**Artificial Intelligence – CSC 362**

**Final Project Report – Team 4 (Gabi & Sarah)**

Report Contents:

* Description of project
* Project organization
* Algorithm components
* Statement of ranking
* Potential improvements for later use

Description of Project:

The main objective of this project is to create a technological agent that operates within a robot on a hospital floor. This robot carries the responsibility of delivering medical supplies, medications, or various items that would be needed on hospital units. The target hospital floor has a layout of the following:



The colored units also each possess a certain priority level that determines their place in the queue for delivery. The top priority (priority 5) includes the Intensive Care Unit, Emergency Room, Oncology, and Burn Ward, while the lowest priority (priority 1) contains the admissions unit and the isolation ward. These priorities are associated with the wards, which are read by the program after the submission of an input file. The input file, submitted by a staff member at the hospital, specifies which algorithm to use, where the robot is to start, as well as all delivery locations that are to be reached. Based on these input parameters, the agent will run the appropriate algorithm to find the optimum path to reach the indicated locations.

Project Organization:

* Color map

color\_map = {

0: 'white',

1: 'grey',

2: 'red',

3: 'yellow',

4: 'lightskyblue',

5: 'pink',

6: 'darkgreen',

7: 'orange',

8: 'powderblue',

9: 'darkseagreen',

10: 'purple',

11: 'coral',

12: 'olive',

14: 'black',

}

This dictionary initializes the key as a number from 0-15 that is associated with a certain color. Numbers zero through twelve each represent a certain ward located on the hospital floor; for example – the maternity ward is indicated with lightskyblue and the medical ward with olive. It is important to note that 3 extra numbers are utilized in this color map but not associated with a ward. White is indicative of hospital hallways while black represents walls (and thus a coordinate that the agent is not allowed to move through). This data structure allows for assignment of maze coordinates to a certain ward, based on their color. The following block of code performs this functionality through traversing through each coordinate when drawing the maze:

for x in range(self.rows):

for y in range(self.cols):

color = color\_map[self.maze[x][y]]

if self.maze[x][y] == 1:

self.cells[x][y].ward = "Admissions"

elif self.maze[x][y] == 2:

self.cells[x][y].ward = "General Ward"

elif self.maze[x][y] == 3:

self.cells[x][y].ward = "Emergency"

elif self.maze[x][y] == 4:

self.cells[x][y].ward = "Maternity"

elif self.maze[x][y] == 5:

self.cells[x][y].ward = "Surgical Ward"

elif self.maze[x][y] == 6:

self.cells[x][y].ward = "Oncology"

elif self.maze[x][y] == 7:

self.cells[x][y].ward = "ICU"

elif self.maze[x][y] == 8:

self.cells[x][y].ward = "Isolation Ward"

elif self.maze[x][y] == 9:

self.cells[x][y].ward = "Pediatric Ward"

elif self.maze[x][y] == 10:

self.cells[x][y].ward = "Burn Ward"

elif self.maze[x][y] == 11:

self.cells[x][y].ward = "Hematology"

elif self.maze[x][y] == 12:

self.cells[x][y].ward = "Medical Ward"

elif self.maze[x][y] == 0:

self.cells[x][y].ward = "Hallway"

elif self.maze[x][y] == 14:

self.cells[x][y].ward = "Wall"

* Matrix

With this color coordination in mind, a 40x40 matrix was created to resemble the hospital’s floor plan with extreme detail. The matrix includes values ranging from zero to twelve, as well as fourteen. These values then are read and traversed through to assign wards to every coordinate.

A portion of the 40x40 matrix is included below:

[0, 0, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 14, 0, 0],

[0, 0, 14, 0, 0, 14, 0, 14, 14, 14, 14, 0, 0, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 14, 14, 14, 14, 14, 0, 0],

[0, 0, 14, 0, 0, 6, 6, 14, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 14, 4, 4, 14, 14, 5, 12, 14, 4, 4, 4, 4, 4, 14, 0, 0],

[0, 0, 14, 0, 0, 6, 6, 14, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 14, 4, 4, 4, 14, 12, 12, 14, 4, 4, 4, 4, 4, 14, 0, 0],

[0, 0, 14, 14, 0, 0, 0, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 14, 4, 4, 4, 14, 12, 12, 14, 4, 4, 4, 4, 4, 14, 0, 0],

[0, 0, 14, 8, 8, 8, 8, 14, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 14, 4, 4, 4, 14, 12, 12, 14, 4, 4, 4, 4, 4, 14, 0, 0],

[0, 0, 14, 8, 8, 8, 8, 14, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 14, 4, 4, 4, 14, 12, 12, 14, 4, 4, 4, 4, 4, 14, 0, 0],

[0, 0, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 0, 0],

* Priority sorting

def search\_and\_sort(self, input\_list):

result = []

locations = []

sorted\_result = []

for priority, positions in self.priorities\_all.items():

for position in self.delivery\_locations:

if position in input\_list:

print(position[0], position[1], self.cells[position[0]][position[1]].priority)

cell\_priority = self.cells[position[0]][position[1]].priority

result.append((self.cells[position[0]][position[1]].priority, position))

break

sorted\_result = sorted(result, key=lambda x: x[0], reverse=True)

# Sort by priority, higher to lower

for item in sorted\_result:

locations.append(item[1])

return locations

The search\_and\_sort() method takes into consideration the priorities of the delivery locations as seen in the input file. Through traversal, the resulting delivery locations and their priorities are added into the result[] list. The blue highlighted line of code specifies how the program sorts the list of target destinations based on their priority - priority 5 units go first, followed by subsequent units of descending priority levels. The method returns the locations in a sorted list that is inputted to the two algorithms to find the optimal paths.

