥	🟥	g=inf h=1.4142	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=1.4142	g=inf h=1.7321	g=inf h=1.4142
🟥	g=inf h=0	🟥	g=inf h=0	🟥	g=inf h=1.7321
g=inf h=0	g=inf h=0	🟥	🟥	🟥	g=inf h=0.0
🟥	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0.0	🟥

Comments:

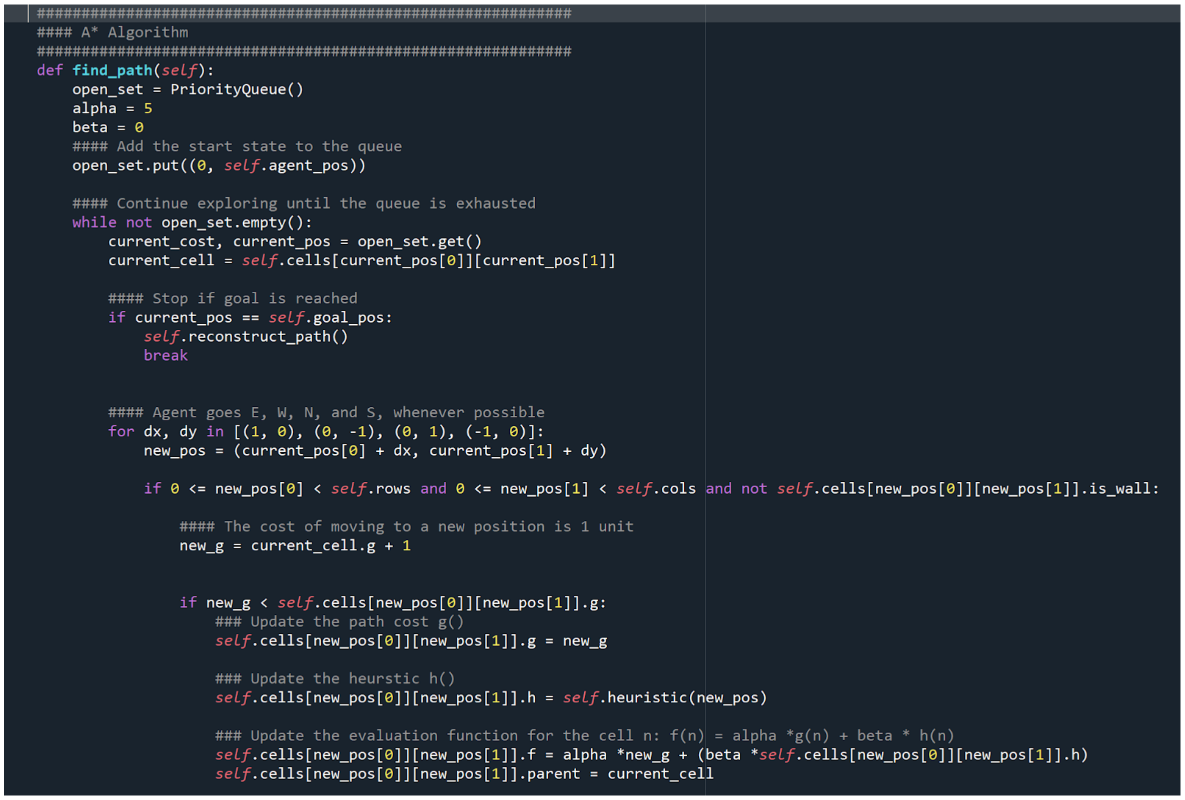
The GBFS did not perform optimally compared to the A* algorithm. The GBFS explores the node at (2,5) instead of skipping to (3,5) since it has a smaller heuristic. Using the Euclidean algorithm, both algorithms perform faster as diagonals are allowed.

Problem 3:

Part 1

A* using the weighted function, $f(n) = \alpha \cdot g(n) + \beta \cdot h(n)$ *where* $\alpha, \beta \geq 0$

First, we reconfigured the function to include alpha and beta and tested for each value of beta and alpha in the table.



