Path Finding Documentation

Jarrod Dowsey
Advanced Diploma in Professional Game Development
Assessment 3 (Game Artificial Intelligence)

Overview

For this assessment it is required that we demonstrate the use of Path Finding techniques such as the A* Algorithm and the Dijkstra's Algorithm. These path finding algorithms need to be used to create an AI Entity within the project that can find Endpoints from any given Start point and/or the shortest paths between any given points on a graph.

Structure of Graph Data

Three Methods of Storing Nodes/Edges:

- ***Using a separate list of nodes and edges/connections.***

Nodes when created would be stored in a List of Node pointers and then their neighbours would be determined and added to another list of node pointers.

```
std::list< Node* > m_graphNodes;
std::list< Node* > m_connections;
```

- ***Using a list of nodes and each node contains a list of edges/connections.***

When Nodes are created they are also stored in a list of Node pointers but in this case each Node has its own list of Node pointers that contain the edge/connection data.

```
std::list< Node* > m_graphNodes;
class Node
{
	std::list< Node* > m_neighbours;
};
```

- ***Using an adjacency matrix to store data about nodes and their neighbours.***

Nodes would be stored in an adjacency matrix along with it's neighbours.

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 & 1 \\
3 & 1 & 0 & 1 & 0 & 1 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

*A **1** means yes it is a neighbour of that node. A **0** means it is not a neighbour to that Node.*

Graph Data Algorithms

Finding a Node in the Graph Based on a Condition.

*For my assessment I have 400 nodes on the screen and at run time you select one start point and then you have the choice of selecting up to three **possible** end nodes in which Dijkstra's algorithm will be used to find the closest one and return the shortest path to it.*

The condition on the nodes is simply whether or not it has been selected to be one of the possible end nodes. If it has been selected it is stored in a vector of Node pointers or in the case of my project, tile pointers.

```
if((*itr)->m_pressed == true && m_rightClicked == true && m_howManyEndTiles <= 3 && m_coolDown < 0.0f)
 {
        m_possibleEnds.push_back( (*itr) );
}
```

The above code is the code i use to say if the mouse is right clicked on any given node add that node to the vector of possible end points.
- m_pressed and m_rightClicked are 2 checks I created to check if the mouse is over nodes and has been right clicked.
- m_howManyEndTiles is the check to make sure there isn't already 3 possible end tiles.
- m_coolDown is just there to stop multiple clicks from happening at the same time on the same node/tile.
- m_possibleEnds is the vector used to store the possible end nodes.

Connecting Two Nodes Within the Graph.

Nodes within the graph can be connected in various different ways. One of these ways would be to give each node a neighbour(s).

This method as simple as it seems can have downfalls when it comes to multiple nodes trying to connect to each other, what i mean by this is, if 20 nodes are all within 20 pixels of each other they will all be added as neighbours of each other and will all have connecting edges meaning each node has far too many neighbours and needs to be cut down for the efficiency of the program.

A fix to this problem is to simply put a limit on the amount of nodes that are allowed to be added to the neighbour list of ny other given node, for example limiting this number to 4 so that each node can only have a maximum of 4 neighbours.

Adding Nodes to the Graph.

The nodes can be added multiple different ways but for mine i have gone with 'Random Generation' and 'Add Your Own" methods.

In my random generation mode i add the nodes to the screen and give them a random value to move to in the x and y axis. The random amount that the nodes can move when first created and drawn on the screen isn't big enough to land nodes on top of each other which allows me to avoid problems with nodes drawing on top of each other. The nodes are then connected if they have neighbours within a certain distance.

```
for(int x = 0; x &lt; TILE_X_RAND; x++)
{
   for(int y = 0; y &lt; TILE_Y_RAND; y++)
   {
      m_xOffset = rand() % 50;
      m_yOffset = rand() % 50;

      float xPos = x * ((m_pGame-&gt;GetWindowWidth() - 30)/TILE_X_RAND) + m_xOffset;
      float yPos = y * ((m_pGame-&gt;GetWindowHeight() - 30)/TILE_Y_RAND)+ m_yOffset;

      AddTile(xPos + 25, yPos + 20);
   }
}
```

The above code snippet is the code that creates the spawn points of each node on the screen.
- TILE_X_RAND and TILE_Y_RAND are variables that set the amount of tiles that will be on each axis.
- m_xOffset and m_yOffset are the variables used to set the random distance that the nodes will offset from there start point, in this case its a random value between 0 and 50 in each axis.
- AddTile is the function written that creates a tile and moves it to it's position.