Algorithm Components

* A-Star Search Algorithm

if self.algorithms[0] == "A\*":

print(self.delivery\_locations)

local\_delivery\_locations = []

local\_delivery\_locations = self.search\_and\_sort(self.delivery\_locations)

print(local\_delivery\_locations)

starting = self.agent\_pos

for pos in local\_delivery\_locations:

print("Start pos:", starting)

self.run\_astar(starting, pos)

starting = pos

print("End pos:", pos)

This block of code specifies that the input user chose to use the AStar algorithm to find the paths. An empty list is initialized, which eventually will contain the returned result from the search and sort method. Since the locations are already sorted, the locations' paths will be found in that order. This block of code will run the a star algorithm for each of these locations as well as update the starting location after finding the path to that location.

def run\_astar(self, start, end):

open\_set = PriorityQueue()

#### Add the start state to the queue

open\_set.put((0, start))

#### Continue exploring until the queue is exhausted

while not open\_set.empty():

print("Queue size", open\_set.qsize())

if open\_set.qsize() > 200:

print("Queue is too long")

self.run\_astar(start, end)

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 == end:

print("Path found!")

open\_set.put((0, end))

self.reconstruct\_path(end)

break

else:

print("No path found.")

#### Agent goes E, W, N, and S, whenever possible

for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

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

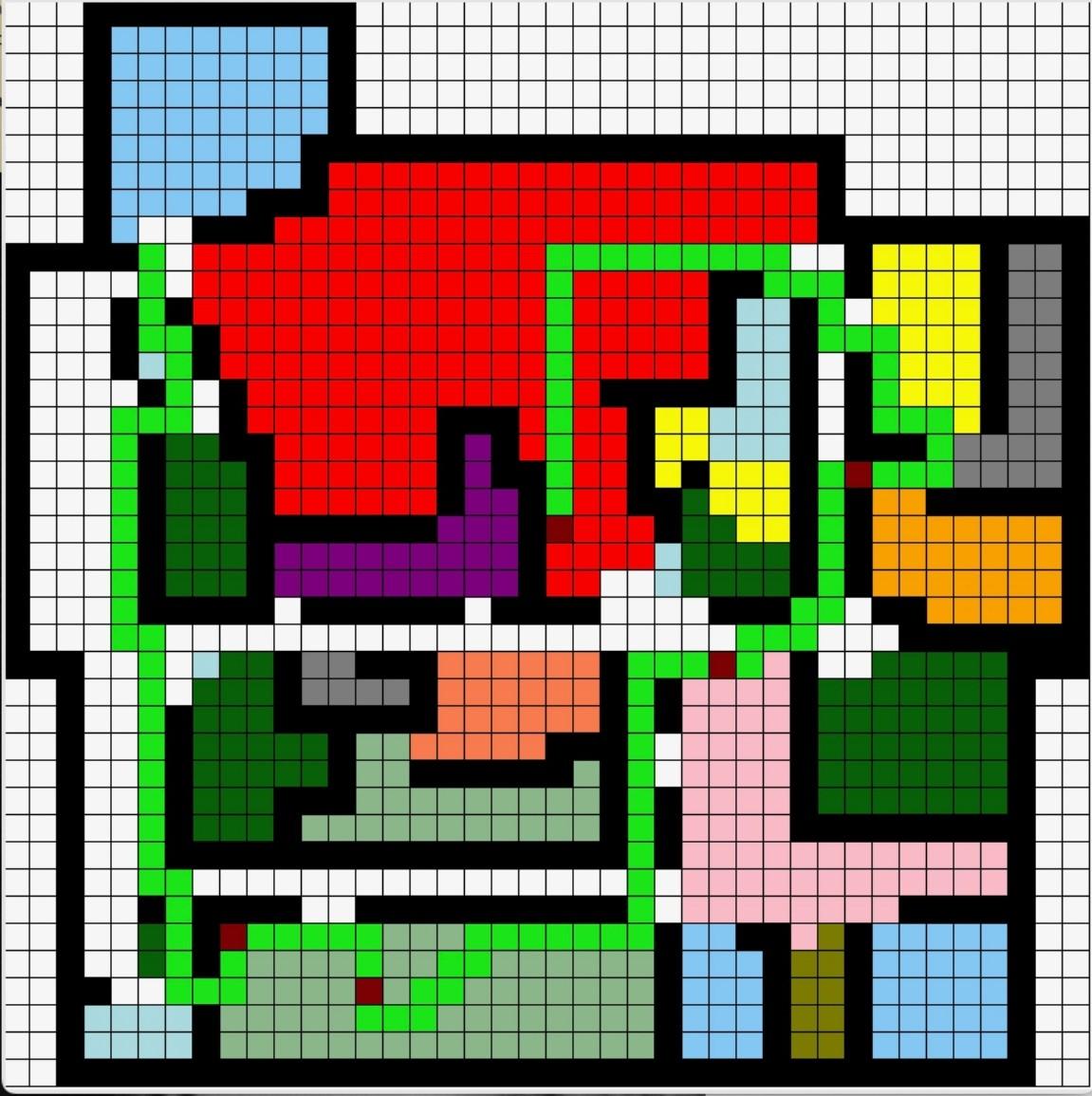
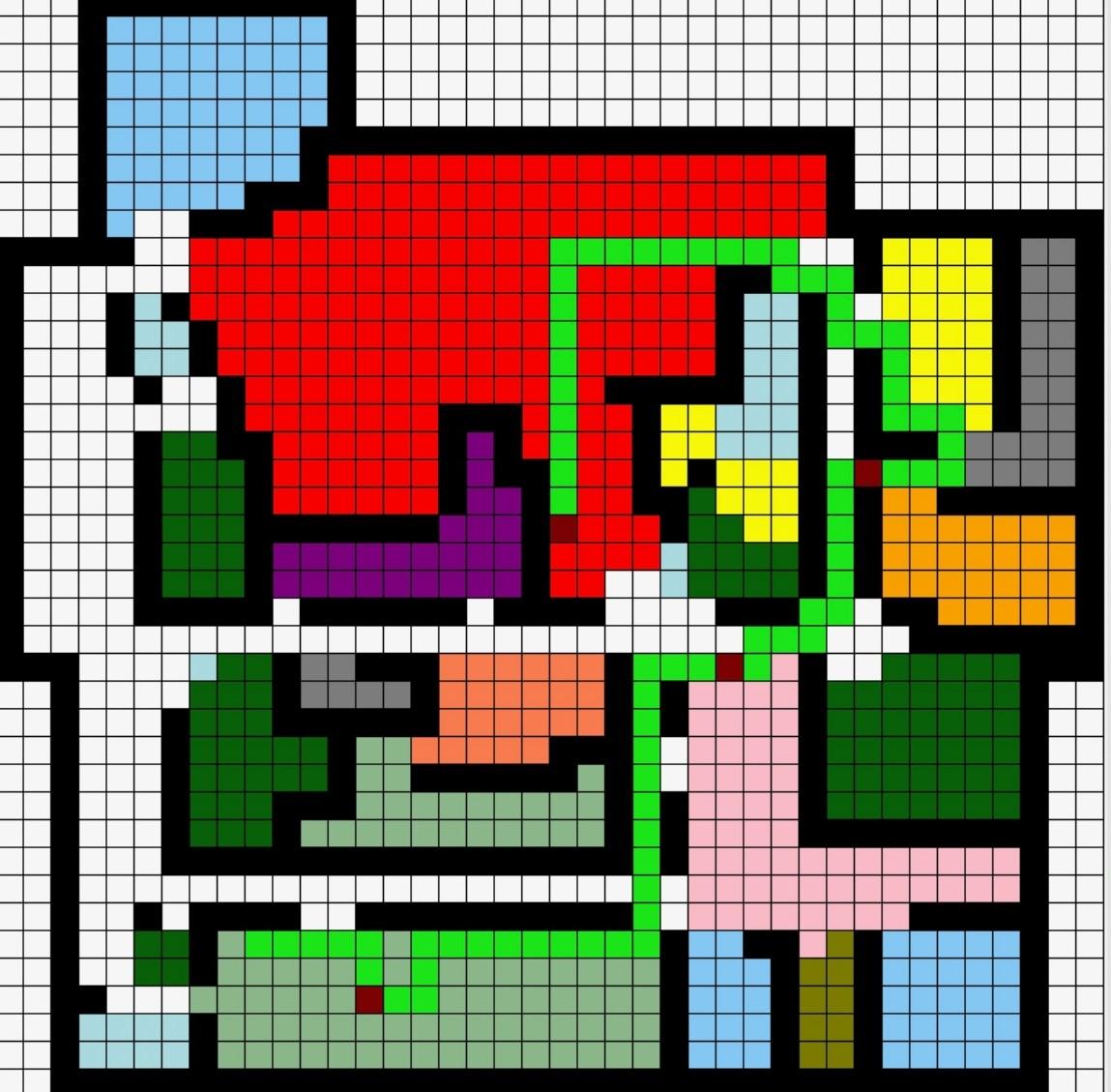
self.cells[new\_pos[0]][new\_pos[1]].parent = current\_cell

#### Add the new cell to the priority queue

open\_set.put((self.cells[new\_pos[0]][new\_pos[1]].f, new\_pos))

The a star algorithm will find paths to delivery locations based on the step cost as well as heuristics. The program uses the Manhattan distance for the heuristic calculation, as well as assigning a cost of one unit per step, to form the evaluation function value (highlighted in blue). Utilizing a star guarantees the finding of an optimal solution.

