AI Final Project: Robot Nurse

Jillian O'Connell and Luke McGinley

Description of the Project:

The purpose of this project was to design a program that would simulate a robot delivering medication to different areas and rooms in a hospital while avoiding walls and prioritizing based on the most urgent deliveries. The image of the hospital layout was provided. In order to begin the program, the hospital layout needed to be divided into a grid and be represented in a way that the computer could understand. For this version, the image was divided into a grid of 30 rows and 38 columns. The image of the hospital layout divided into the grid that was used for this implementation can be seen in Figure 1.

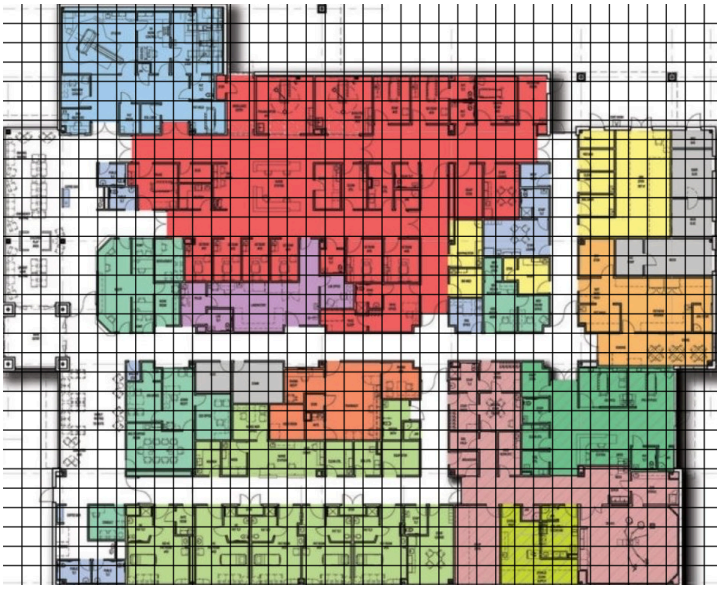


Figure 1: The hospital layout divided into a 30x38 grid.

Once the layout was divided into smaller portions, it could be represented in the program. For this project, an adjacency list was used to represent the floorplan. Each row and column was assigned a value between 0 and 30 and 38 exclusive for the rows and columns respectively.

Every cell was then assigned a tuple equal to its x and y coordinates. These tuples were what were used in the adjacency list for representing the floorplan (Figure 2).

Once the list was created, each cell needed to be assigned its ward and attribute. To do so, a string data field to represent the ward was added to the cell class. An integer data field to represent the priority was also added to the cell class. Each individual cell was then assigned its ward based on the original image divided into the 30 by 38 grid (Figure 3). While the cells were being assigned their wards, a second floor plan image was created based on what ward each cell was assigned (Figure 4). This was helpful later on to know how a cell was assigned its ward if a wall went through the middle of the cell in the floor plan image with the grid. It prevented having to search back through the code to see which ward was manually assigned.

With each cell being assigned its ward, the priority could be assigned as well. Assigning the priorities to each cell was completed in a separate method. This method looped through every coordinate in the list and assigned it a priority based on the ward that was assigned to it (Figure 5).

