Project 10

Added by Christopher Hoder, last edited by Christopher Hoder on Dec 09, 2011

Project 10 : Hunt the Wumpus

Outline:

The goal of this project was to implement a version of the classic computer game, Hunt the Wumpus. A version of the game can be found here. Here is an outline of the game: The Wumpus is an ugly, smelly monster that lives in a cave. You are a hunter who has been given the task of finding the Wumpus and killing it. The Wumpus, however, lurks in the dark shadows of the cave system and has a good sense of hearing. You initially have no idea how the cave is laid out and it can include any number of openings and tunnels. As you move from room to room the cave layout becomes mapped and appears to you. You will also pick up clues as to where the Wumpus is. If the Wumpus is two or fewer rooms away, you can smell it. This smell is conveyed with a change in cave color from black boxes to red boxes.

If you travel to close to the Wumpus (i.e. into the same cave opening) it will hear you and eat you. However, you have 1 arrow with which to slay the Wumpus. You can fire from one adjacent room to another where you suspect the Wumpus to be hiding. If you fire the arrow correctly, you will slay the Wumpus and win the game. If not the Wumpus will hear the arrow hit the cave walls and come eat you and you will loose.

Game Play Controls:

The game controls are as follows:

p : pause and play the game

r : reset the game

q : quit the game

Arrow keys : move the Hunter from room to room

a : arm the arrows, the arrow keys are then used to fire the arrows in the given direction

s : begin / end saving every 50th iteration to an image file

h: bring up the help menu

Solution / Coding

To implement this game I reused the SimObject, SimObjectList, Landscape and LandscapeDisplay classes from previous projects. This code is unchanged and the Hunter and Wumpus were saved on the landscape as agents while the Vertices are saved as resources. The graph was not saved on the Landscape, because i felt that it was specific to the game and not the landscape. The distance a particular vertex was from the vertex containing the Wumpus was determined using a single-source shortest path algorithm so that we know the distance every vertex is from the vertex where the Wumpus is. This algorithm was modelled off of the Dijkstra's algorithm, though some parts had to be changed slightly to work with my dynamic cave implementation that I did as part of an extension (this is talked about later). To implement this algorithm I made the Vertex class implement the Comparator interface (comparing costs) so that I could use a priority queue to sort the Vertices during the shortest path calculation.

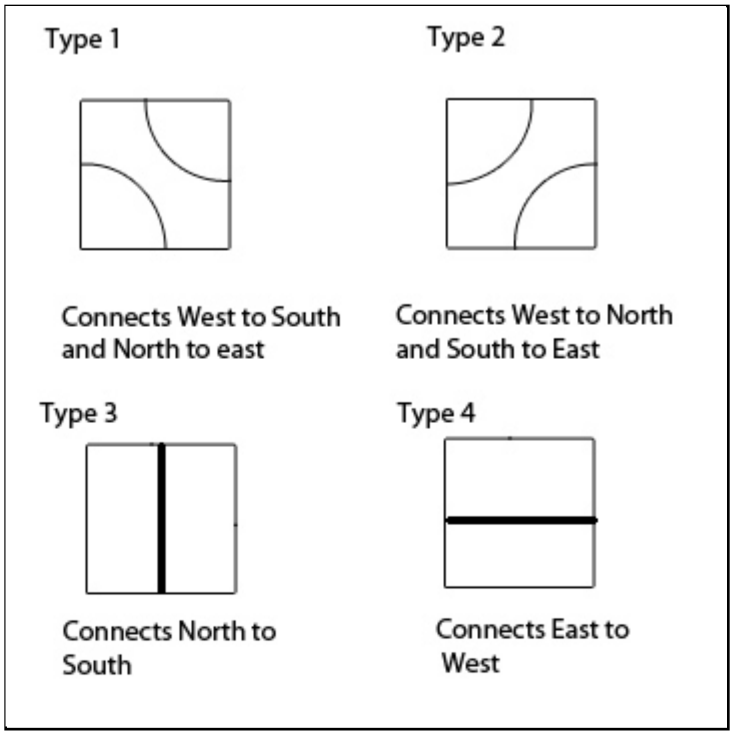
I tried to make the drawings of the Hunter and Wumpus a little bit more interesting by adding eyes and a mouth to each. The Hunter is a circle that is blue when it is alive and red when it is dead. Additionally, The hunter's smile turns to a frown when it dies. The Wumpus is a green square with a smiley face if it is alive and a red square with a frown if it is dead. I also have the Vertices showing the paths between the different openings. Withouth an easy way to make curves between the different connections they are linear lines. I varied the color of paths when 2 paths came close to each other. If two lines intersect, the Hunter will follow the same color that it come in on. (i.e. see images below and the intersections of black and blue lines).