For my Add your Own method a node is added to the screen on a mouse click and only if there is a certain distance between it and the last one placed to ensure nodes cannot be placed on top of each other causing problems with connections of nodes and their neighbours.

```
double xPos, yPos;
 if( m_mouseClicked == true && m_howManyTiles <= 500.0f && m_coolDown <= 0.0f)
 {
          glfwGetCursorPos(m_pGame->GetWindow(), &xPos, &yPos);
          AddTile(xPos, yPos);
          m_howManyTiles += 1.0f;
          m_mouseClicked = false;
 }
```

The above code snippet shows how nodes are added to the screen in the create your own mode.
- m_mouseClicked is a true or false that determines whether or not the mouse has been clicked.
- AddTile is a function that adds the tiles to the screen.
- m_howManyTiles is a variable that increases every time a tile is created to ensure a maximum of 500 tiles can be on the screen at once.
When the mouse is clicked the location is tracked and that becomes the x and y position for the tile to be added to the screen, once this is completed the program will move through to draw the tile where it is currently located on the screen.

In both methods I use distance and whether or not a node is a neighbour to determine whether or not the nodes will be connected. If the nodes are within the correct distance and they are neighbours of each other they will automatically be connected.

(Please note I no longer use the add your own method within my project but have decided to keep it in the documentation as an alternate way of adding nodes to the screen).

Finding Neighbours of a Given Node.

In my project I determine neighbours through the use of distance. What I am referring to is if, for example, there is a node that is within 20 pixels of a another node it would be added to the nodes list of neighbours.

As previously mentioned these nodes would be stored in lists and returned through the use of custom iterators. Neighbours in the graph determine which nodes have connections to each other, if a node isn't a neighbour with another node then there is no connection between those two nodes. I believe it is also worthy of taking note that i included a neighbour limit meaning each node has a maximum number of neighbours.

The below code demonstrates how i determine neighbours within my program.
- I loop through the entire vector of tiles and then search for nodes within the correct range to be neighbours
- The equation is used to determine length between nodes.
- If there are any nodes within the correct range i then add them (push_back) to the list of neighbours.

```
for(int i = 0; i < m_tiles.size(); i++)
{
 for(int j = 0; j < m_tiles.size(); j++)
 {
		Vec2 lengthBetween = m_tiles[j]->Transform()->GetTranslation();
		lengthBetween.x -= m_tiles[i]->Transform()->GetTranslation().x;
		lengthBetween.y -= m_tiles[i]->Transform()->GetTranslation().y;
		if(j != i)
		{
			if(m_tiles[i]->m_neighbours.size() < 4)
			{
```

```
                    if(m_tiles[j]->m_neighbours.size() < 4)
                    {
                            if(lengthBetween.Len() < 65.0f)
                            {
                                    m_tiles[j]->m_neighbours.push_back(m_tiles[i]);
                                    m_tiles[i]->m_neighbours.push_back(m_tiles[j]);
                            }
                    }
            }
        }
}
}
```

How can this Data be Saved and Loaded from File.

The data could be saved in many different ways and many different things could be saved. A simple thing to save would be the graph itself, what I mean by this is the positions of every node on the screen could be saved to file and then those position could then be re loaded back into the project and the nodes would be in the same positions that were saved out from.

File Input Output would be a simple way of saving these values out to a text file.

Path Finding

How Have You Implemented Dijkstra's Algorithm and How is it Calculated?

I have implemented Dijkstra's Algorithm through the use of having start and finishing nodes/ nodes with certain conditions. My implementation of the Dijkstra's Algorithm searches through my graph by traversing through nodes and their neighbours until an end node is reached.

Firstly it determines the start node and then from there adds it to a closed list and processes it to gather the information about its neighbours.

```
while(m_priorityList.empty() != true)
 {
        currentTile = m_priorityList.back();
        m_priorityList.pop_back();
        m_closedList.push_back( currentTile );

        /// Processing of Current nodes.
        for(auto itr = currentTile->m_neighbours.begin(); itr != currentTile->m_neighbours.end(); itr++)
        {
                bool canAddNode = true;

                ///Checking for node on closed list
                for(auto iter2 = m_closedList.begin(); iter2 != m_closedList.end(); iter2++)
                {
                    if( (*itr) == (*iter2))
                    {
                        canAddNode = false;
                        break;
                    }
                }

                ///Checking for node on open list
                for(auto iter2 = m_priorityList.begin(); iter2 != m_priorityList.end(); iter2++)
                {
                    if( (*itr) == (*iter2))
                    {
                        canAddNode = false;
                        break;
                    }
                }

                if( canAddNode )
                {
                    m_priorityList.push_back((*itr));
                }
        }
```

The above code is how the current node being processed is added to the closed list and how its neighbours are then processed and added to the Priority List.