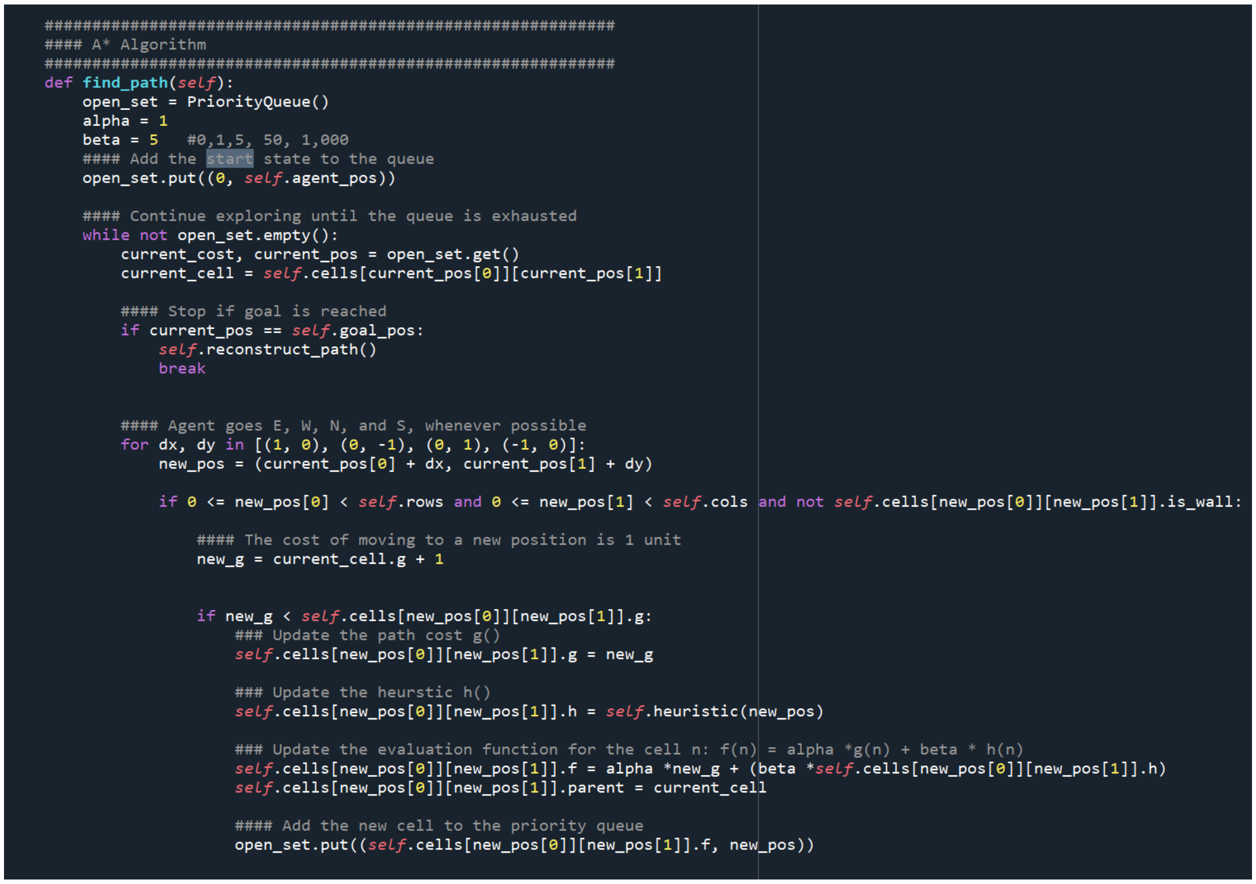
```
############################################################
#### A* Algorithm
############################################################
def find_path(self):
    open_set = PriorityQueue()
    alpha = 5
    beta = 0
    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))

    #### 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]]

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


        #### Agent goes E, W, N, and S, whenever possible
        for dx, dy in [(1, 0), (0, -1), (0, 1), (-1, 0)]:
            new_pos = (current_pos[0] + dx, current_pos[1] + dy)

            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
                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

                    ### Update the heurstic h()
                    self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

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

α	β	Observed Behavior
5	0	The weight is 0, and f(n) = g(n) which is optimal
5	5	The weight is 1, and is always optimal
6	18	The weight is 3, and is not optimal
18	6	The weight is ⅓ and is optimal
5	2500	The weight is 500 and is not optimal and performs worse compared to the the weight of 3

Part 2:

Manipulating the bias, β

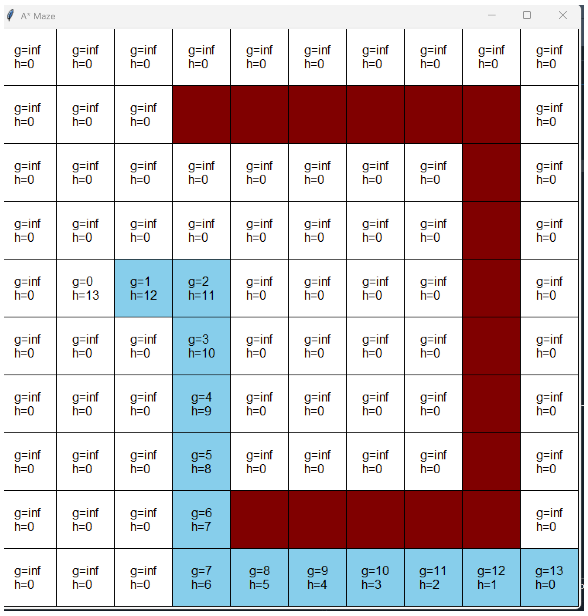
Here, we set the α to constant 1 and β is reconfigured to 0,1,5, 50, 1,000.