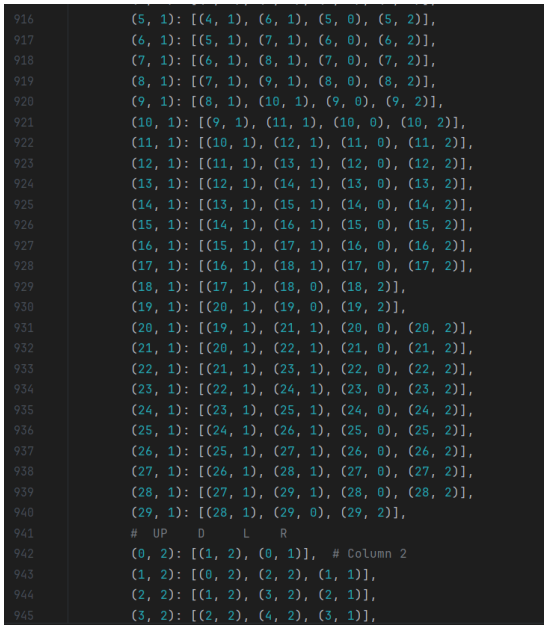


Figure 2: A snippet of the adjacency list. Each cell was connected to its neighbors. If there was a wall between a cell and one of the adjacent cells, the adjacent cell would not appear in the original cell's list.

```
        # rows 4 - 6
        y = 3
        while y < 11:
            self.cells[4][y].ward = "Maternity"
            self.cells[5][y].ward = "Maternity"
            y += 1
        self.cells[4][11].ward = "General"
        self.cells[5][11].ward = "Maternity"
        y = 12
        while y < 30:
            self.cells[4][y].ward = "General"
            self.cells[5][y].ward = "General"
            self.cells[6][y].ward = "General"
            y += 1
        y = 30
        while y < 38:
            self.cells[4][y].ward = "Hallway"
            self.cells[5][y].ward = "Hallway"
            y += 1
        self.cells[6][0].ward = "Hallway"
        self.cells[6][1].ward = "Hallway"
        self.cells[6][2].ward = "Hallway"
        self.cells[6][3].ward = "Maternity"
        self.cells[6][4].ward = "Maternity"
        self.cells[6][5].ward = "Hallway"
        self.cells[6][6].ward = "Maternity"
```

Figure 3: A snippet of assigning each cell its ward. In some cases, as seen above, a loop could be used to assign the same ward to a large block of cells. In other cases, as seen above, each cell was individually assigned its ward.

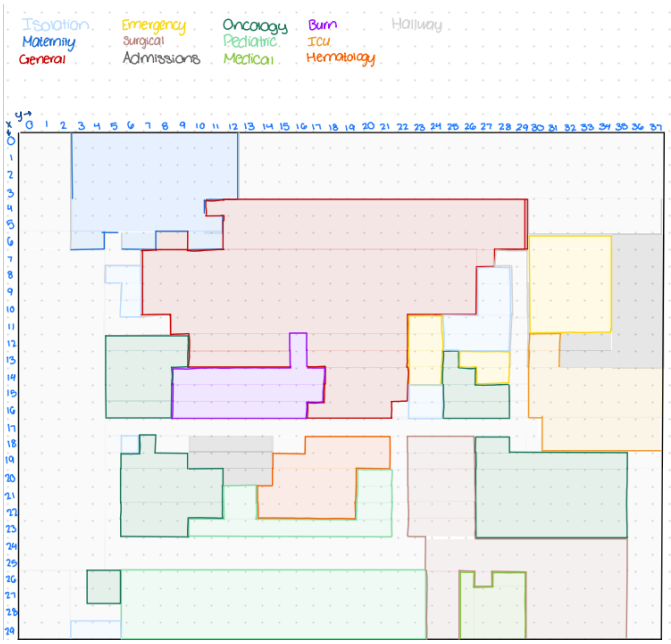


Figure 4: The image created while assigning the ward to each cell. This image is aligned with what ward each cell was assigned unlike the original image with the grid which is ambiguous at parts.

```
192     def assign_priorities(self):
193         #function to check the ward of the cell at the given position and assign priority accordingly
194         for y in range(self.cols):
195             for x in range(self.rows):
196                 if self.cells[x][y].ward == "ICU" or self.cells[x][y].ward == "Emergency":
197                     self.cells[x][y].priority = -5
198                 elif self.cells[x][y].ward == "Oncology" or self.cells[x][y].ward == "Burn":
199                     self.cells[x][y].priority = -5
200                 elif self.cells[x][y].ward == "Surgical" or self.cells[x][y].ward == "Maternity":
201                     self.cells[x][y].priority = -4
202                 elif self.cells[x][y].ward == "Hematology" or self.cells[x][y].ward == "Pediatric":
203                     self.cells[x][y].priority = -3
204                 elif self.cells[x][y].ward == "Medical" or self.cells[x][y].ward == "General":
205                     self.cells[x][y].priority = -2
206                 elif self.cells[x][y].ward == "Admissions" or self.cells[x][y].ward == "Isolation":
207                     self.cells[x][y].priority = -1
208                 elif self.cells[x][y].ward == "Hallway":
209                     self.cells[x][y].priority = 0
210                 else:
211                     self.cells[x][y].priority = 1 #if ward not correct - assigns priority -1
```

