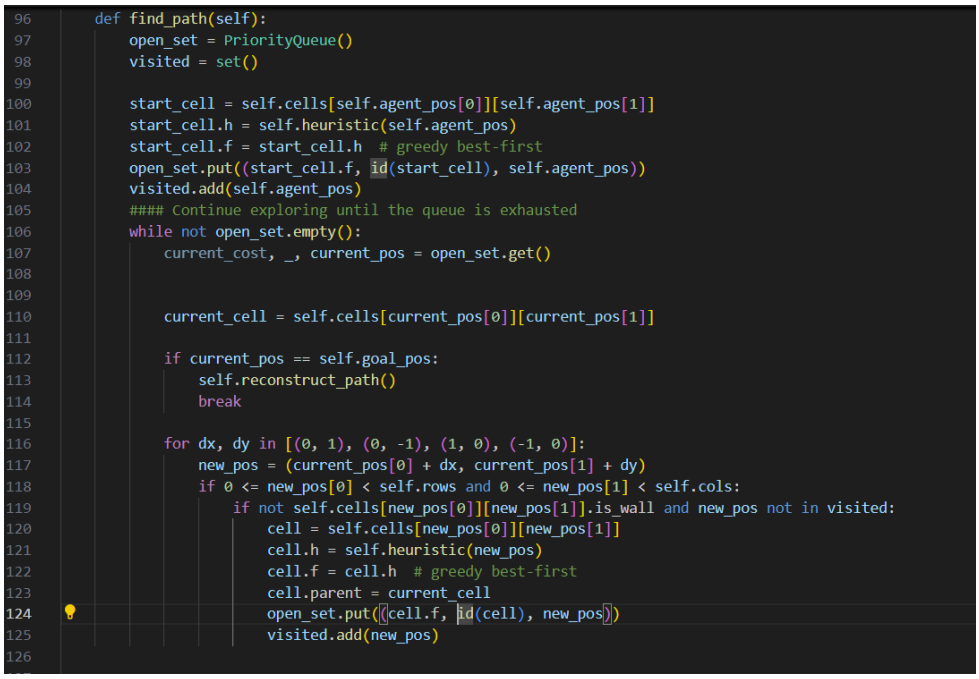
Name: Joe Pepe
Purpose: to explain the steps taken to complete HW3 and demonstrate understanding of Greedy BFS, Euclidean Distance heuristic, and altering the A* evaluation function

Problem 1:

In problem 1, we must turn A* into a greedy best first search. This is the code after it has been altered. The most important parts are on lines 98, 104, 122, 125. Line 98, and 104 both relate to a visited queue that I created. This is a necessary step because the greedy algorithm can get stuck in an infinite loop if it does not account for previously visited tiles. When encountering the wall, it can be forced to move somewhere else, then move right back to where it was since that tile has a better heuristic value.

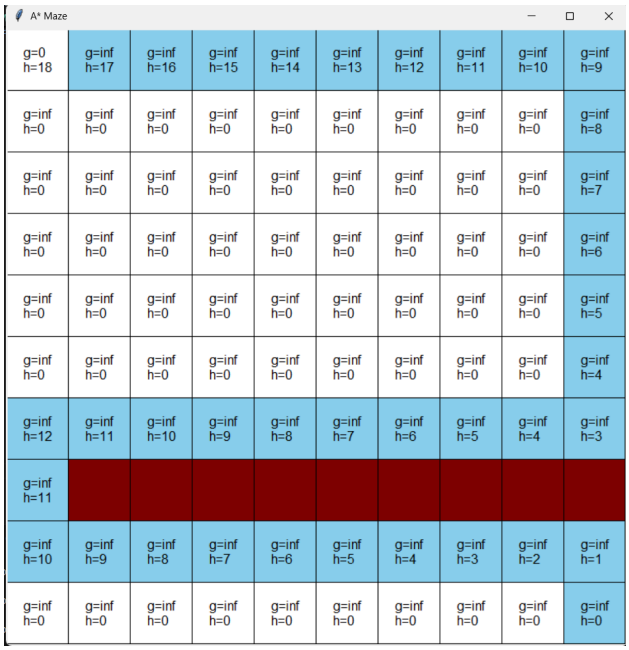
On lines 122 and 125, the evaluation function is altered, which gets into the heart of what a greedy algorithm actually is. On line 122, the overall value of the function is set to be equal to the heuristic value. This means that past cost has no say in what node is selected. All remnants of past node cost, g, has been deleted.



```
96      def find_path(self):
97          open_set = PriorityQueue()
98          visited = set()
99
100         start_cell = self.cells[self.agent_pos[0]][self.agent_pos[1]]
101         start_cell.h = self.heuristic(self.agent_pos)
102         start_cell.f = start_cell.h  # greedy best-first
103         open_set.put((start_cell.f, id(start_cell), self.agent_pos))
104         visited.add(self.agent_pos)
105         #### Continue exploring until the queue is exhausted
106         while not open_set.empty():
107             current_cost, _, current_pos = open_set.get()
108
109
110             current_cell = self.cells[current_pos[0]][current_pos[1]]
111
112             if current_pos == self.goal_pos:
113                 self.reconstruct_path()
114                 break
115
116             for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
117                 new_pos = (current_pos[0] + dx, current_pos[1] + dy)
118                 if 0 <= new_pos[0] < self.rows and 0 <= new_pos[1] < self.cols:
119                     if not self.cells[new_pos[0]][new_pos[1]].is_wall and new_pos not in visited:
120                         cell = self.cells[new_pos[0]][new_pos[1]]
121                         cell.h = self.heuristic(new_pos)
122                         cell.f = cell.h  # greedy best-first
123                         cell.parent = current_cell
124                         open_set.put((cell.f, id(cell), new_pos))
125                         visited.add(new_pos)
126
```

