**Project Summary**

The data structure that I implemented in this final project of CS231 was Graph, which is composed of a set of vertices and edges. I also implemented the Dijkstra’s algorithm which carries out a breadth-depth search in the graph and finds the number of steps to get from the root to each other vertices in the graph.

The goal of the project was to use graph to build the Hunt the Wumpus game. Each vertex in the graph is a “room” connected to its neighbors. The UI designs in the main code the executes the game allow users to interactively control the hunter, who can move around the group or shoot at a direction for the Wumpus.

**Task Solutions**

Vertex

The Vertex class holds fields of an array that stores its neighbors in the four direction. The directions are stored in its own class in an enum, which allows me to use the .ordinal() method to convert a direction to an integer and thus access the respective indexed position in a vertex’s neighbor array. The vertex class implements the Comparable interface and its method compareTo() that compares the vertices based on their cost field, which was used in the graph class when I was implementing the Dijkstra’s algorithm. The draw method visualizes each vertex in the game as a box and draws connections to its neighbors at a certain direction (if it has one). A vertex is red if it is less or equal to two steps away from the position of the Wumpus and black otherwise.

Graph

The Graph class has an arraylist of all the vertices in the graph. The method addEdge first adds the two vertices into the arraylist and because the graph used in the game is undirected, it creates a bi-directional link between the two vertices.

The method shortestPath implements Dijkstra’s algorithm given a root vertex. It first set the cost of all vertices to Integer.MAX\_VALUE, and put the root into a priority queue that sorts the vertices in ascending order based on their costs. Then I used a while loop to keep removing the first element in the queue, mark it as visited and loops through all its neighbors and update their costs if a shorter path is available.

The Landscape and Agent class are updated from prior projects to meet the design of the game.

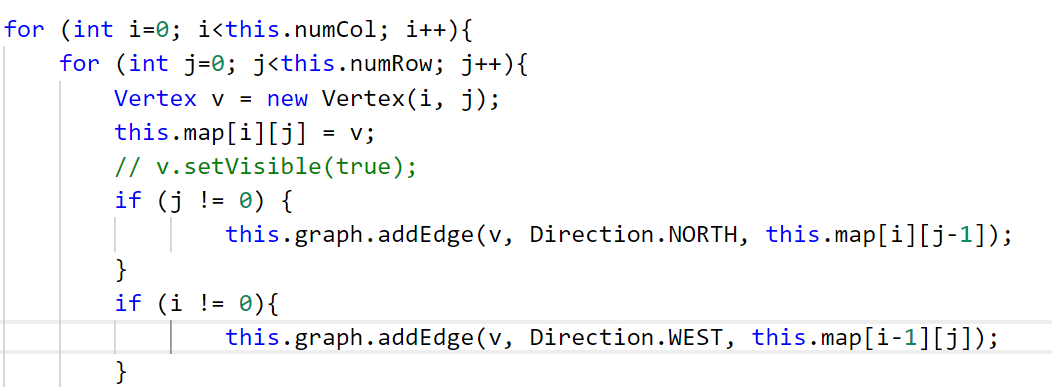
Hunter

The Hunter class stores a reference to its position vertex, which can be updated through the method updatePostion. The constructor takes in the x and y position of the hunter and the vertex of its position. It also stores two Boolean fields that indicate if the hunter is armed or alive.

The Wumpus class follows similar structure as the Hunter class.

Hunt the Wumpus

This is the class that contains the UI designs and sets up the game. I created the class using the templet of InteractiveLandscapeDisplay. It has fields for creating the game windows as well as elements of the game – Graph, Hunter and Wumpus. Because the vertices in the game will be mapped out to a 2D landscape, I also created a 2D array *map* to manage the vertices more conveniently. The constructor initializes the main window and calls the setGame() method, which initializes the graph, the hunter and the Wumpus. I used a nested for loop to traverse through each column and each row and create vertex for each position, and connects it to the previous vertex in the same row or column.



After initializing the hunter and the Wumpus, the method calls the shortestPath method of Graph to generate each vertex’s position with respect to the Wumpus and draws the vertices in different color.

The private class control is where the game and keyboard actions are defined. When the hunter is alive, I used several if/else if statement so that pressing “a”, ”w”, “s”, “d” respective moves the hunter going left, up, down and right, by calling the updateHunter method in HuntTheWumpus class, if the vertex of its current position has a neighbor at that direction. if the spacebar is pressed, the hunter will be armed.

The method updateHunter decides the consequences of the keyboard actions. A vertex will only be visible if the hunter has entered it or shoots at it. When the hunter is not armed, the method will simply move the hunter and updates its vertex reference.

If the hunter enters the same vertex that the Wumpus is in, the alive field of the hunter will be set to false, a JOptionPane will pop up saying the user lost the game, and the game will stop.

If the hunter is armed, the method will check if the Wumpus is in the direction it shoots, and if yes, the hunter wins the game and otherwise it loses. The main method of the class executes the game using while loop that keeps running so long as the state enum of the game is PLAY.

**Extensions**

Improve Visualization

I used an image of the Smurf for the hunter and an image of Gargamel for the hunter. The Smurf is trying to escape from Gargamel otherwise he will be kidnapped. In the draw method of both the Hunter and the Wumpus class, I used a BufferedImage object to read in the image from file. I also generated a black-and-white version of the Smurf when it is dead, i.e. the user lost the game.

The method also draws a 3Drectangle when the smurf is armed.

Use my own priority queue and compare performance

I created the class MyGraph which is child class of Graph. The difference is that it overwrites the parent’s shortestPath method and uses my own PQHeap from project 8. Then I changed the Graph field in HuntTheWumpus to MyGraph to use my own PQHeap and it generates the same result.

Generate random graph, positions for characters, and connections between vertices

I modified the setGame method in HuntTheWumpus, allowing it to generate random connections between vertices (i.e. making some vertices disconnected with its neighbors) by adding an if statement in the nested for loop.

PIC

Then I used Random.nextInt to generate random positions for the hunter and the Wumpus and used two while loop to reselect their positions if the hunter and the Wumpus are generated into the sampe vertex or in a vertex that has too few paths out.

PIC

Add Restart button

I added a start button to the game window by creating a new JButton object *restart* and adding a separate if statement in the actionPerformed method in the Control class, which calls the reset method that I created in HUntTheWumpus class.

The reset() method initializes the game by calling setGame and requests focus of the window.

Conclusion

This has been a fun (and challenging) project in which I learnt the data structure of Graphs and the Dijkstra’s algorithm. When designing the game, I also learnt how to use simple UI designs to detect user’s inputs and carries out respective actions. I gained a better understanding of the data structures and algorithms while experimenting with different game setup customizations, both visually and application-wise.

P.S. Thanks for a great semester!