I decided to keep the fire method, which controls the firing of the Hunter's arrow, inside the WumpusHunt game class because it required the use of both the landscape and the graph in order to determine the cave where the arrow is shot into (due to the series of tunnels that exist). This method could easily be moved into the Hunter class (who fires the arrow)

Extensions

As part of an extension I made the cave system that is created with each game and is a new random map each time. This cave is going to have a random number of cave openings (i.e. spots where the hunter can stop and the Wumpus could be). These openings can be connected by a convoluted series of tunnels, creating a twisty maze of passages between these different openings. These passages are drawn in the landscape so when you traverse them they will become visible along with the cave openings. This allows the user to see where each connection goes without a mental map of the connections. There is code to ensure that the cave created is fully connected and each vertex can be reached from every other vertex.

Types:


Type 1
Type 2
Connects West to South and North to east
Connects West to North and South to East
Type 3
Type 4
Connects North to South
Connects East to West

The way that I implemented this extension not the most ideal. In retrospect, I wish that I had created an edge class that stored information about each edge and added a dynamic edge weight data field to make calculating the single - source shortest path a bit easier and maybe less complex. However, I was able to create the dynamic board by doing the following. First a static grid of Vertices is initialized on the landscape. Then random connections are made between neighboring vertices (in the north, south, east, west directions). Random connections are added to the graph until we have created a connected graph. At this point we have a connected graph that is a grid of Vertices with straight edge connections only.

Then, based on a static probability, a random number of vertices with either 2 or 4 connections will be turned from cave openings to tunnels between openings. Based on the type of connections one of 4 types of tunnel vertices are created. The image below illustrates the 4 different types of connections there can be. The Image below illustrates the different type of connections that can be made.

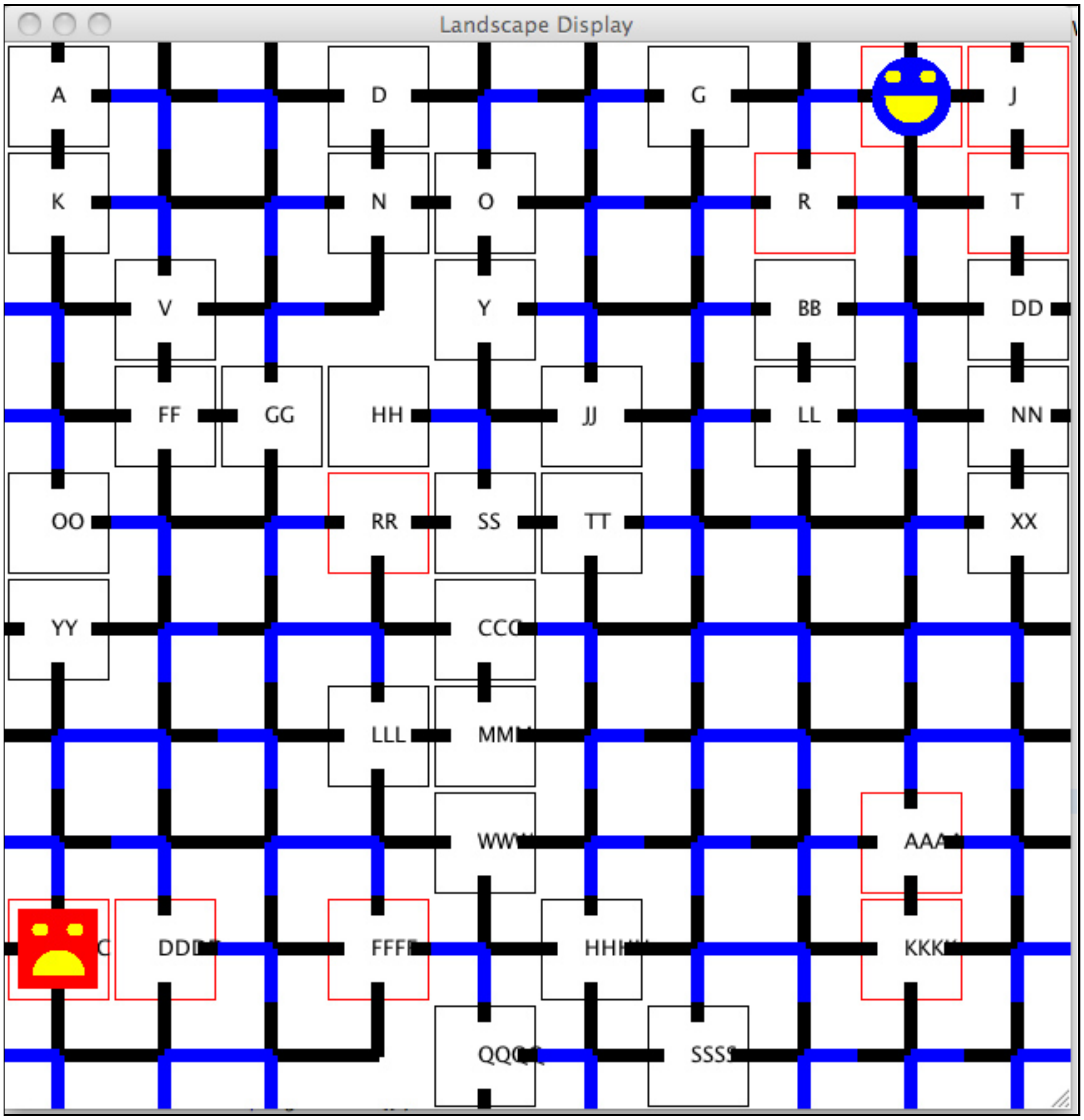
Once this dynamic cave code was complete it allowed for an easy method of introducing different levels of difficulty into the game. I did this by simply saying that an easy game would have almost no crazy connections while the hard version of the game would have as many as possible. I implemented this by allowing the Connections Probability to change between games using the drop down menu at the bottom of the landscape. I referenced the following page for help on implementing this drop down box: http://www.java-forums.org/java-tutorials/3181-java-swing-tutorial.html