Figure 5: The method created to assign a priority to each cell based on the ward assigned to it. Each priority had to be made negative because the priorities were defined from highest to lowest, but the build in priority queue class in python organizes from lowest priority to highest priority.

In this project, two different algorithms were implemented for the robot to reach the delivery locations. These algorithms were Dijkstra's and A*. The algorithms themselves find the path in the same way; the main difference is the heuristic value. A* uses a heuristic coupled with the path cost to determine the next optimal move (Figure 6). Dijkstra's only uses the path cost to determine the next optimal move (Figure 7). Because the step cost is 1 no matter the move, Dijkstra's algorithm, in this application of it, guarantees completeness but not optimality. A* always guarantees both completeness and optimality. To implement these algorithms in the simplest way possible, two different heuristic functions were created. When calling the heuristic function, the program simply checks which algorithm is being used and calls the applicable method.

```
688     ############################################################
689     #### Manhattan distance - heuristic for A* algorithm
690     ############################################################
        3 usages   jo744669
691     def Astar_heuristic(self, pos):
692         return (abs(pos[0] - self.goal_pos[0]) + abs(pos[1] - self.goal_pos[1]))
```

Figure 6: The heuristic method used when the robot is finding the path using the A* algorithm. For this project, the Manhattan Distance was used as the heuristic because no diagonal moves were allowed.

```
694     ############################################################
695     #### Dijkstra heuristic - always 0 because Dijkstra only relies on true path cost
696     ############################################################
        3 usages   jo744669
697     def Dijkstra_heuristic(self, pos):
698         return 0
```

Figure 7: The heuristic method used when the robot is finding the path using Dijkstra's algorithm. Because Dijkstra only relies on path cost, this method simply returns zero.

For this project, the robot is able to begin anywhere in the hospital that is within the bounds of the grid. That is, as long as the row is less than 30 and the column is less than 38, the robot can begin there.

There are a few different variables that must be initialized for use in this project. For goal positions, the robot prioritizes which order to make deliveries to the goal locations based on the priority of the ward that the cell is in. This is where the previously defined priorities of each cell is used. Because of this prioritizing, the program utilizes a priority queue to keep track of all goal locations. Additionally, a list of locations is used so the program has access to all delivery locations when it needs to examine all of them at once. The two data structures are needed because a priority queue only allows access to the top element whereas a set, in Python, allows access to all elements of the list.

Additionally, there are 2 deque variables used to keep track of the path that the robot is taking. One of those variables keeps track of the temporary path which is the path from the starting location to the goal location. Essentially, the temporary path keeps track of the path one goal location at a time. The full path variable keeps track of the entire path the robot is taking from the start location to every goal location.

Finally, the visited list is used when redrawing the maze including every step that the robot took. This allows the cell to be a different color if it is the first time the robot is visiting that coordinate versus a repeat time that the robot is visiting that coordinate.

```
61          ### Read from input file after assigning priorities and wards and buidling cells
62          self.locations = set()
63          delivery_locations = PriorityQueue()
64          self.fullPath = deque()  # keeps track of full path to every location in order
65          self.temporaryPath = deque() #keeps track of path from start to goal to build backwards
66          self.visited = set() #for GUI updates
```

Figure 8: Different variables that were necessary in implementing the robot moving through the hospital and displaying the path that it took.

