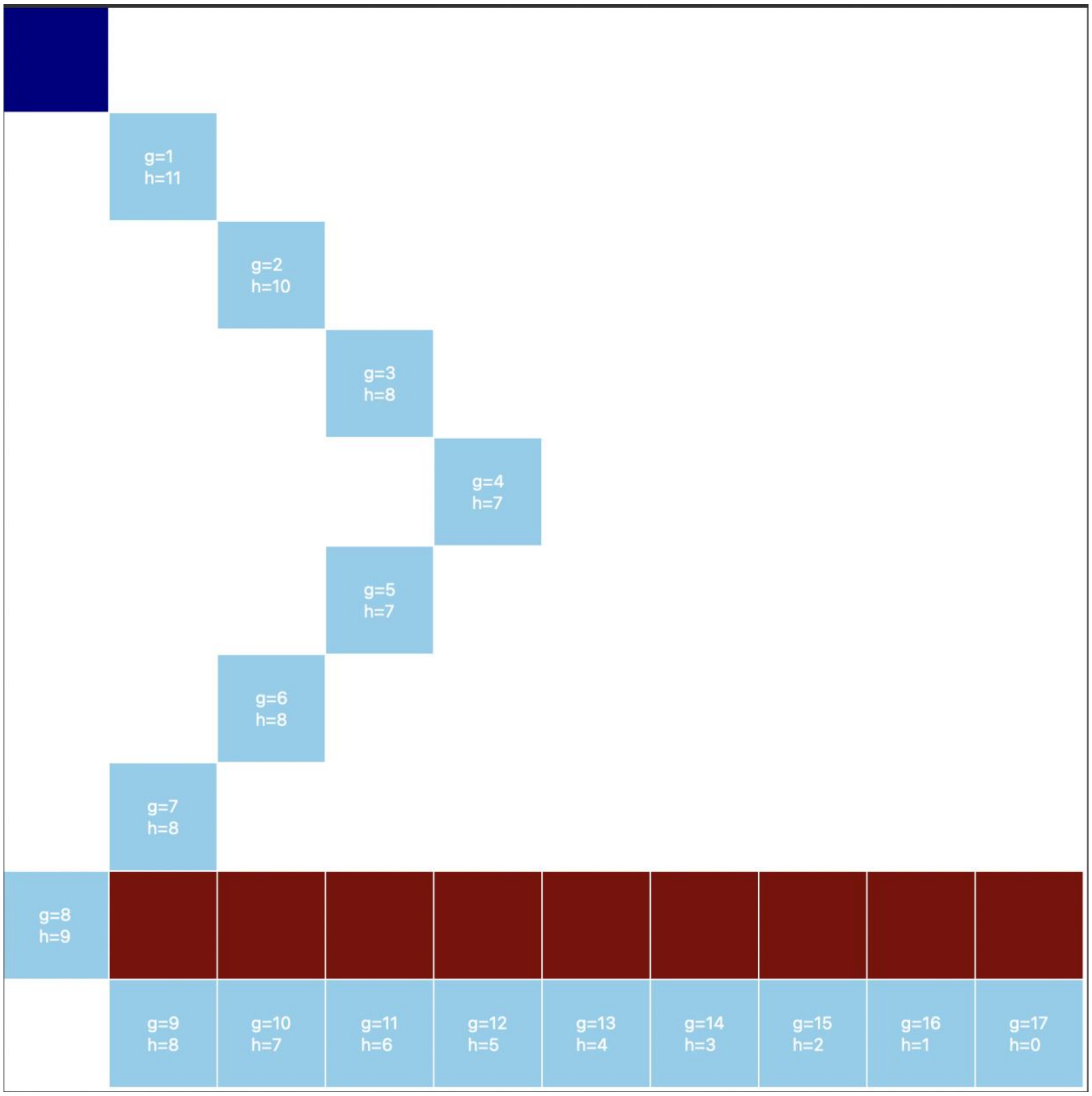
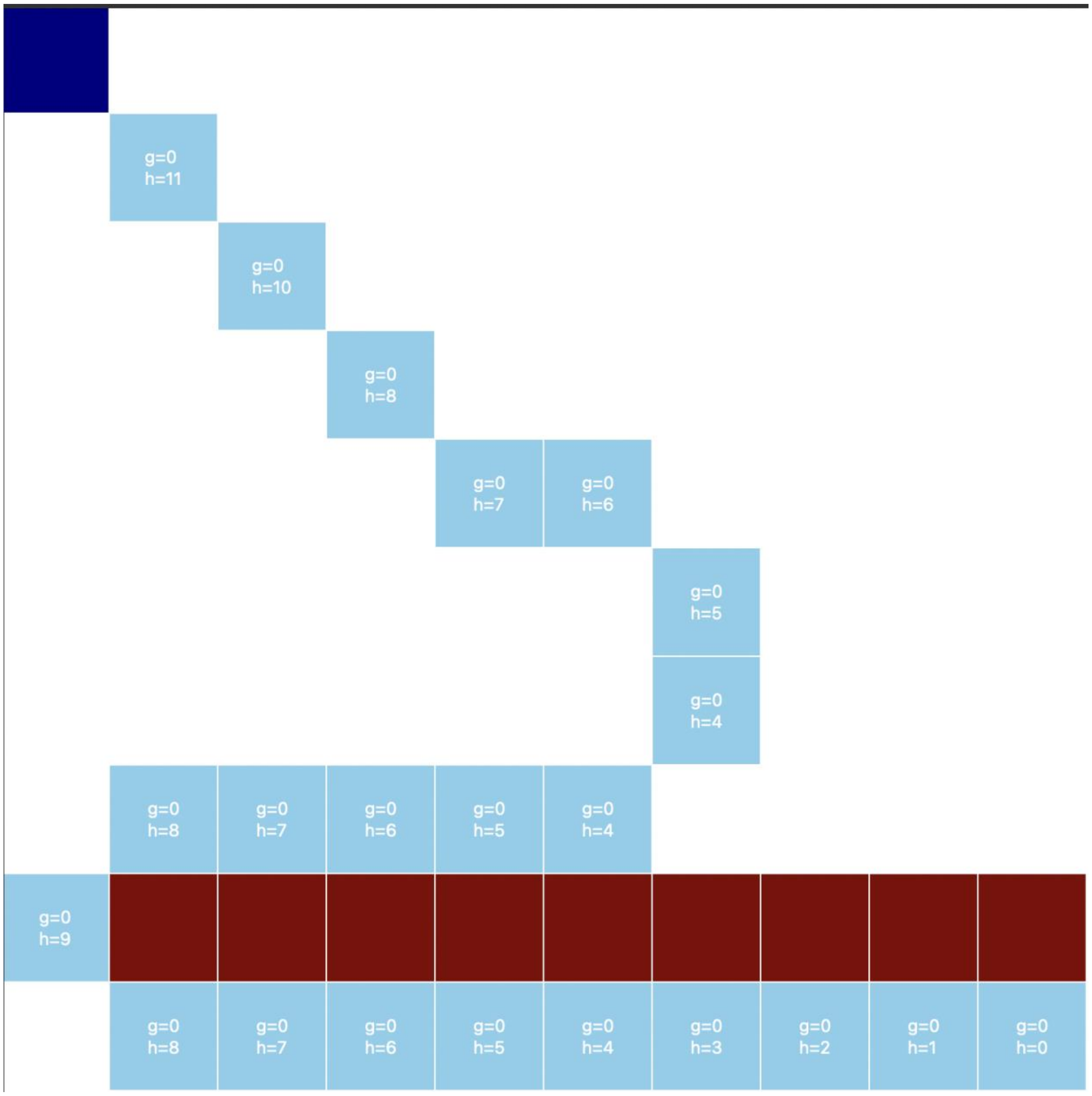
Question 2:

