Path Finding Documentation

Iain Dowling *Advanced Diploma of Professional Games Development*
Assessment 3 (Game Artificial Intelligence)

Overview

For this assessment we are required to demonstrate knowledge of various artificial intelligence pathfinding techniques and behaviours, as well as implementing said techniques into our Tiny Tanks game.

Structure of Graph Data

- **Node and edge objects**

  The nodes and edges are both objects. The nodes contain an array of edges, and each edge contains a pointer to another node.

  Container type:

  The best container for this type of structure would be a vector. Multiple vectors would need to be used, one to store the nodes, and one for each node to store its edges. The use of vectors would allow addition of new nodes and new edges, and fast referencing, however risks nots being cache coherent.

Code snippet:

```
void Graph::CreateGraph(int width, int height)
{
    //width = number of columns
    //height = number of rows

    for (int i = 1; i < width + 1; i++)
    {
        for (int j = 1; j < height + 1; j++)
        {
            //create a new node on the heap
            GraphNode* node = new GraphNode(Vector2((m_game->GetScreenWidth() / (width + 1)) * i,
                                                    (m_game->GetScreenHeight() / (height + 1)) * j));
            //push the node into the vector (m_nodes)
            m_nodes.push_back(node);
        }
    }

    //determines the max distance that 2 nodes must be from each other in order to connect
    float radius = sqrt(((m_game->GetScreenHeight() / height + 1) * (m_game->GetScreenHeight() / height + 1))
                        + ((m_game->GetScreenWidth() / width + 1) * (m_game->GetScreenWidth() / width + 1)));

    for (int i = 0; i < m_nodes.size(); i++)
    {
        for (int j = 0; j < m_nodes.size(); j++)
        {
            //find the distance between two nodes
            Vector2 distance = m_nodes[i]->GetPosition() - m_nodes[j]->GetPosition();
            //check if the two nodes are within connecting distance
            if ( distance.GetLength() < radius && j != i)
            {
                //creates a new edge
                GraphEdge* edge = new GraphEdge();
                //sets the edge data
                edge->From = m_nodes[i];
                edge->To = m_nodes[j];
                edge->Weight = distance.GetLength();
                //pushes the edge into the nodes vector (m_edges)
                m_nodes[i]->AddEdge(edge);

            }
        }
    }
}
```

Pros and cons:

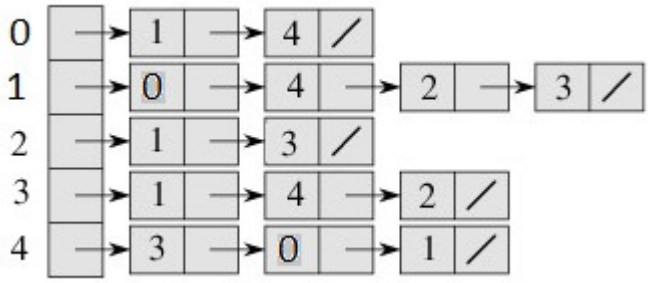
Easy to create and to access. Adding and deleting nodes and edges is easy. However, this method takes up lots of memory and is not cache coherent.

- **Adjacency List.**

  An array of Linked lists. Each list in the array shows the connections for one node to other nodes.

  The example on the right shows the connections for a graph of 5 nodes.

  The first node in the Graph, represented by a 0, is connected to the 2nd and 5th nodes, represented by 1 and 4 respectively.



  Container type:

  Depending on whether the Graph is dynamic or not, an array of linked lists or a vector of linked lists could be used to represent the adjacency list:

  - An array of Linked lists would be the most memory efficient option for a graph that is not dynamic. As such, the array would not be expected to resize and so the data contained will stay uniform in memory. The array would allow fast retrieval of data via the array indexing operator.

  - A Vector of Linked lists would be the better option for a graph that is expected to be dynamic as it would allow for the addition and deletion of nodes via its resizing capabilities. The drawback is that each time the vector resizes, the data inside will be shuffled around in memory, making accessing it more memory inefficient. Retrieving data from a vector is as easy as using an array as the vector can also make use of the array indexing operators.

  - Linked lists would be the best option to hold the connection data for both arrays and vectors as they are dynamic and would allow new connections to be made and old connection to be deleted easily.