In order for the program to know the start position, all delivery locations, and the algorithm being used by the robot to find the optimal path to all delivery locations, it must get the values from an input file. This input file is read by the program and all of the different variables are interpreted and then assigned. Within reading from the input file, error checking was added to ensure that everything is defined correctly and the program will not fail. Some of this error checking includes ensuring that all locations that are entered are in bounds, and each location is a number and not words (Figure 9).

```
        for line in lines:
            if line.startswith('Delivery algorithm:'):
                delivery_algorithm = line.split(':')[1].strip()
            elif line.startswith('Start location:'):
                start_location_str = line.split(':')[1].strip()
                # Correctly parse the start location string
                if not start_location_str[0].isdigit():
                    print("Start location entered is not a number. Try again")
                    return
                start_location = tuple(map(int, start_location_str.split(',')))
            elif line.startswith('Delivery locations:'):
                delivery_locations_str = line.split(':')[1].strip()
                # Correctly parse the delivery locations string
                delivery_locations = [loc.strip() for loc in delivery_locations_str.split(',')]
                #convert the string of delivery locations to a list of tuples
                index_x = 0; index_y = 1
                counter = 0
                while counter < len(delivery_locations):
                    if not delivery_locations[index_x].isdigit():
                        print("Delivery location entered is not a number. Try again")
                        return
                    deliveries.add((tuple(map(int, (delivery_locations[index_x], delivery_locations[index_y])))))
                    counter += 2; index_x += 2; index_y += 2
        file.close()

        if delivery_algorithm is None or start_location is None or not delivery_locations:
            raise ValueError("Input file format is incorrect.")
```

Figure 9: A snippet of the code used to read the values specified in the input file.

Another feature that is in our program is the way that the robot delivers to the different specified delivery locations. When originally reading from the input file, the locations are placed into the general locations list. They are then pushed into the priority queue which prioritizes them by the priority assigned to the cell. There are multiple wards in the hospital that all have the same priority. If a location is being added to the priority queue and it has the same priority as a

different location in the list, they are sorted by first come first served order. That being said, it is possible for the locations to be organized in a way that the robot would deliver to a certain ward, leave to deliver to a different ward of the same priority, then come back to the original ward to make another delivery. This is an inefficient way for the robot to make its deliveries.

To combat this, the program checks the general locations list for a delivery in the same ward (Figure 10). If there is one, it makes that delivery next. If there is no delivery in the same ward, then the program moves on to the next goal in the priority queue. Each goal that receives a delivery is added to a list of completed goals once the delivery is successfully completed. When a goal is removed from the priority queue, the program first checks if it has been visited already. This is to ensure that the robot did not already deliver to that location because it was already in the ward.

```
127         #### Display the optimum path in the maze
128         while not delivery_locations.empty():
129             # check if there are anymore deliveries in the list in that ward
130             # get the current ward
131             current_ward = self.cells[self.agent_pos[0]][self.agent_pos[1]].ward
132             for x in self.locations:
133                 #if it is in the same ward and has not been visited already
134                 if len(self.fullPath) != 0: #not the first iteration bc already found a path
135                     current_ward = self.cells[self.agent_pos[0]][self.agent_pos[1]].ward
136                 if self.cells[x[0]][x[1]].ward == current_ward and x not in self.goals_completed:
137                     self.goal_pos = x
138                     self.find_path()
139                 # if not delivery locations not yet visited in same ward as current ward, look at priority queue
140                 else:
141                     #check the next element in the priority queue
142                     goal = delivery_locations.get()
143                     #make sure goal has not yet been visited - if it has, pop until you find oen that hasn't been
144                     while goal in self.goals_completed:
145                         goal = delivery_locations.get()
146                     self.goal_pos = goal[1]
147                     self.find_path()
148             while not delivery_locations.empty():
149                 delivery_locations.get()
150         if self.flag == 0:
151             print("SUCCESS")
```

Figure 10: These loops ensure that a robot makes all of the deliveries in its current ward before moving to a different ward. If it is getting its goal location from the priority queue, it checks that it has not already successfully made that delivery.