A* Algorithm



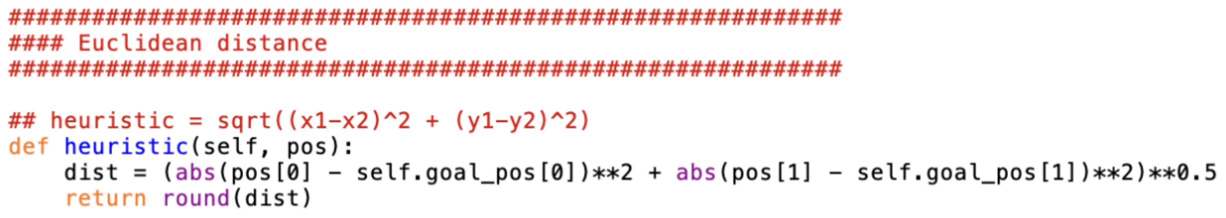
Explanation: A* algorithm functions the same as previously, considering both path cost and future cost. When using the Euclidean distance heuristic, the path to the goal is allowed to use diagonal moves (rather than the Manhattan Distance's strict four directly movement). Euclidean calculates distance in a straight line, or "as the crow flies", using this formula: d = sqrt((x2 - x1)^2 + (y2 - y1)^2). Euclidean prioritizes diagonal movements and will curve around obstacles to reach the goal.

