CMP2020M Artificial Intelligence

Games Workshop 3: A* Algorithm

Objectives

1. Modify Dijkstra's algorithm to implement A*
2. Create a bot to follow the path generated by A*
3. ***Optionally***, optimise the algorithm further

Duration: 1 week

Tools / Libraries

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

This workshop is not directly assessed, and will not count to your final mark for this module. ***However, you may asked questions about it on your examination***. You should therefore make sure that you complete it and discuss any problems with a workshop demonstrator.

Introduction

We are going to implement the A* algorithm. A* is an extension of Dijkstra, so we are going to modify your solution to the last workshop. Before starting, you should make sure you have a back up of your version of Dijkstra! As before, we will implement A* in a series of steps.

Re-read the lecture notes, so that you are familiar with how the A* algorithm works.

Task A

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

```
sing 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 AStar
      {
      }
}
```

Copy all the code from your Dijkstra class to your AStar class, so that they are identical (apart from their name, and the name of their constructors, of course).

A* is very similar to Dijkstra, but adds an extra "heuristic" value to each location **when choosing which open location to close next**. This helps to focus the search onto more likely pathways. The costs of nodes neighbouring a newly closed node are re-calculated using exactly the same formula as for Dijkstra:

new cost = cost of parent + 1 //for left,right,up,down
OR *new cost = cost of parent + 1.4 //for diagonals*

You do not need to change this!

To implement A*, you simply need to change the node selection process. The next node to be closed is given by choosing the open node with the lowest value of :

cost value + heuristic value

As in the lectures, we can calculate the heuristic value using the "city block" (or "Manhattan") method: the difference in X between the node and the player position + the difference in Y (make sure you use absolute differences – **we do not want negative values**!). A good approach is to pre-calculate the heuristic values, and store them in a array (like the array used to keep track of the node cost values).

When you have finished, change the Level class so that an AStar object is created rather than a Dijkstra object. Test your code.

Task B

Compare the path generated using A* with that using Dijkstra. You should notice some difference in the path, and also some difference in the spread of closed nodes generated by the respective algorithms (if you implemented the optional part of the last workshop). Although the spread is only slightly smaller for A*, the difference becomes significant for pathfinding on larger grid sizes.

Task C

You should now be able to create a bot which can follow the path generated by A*. Modify the AStar class so that the path coordinates are stored as a sequential list in a 1D array. The first item in the array should be the player location, and the last should be the bot location. In between should be the intermediate locations, in sequence. You can implement this by adapting the code that builds the "inPath" array.

The bot simply needs to read coordinates from this array in turn. It will therefore need one member variable which is an index into this list. The index should initially be set to the last entry in the location list (the bot location), and decremented on each call to *ChooseNextGridLocation()*. When it reaches zero, the bot will have finished following the path.

Task D

Of course, the player may move, and if this happens the path to the player will need to be updated. You could do this either by recalculating the path (calling *AStar::Build()*) whenever the player moves, or simply by just calling *Build()* every few frames.

Task E (Optional : Optimsation)

Our implementation of AStar is not very optimal. The loop inside *build()* searches for a new location to close on each iteration: to do this, it considers every location in the grid. For a small grid like ours this is ok, but for a large grid we would spend a lot of time searching through many locations unnecessarily.

A more optimal approach is to maintain a list of “open locations”. These are locations which have not yet been closed, but which have had their cost value re-calculated at least once. This is a quite a small subset of all the locations in the grid, but in fact we only need to search this list to find the next location to close.

The list should be initialised by first adding only the bot location. Each time a neighbour has its cost location recalculated it would need to be added to the open list (if not already on it). When a location is closed, it would need to be removed from the open list, as well as being marked as closed in the existing closed array. There is some additional processing to maintain the open list, but for large maps (eg 1000x1000) there is a big saving in computational time when searching for the next location to close.