*From there the neighbours are then added to a priority list if they aren't already on one and are then sorted through the use of **G Score**. The G Score is calculated for each node and is determined by its distance from its parents node added with the already present G Score of the parent node. The smallest G Score will be sorted as the top of the priority queue.*

```
float distanceBetweenTiles =  Vec2(currentTile->Transform()->GetTranslation().x - (*itr)->Transform()->GetTranslation().x,

currentTile->Transform()->GetTranslation().y - (*itr)->Transform()->GetTranslation().y).Len();

float tempGValue = currentTile->GetGScore() + distanceBetweenTiles;

(*itr)->SetGScore(tempGValue)
```

The above code is the code used to determine the G values of nodes. I get the distance between the two Nodes that are neighbours and then set that distance to the current G Score and add on any previous G Score the parent node may have had.

The process continues to repeat until the ending node is found. Once the ending node is found the process in a way reverses itself. Starting at the ending node all of the parents of each node used to find the end node are added to a list that is the shortest path list. The nodes that are saved into this list are the connections needed to show the shortest path from the starting point to the ending point.

```
if((*itr) == end)
{
 (*itr)->m_active = true;
 currentTile = (*itr);
 endFound = true;

 Tile* current = end;

 while( current->GetParent() != start)
 {
		outPath.push_back( current );
		current = current->GetParent();
 }
break;
}
```

The above code is the process used to add the parent nodes to the list that determines the shortest path between start and end points. It checks to see if the end node has been found and if it has the while loop starts and all the parents that are along the path are added to a list that is later used to draw the shortest path.

From there i just draw the connections between the nodes on the list in a different colour so it is clear what the shortest path is between start and end nodes.

How is the Path Stored?

*The shortest path is stored by adding the parents of each node that had the lowest **G Score.** Those parent nodes are pushed onto the list and then when it comes time to draw the shortest path the nodes on that list are recalled and i change the colour of the lines between the nodes that are on the list.*

```
while( current->GetParent() != start)
{
 outPath.push_back( current );
 current = current->GetParent();
}
```

OutPath is the list that the parent nodes are sent to for the path to be later recalled.

How have I Modified the Dijkstra’s Algorithm to use the A Algorithm?*

There are only a few small changes that needed to be made to the Dijkstra’s Algorithm for it to change into using the A Algorithm.*

*The main thing that needed to be changed was the way the neighbours are sorted when being processed. While using the Dijkstra’s Algorithm the neighbours are sorted by the **G Score** which is ultimately the distance between two nodes. For the sort function to be changed two new scores needed to be added, the **H Score** and **F Score**. The H score or Heuristic is an estimate of the distance from the current node and the finish node and the F score is the G Score and the H Score added together. The sort function for A* uses the F score instead of the G score.*

Another thing that is only a small change for the A algorithm is that it required me to use only one single finish point instead of m=having multiple possible end points.*

```
m_priorityList.sort( [](Tile* const a, Tile* const b)
{
        if(a->GetFScore() == b->GetFScore())
         return a->GetId() > b->GetId();

         return a->GetFScore() > b->GetFScore();
});
```

The above code snippet is the sort function used within the A Algorithm search.*

What are Some Scenarios that you could use the A Algorithm in?*

The A Algorithm is used to find the shortest path between a start point and a **known** finish point. This can be useful in situations such as making an enemy close in on a player using the shortest possible path and on a bigger scale it could be used to draw a GPS line across an entire map showing the shortest possible route to your destination (“Grand Theft Auto 5” and “Watch Dogs” are good examples of this).*

The A Algorithm is a much faster process as it doesn’t require searching as many nodes as the Dijkstra’s Algorithm would need to search. A negative for this algorithm is it requires a finish point to run.*

What are Some Scenarios that you could use the Dijkstra's Algorithm in?

Dijkstra's Algorithm is used to find the shortest path to an object when the location of this object is ***unknown.*** *An example of when this can be used would be in a program the user is in need of health and the Dijkstra's Algorithm would be used to find the closest health pack to the user. The Dijkstra's Algorithm can be used in various different scenarios given that the location of objects are unknown.*

A positive about the Dijkstra's algorithm is it can search for things that are closest to you instead of needing a specific finish point. A downside to the Dijkstra's algorithm is it is a much slower search because it searches in every direction instead of toward the possible ends.

REFERENCES:

- AIE PDF'S
- http://www.policyalmanac.org/games/aStarTutorial.htm