Some AStar Path Results:



* Dijkstra's Search Algorithm

elif self.algorithms[0] == "Dijkstras":

print("Running Dijkstras")

starting = self.agent\_pos

print(self.delivery\_locations)

local\_delivery\_locations = []

local\_delivery\_locations = self.search\_and\_sort(self.delivery\_locations)

print(local\_delivery\_locations)

for pos in self.delivery\_locations:

print("IN MAZE GAME IF STATEMENT")

print(f"Goal location: {pos}")

self.run\_dij(starting, pos)

starting = pos

if not self.path\_found:

print("No path able to be found. Please check input coordinates.")

messagebox.showerror("Error","No path found in the maze.")

print("BACK TO MAZE GAME IF STATEMENT = End pos: ", pos)

This block of code specifies that the input user chose to use Dijkstra's algorithm to find the paths. An empty list is initialized, which eventually will contain the returned result from the search and sort method. Since the locations are already sorted, the locations' paths will be found in that order. This block of code will run the algorithm for each of these locations as well as update the starting location after finding the path to that location.

def run\_dij(self, start, end):

open\_set = PriorityQueue()

#### Add the start state to the queue

open\_set.put((0, start))

self.path\_found = False

#### 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 == end:

print("Path found!")

self.reconstruct\_path(end)

self.path\_found = True

break

#### Agent goes E, W, N, and S, whenever possible

for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

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 evaluation function for the cell n: f(n) = g(n)

self.cells[new\_pos[0]][new\_pos[1]].f = new\_g

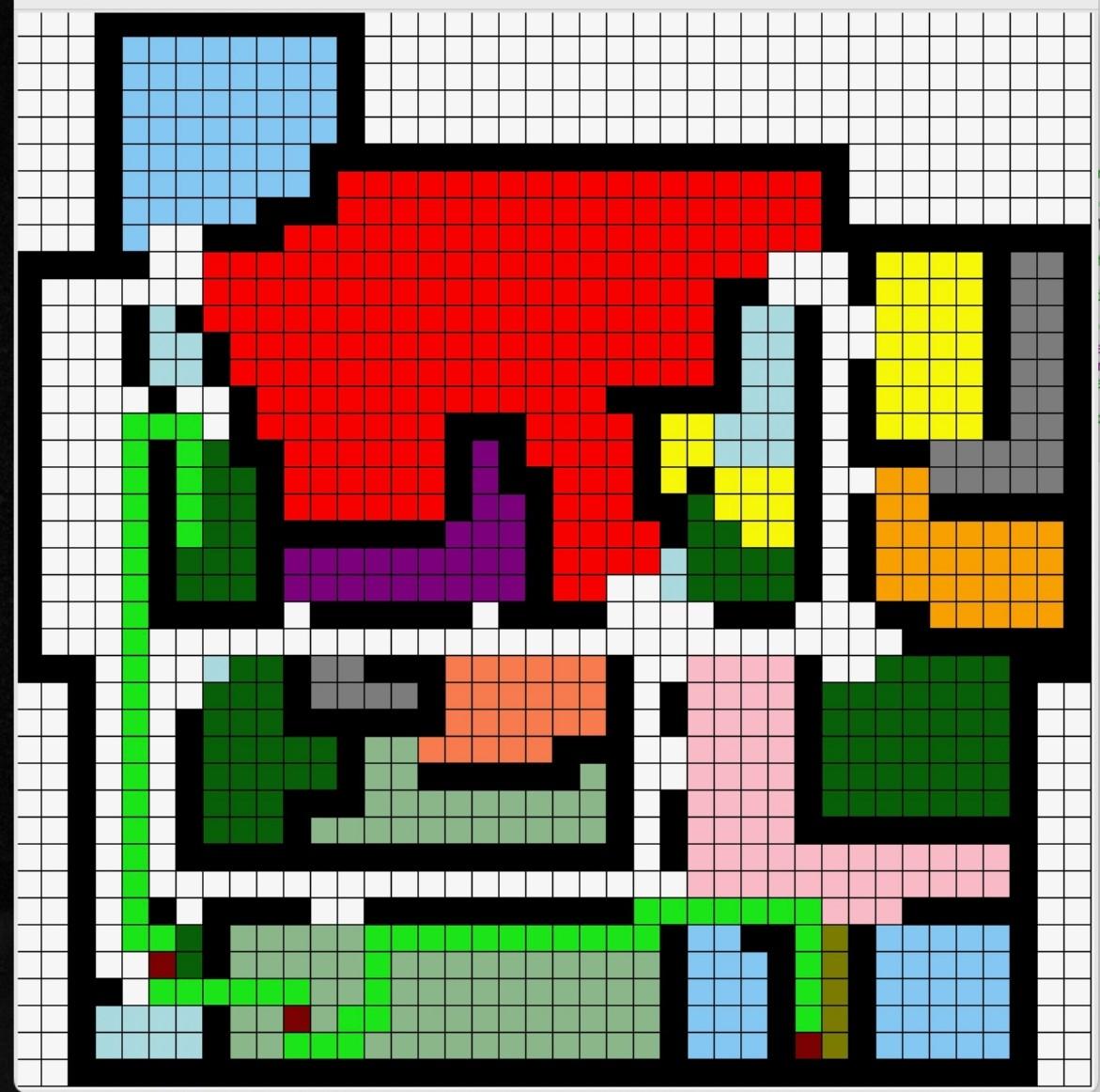
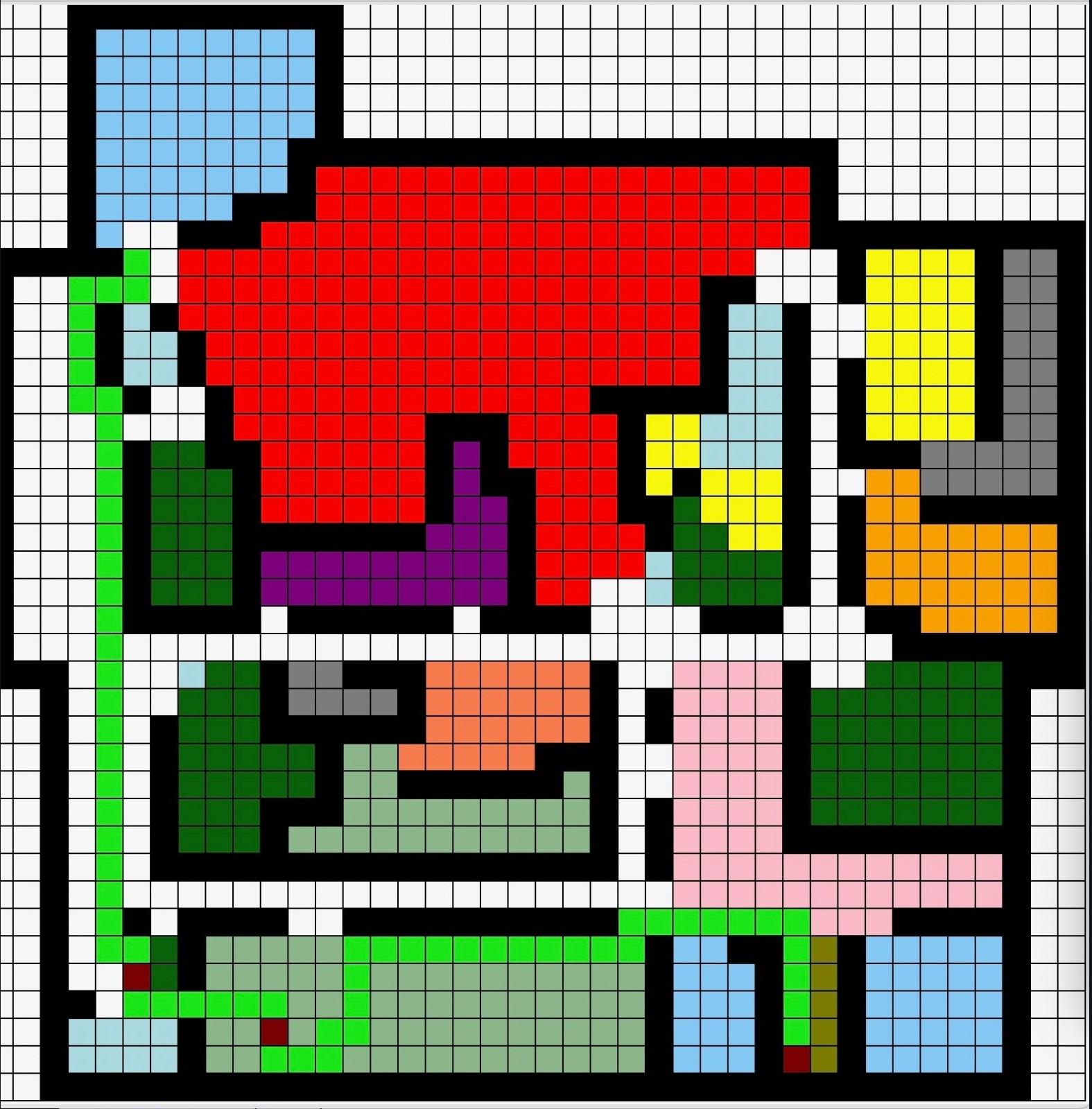
self.cells[new\_pos[0]][new\_pos[1]].parent = current\_cell