```
############################################################
#### A* Algorithm
############################################################
def find_path(self):
    open_set = PriorityQueue()
    alpha = 1
    beta = 5   #0,1,5, 50, 1,000
    #### Add the start state to the queue
    open_set.put((0, self.agent_pos))

    #### 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]]

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


        #### Agent goes E, W, N, and S, whenever possible
        for dx, dy in [(1, 0), (0, -1), (0, 1), (-1, 0)]:
            new_pos = (current_pos[0] + dx, current_pos[1] + dy)

            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
                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

                    ### Update the heurstic h()
                    self.cells[new_pos[0]][new_pos[1]].h = self.heuristic(new_pos)

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

                    #### Add the new cell to the priority queue
                    open_set.put((self.cells[new_pos[0]][new_pos[1]].f, new_pos))
```

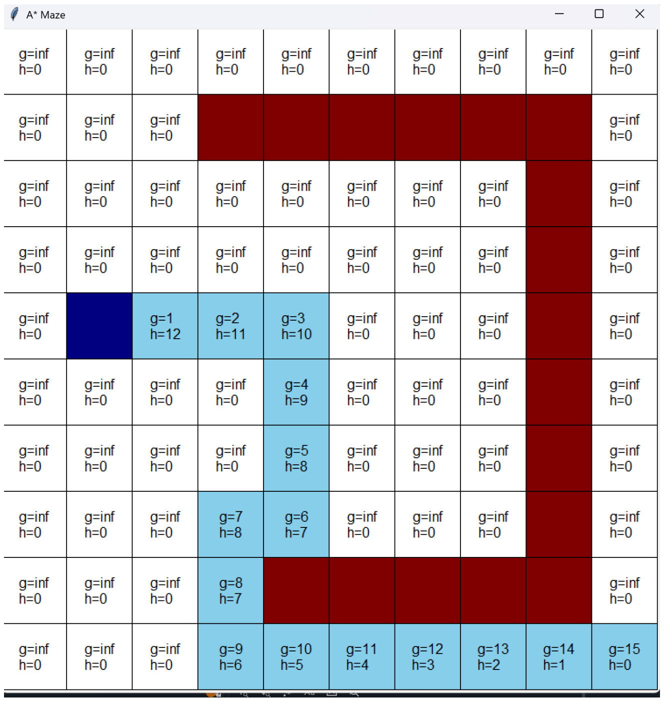
Result:

Bias of 0 and 1 had the same result and is optimal:


A* Maze

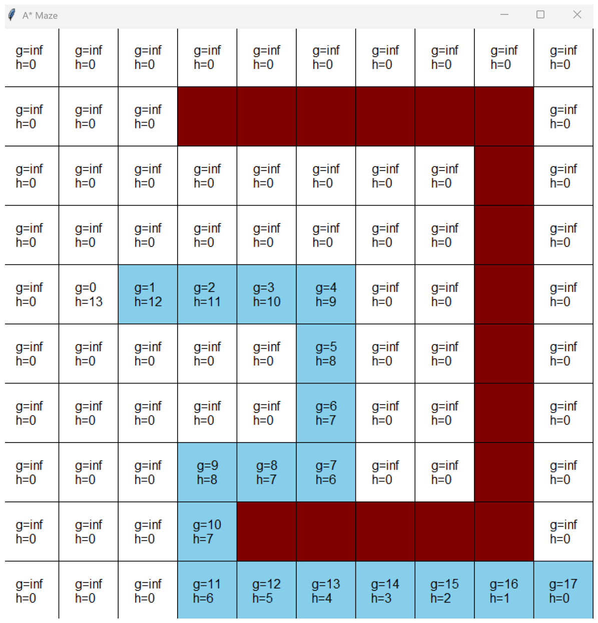
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	🟥	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=0 h=13	g=1 h=12	g=2 h=11	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=3 h=10	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=4 h=9	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=5 h=8	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=6 h=7	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=7 h=6	g=8 h=5	g=9 h=4	g=10 h=3	g=11 h=2	g=12 h=1	g=13 h=0

Bias of 5 was not optimal:


A* Maze

