Devon Reing and Erin Zahner

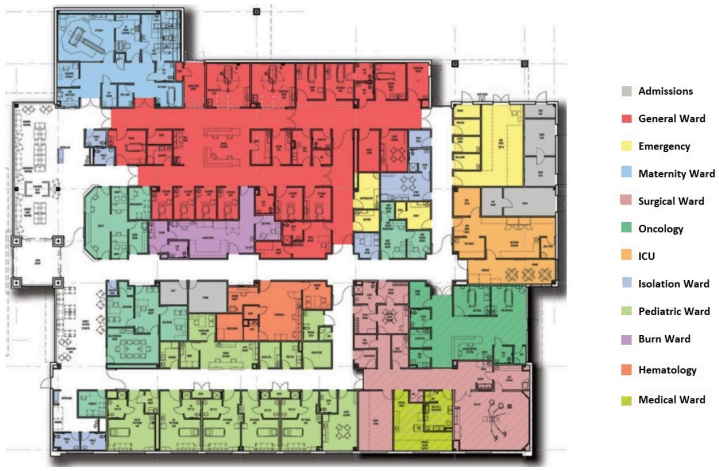
CSC 362

7 May 2024

Final Project: Robot Nurse

Goal of the Project

Robot nurses are a fairly new addition to hospital systems that utilize artificial intelligence to automate tasks such as delivering needed medications to different wards in the hospital. The specific type of artificial intelligence used by these robots are variations of the A* algorithm, which is used to find a complete and optimal path between two points. This algorithm considers obstacles within the hospital as well as walls. In this specific project, we look at A* and its variant Dijkstra's algorithm to navigate a robot nurse through the hospital floorplan shown below based on ward priorities. As shown in the diagram, there are twelve wards in this hospital, each of which is assigned a priority. A higher number indicates a higher priority and the categorizations of the wards are as follows: ICU, ER, Oncology, and Burn Wards are priority 5, Surgical and Maternity Wards are priority 4, Hematology and Pediatric Wards are priority 3, Medical and General Wards are priority 2, and Admissions and Isolation Wards are priority 1.



The robot can be programmed to start at any location in the hospital and will navigate to the next delivery location if there is a clear path based on whichever delivery location has the highest priority. The only exception to this method of choosing the next location for the robot is if there is a delivery location within the same ward that it is currently located within. In that case, the robot will finish its tasks within that ward before moving to another area of the hospital to continue carrying out its tasks. The robot continues carrying out tasks until no more tasks have been inputted for it to complete or no more clear paths exist between its current location and a delivery location.

Implementation Details:

The robot nurse receives its tasks from an input file that is passed in through a command line argument at runtime. This text file that is passed in must follow a specific format listing the algorithm (either A* or Dijkstra) with no mistakes on the first line, followed by the start state entered as a tuple on the second line, and finally a comma-separated list of destinations entered on tuples on the third line. If these conditions are not met, error checking will be utilized to give the user an error message detailing what is wrong with the input file so it can be fixed and run again correctly. Once the program is started correctly, the program will begin with initializing all the cells with its x and y coordinates in the floorplan itself, as well as a boolean value stating if it is a wall or not. This boolean value is assigned based on whether the corresponding entry in the walls matrix titled maze has a 1 at those x and y coordinates. The cell is also loaded with initial f, g, and h values which will updated when the cell is accessed in the path-finding portion of the algorithm. Finally, each cell is also initialized with ward and priority attributes using two loops. The first loop will grab each cell and find the corresponding entries from the wards matrix using the x and y coordinates. The wards matrix has the floorplan similar to the maze matrix, but with

the 0s inside the hospital walls replaced with a corresponding letter that is used to assign the wards. The wards are labeled as follows: c is ICU, e is ER, o is Oncology, b is Burn, m is maternity, s is Surgical, h is hematology, p is Pediatric, d is Medical, g is General, a is Admissions, and i is Isolation. Using the ward property that is now properly assigned to each cell, the second loop goes through and uses a series of if statements to correctly assign the priorities associated with each ward to the cells. If a cell does not have a ward assigned, it is given priority -1 so as not to interfere with the robot finding optimal paths to actual ward locations. The ward attribute is also used in the draw_maze method to assign the cell color in the graphics to correctly correlate to the floor plan attached in the image above.