The game board can also be run using different widths and heights. This can be changed using a command line arguments. The syntax for this is as follows: WumpusHunt [~~w,~~-width] [-h, --height]. The arguments are parsed using the CmdLineParser. Additionally, there is code for a static 4x4 game baord as well as creating a completely connected cave in the WumpusHunt class but they are unused in the game.

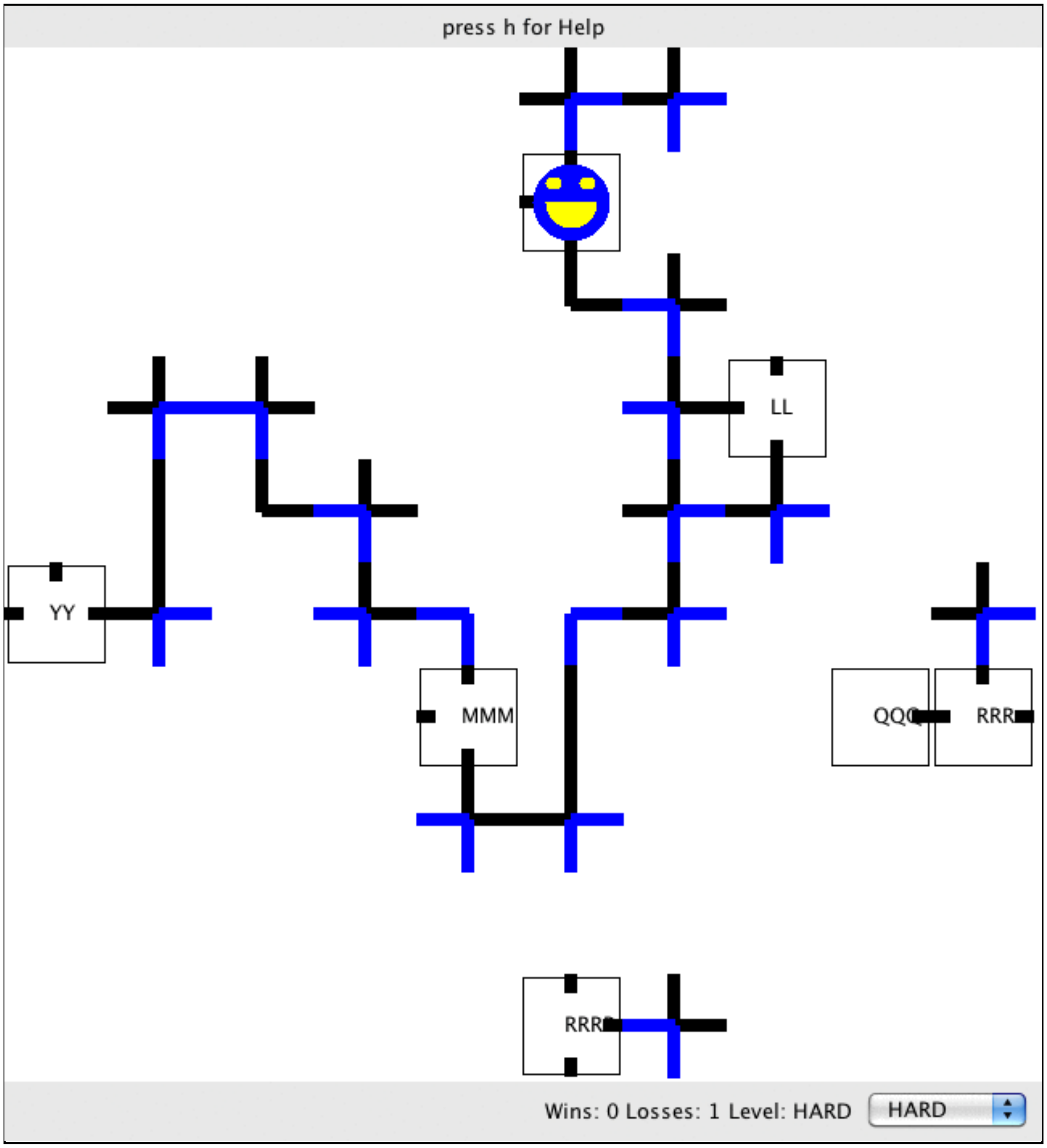
I also added a help message dialog box. If you press h while playing the game, a message box will appear with all of the controls for the game. I also put a text JLabel on the top of the Landscape that tells you to press h for help. Additionally, I modified the LandscapeDisplay class to take the display title as an input so that I could get it to say Wumpus Hunt at the top of the game.