The results below speak for themselves. On the left side, we see the greedy algorithm, which starts going right because it lowers the heuristic value. When it encounters the wall, it realizes that it will be forced to go all the way around, which will result in a cost much higher than what was necessary. Since it does not care about the past cost that it is incurring, however, it is fine with having this very high cost compared to the optimal solution.

A*, on the other hand, is able to find the optimal solution. While exploring nodes, it is able to see that going down is the only way to balance past cost and the heuristic. A* is able to find the optimal solution. This balance between past cost and a heuristic makes it significantly better than the greedy BFS.



Problem 2:

Lets repeat the same experiment but use Euclidean distance instead of Manhattan distance. There is only one minor change to make to the previous code. We will take the square root instead of the absolute value to calculate the distance between the start and the goal as if there was a straight line between them.

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

Now that we have changed the heuristic to take the exact distance between two points, we need to update the ways that the agent can move. It needs to be able to move diagonally.

```
38  class MazeGame:
150     def move_agent(self, event):
153         if event.keysym == 'Right' and self.agent_pos[1] + 1 < self.cols and not self.cells[self.agent_pos[0]][self.agent_pos[1] + 1].is_wall:
154             self.agent_pos = (self.agent_pos[0], self.agent_pos[1] + 1)
155
156
157         #### Move Left, if possible
158         elif event.keysym == 'Left' and self.agent_pos[1] - 1 >= 0 and not self.cells[self.agent_pos[0]][self.agent_pos[1] - 1].is_wall:
159             self.agent_pos = (self.agent_pos[0], self.agent_pos[1] - 1)
160
161         #### Move Down, if possible
162         elif event.keysym == 'Down' and self.agent_pos[0] + 1 < self.rows and not self.cells[self.agent_pos[0] + 1][self.agent_pos[1]].is_wall:
163             self.agent_pos = (self.agent_pos[0] + 1, self.agent_pos[1])
164
165         #### Move Up, if possible
166         elif event.keysym == 'Up' and self.agent_pos[0] - 1 >= 0 and not self.cells[self.agent_pos[0] - 1][self.agent_pos[1]].is_wall:
167             self.agent_pos = (self.agent_pos[0] - 1, self.agent_pos[1])
168
169         elif event.keysym == 'Northeast' and self.agent_pos[0] - 1 >= 0 and not self.cells[self.agent_pos[0] - 1][self.agent_pos[1] +1].is_wall:
170             self.agent_pos = (self.agent_pos[0] - 1, self.agent_pos[1] + 1)
171
172         elif event.keysym == 'Northwest' and self.agent_pos[0] - 1 >= 0 and not self.cells[self.agent_pos[0] - 1][self.agent_pos[1] - 1].is_wall:
173             self.agent_pos = (self.agent_pos[0] - 1, self.agent_pos[1] -1)
174
175         elif event.keysym == 'Southeast' and self.agent_pos[0] - 1 >= 0 and not self.cells[self.agent_pos[0] +1][self.agent_pos[1] +1].is_wall:
176             self.agent_pos = (self.agent_pos[0] + 1, self.agent_pos[1] + 1)
177
178         elif event.keysym == 'Southwest' and self.agent_pos[0] - 1 >= 0 and not self.cells[self.agent_pos[0] - 1][self.agent_pos[1] -1].is_wall:
179             self.agent_pos = (self.agent_pos[0] + 1, self.agent_pos[1] -1)
```

I added 4 new types of movements, accounting for every diagonal movement it could make. The difference between these movements and the others is that it needs to be able to move in two directions at once. On lines 170, 173, 176, and 179 you can see that each one has both positions being updated in some fashion, rather than one at a time.