Once all the cells are properly initialized, the data in the input file will be added to the goal_pos_list. This list is traversed through and added to the destinations priority queue to be utilized by the rest of the algorithm to find paths. The destinations priority queue locations are also added to a list called goals_left. Priority queues in Python favor lower priorities, but our robot needs to favor higher-priority goal states. To rectify this difference, when a goal is added to the queue from goal_pos_list, the priority associated with it is multiplied by -1. The agent_pos is also updated to be the start position indicated in the input file.

At this point, the path-finding portion of the algorithm can begin. While the destinations priority queue is not empty, the algorithm will begin by determining the next goal state. It first checks if there are any entries in the goals_left list that have the same ward attribute as the agent_pos. If there are none it then proceeds to pick the next location in the destinations queue. Before proceeding, the goal_pos from the destinations queue is also compared to the entries in the goals_completed list to ensure a destination is not completed twice if it was already completed using the same ward distinction. Once a valid goal position has been designated, the

find_path method is invoked. The find_path method is mostly untouched from the original A* implementation from earlier in the semester. The only changes we made were to add the variable success_flag and making it true if a path was found as well as then adding that goal to the success_goals list. This flag is used to print out success finding an optimal path at the end of the program running if at least one of the goal states had a clear path found. The success_goals list is traversed through at the end of the program running to highlight the goal states in a lighter blue so that they stand out in the path that is colored a dark blue. When reconstruct_path is called in the find_path method, it follows the same logic as before but is split up into two methods to allow us to be able to do a slow reveal of the path. The draw_path_with_delay method that is called in reconstruct_path uses recursion to pop a stack and redraw each square to be a new color. The slow reveal is done by using calling time.sleep(0.1) to delay the next step long enough for people to notice the change over time.