As with the last project, I made it so that the game board does not display when you pause the game, instead it displays a static image of the origional Wumpus Hunt game picture.

Since, this is applied on a per Vertex basis there is nothing that prevents multiple tunnels from linking together creating a complex maze. We therefore can create very complex mazes now that we know are connected. Below is an image of a particularly complex system that was created using this code. This is an older picture so it does not show some of the added extensions that exist in my final versions.

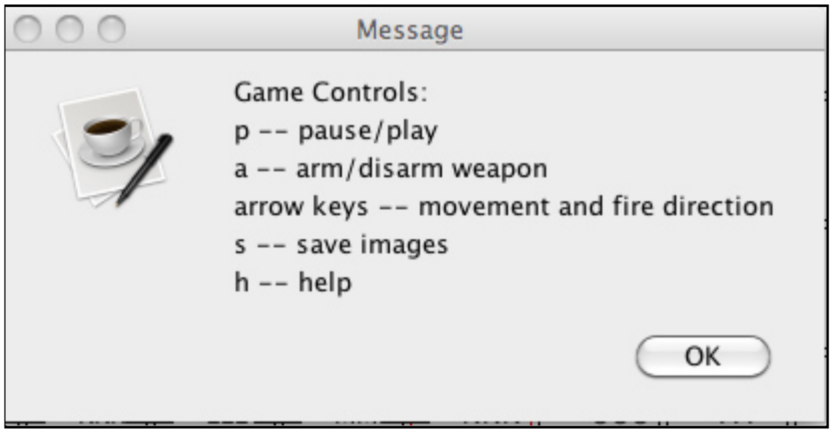


Below is a gif of me playing the game on the hard setting:



Wins: 0 Losses: 1 Level: HARD

Also below is an image of the help dialog box that pops up when you press h for help:



Game Controls:
p -- pause/play
a -- arm/disarm weapon
arrow keys -- movement and fire direction
s -- save images
h -- help

Additionally, I created documentation for all of my code which can be found in the index.html file in the html folder included in my solution.

Conclusions

This project taught me a lot about graphs and how to traverse them in different ways both iteratively and recursively. Implementing the dynamic cave with non-straight connections proved to be really challenging and looking back over my code I see several areas that could be improved with more time. Some methods are not in the ideal class placement (i.e. methods dealing with individual vertices are in the graph class because they require knowledge of the graph as a whole). Some of these methods could be moved around for better coding. Additionally, by implementing an edge class I think I could more efficiently calculate the single-source shortest path with these crazy connections that I have. I also learned methods of testing that the graph was connected by using the single-source shortest path method already used, instead of writing a method modeled off the isConnected() method from class.

Labels

cs231f11project10

Printed by Atlassian Confluence 3.3.1, the Enterprise Wiki.