#### Add the new cell to the priority queue

open\_set.put((self.cells[new\_pos[0]][new\_pos[1]].f, new\_pos))

Unlike a star, Dijkstra's algorithm utilizes the evaluation function solely based on step cost. In this sense, an optimal solution is not guaranteed but a path is found. The highlighted blue line shows the assignment of the evaluation function to the open cell not including the heuristic but just the accrued cost of the current path.

Some path results utilizing Dijkstra's:



* Drawing the path

def reconstruct\_path(self, end):

current\_cell = self.cells[end[0]][end[1]]

path = []

while current\_cell.parent:

x, y = current\_cell.x, current\_cell.y

path.append((x, y))

current\_cell = current\_cell.parent

path.reverse()

def draw\_next\_square():

nonlocal path\_index

if path\_index < len(path):

pos = path[path\_index]

x, y = pos

if pos in self.delivery\_locations: # Check if the position is one of the goal locations

self.cells[x][y].visited = True

self.canvas.create\_rectangle(y \* self.cell\_size, x \* self.cell\_size, (y + 1) \* self.cell\_size,

(x + 1) \* self.cell\_size, fill='maroon')

else:

self.cells[x][y].visited = True

self.canvas.create\_rectangle(y \* self.cell\_size, x \* self.cell\_size, (y + 1) \* self.cell\_size,

(x + 1) \* self.cell\_size, fill='green2')

path\_index += 1

self.root.after(100, draw\_next\_square) # Schedule the next square to be drawn after 100ms

path\_index = 0

draw\_next\_square()

This method takes the current path and reverses it so that the path is drawn from the start to the goal state. The draw\_next\_square() method checks every cell in the path list to see if it is a goal location. If it is, the GUI will color the goal cell in maroon while the rest of the path is highlighted in green. It proceeds this checking process for each cell coordinate pair in the found path until it draws the path to all goal locations.