g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	🟥	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	🟦	g=1 h=12	g=2 h=11	g=3 h=10	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=4 h=9	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=5 h=8	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=7 h=8	g=6 h=7	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=8 h=7	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=9 h=6	g=10 h=5	g=11 h=4	g=12 h=3	g=13 h=2	g=14 h=1	g=15 h=0

Bias of 50 and 1000 were not optimal and performed worse:



g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	🟥	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=0 h=13	g=1 h=12	g=2 h=11	g=3 h=10	g=4 h=9	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=5 h=8	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=inf h=0	g=6 h=7	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=9 h=8	g=8 h=7	g=7 h=6	g=inf h=0	g=inf h=0	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=10 h=7	🟥	🟥	🟥	🟥	🟥	g=inf h=0
g=inf h=0	g=inf h=0	g=inf h=0	g=11 h=6	g=12 h=5	g=13 h=4	g=14 h=3	g=15 h=2	g=16 h=1	g=17 h=0

Comments: The algorithm is optimal for weights 0 and 1 but is not for values of 5,50, and 1000 as expected. It would always be optimal for 1 as there is no effect on g(n) and h(n) as performs as A* . In addition, when h(n) is 0, the algorithm is Dijkstra, which is optimal. Even when the bias is high, it never truly operates as a greedy best first search since it still considers g(n), the actual cost. In addition, the algorithm was optimal until it reached a bias of 3, and performed worse after a bias of 5.

Conclusion: The A* algorithm guarantees optimality as demonstrated in the Manhattan and Euclidean distance experiments. The weighted A* algorithm performed greedier as it prioritized the heuristic, this allows it to be faster in choosing the next step. However this does not guarantee optimality . Although not depicted in this report, GBFS has also performed optimally in the first experiment under a different maze, however optimality isn't guaranteed.