Another feature that was added to improve the functionality of the robot finding the path was resetting the value of the heuristic, path cost, and evaluation function for every cell that the robot looked at (Figure 11). After finding a path to one of the goals, all of these values were reset before finding the path to the next goal. This way, the robot was finding the optimal path for each individual path instead of being influenced by the route it took to get to the previous delivery location. This resetting was done in a method that was called when the robot reached its goal (Figure 12).

Additionally, once the robot reached its goal the open and closed lists that were used to find the optimal path were cleared. This allowed the robot to have no memory of where it had been previously besides what was stored in the temporary path (Figure 11).

Finally, when the robot reached a goal, the path needed to be reconstructed. This work was done in a method that was called when that goal was reached (Figure 11). In this method, the temporary path was built using a deque (Figure 13). This is because the path was rebuilt backwards based on the parent assigned to each cell. This temporary path was then emptied from the front of the deque to add to the full path and print out the temporary path.

```
740             # if current node is goal node, reconstruct the path
741             if current_pos == self.goal_pos:
742                 self.goals_completed.add(self.goal_pos)
743                 self.reset_values(open, closed)
744                 open.clear()
745                 closed.clear()
746                 self.reconstruct_path()
747                 return self.fullPath
```

Figure 11: This code snippet shows the work done once a path was found to one of the goals. The goal itself was added to the goals completed list to inform the program that the goal had been successfully found. The values were reset, open and closed lists emptied, and the path was reconstructed.

```
    ############################################################
    #### Reset g, h, and f values
    #### Before finding next path
    ############################################################
    1 usage   jo744669
    def reset_values(self, open, closed):
        while len(open):
            cost, node = open.popleft()
            self.cells[node[0]][node[1]].g = float("inf")
            self.cells[node[0]][node[1]].h = 0
            self.cells[node[0]][node[1]].f = float("inf")
        while len(closed):
            node = closed.pop()
            self.cells[node[0]][node[1]].g = float("inf")
            self.cells[node[0]][node[1]].h = 0
            self.cells[node[0]][node[1]].f = float("inf")
        return None
```

Figure 12: This is the method used to reset the heuristic, path cost, and evaluation function values once the robot found the optimal path to one cost. This way, every path to every goal that was found was optimal.

```
    ############################################################
    #### To rebuild the path and keep a running total of where agent has been
    #### This is also for the GUI part
    ############################################################
    1 usage   jo744669
    def reconstruct_path(self):
        #rebuilds the path using the temporary and full list
        current_cell = self.cells[self.goal_pos[0]][self.goal_pos[1]]
        self.temporaryPath.clear()
        while current_cell.parent:
            #build the temporary path of just this portion of movement
            self.temporaryPath.appendleft((current_cell.x, current_cell.y))
            if current_cell.x == self.agent_pos[0] and current_cell.y == self.agent_pos[1]:
                break
            current_cell = current_cell.parent

        if self.agent_pos != self.goal_pos:
            print("Path to {}".format(self.goal_pos))

        # add the temporary path to the running total once it is complete
        while self.temporaryPath:
            x, y = self.temporaryPath.popleft()
            print((x, y), end=", ")
            self.fullPath.append((x, y))

        print()

        self.agent_pos = self.goal_pos
```

Figure 13: This code snippet is the reconstruct path method that is called once the robot finds the path to a goal. It builds the temporary path, from where the robot started to only the goal it just found, using the parent attribute of the cell class that was updated when the robot was finding the path. It then prints what goal it is printing the path to and the path from where it started to the

current goal position. It also adds this temporary path to the running total that is the full path of where the robot has been.

