CMP2020M Artificial Intelligence

Games Workshop 2: Dijkstra's Algorithm

Objectives

1. Implement Dijkstra's algorithm to find a path from a bot to the player.

Duration: 1 week

Tools / Libraries

1. Monogame v3.6
2. Microsoft Visual Studio 2015 (including Community Edition)

This workshop is directly assessed, and will count towards your final mark for this module. **You should ask a demonstrator to tick off the task when it is complete. *You may also be asked questions about it on your examination***. You should therefore make sure that you complete it and discuss any problems with a workshop demonstrator.

You may need to spend time on this task outside of your timetabled workshop, but please make sure that you finish it for next week!

Introduction

Using the Pathfinder project, we will implement Dijkstra's algorithm (which we have covered in the last lecture). Note that you do not need your solution for workshop 1 to undertake this workshop task.

Dijkstra's algorithm computes an optimal (shortest) path between two nodes – in this case from the bot's grid position to the player's grid position. Once the algorithm has terminated, a bot could move to the player's position by following the final node links set by the algorithm (see notes for lecture 2).

However, we will not create a new bot this week: just implement Dijkstra's path finding code, as a completely new object, and inspect the results. This workshop will show you how to do this, step-by-step.

Task A

To begin, create a new source code file, containing a new class called Dijkstra. The file should initially look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Diagnostics;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Audio;
using System.IO;
```

```
namespace Pathfinder
{
	class Dijkstra
	{
	}
}
```

Dijkstra's algorithm needs to maintain the following data:

1. Which nodes (grid locations) are closed
2. A cost value for each node
3. A link for each node (which shows the best way to get there)
4. Which nodes form the final path

The easiest way to store this data is to setup a corresponding set of 2D arrays (size 40x40). Each of the can be used to store one of the above data items for every location in the grid. A add the following members to the Dijkstra class:

```
class Dijkstra
	{
		public bool[,] closed; //whether or not location is
	closed
		public float[,] cost; //cost value for each location
		public Coord2 [,] link; //link for each location = coords
					//of a neighbouring location
		public bool[,] inPath; //whether or not a location
					//is in the final path
```

Remember, these items need to be created using 'new' as 40x40 arrays, and initialised, in the class constructor. For example:

```
	public Dijkstra()
	{
		closed = new bool[40, 40];
		//etc..
	}
```

Thus, using these data structures, if we wanted to mark grid location {10,10} as closed, we could just write:

```
closed[10,10] = true;
```

Or, if we wanted to set the cost of location {27,5} to zero, we could just write:

```
cost[27,5] = 0;
```

And so on....

Task B

Create a new class function called *build()*. This function will calculate the path, and fill out the data arrays we defined above. The build function will need to take some parameters:

1. A reference to a bot, to get the starting location
2. A reference to the player, to get the end location
3. A reference to a level, so we can determine which grid locations are blocked by walls

So the function declaration should look something like this:

```
public void Build(Level level, AiBotBase bot, Player plr)
{
      ...
```

The first thing the function should do is initialise the data arrays:

1. All nodes should be marked as open
2. Each node's cost should be initialised to a very big number (1,000,000 for example)
3. Links are not important at this stage, so just set them all to {-1,-1}
4. Every value of "inPath" should be initialised as false.

To start the search, the bot's position should be marked as open, eg:

```
closed[bot.GridPosition.X, bot.GridPosition.Y] = false;
```

Similarly, the cost at the bot's position should be set to zero

.

The function should then execute a repeating loop, using a while() statement. The loop should execute the following processes:

1. Find the grid location with the lowest cost, that is not closed, and is not blocked on the map layout (on the first iteration of the loop, this should actually be the bot's position). If there are locations with equal lowest cost, it doesn't matter which you choose.

2. Mark this location as closed

3. Calculate a new cost for the 8 locations neighbouring this one: this will be equal to the cost of the location that was just closed (the *parent*) + the cost of moving from there to the neighbour (so, +1 for left,right,up,down, +1.4 for diagonals) : **BUT only change the cost of the neighbouring square in the cost array IF**:

    a. The new cost value found for the neighbour is lower than its current cost value.

b. The neighbouring location is not blocked on the map layout. Remember: You can check if a grid location is blocked, or off the edge of the map, using the *level.ValidPosition()* function.

4. If you have set a new cost value for a neighbour, then also set its corresponding link value to be equal to the coordinates of its parent (the location that was closed)

5. This should be repeated until the location occupied by the player is closed, at which point the while loop should terminate.

Task C

When the loop has terminated (the player's location has been closed), the array of links should represent a traceable path from the player back to the bot. This path is defined by the values in the link array. The next task is to extract these locations.

Starting from the player coordinates, trace back through the links array and mark the corresponding entries in the "inPath" array to true. The code for this should look something like:

```
bool done = false; //set to true when we are back at the bot position
Coord2 nextClosed = plr.GridPosition; //start of path
while (!done)
{
	inPath[nextClose.X, nextClose.Y] = true;
	nextClose = link[nextClose.X, nextClose.Y];
	if (nextClose == bot.GridPosition) done = true;
}
```

Task D

The inPath array will now tell us which grid locations are in the path. We would now like to actually create a path using the Dijkstra::Build() function, and visualise it on the grid so that we can check that it works.

Include an object of type "Dijkstra" in the Level class, and create it using "new" in the Level constructor function. Make sure it has a *public* access specifier. Set the game map to "1.txt".

We want to call the Dijkstra::Build() function at some suitable point in the code. It is useful to be able to decide at run-time when this happens, so we want to call it when a key is pressed. Look at the *Update()* function in the Game1 object. This is where keypresses are detected. Add a new "else if" clause to the keypress detection code, to detect a keypress (of your choice). Add the call to Dijkstra::Build() to this clause. The code should look something like this:

```
else if (keyState.IsKeyDown(Keys.P))
{
	level.dijkstra.Build(level, bot, player);
}
```

Next, we need to add some code to draw the path on the map.

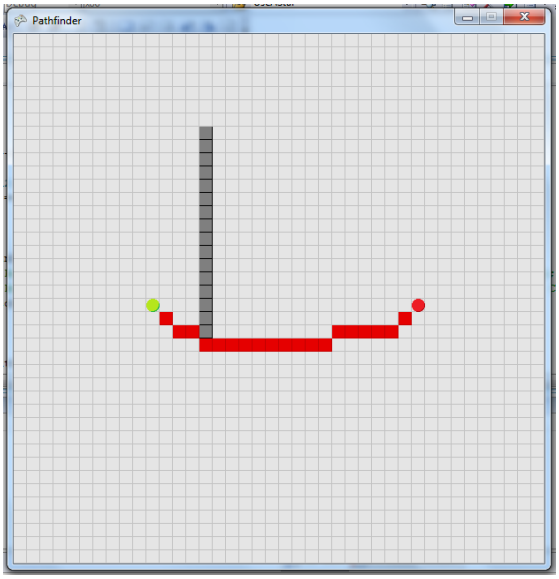
Look at the DrawGrid() function in class Game1. You will see some code like this:

```
if (level.tiles[x, y] == 0)
{
      spriteBatch.Draw(tile1Texture, pos, Color.white);
}
else
{
      spriteBatch.Draw(tile2Texture, pos, Color.White);
}
```

Firstly change this code so that tiles which are in the path are drawn red. You can do this as follows:

```
if (level.tiles[x, y] == 0)
{
      if (dijkstra.inPath[x, y])
      {
            spriteBatch.Draw(tile1Texture, pos, Color.Red);
      }
      else
      {
            spriteBatch.Draw(tile1Texture, pos, Color.white);
      }
}
else
{
      spriteBatch.Draw(tile2Texture, pos, Color.White);
}
```

Now test your code! Run the programme, and press the key to run the *build()* function. If all is well, you should see something like the following:



The red line shows the path from the player to the bot.
Repeat this with the other built in maps to check the algorithm works correctly.

Note: You need to ensure that your implementation of Dijkstra is working before you move on to the next workshop.

Optional: Task E

If you have time, add extra code to draw all closed locations in a different colour (blue?). Also, draw locations which are not closed, but which have had a new cost value calculated in yet another colour (light blue). This will show you how the algorithm has “spread” from the initial bot position to find the player.