Chinonso Akujuobi

Intro to Artificial Intelligence

07 March 2021

Lab 1 Write-up

A design choice that I felt influenced many other parts of my implementation has to be my definition of “x” and “y.” The reason I even had to make this choice was because the terrain map’s dimensions are 395 pixels width (corresponding to columns) and 500 pixels height (corresponding to rows). So my 2D pixel array had to be of size [395][500]. The problem though is that this represents a shape that is 395 rows high and 500 columns wide, which is obviously not the case. So in my implementation, I had to switch “x” to y and “y” to x. Therefore, when I refer to x and y in this write-up I really mean them the other way around.

**Cost Function**

The formula I used for g(n) is the following:

*g(n) = g(parent) + (speed modifier of the node times the latitude or longitude (based on the direction we’re going)).*

I’d retrieve the G(n) cost of the parent using a getter function for G(n) I had created for the Pixel class. Then, I compared the x values of the parent Pixel and the current Pixel. If these were the same, it means that we’re moving vertically. I’d return the sum of the parent’s G(n) and 10.29 times the speed modifier of the current node. The same is done with the y values, which means we’re moving horizontally, and we multiply the speed modifier by 7.55. I didn’t know if this was necessary, but I also had a case for moving diagonally. The thing multiplied by the speed factor here is the hypotenuse of the right triangle formed by the legs of 10.29 and 7.55. I figured that I had to do this because I’m checking diagonal neighbors, which of course involves diagonal movement.

**Heuristic**

My heuristic was the formula:

*h(n) =* 3*D distance between current node n and destination node, times the speed modifier of a footpath pixel*

My reasoning was simple: to form an admissible heuristic, I wanted to make sure it was an under-bound of the g(n). In other words, I wanted to ensure that h(n) never exceeded g(n). That’s why I’m always multiplying the distance by the fastest speed, which happens to be held by footpath pixels.

**Fall**

To implement fall, I looped over my 2D array of pixels and checked each pixel to see if it was a footpath pixel. If it is, I generate its neighbors and loop over them. If any of the neighbors are easy movement forest pixels, I set the original pixel’s terrain type to “leafy footpath” and break out of the neighbor loop. The speed of a leafy footpath pixel is 1.5 in my implementation; I figured that going through paths with leaves scattered all over them would be just a bit slower than open land paths (which have a speed of 1.2).

**Winter**

For winter, I iterated through all the pixels again, this time checking that each pixel is a water pixel. If so, its neighbors are generated. If any of the neighbors are non-water pixels, the original pixel is added to a list named “waterEdgePixels.” Then, we do a BFS, adding all the pixels from this list to a queue of pixel-integer tuples (with the integer representing depth). The depth for all pixels in the queue is initialized to zero. While the queue isn’t empty, we pop off a tuple and extract its pixel and depth. If the depth is 7, we break out of the while loop. If not, we proceed, generating neighbors for the current extracted pixel. I iterate through the neighbors and see if any of its neighbors are water pixels. Then we check is it’s not in our visited set. If these conditions are true, we make a new tuple consisting of the neighbor and the original depth + 1 and add the tuple to the queue. We also add it to our visited set, which will contain all the pixels that need to be changed to ice. As for ice’s speed, I decided to make it 3.4. Ice is slippery, but smooth, so I figured it would be a bit faster than most paths on the map.

**Spring**

Spring was difficult. I had to iterate over all the pixels like the previous two seasons. I checked each pixel to see if it was a water pixel. If so, I got its neighbors. If any of them were non-water, I added the neighbor to a shorePixels set and placed it in a neighborMap with the key being the neighbor and the value being the original pixel. I then iterated over the shorePixels and added pixel-depth tuples to initialize my queue with, like in winter. Before I began BFS, I also initialized a visited set with the pixels in the shoredPixels.

While the queue isn’t empty, I pop the tuple from the queue. I extract the pixel and its depth and get the pixel’s elevation. Then, using the neighborMap, I get the water pixel predecessor of the pixel and get the predecessor’s elevation. I subtract the water elevation from the pixel elevation, and if the difference is less than or equal to 8, I add it to set of pixels that will be changed to muddy pixels. **The reason I’m using the arbitrary number 8 instead of 1 (like the writeup says) is because I had trouble matching the test output when I was checking if the difference was less than or equal to 1. I had much better luck with 8.** I also check if the depth is 15; if it is we break. I continue after that, looping through the generated neighbors of the pixel. If any of the neighbors are non-water and if the difference between the neighbor pixel’s elevation and water pixel’s elevation are again less than 8, I add it to the set of pixels we’re changing. If the neighbor is non-water and the visited set doesn’t contain it, I take the steps necessary to make a new tuple consisting of the neighbor and the original depth + 1 and add the tuple to the queue. In addition, I also add the neighbor to the neighborMap and the visited set. As one might expect, mud slows you down, so I decided to make the muddy terrain have a speed of 6.2.

**Human Readable Output**

My algorithm for human readable output is as follows: I call my winter or spring function according to what the input season is, with the result placed in a set called affectedPixels. I then iterate through the set, changing the color of each pixel in affectedPixels to light blue or brown (depending on if we’re doing winter or spring, respectively). This is all the pre-processing phase, before the actual path drawing on the map. All the points in our specific routes returned by A\* are stored in a two-dimensional ArrayList named *event*. We iterate through each and every pixel in event, coloring it to red (our chosen path color) using image.setRGB(x coordinate of pixel, y coordinate of pixel, rgb of pixel). As for the total path length in meters, I do that before all of this, in a loop where I call my A\* method over and over again with points from the path file. This is simply a running sum of the 3D distances between the start pixel and the goal pixel of every path from the path file.