Authors: A.J. Litchfield and Brittany Klose
AI CSC 362
Submitted 5/7/24
Assignment 8: Final Project

<u>Statement of Ranking</u>

Member 1: A.J. Litchfield
My teammate and I agreed that I handled 50% of the overall project. My specific tasks included:

Task 1: I researched how to accept multiple inputs as a list, and how to store that information in the queue with the appropriate priority and coordinates associated with the respective wards.
Task 2: I designed and implemented the program module that accepted input from the user to collect the start state and the destinations. This part of the program also has error checking to prevent illegal start states and incorrectly/invalid entered destinations.
Task 3: I also wrote the portion of code in the beginning that allows the user to pick which algorithm they want to use to find the path through the entered destinations.
Task 4: I helped work on the AStar algorithm.

Member 2: Brittany Klose
My teammate and I agree that I handled 50% of the overall project. My specific tasks included:

Task 1: I designed and implemented the original maze structure and design to most accurately represent the floorplan of the hospital.
Task 2: I worked on the AStar algorithm to find multiple destinations. Including setting up the initial while loop in the def_init_ to reset f, g, h values and continue calling the find_path and reconstruct_path.
Task 3: I worked on the Dijkstra algorithm to find paths solely by distance and without use of a heuristic.
Task 4: I updated the reconstruct_path method to draw multiple paths. This included updating the GUI to change the colors of the goal and start cells, and researching into implementing the root.after method to show the drawing of a path one cell at a time.

<u>Report</u>

<u>Overview</u>
For the final project, we were asked to design a program for a robot nurse that would allow it to navigate the floorplan at a hospital. To explain, this program allows the robot to take in multiple destinations- or wards, queue those locations based on the priority associated with the level of care needed, and distance from the current position. The program then finds each path from start or current position to the goal using either the A* algorithm or Djikstra's algorithm.

<u>The Maze</u>

A crucial part of this program was having an accurate representation of the floorplan for the agent to traverse. This included using the provided photo tool to superimpose a grid over the original floor plan image, and then using the provided converter to convert the image into 0s and 1s. Our end maze size was 50x50. There are multiple tools available to then easily adjust the maze. I personally was unable to use the Text-Pad Editor on my Mac so I used Visual Studio Code to first replace the spaces with ", " and then I accessed the command palette which let me add brackets to all the end lines at once. From there we simply modified the walls and colors to best match the desired floor plan.

The original maze from the converter program was constructed of mostly 1 for walls. It made every colored cell whether black or other colored a wall and only the white spaces were written as 0s or 'open'. Thus, we had to update many 1s inside the maze to become open spaces inside each ward. I also made the surrounding spaces all walls so if the user called the program to begin outside the floor plan where there was no entrance inside, it would be an illegal start space, from which AJ created an error message for. From there I updated the GUI in the draw_maze function to add specific colors to represent certain wards. Cell numbers ranged from 2 to 13. Inside the main loop of the maze game the line for " self.cells = [[Cell(x, y, maze[x][y] == 1) for y in range(self.cols)] for x in range(self.rows)]" establishes that anything that's not a 1 will be considered open space, letting the path travel over ward areas even if they weren't labeled 0. While each color displayed an entire ward, or section of a particular ward, we programmed only one specific cell to be the actual goal location of each ward.

User Input

We wanted our program to resemble something that would be used in the real-world as closely as possible. With that in mind, we opted to collect the input for our program directly from the user via text inputs. The problem with that, however, is the possibility–and likelihood–of human error. We made sure to incorporate a few levels of error checking to ensure that our program would re-prompt the user whenever it received incorrect or invalid inputs. We executed this with multiple 'try and except blocks' which allowed the program to collect the input, check if it was valid, and either move on if it was, or re-prompt if it was not.

With this structure in place, we set up code that asked for the user to input a start state, provided as coordinates on the grid. Next, we gathered the wards for our "robot" to hit. The user enters this in the form of a comma-separated list and we then cycle through that list, check the name of the provided ward at each index in the list, and then add the corresponding priority and goal location (which we track in a dictionary of wards and goals) to our priority queue in the correct order. Finally, we ask the user to choose which path the agent will take between destinations. They can choose either A* or Djikstra's. Based on the user's input, a decision tree with an if-statement is set up to call a function that finds the path using the provided algorithm.