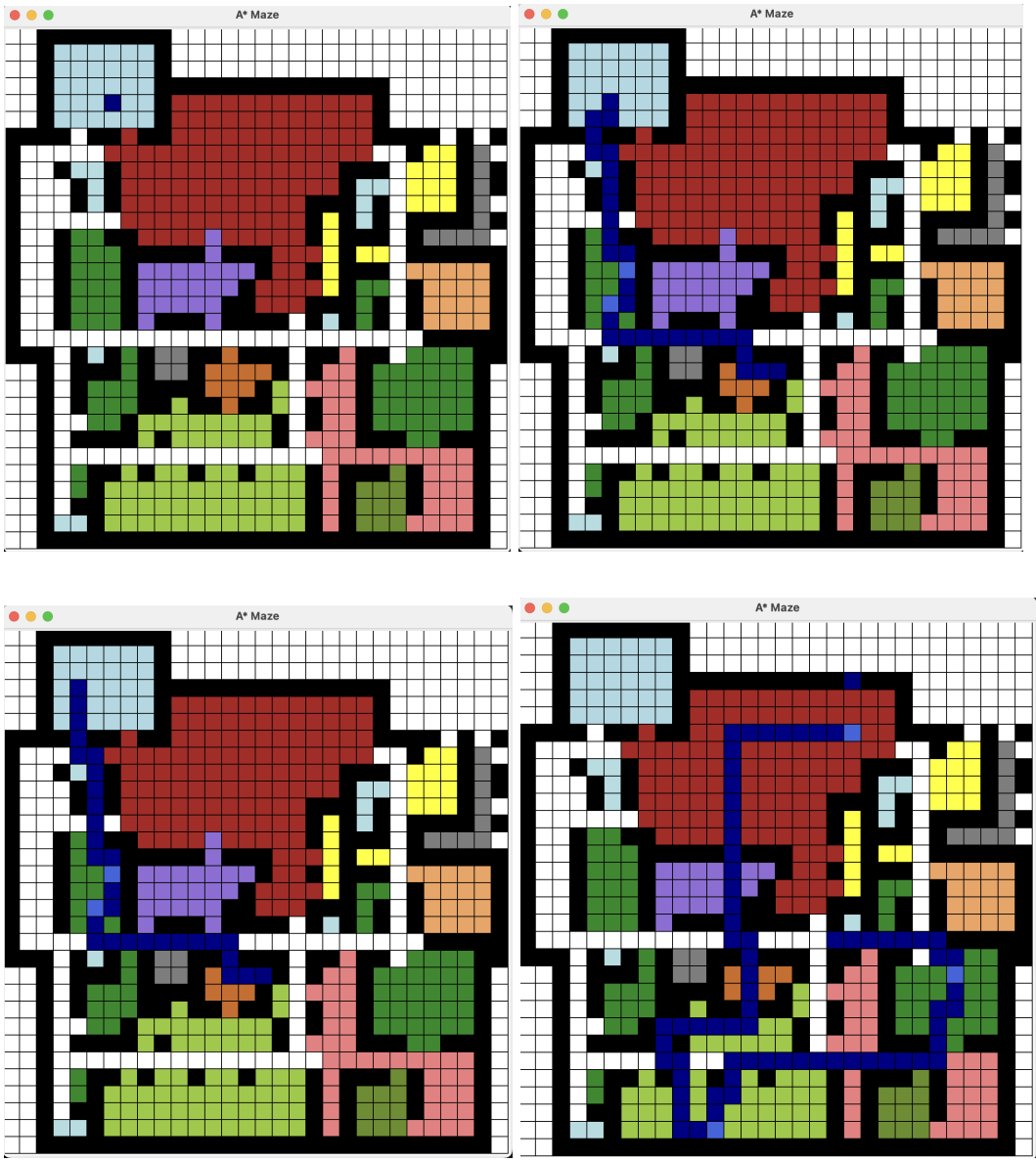
In addition to these small changes made in find_path, another small change was made to the heuristic method that is called from within this method to find the optimal path. This method now has an additional alg parameter that is used to determine which heuristic value to return to populate the cell's h value. This parameter is filled with the MazeGame object's alg attribute which is initialized based on the first line of the input file. If the algorithm in the alg attribute reads "astar", then the heuristic method returns the Manhatten distance. If the algorithm currently being used is Dijkstra's algorithm, the method then returns 0 as this algorithm is greedy and does not utilize the heuristic values. This results in this algorithm finding a complete but not optimal path.

Once the find_path method is done running for the specified goal state, the goals_left list is updated to no longer have that goal state as an entry and the goal is added to the

goals_complete list. The same_ward_flag is reset so that the next goal state can correctly assign the next goal state as well. Finally, the agent_pos is updated to be the goal_pos before the loop returns to the beginning to repeat this process until the destinations queue is empty.

Testing:

To ensure that the robot nurse was delivering correctly to all locations in the hospital, we performed a few key tests. First, we sent the robot nurse outside of the hospital and provided it with unreachable delivery locations. When given this input, the robot correctly returned “Failure: unable to find a path to goal states”. Next, we made sure that the robot nurse would deliver within the current ward before moving to other wards in the hospital. To test this, we gave the robot a starting location of (3, 5) in the maternity ward and gave it the delivery locations of (16, 5), (20, 15), (14, 6). The coordinates (16, 5) and (14, 6) are both in the oncology ward, while (20,15) is in the hematology ward. Our test showed that the nurse would deliver to both locations in oncology before moving to the location in hematology. We used both A* and Dijkstra’s for this test to show the difference in the path taken by the agent. While the difference was only slight, it showed that the nurse will move differently if it is not given any heuristic to follow. Finally, we tested what would happen if one of the delivery locations was a wall. In this test, the robot nurse simply ignored the wall location and delivered medicine to all other reachable locations.