The last main feature that was implemented was the graphics work. The image of the graphic created for the hospital layout without any path drawn can be seen in Figure 14. The first part of the graphics was coloring the image that would be displayed. Every ward was assigned a different color that closely resembled the original colors shown in Figure 1. Each cell was assigned a color based on the ward it was assigned at the beginning of the program (Figure 15). Additionally, when the maze was drawn, each cell was drawn without an outline. This allowed the image and walls within the layout to be easily understood.

Each wall was drawn depending on if the adjacent cell appeared in the current cell's adjacency list (Figure 16). If it did not appear, the program interpreted there to be a wall between the 2 cells and drew a line. To draw the line, the create line built in method was used along with calculations to determine the x and y coordinate of where the line should start and end.

Finally, a method was added to allow the user to click the down arrow key to display the path one step at a time (Figure 17). As the path was being drawn, if a cell was being visited for the first time it would display as white, and it would be added to the visited list. If a cell was already in the visited list, it would display as a gray color. This way, if the agent was visiting a cell again, the user would be able to see that in the GUI.

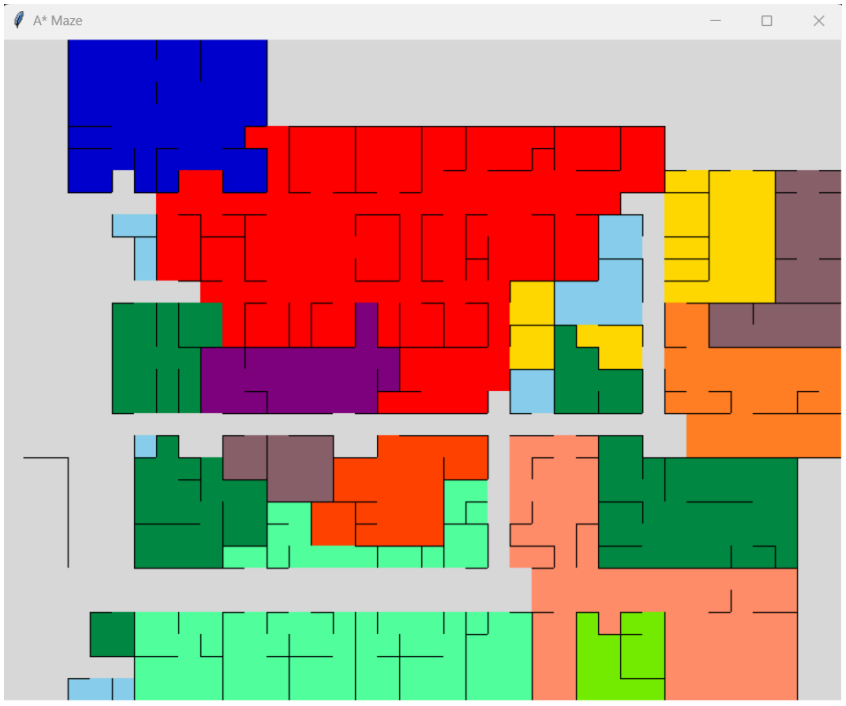


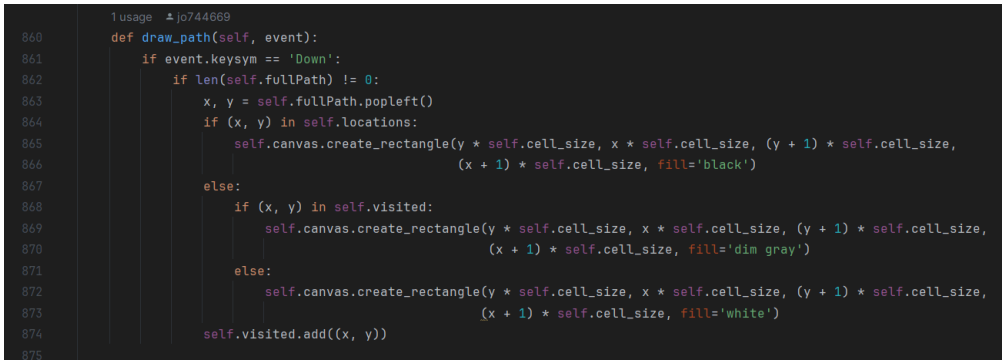
Figure 14: The hospital layout created for the GUI and displayed without any path drawn. All of the wards appear as a different color, and the walls are drawn as lines. The robot is not able to move to a location if there is a wall there.

```
812         def draw_maze(self):
813             for x in range(self.rows):
814                 for y in range(self.cols):
815                     if self.cells[x][y].ward == "General":
816                         color = 'red'
817                     elif self.cells[x][y].ward == "Maternity":
818                         color = 'medium blue'
819                     elif self.cells[x][y].ward == "Isolation":
820                         color = 'sky blue'
821                     elif self.cells[x][y].ward == "Emergency":
822                         color = 'gold'
823                     elif self.cells[x][y].ward == "Surgical":
824                         color = 'salmon1'
825                     elif self.cells[x][y].ward == "Admissions":
826                         color = 'pink4'
827                     elif self.cells[x][y].ward == "Oncology":
828                         color = 'SpringGreen4'
829                     elif self.cells[x][y].ward == "Pediatric":
830                         color = 'SeaGreen1'
831                     elif self.cells[x][y].ward == "Medical":