A* Algorithm

The A* algorithm is an informed search algorithm that uses a heuristic to make an educated guess about the best next move to make. The evaluation function is: $f=g(n)+h(n)$, where h is the heuristic. This was calculated via the Manhattan distance function. This distance function was selected because it is admissible and our agent is prohibited from making diagonal moves, it can only move up, down, and side to side. The A* find_path we programmed was essentially the same as the example given for finding a single destination. It creates a priority queue of paths to visit and the cost of that path. First the initial starting position with a cost of 0 is added. While the queue isn't empty it continues to evaluate each path. If the current position is the goal it stops and will reconstruct the path. If it is not the goal state, we continue to cycle through the available/legal moves and update the evaluation function f(n) = g(n) + h(n) where g(n) is our true path cost and h(n) is the heuristic or predicted distance from the current state to the goal state. The move that results in the most favorable evaluation function value is chosen and the agent follows that path accordingly. The cost of moving to one cell is 1. We added in weights of 2 and 3 for g and h respectively to add emphasis on the actual cost versus the predicted to give a slightly more optimal path. Additionally, we wanted this function to repeat for each new goal. So we created a priority queue in the main maze_game loop that included the list of goals, and then a while loop was made. While the priority queue of tasks was not empty it would reset the values for g, h, and f, call the find_path method and then the current goal position would be updated to become the new start position for the next goal.

Dijkstra's Algorithm

Djikstra's algorithm serves as a useful comparison in this scenario because it is an uninformed greedy search algorithm. This can be seen in our function definition for calling Djikstra's on the queue of destinations entered. The set up is largely the same as it was for A*, but without the presence of the heuristic value. To elaborate, another priority queue is set up to keep track of the visited cells and their costs and if the current cell is the goal it will break. For new positions only g and f are updated where f(n) = g(n), so when called it finds a path to the goal, but not necessarily the optimal path. Whereas A* will always be complete and optimal because it utilizes both g() and h() values. When both Dijkstra and A* algorithms were run, we did not see a huge variation in the path chosen. This was likely because we labeled only one cell in each ward as the goal. If we had instead marked the entire ward as the goal there likely would have been more distinct paths between the two algorithms. However, the small changes we did see still highlight the higher advantages of using A* in more complex problems to guarantee an optimal path every time.

Path Reconstruction

For the path reconstruction algorithm I assigned the color brown to mark the initial start location as well as the start for each next path. I also coded for the final end goal to be colored gold. Thus, at the end of the program every goal should be colored brown along with the original start spot and the final goal will be shown in gold. Each path was drawn over the cells in blue. I researched the root.after method to delay the draw time of the actual path, so viewers could see the path forming in real time. Unfortunately, I was unable to get it working. Instead of a simple delay in drawing the path it caused the whole program to delay including the initial construction of the overall maze. To rectify this problem I searched for alternative methods such as root.update and importing time. While this method did work in drawing the path in real time, it ended up drawing the path backwards from goal to start for each path.

Termination Conditions

To inform the user that our program has run and has either found path's to the entered wards, we implemented termination conditions. In addition to informing the user where it is starting and what ward is the next goal as it cycles through the queue, when each path is found the while loop for the queue of visited cells terminates and the program prints "ward found!". Then when the priority queue of destinations is exhausted, the while loop holding the priority queue of tasks is terminated and the programi prints "All tasks completed successfully!"

If there is no path found to a ward due to an obstacle/wall then the program will say "Failure, no path found to ward" after checking to make sure the queue of possible moves in either the A* or Djisktra's functions are empty and checking one last time to see if the current state is the goal state.

Conclusions

This project was an excellent culmination of all the AI topics that we covered this semester and provides us with the opportunity to implement them in a real-world scenario. We experienced some challenges along the way with drawing the maze and updating the program to find multiple goals, but for the most part it was a very enjoyable experience and a cool project. If we had more time we would have liked to add in more extra functionalities like adding in more tasks once the robot was finished, and popping up obstacles. Overall, a great project that clearly demonstrates the advantages of the A* algorithm as well as the benefits that AI provides for creating solutions to solve complex real world problems.