Statement of Rankings

**Member 1**: Sarah Groark   
My teammate and I agree that I handled **X%** of the overall project. My specific tasks included:

Task 1: I designed and wrote the matrix that represents the hospital floor plan.

* The 2d array is a 40x40 structure that color coordinates the hospital wards.
  + Matrix values include integers from 0-15
* Careful consideration was put into the representation of the walls and hallways, as well as entrances and exits to each unit.

Task 2: I wrote and implemented error handling in the following areas of the program =

* Input file reading
  + The following code block shows the error handling used in the read\_input\_file() method.
  + The code throws an error if the formatting of any of the inputs is incorrect and indicates which input field is in need of correction
    - For example - an odd number of locations was provided, and the program cannot map them into tuples
  + The code also delivers an error message if the specified input file is not found on the user's device
  + Finally, if any other unaccounted for error occurs, the user is notified

try:

with open(filename, 'r') as file:

for line in file:

if line.startswith("Delivery algorithm"):

try:

algo\_values = line.split(":")[1].rstrip('\n').split(",")

algorithms.extend(algo\_values)

except IndexError:

print(f"Error: Incorrect format in line '{line.strip()}'")

elif line.startswith("Start location"):

try:

coordinates = line.split(":")[1].strip()[1:-1].split(",")

start\_locations = tuple(map(int, coordinates))

except (ValueError, IndexError):

print(f"Error: Invalid start location format in line '{line.strip()}'")

elif line.startswith("Delivery locations"):

try:

loc\_str = line.split(":")[1].strip()

locations = loc\_str[1:-1].split("), (")

for loc in locations:

try:

x, y = map(int, loc.split(","))

delivery\_locations.append((x, y))

except (ValueError, IndexError):

print(f"Error: Invalid delivery location format in line '{line.strip()}'")

except IndexError:

print(f"Error: Incorrect format in line '{line.strip()}'")

return algorithms, start\_locations, delivery\_locations

except FileNotFoundError:

print(f"Error: File '{filename}' not found")

except Exception as e:

print(f"An unexpected error occurred: {e}")

* Algorithm choicing
  + This simple line of code takes into account whether the algorithm specification indicates to use AStar or Dijkstras. If neither of those options are indicated, then program prints an error message indicating to the user that the choice of algorithm is invalid.

else:

print("Ineligible algorithm type. Please specify whether to use A\* or Dijsktra's search algorithm.")

* Result of no path found
  + The error handling for the inability to find a path to a given goal location includes the printing of an error message to the console as well as a pop up message block.
    - Should there be no path found, the user is alerted to check the input coordinates in case the goal location is a wall or is an unreachable part of the floor plan.
  + An image of what the resulting alert message looks like is included.

elif self.algorithms[0] == "Dijkstras":

print("Running Dijkstras")

starting = self.agent\_pos

print(self.delivery\_locations)

local\_delivery\_locations = []

local\_delivery\_locations = self.search\_and\_sort(self.delivery\_locations)

print(local\_delivery\_locations)

for pos in self.delivery\_locations:

print("IN MAZE GAME IF STATEMENT")

print(f"Goal location: {pos}")

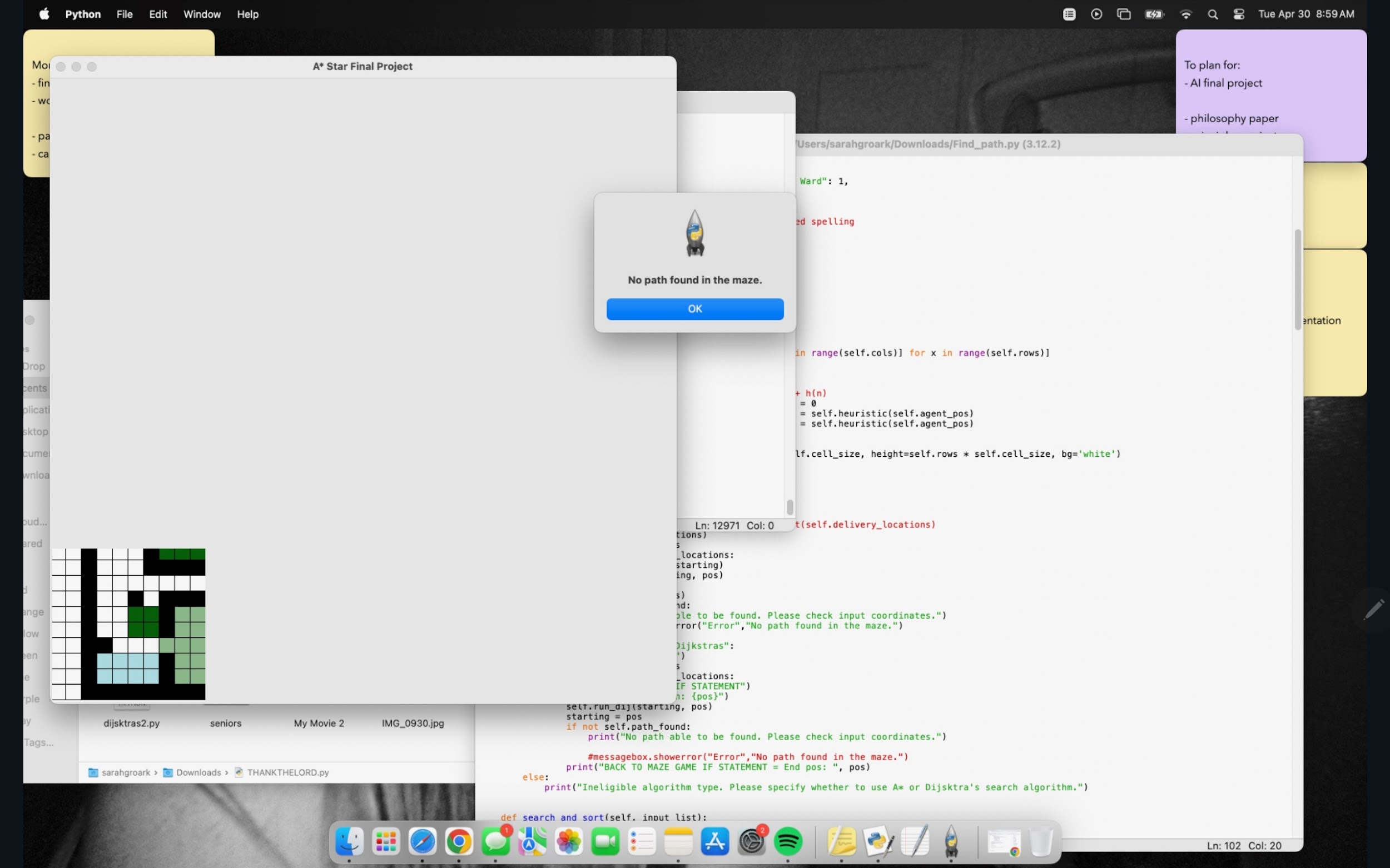
self.run\_dij(starting, pos)