832                         color = 'chartreuse2'
833                     elif self.cells[x][y].ward == "Burn":
834                         color = 'purple'
835                     elif self.cells[x][y].ward == "ICU":
836                         color = 'chocolate1'
837                     elif self.cells[x][y].ward == "Hematology":
838                         color = 'orange red'
839                     else:
840                         color = 'grey85'
```

Figure 15: Within the draw maze method, each cell is assigned a color based on the ward that is assigned to it. This method is called after the method to assign the wards is called, so every cell has its ward attribute populated.

```
845         #look whether or not there should be walls
846         for x in range(self.rows):
847             for y in range(self.cols):
848                 neighbors = self.maze[(x, y)]
849 
850                 ##decide the if there should be a wall or not
851                 if x + 1 < 30:
852                     if (x + 1, y) not in neighbors:
853                         self.canvas.create_line(y * self.cell_size, (x + 1) * self.cell_size, (y + 1) * self.cell_size,
854                                                 (x + 1) * self.cell_size, fill="black")
855                 if y + 1 < 38:
856                     if (x, y + 1) not in neighbors:
857                         self.canvas.create_line((y + 1) * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
858                                                 (x + 1) * self.cell_size, fill="black")
859 
```

Figure 16: This code snippet shows the loops that draw the walls if an adjacent cell does not appear in the adjacency list. It also checks that the cell that it is looking to be connected to the current cell is within the bounds of the maze.



```
1 usage   jo744669
def draw_path(self, event):
    if event.keysym == 'Down':
        if len(self.fullPath) != 0:
            x, y = self.fullPath.popleft()
            if (x, y) in self.locations:
                self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
                                             (x + 1) * self.cell_size, fill='black')
            else:
                if (x, y) in self.visited:
                    self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
                                                 (x + 1) * self.cell_size, fill='dim gray')
                else:
                    self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
                                                 (x + 1) * self.cell_size, fill='white')
            self.visited.add((x, y))
```

Figure 17: The method created to allow the user to use the down arrow key to display the path one step at a time. If the cell that the robot is moving to has already been visited, then it is redrawn in gray instead of white.