Finally, we must update the find_path method. We need to allow our greedy algorithm to actually consider the diagonal cells. Therefore, I added 4 more pairs to be considered, (1,1), (-1,-1), (1,-1), and (-1, 1).

```
115
116             for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]:
117                 new_pos = (current_pos[0] + dx, current_pos[1] + dy)
118                 if 0 <= new_pos[0] < self.rows and 0 <= new_pos[1] < self.cols:
119                     if not self.cells[new_pos[0]][new_pos[1]].is_wall and new_pos not in visited:
120                         cell = self.cells[new_pos[0]][new_pos[1]]
121                         cell.h = self.heuristic(new_pos)
122                         cell.f = cell.h  # greedy best-first
123                         cell.parent = current_cell
124                         open_set.put((cell.f, id(cell), new_pos))
125                         visited.add(new_pos)
126
```

Okay, we have made all of the code changes necessary to our greedy Euclidean algorithm. Lets see what the result is.

On the left, we have our greedy bfs. We see that it darts straight at the goal, and eventually is forced to move around and incur a longer path cost, just like what happened with the Manhattan distance example. The only difference is that this algorithm was able to move diagonally.

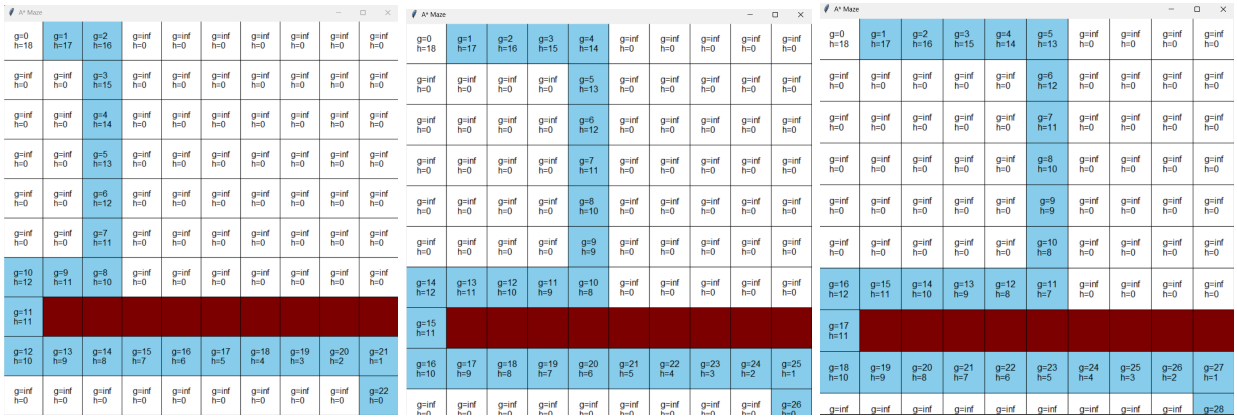
On the right, we have A*. At first glance, the algorithm looks a little strange - it starts off by heading straight for the goal just like the greedy BFS. There is a reason for this. In the beginning, there is not a log of incurred cost, and the heuristic dominates the function. This causes it to dip towards the goal even though there is a wall. Since there is diagonal movement though, A* can still reach the edge of the wall in 6 moves. Even if it just went straight down, it can still only get there in 6 moves. Even though it is dipping towards the goal, A* still reaches the goal in the optimal number of moves thanks to the new diagonal movement.

Problem 3:

Lets complete the final problem. Our evaluation function can be changed so that it has weights, or coefficients. This will cause A* to still evaluate both the heuristic and the past cost, but give more favor to one or the other.

```
130         self.cells[new_pos[0]][new_pos[1]].f = (new_g) + (3 * self.cells[new_pos[0]][new_pos[1]].h)
131         self.cells[new_pos[0]][new_pos[1]].parent = current_cell
132
```

There is only one thing that is to be changed. On line 130, I give 3x the weight to hour heuristic. Any changes I make in the future will be on this line, but with different weights. I will say what I changed the weight to before showing the results. So let's look at the results for 3x, 10x and 100x weight for the heuristic.



Here are the results for 3x, 10x, and 100x respectively. In the first one, we notice that the solution is not optimal. The algorithm shows bias for a node to the right that is closer to the goal state. It is beginning to act like the greedy algorithm. It only moves one node in the wrong direction though before it corrects itself and starts moving towards the wall and towards the correct solution, however.

In the next image, we see a 10x bias for the heuristic. This algorithm moves several nodes in the wrong direction before correcting itself, incurring a lot of unnecessary cost. This algorithm looks even more like the greedy algorithm than the last one.

In the rightmost image, we see a 100x bias for the heuristic. This one goes further out of the way than the last one, incurs even more unnecessary cost.

Let's look at one last image. What will happen if we add 100x bias to g? It will not look too interesting in the image, it will just place a very high weight on the cost. Therefore it will find an optimal solution just like A* would have, but it will be more similar to BFS since it doesn't place a lot of emphasis on the heuristic.