starting = pos

if not self.path\_found:

print("No path able to be found. Please check input coordinates.")

messagebox.showerror("Error","No path found in the maze.")

print("BACK TO MAZE GAME IF STATEMENT = End pos: ", pos)

Task 3: I researched and implemented the inclusion of inputting a file from the command line.

* The code that accepts the input file as an argument in the command line utilizes the system library from Python.
  + The sys.argv checks the input line for two arguments - the Find\_Path.py file and the input file.
    - If the length of arguments isn't two, an error message is displayed.
  + The first index of the command line argument (which should be the input file) is assigned to the input\_file variable.

if len(sys.argv) != 2:

print("Usage error")

else:

input\_file = sys.argv[1]

# Read input file

algorithms, start\_locations, delivery\_locations = read\_input\_file(input\_file)

Task 4: I integrated the Dijkstra's Search Algorithm as well as changes made after significant debugging of both search algorithms.

* See program components - Dijkstra's Search Algorithm above for details.

Task 5: Wrote description of project, project organization, project components, and potential improvements/additions for the final report.

**Member 2**: Gabi Huegel  
My teammate and I agree that I handled **Y%** of the overall project. My specific tasks included:

Task 1: I created the GitHub repository to store all of our files throughout the project.

<https://github.com/gabriela-148/Final_Project/blob/main/Find_path.py>

Task 2: I designed and implemented the program module that assigns the priority and ward name to the maze. When first tackling this problem, it took some trial and error to find a way to check for priority of the wards without a lengthy, time consuming if-statement. The solution I came up with was to create an adjacency list of all the cells in the maze. In the initialization of the MazeGame I created the following code:

self.priorities\_all = {

1: [],

2: [],

3: [],

4: [],

5: []

}

Here I created the adjacency list from numbers 1-5, since there are only priorities labeled 1-5 in this particular problem. Next, I edited the Cell class to include two new attributes: ward and priority.

class Cell:

def \_\_init\_\_(self, x, y, is\_wall=False):

self.x = x

self.y = y

self.is\_wall = is\_wall

self.g = float("inf")

self.h = 0

self.f = float("inf")

self.parent = None

self.visited = False

self.priority = 0

self.ward = ""

def \_\_lt\_\_(self, other):

return self.f < other.f

After adding the two attributes, I edit the draw\_maze method in MazeGame so that as the program checks the value associated with the cell to assign it a color of the ward, I can also assign the ward name, and priority number to the cell:

for x in range(self.rows):

for y in range(self.cols):

color = color\_map[self.maze[x][y]]

if self.maze[x][y] == 1:

self.cells[x][y].ward = "Admissions"

self.cells[x][y].priority = 1

self.priorities\_all[1].append((x,y))

print(self.cells[x][y].ward, self.cells[x][y].priority)

This is just the first if statement, but it continues for a long if-else statement to assign all the wards and priorities. It also appends the x,y coordinates of the cell to its priority. In the case above, the Admissions ward is a priority one, so the x,y coordinates would be added to the list at one. This process repeats for each cell in the maze. For walls and hallway the priority is 0 and the ward name is either wall or hallway.