Member 1: Devon Reing

My teammate and I agree that I handled **55%** of the overall project. My specific tasks included:

Task 1: I designed and implemented the program module that assigned the wards and priorities based on the wards matrix. This was done through two loops I added and ward and priority attributes added to the cell object. The loops first went through the wards matrix and assigned the ward cell attribute based on the entry at the corresponding location. Based on the ward

assignment the second loop then went through and assigned the appropriate priority based on the project guidelines provided.

Task 2: I wrote the code that handled picking the next goal state based on priority or the same ward. This involved adding goal_complete and goals_left to keep track of what goals were completed since the destinations priority queue would not be up to date if a goal in the same ward was selected.

Task 3: I designed the map that was used as a reference picture for what ward each cell was a part of and where the walls would be. This can be found in the connections map pdf.

Task 4: I handled updating the agent position to be the goal position of the previous goal found so that an optimal path would be found from the correct location instead of the original start location.

Task 5: I wrote the code that highlighted the found goal states in the path to be a different shade of blue so that they stuck out against the rest of the path for ease of the user in finding each goal state. This was done by looping through a list titled success_goals that was populated in the find_path method with any goals that successfully found a path from its respective start state to the goal position. If a goal was in that list, the cell was then redrawn with a separate color.

Task 6: I wrote the code to print out if the algorithm was able to successfully find a path or if it failed. This was done through a flag that was added in the find_path method which was updated to be true if at least one of the goal states successfully found a path from the start state.

Task 7: I handled the algorithm determining what heuristic should be used based on the input file by creating a variable that would be checked in an if statement in the heuristic method. This would return the Manhattan Distance if the variable indicated the A* method is to be used or a 0 if Dijkstra's Algorithm was being used.

Task 8: I handled debugging the input file with how it loaded the destinations and start position as well as a bug that would freeze the program if only one goal state was given or if certain final goal states were added to the file.

Task 9: I handled testing before we added the input file to make sure the algorithm was working before integrating the input file through a command line argument. This was done through hard coding the values such as the algorithm to be used, the goals inputted in the destinations queue, and the start state. This made the transition to incorporating the input file much smoother as we knew that everything should work as long as the input file code is implemented correctly.

Member 2:Erin Zahner

My teammate and I agree that I handled **45%** of the overall project. My specific tasks included:

Task 1: I designed and implemented the program module that read from the input file (parse_input_file). This involved being able to read and store the information properly, as well as handle error checking to ensure the input file was formatted correctly. Screenshots of the code can be seen below. First, I converted the whole file to lowercase to avoid having to worry about upper or lowercase words. Then, I wrote four if statements to check for formatting errors within the file to ensure that the file was three lines long, had a delivery algorithm of either A* or Dijkstra specified, and had lines for start location and goal locations. After ensuring the general format was correct, I performed more error checking as I handled saving the values. To grab the algorithm, I simply split the line after the colon and saved what was written after. Saving the start and goal locations was a bit more complicated. To ensure they were in the correct format, I utilized regular expressions that guaranteed they were in the form ‘(x,y)’. If they did not match

that format, a value error was returned and the program would not continue. This ensured the program could read and save the coordinates as a tuple for the start position and a list of tuples for the goal positions.

Task 2: I dedicated time to creating the two matrices we used to represent the hospital floor plan. I first created the ward matrix that consisted of only hallways, wards, and outer walls (ignoring any inner walls) using a mix of 0s, 1s, and letters to represent the wards. Then, to implement the inner walls, I created a matrix of 0s and 1s. As Devon said above, she then wrote the program that connected these two matrices together.