Greedy Best-First Algorithm



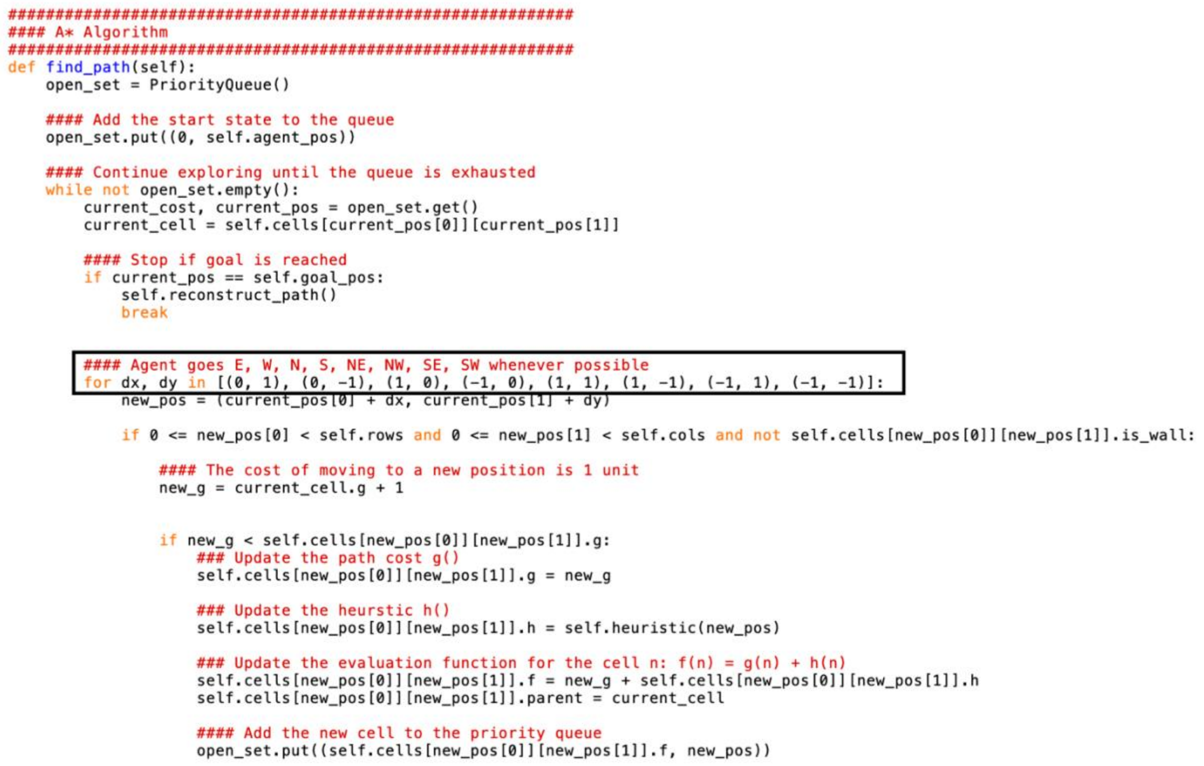
Explanation: Greedy Best-First algorithm functions the same as the previous problem, by ignoring path cost and only taking into consideration future cost. Using the Euclidean distance heuristic, it prioritizes diagonal movements and moves towards the goal in a straight line. The straight line is blocked by an obstacle so it must move along it, and curves around it towards the goal.

Euclidean Distance



```
##############################################################
#### Euclidean distance
##############################################################

## heuristic = sqrt((x1-x2)^2 + (y1-y2)^2)
def heuristic(self, pos):
    dist = (abs(pos[0] - self.goal_pos[0])**2 + abs(pos[1] - self.goal_pos[1])**2)**0.5
    return round(dist)
```



```
############################################################
#### A* Algorithm
############################################################
def find_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, 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))
```

Explanation: In the AStarMaze code, I changed the heuristic so it represent the Euclidean distance formula: d = sqrt((x2 - x1)^2 + (y2 - y1)^2). I also altered the directions the agent is able to move (originally from the four cardinal directions to adding NE, NW, SE, SW). With these combined always the program to bias diagonal movements and move in those diagonal movements towards the goal.