Task 3: I implemented the sorting algorithm for the delivery locations. I implemented the previous code of the ward and priorities to help with this sorting algorithm. This is the sorting method:

def search\_and\_sort(self, input\_list):

result = []

locations = []

sorted\_result = []

for priority, positions in self.priorities\_all.items():

for position in self.delivery\_locations:

if position in input\_list:

print(position[0], position[1], self.cells[position[0]][position[1]].priority)

result.append((self.cells[position[0]][position[1]].priority, position))

break

sorted\_result = sorted(result, key=lambda x: x[0], reverse=True) # Sort by priority, higher to lower

for item in sorted\_result:

locations.append(item[1])

return locations

In this method it takes a list of tuples and then creates three temporary lists: result, locations, sorted\_result. First, it loops through the adjacency list and for every tuple in the first priority it checks to check if that x,y tuple is in the input list (meaning that the x,y tuple is in the delivery locations). If yes, it adds the x,y tuple and its priority to the results list. This repeats until there are no more delivery locations to check. Then it uses the built in sorted() function to sort the tuples based on their priority from high to low. Then it loops through the sorted list that has both the x,y and priorities and appends just the x,y tuples to the locations list and finally returns it.

Task 4: I implemented the input checking. I created this method to read the input from a .txt file to then be able to run the program. (Sarah implemented the error checking throughout this method)

def read\_input\_file(filename):

algorithms = []

start\_locations = ()

delivery\_locations = []

priorities = {

"ICU": 1, "Emergency": 1, "Oncology": 1, "Burn Ward": 1,

"Surgical Ward": 2, "Maternity Ward": 2,

"Hematology": 3, "Pediatric Ward": 3,

"Medical Ward": 4, "General Ward": 4,

"Admission": 5, "Isolation Ward": 5 # Corrected spelling

}

ward\_locations = {

"General Ward": (21, 23),

"Surgical Ward": (24, 25),

"Admissions": (17, 34),

"Emergency": (11, 31),

"Maternity": (8, 5),

"Oncology": (24, 30),

"ICU": (17, 31),

"Isolation Ward": (10, 27),

"Pediatric Ward": (33, 11),

"Burn Ward": (22, 11),

"Hematology": (24, 16),

"Medical Ward": (34, 30),

}

try:

with open(filename, 'r') as file:

for line in file:

if line.startswith("Delivery algorithm"):

try:

algo\_values = line.split(":")[1].rstrip('\n').split(",")

algorithms.extend(algo\_values)

except IndexError:

print(f"Error: Incorrect format in line '{line.strip()}'")

elif line.startswith("Start location"):

try:

coordinates = line.split(":")[1].strip()[1:-1].split(",")

start\_locations = tuple(map(int, coordinates))

except (ValueError, IndexError):

print(f"Error: Invalid start location format in line '{line.strip()}'")

elif line.startswith("Delivery locations"):

try:

loc\_str = line.split(":")[1].strip()

locations = loc\_str[1:-1].split("), (")

for loc in locations:

try:

x, y = map(int, loc.split(","))

delivery\_locations.append((x, y))

except (ValueError, IndexError):

print(f"Error: Invalid delivery location format in line '{line.strip()}'")

except IndexError:

print(f"Error: Incorrect format in line '{line.strip()}'")

return algorithms, start\_locations, delivery\_locations

except FileNotFoundError:

print(f"Error: File '{filename}' not found")

except Exception as e:

print(f"An unexpected error occurred: {e}")

This method reads the input line by line and everything after the “:” it appends to either the algorithms, delivery or starting locations list depending on the line the program is reading.

Task 5: I implemented the A-Star algorithm to find paths between wards and continuously update the starting and ending positions. In the original implementation of this A-Star maze program, it had a static starting and ending position for the path finding algorithm. This obviously had to change for this maze. When testing out different code, I found that updating the starting and goal position around the A\* method was more successful than updating it in the method. So when the input is checked to run either A\* or Dijkstra, it sets the starting and end positions to what they currently assigned as, then updates as the algorithm is executed:

if self.algorithms[0] == "A\*":

print(self.delivery\_locations)

local\_delivery\_locations = []

local\_delivery\_locations = self.search\_and\_sort(self.delivery\_locations)

print(local\_delivery\_locations)

starting = self.agent\_pos

for pos in local\_delivery\_locations:

print("Start pos:", starting)

self.run\_astar(starting, pos)

starting = pos

print("End pos:", pos)

elif self.algorithms[0] == "Dijkstras":

print("Running Dijkstras")

starting = self.agent\_pos

print(self.delivery\_locations)

local\_delivery\_locations = []

local\_delivery\_locations = self.search\_and\_sort(self.delivery\_locations)

print(local\_delivery\_locations)

for pos in self.delivery\_locations:

print("IN MAZE GAME IF STATEMENT")

print(f"Goal location: {pos}")

self.run\_dij(starting, pos)

starting = pos

if not self.path\_found:

print("No path able to be found. Please check input coordinates.")

messagebox.showerror("Error","No path found in the maze.")

print("BACK TO MAZE GAME IF STATEMENT = End pos: ", pos)

else:

print("Ineligible algorithm type. Please specify whether to use A\* or Dijsktra's search algorithm.")

Task 6: I wrote up all of my taks for the Final Report of this project.

Potential Additions or Improvements

* Inclusion of ward names as viable input parameters for goal locations (rather than just specific coordinates)
* Interactive floor plan map that allows users to click on a delivery location to find a path there
* Time estimates for robot trips
* Integration of a 'homebase' that the robot can return to when no requests are being made
* Ability to travel to a supply unit/room before delivery to gather supplies