Task 3: I added styling to our graphics, coloring each ward to be as close to the original floor plan as possible. This helped the user know exactly what ward the agent was in and made the program more pleasing to look at. I used Matplotlib's website to find the shades that best fit as an added touch and attention to detail. I then colored the walls to be black to stand out against the wards and hallways and made the path of the agent dark blue so it would be clear at any part of the hospital. (https://matplotlib.org/stable/gallery/color/named_colors.html)

Task 4: I implemented a slow reveal of the path of the agent in the draw_path_with_delay method. This method operates recursively to draw each step of the path with a small delay between each step. To do this, it first checks if there are still cells in the path stack (self.path_stack). If there are, it pops the top cell from the stack, draws it on the canvas as a green rectangle, updates the GUI to show the drawn path, and adds a small delay using time.sleep(0.1), redraws the cell with its original color and then calls itself recursively to draw the next step of the path. This method is then called in the reconstruct_path method to construct it from the goal state back to the start state.

Task 5: I created test input files that demonstrated the capabilities of our agent. These files showed that our agent can deliver within the same ward before it goes to the next priority using both A* and Dijkstra, knows when not to move if a location is a wall, and returns failure if it is in an unreachable state, such as outside the walls of the hospital.

Input File

```python
def parse_input_file(self, file_path):
    with open(file_path, 'r') as file:
        # Read from file, turn everything to lower case
        lines = [line.strip().lower() for line in file.readlines()]

        # Check if the file is 3 lines long
        if len(lines) < 3:
            raise ValueError("Error: Input file length must be 3 lines")

        # Check if the first line contains the delivery algorithm
        if not (lines[0].startswith("delivery algorithm: astar") or
                lines[0].startswith("delivery algorithm: dijkstra")):
            raise ValueError("Error: First line should specify the delivery algorithm")

        # Check if the second line contains the start location
        if not lines[1].startswith("start location:"):
            raise ValueError("Error: Second line should specify the start location")

        # Check if the third line contains delivery locations
        if not lines[2].startswith("delivery locations:"):
            raise ValueError("Error: Third line should specify the delivery locations")
```

```python
# Get algorithm
self.alg = lines[0].split(":")[1].strip()

# Get start position using regular expression
start_match = re.match(r'start location:\s*\((\d+),\s*(\d+)\)', lines[1])
if not start_match:
    raise ValueError("Start location format is incorrect")
self.start_pos = tuple(map(int, start_match.groups()))

# Get delivery locations using regular expression
delivery_match = re.findall(r'\((\d+),\s*(\d+)\)', lines[2])
if not delivery_match:
    raise ValueError("Delivery locations format is incorrect")
self.goal_pos = [(int(x), int(y)) for x, y in delivery_match]

return self.alg, self.start_pos, self.goal_pos
```