Code snippet:

```
//graph struct that holds array of linked lists
struct graph
{
    int sizeOfGraph;
    struct adjList* connectingNodes;
};

//head of the linked list
struct adjList
{
    struct adjListNode* headNode;
};

//nodes in the graph
struct adjListNode
{
    //node data
    int data;
    //pointer to the next node in the list
    struct adjListNode* nextNode;
};
```

```
struct adjListNode* NewAdjListNode(int a_data)
{
    //create a new node on the heap
    adjListNode* node = new adjListNode;
    //set its data
    node->data = a_data;
    //set nextNode as a null pointer
    node->nextNode = nullptr;
    return node;
}

struct graph* CreateGraph(int size)
{
    //create a graph on the heap
    graph* nodeGraph = new graph;
    nodeGraph->sizeOfGraph = size;

    //create an array to house the linked lists
    nodeGraph->connectingNodes = new adjList[size];

    //set the head of each list as a null pointer
    for (int i = 0; i < size; ++i)
        nodeGraph->connectingNodes[i].headNode = nullptr

    //return the graph
    return nodeGraph;
}
```

Pros and cons:
Adding and removing nodes is easier with this method. This method is also more memory efficient for larger graphs with lots of connections. However it is harder to implement and accessing connections requires looking through an entire linked list that could potentially be hundreds of connections large.

- **Adjacency matrix.**

  A 2D array of size [j ^2] where [j] equals the number of nodes in the graph.
  An intersection between a column and row shows the relationship between the nodes with the respective column/row index.

  The example on the right shows the connections for a graph of 5 nodes.

  The 1 in the ‘row 2’, ‘column 3’ intersection shows that there is an edge between the 3rd and 4th nodes in the Graph.

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Container type:
Depending on whether the Graph is dynamic or not, an array or vector could be used to represent the adjacency matrix:

- A 2D array of nodes would be most memory efficient for a graph that is not dynamic, and as such cannot be expected to resize. It would also allow for fast retrieval of connections data with the use of the array indexing operator.

- A 2D vector would be more effective for a dynamic array, as it can resize to account for the creation or deletion of new nodes. A vector also allows the use of the array indexing operator so that retrieving connection data would be fast. However, every time the vector resizes its memory will be shifted around, making it less efficient the more it resizes

Code snippet:

```
int matrixSize = 5;
//create an array with 5 rows and columns
bool matrix[matrixSize][matrixSize];

for ( int i = 0; i < matrixSize; i++)
{
    for ( int j = 0; j < matrixSize; j++)
    {
        //i = rows, j = columns
        if ( /* connection should exist */ )
            //set connection to true
            matrix[i][j] = true;
        else
            //set connection to false
            matrix[i][j] = false;
    }
}
```

Pros and cons:

This method is easy to implement and to understand and querying for connections is fast. Adding and removing connections is also fast and easy. However adding and removing nodes can be time consuming and larger graphs with many nodes can take up large amounts of memory. Also requires memory for edges that don't exist.

Graph Data Algorithms

The following algorithms will be based off the graph and edge object method for storing graph data.

- **Finding a node in the graph based on a condition.**

The following code will look through a vector of nodes, named m_graph, and will return true if a node is found at the coordinates that are passed in. Otherwise, it returns false.

```
bool IsNodeAtCoords(Vector2 coords)
{
    for ( int i = 0; i < m_graph.size(); i++)
    {
        if ( m_graph[i]->GetPosition() == coords )
        {
            return true;
        }
    }
    return false;
}
```

- **Connecting 2 nodes within the graph**

The following code searches through the entire vector of nodes and checks the distance between them, creating an edge between them if they are close enough.

```
//determines the max distance that 2 nodes must be from each other in order to connect
float radius = sqrt(((m_game->GetScreenHeight() / height + 1) * (m_game->GetScreenHeight() / height + 1))
                  + ((m_game->GetScreenWidth() / width + 1) * (m_game->GetScreenWidth() / width + 1)));

for (int i = 0; i < m_nodes.size(); i++)
{
    for (int j = 0; j < m_nodes.size(); j++)
    {
        //find the distance between two nodes
        Vector2 distance = m_nodes[i]->GetPosition() - m_nodes[j]->GetPosition();
        //check if the two nodes are within connecting distance
        if ( distance.GetLength() < radius && j != i)
        {
            //creates a new edge
            GraphEdge* edge = new GraphEdge();
            //sets the edge data
            edge->From = m_nodes[i];
            edge->To = m_nodes[j];
            edge->Weight = distance.GetLength();
            //pushes the edge into the nodes vector (m_edges)
            m_nodes[i]->AddEdge(edge);
            m_edges.push_back(edge);
        }
    }
}
```

This method will cause 2 edges to exist between every node, however this has no ill effect other than using more memory than what's optimal.
A possible error could occur if a node tries to connect an edge to itself, however the code above deals with that situation.