Statement of Ranking:

Member 1: Jillian O'Connell

My teammate and I agreed that I completed 55% of the overall project. My specific tasks included:

- I completed the graphics of the maze. This included assigning each cell a different color based on what ward it is in. I researched what colors were available within the canvas and chose colors that closely resembled the original image.
- I completed the graphics to draw the walls between the cells. This included researching how to draw a line using canvas and tkinter in Python. I then calculated the x and y coordinates that the lines start and end at based on if the line was being drawn to the right or left of the cell or above or below the cell.
- I completed the method to allow the user to use the down arrow key to move through the path that the robot took one step at a time. Implementing this included researching how to bind a keypress to the root in the GUI. This also included implementing the visited list to

keep track of which cells the robot had passed over already. I implemented the difference in drawing the path in white and gray. If the cell is being visited for the first time then the cell was drawn in white. If the cell was not being visited for the first time then the cell was drawn in a gray color. For drawing the path, the border around each cell benign drawn was kept to help distinguish between the path being drawn and the original graphic.

- I implemented the priority queue for the delivery locations. This included using 2 separate lists. One list was a set, and it was populated from the input file. The other was the priority queue that was populated using the set. The priority queue would sort the delivery locations based on their assigned priorities.
- I implemented the logic for the robot to stay in the same ward and finish the deliveries there before leaving the ward. This is where the 2 separate lists became useful. First, the agent would check the basic set for any other delivery locations in the same ward as its current ward. If there was another delivery, it would assign that location as the goal and find the path. If there was not another delivery in the same ward, it would pop from the priority queue. Additionally, when removing from the priority queue, the agent would check if it had already been to that delivery location by checking the list containing the completed goals. Once all locations in the set had been checked, the remaining locations were popped from the priority queue. This is because the main loop here uses the priority queue length to know when to terminate. If a goal was taken from the set because it was in the same ward as a different goal, then it would not necessarily be removed from the priority queue. This would allow the agent to make all deliveries without fully emptying

the priority queue. Emptying the priority queue at the end prevented the agent from being caught in an infinite loop.

Member 1: Luke McGinley

My teammate and I agreed that I completed 45% of the overall project. My specific tasks included:

Implement Maze Format - I fully designed the layout of the maze floor plan using an adjacency list. This took the longest to complete as there were over 1,000 individual cells that had to be carefully examined to see if it has UP DOWN LEFT RIGHT neighbors. These neighbors determine whether or not there are walls..

Assigned Wards - Wards were assigned to individual cells, sometimes looping through sections of the maze to speed up the process of assignment. Wards included sections such as “emergency”, “general” and “burn” and more. Wards were also assigned a numerical value to determine its priority when the robot traverses the hospital floor plan. A ward with the value 5 is the highest priority such as the “emergency ward”, and wards valued 0 are the lowest priority - hallways.

Implemented A* & Djikstra algorithms - Both A* and Dijkstra algorithms were implemented in order for the program to find the best pathing for the robot. A* star uses a heuristic value with the path cost to determine the optimal path whereas Dijkstra only uses path cost to find the next optimal move.

Implemented Reading From input file - In order to feed the program information for testing, the program reads from an input .txt that contains the following information; Delivery algorithm, starting position and delivery locations. With this information being read by the program, it can now execute the final product and complete a run based on the defined information.

Implemented Error handling - Error handling is implemented here if the user's input file has incorrect/invalid testing scenarios, such as unreachable paths, checking to see if location is a number, and if all necessary information is provided in the file. Program also prints “SUCCESS” or “FAILURE” when there is a successful and complete traversal, and failure if there is not.

Testing - Throughout the project we had to test a lot of different scenarios near the end to ensure our program was running correctly, and to see how the program handled these scenarios. I tested adding to the priority queue to ensure that the robot knew correctly what priorities each delivery location had during the run. I also tested how the program would react if the robot was sent to an unreachable goal; we properly handled this by throwing an error telling the user this is not a possible delivery goal. Staying in the same ward was also tested to see how the robot would handle this scenario on queue. I tested the agent_pos to check if it was being updated after each goal. I checked to see if the program properly handled if words were placed in the input file as the locations, this would throw an error handling message. Lastly, I tested prioritizing locations to deliver to, to ensure that the robot would go to the correct prioritized ward depending on its priority value.