Ward Matrix

```python
floor_plan = (
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 'm', 'm', 'm', 'm', 'm', 'm', 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 'm', 'm', 'm', 'm', 'm', 'm', 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 'm', 'm', 'm', 'm', 'm', 'm', 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 'm', 'm', 'm', 'm', 'm', 'm', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 'm', 'm', 'm', 'm', 'm', 'm', 'm', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 1, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 'm', 0, 'm', 'm', 'g', 'm', 'm', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 0, 0, 'e', 'e', 'e', 'a', 'a', 1],
    [1, 0, 0, 0, 'i', 'i', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'i', 'i', 0, 'e', 'e', 'e', 'a', 'a', 1],
    [1, 0, 0, 0, 0, 'i', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'i', 'i', 0, 'e', 'e', 'e', 'a', 'a', 1],
    [1, 0, 0, 0, 0, 'i', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'i', 'i', 0, 'e', 'e', 'e', 'a', 'a', 1],
    [1, 0, 0, 0, 0, 0, 0, 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'e', 'i', 'i', 'i', 0, 'e', 'e', 'e', 'a', 'a', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'g', 'g', 'g', 'g', 'b', 'g', 'g', 'g', 'g', 'g', 'g', 'e', 'i', 'i', 'i', 0, 'c', 'a', 'a', 'a', 'a', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'g', 'g', 'g', 'g', 'b', 'g', 'g', 'g', 'g', 'g', 'g', 'e', 'o', 'e', 'e', 0, 'c', 'a', 'a', 'a', 'a', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'g', 'g', 'g', 'g', 'e', 'o', 'e', 'e', 0, 'c', 'c', 'c', 'c', 'c', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'b', 'b', 'b', 'b', 'b', 'b', 'g', 'g', 'g', 'g', 'g', 'e', 'o', 'o', 'o', 0, 'c', 'c', 'c', 'c', 'c', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'b', 'b', 'b', 'b', 'b', 'g', 'g', 'g', 'g', 'g', 'g', 'i', 'o', 'o', 'o', 0, 'c', 'c', 'c', 'c', 'c', 1],
    [1, 0, 0, 0, 'o', 'o', 'o', 'o', 'b', 'b', 'b', 'b', 'b', 'g', 'g', 'g', 'g', 0, 0, 'i', 'o', 'o', 'o', 0, 'c', 'c', 'c', 'c', 'c', 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'c', 'c', 'c', 'c', 1],
    [1, 1, 1, 0, 0, 'i', 'o', 'o', 'a', 'a', 'a', 'a', 'h', 'h', 'h', 'h', 'h', 0, 's', 's', 's', 'o', 'o', 0, 'o', 'o', 'o', 'o', 1, 1],
    [0, 0, 1, 0, 0, 'o', 'o', 'o', 'a', 'a', 'a', 'a', 'h', 'h', 'h', 'h', 'h', 0, 's', 's', 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 1, 0],
    [0, 0, 1, 0, 0, 'o', 'o', 'o', 'o', 'o', 'p', 'h', 'h', 'h', 'h', 'h', 'p', 0, 's', 's', 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 1, 0],
    [0, 0, 1, 0, 0, 'o', 'o', 'o', 'o', 'o', 'p', 'h', 'h', 'h', 'h', 'h', 'p', 0, 's', 's', 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 1, 0],
    [0, 0, 1, 0, 0, 'o', 'o', 'o', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 0, 's', 's', 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 1, 0],
    [0, 0, 1, 0, 0, 'o', 'o', 'o', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 0, 's', 's', 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 1, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 's', 's', 's', 's', 's', 's', 's', 's', 's', 1, 0],
    [0, 0, 1, 0, 'o', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 's', 'd', 's', 'd', 'd', 's', 's', 's', 's', 1, 0],
    [0, 0, 1, 0, 'o', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 's', 'd', 'd', 'd', 'd', 's', 's', 's', 's', 1, 0],
    [0, 0, 1, 0, 0, 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 's', 'd', 'd', 'd', 'd', 's', 's', 's', 's', 1, 0],
    [0, 0, 1, 'i', 'i', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 's', 'd', 'd', 'd', 'd', 's', 's', 's', 's', 1, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0])
```

Draw Path with Delay

```python
def draw_path_with_delay(self):
    if self.path_stack:
        current_cell = self.path_stack.pop()
        x, y = current_cell.x, current_cell.y
        self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
                                     (x + 1) * self.cell_size, fill='green')
        self.root.update()  # Update the GUI to show the drawn path
        time.sleep(0.1)  # Add a delay between steps

        # Redraw cell with updated g() and h() values
        color = 'darkblue'
        self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size,
                                     (x + 1) * self.cell_size, fill=color)
        self.draw_path_with_delay()  # Recursively draw the next step

1 usage  erinzahner
def reconstruct_path(self):
    current_cell = self.cells[self.goal_pos[0]][self.goal_pos[1]]
    while current_cell.parent:
        self.path_stack.append(current_cell)
        current_cell = current_cell.parent

    self.draw_path_with_delay()  # Start drawing the path with a delay
```