● **Adding Nodes to the graph**.

This code shows how nodes are added to the graph when the graph is being created.

```
//width = number of columns
//height = number of rows

for (int i = 1; i < width + 1; i++)
{
    for (int j = 1; j < height + 1; j++)
    {
        //create a new node on the heap
        GraphNode* node = new GraphNode(Vector2((m_game->GetScreenWidth() / (width + 1)) * i,
                                                (m_game->GetScreenHeight() / (height + 1)) * j));
        //push the node into the vector (m_nodes)
        m_nodes.push_back(node);
    }
}
```

Using this method, no two nodes will be created on top of each other. However, 2 nodes existing on each other is a possibility, except that it will cause no issue with the graph and path-finding algorithms.

Edges are created automatically is a graph is generated, however they are not added if a node is created after the graph is generated and they will need to be created manually.
E.g.

```
void AddEdge(GraphEdge* edge)
{
    for ( int i = 0; i < m_graph.size(); i++)
    {
        if ( m_graph[i] == edge.From )
        {
            m_graph[i].AddEdge(edge);
            continue;
        }
    }
}
```

This code looks through all the nodes in the vector and adds the edge to the node that matches its 'From' pointer.

● **Finding Neighbors of a given node**

To find Neighbours of a given node I loop through the entire vector of nodes and check the distance between the nodes in the vector and the given node.

```
//determines the max distance that 2 nodes must be from each other in order to connect
float radius = sqrt(((m_game->GetScreenHeight() / height + 1) * (m_game->GetScreenHeight() / height + 1))
                    + ((m_game->GetScreenWidth() / width + 1) * (m_game->GetScreenWidth() / width + 1)));

for (int i = 0; i < m_nodes.size(); i++)
{
    for (int j = 0; j < m_nodes.size(); j++)
    {
        //find the distance between two nodes
        Vector2 distance = m_nodes[i]->GetPosition() - m_nodes[j]->GetPosition();
        //check if the two nodes are within connecting distance
        if ( distance.GetLength() < radius && j != i)
        {
            //creates a new edge
            GraphEdge* edge = new GraphEdge();
            //sets the edge data
            edge->From = m_nodes[i];
            edge->To = m_nodes[j];
            edge->Weight = distance.GetLength();
            //pushes the edge into the nodes vector (m_edges)
            m_nodes[i]->AddEdge(edge);
            m_edges.push_back(edge);
        }
    }
}
```

The neighbours are then connected via an edge. The edges are stored in a vector belonging to the given node searching for neighbours. The edges can then be accessed from the node using a function such as:

```
vector<GraphEdge*> GetEdges()
{
        return m_edges;
}
```

● **How can this data be saved and loaded from file.**

This data can be stored in a text document.

The best and fastest way to store the data would be as a series of vector2 positions. In this way, the game could load in these vector2 positions, create nodes at each of those positions, then generate edges between nodes depending on the distance between them.

A vector2 would first need to be split into two floats, its x and y position, then converted to a string for input into a file. Code to convert would look similar to this:

```
std::string Convert (float number)
{
    std::ostringstream buffer;
    buffer<<number;
    return buff.str();
}
```

Path Finding

Dijkstra's Algorithm:
In my tanks game, the enemy tank will use the Dijkstra's search algorithm to path towards the players tank.

```
bool Graph::SearchDJK(GraphNode* start, GraphNode* end)
{
    std::vector<GraphNode*> open;
    std::vector<GraphNode*> closed;
    std::map<GraphNode*, GraphNode*> parent;
    std::map<GraphNode*, float> runningCost;

    open.push_back(start);
    runningCost[start] = 0;

    while (!open.empty())
    {
        //choose item with lowest gScore
        GraphNode* current;
        float lowestG = runningCost[open[0]];
        current = open[0];
        for (int  i = 0; i < open.size(); i++)
        {
            if ( runningCost[open[i]] < lowestG)
            {
                lowestG = runningCost[open[i]];
                current = open[i];
            }
        }

        //close node and pop from queue
        closed.push_back(current);
        auto i = std::find(open.begin(), open.end(), current);
        if (i != open.end())
            open.erase(i);


        //check if current node == end node
        if ( current == end )
        {
            m_path->push_back(ReconstructPath(parent, end));
            m_path->push_back(end);
            return true;
        }
```

This code is the first half of the Dijkstra algorithm and will find the node in the open list with the lowest g-score, add it to the closed list and process it.

```
        //add children to open stack
        for ( int j = 0; j < current->GetEdges().size(); j++)
        {
            GraphNode* child = current->GetEdges()[j]->To;

            if (!(std::find(closed.begin(), closed.end(), child) != closed.end()))//does this work
            {
                float childRunningCost = current->GetEdges()[j]->Weight + runningCost[current];

                if ( std::find(open.begin(), open.end(), child) != open.end())
                {
                    auto i = std::find(open.begin(), open.end(), child);

                    if (childRunningCost < runningCost[(*i)])
                    {
                        open.erase(i);
                        runningCost[child] = childRunningCost;
                        parent[child] = current;
                        open.push_back(child);
                    }
                }
                else
                {
                    runningCost[child] = childRunningCost;
                    parent[child] = current;
                    open.push_back(child);
                }
            }
        }
    }
    return false;
}
```

This code is the second half of the Dijkstra search. It will loop through the current nodes children adding any that aren't in the closed list to the open list. It will also check the nodes children with those in the open list, replacing them if the running cost is lower.

As can be seen above, if the end node is found, a new function is called that reconstructs the path and puts it inside a vector<GraphNode*>*. This allows easy access to the path via the array indexing operators, and will not affect how the nodes are stored in the heap.
Retrieving the path is also easy with the use of a function such as the following:

```
vector<GraphNode*> GetPath() const;
```

Which will return m_path (the vector holding the path of nodes).

The Dijkstra algorithm could also be used to path to health packs or other types of power-ups. The end node does not need to be known, the algorithm just needs to know what its looking for.

Path Smoothing:

One method that could be used to smooth a path would be to use Bezier curves. This would remove the jagged nature of the path and create much nicer and natural looking movement.

Another method would be to use post-processing smoothing operations. These operations would examine the path after it has been created and remove unnecessary nodes to create shorter and better looking paths.

A* Algorithm:

In order to convert Dijkstra to A*, I changed the running cost/g-score checks to f-score checks, which a heuristic estimate as well as the running cost.

```
std::map<GraphNode*, float> gScore;
std::map<GraphNode*, float> fScore;

open.push_back(start);
gScore[start] = 0;
Vector2 heuristicEstimate = end->GetPosition() - start->GetPosition();
fScore[start] = gScore[start] + heuristicEstimate.GetLength();
```

The above shows that f-score is the sum of the heuristic estimate and the g-score.

```
//choose item with lowest Score
GraphNode* current;
float lowestF = fScore[open[0]];
current = open[0];
for (int  i = 0; i < open.size(); i++)
{
    if ( fScore[open[i]] < lowestF)
    {
        heuristicEstimate = end->GetPosition() - open[i]->GetPosition();
        lowestF = gScore[open[i]] + heuristicEstimate.GetLength();
        current = open[i];
    }
}
```

This code shows the f-score being used in place of the g-score when finding the next node to process.

I will use the A* path finding algorithm to path to a node that is closest the the mouse when the left mouse button is pressed. It could also be used to path to the player tank or to other random locations, as long as the end node is known.

Pros and cons of the different path finding algorithms:

Dijkstra is most useful when searching for a node based on a condition, and does not need to know the exact node it is looking for. However it this search generally takes longer to complete and searches through more nodes than its alternative, A*.

A* is a very fast search algorithm that will always find the shortest path to a node, as long as a path exists. However, A* cannot be used if the end node is not known, and so cannot search for a node based on a condition.

References

http://en.wikipedia.org/wiki/B%C3%A9zier_curve

http://gamedev.stackexchange.com/questions/26543/smoothing-found-path-on-grid

http://gamedevelopment.tutsplus.com/tutorials/create-custom-binary-file-formats-for-your-games-data--gamedev-206

http://www.daniweb.com/software-development/cpp/threads/146718/conversion-from-float-to-string

http://www.seas.gwu.edu/~simhaweb/alg/lectures/module7/module7.html

http://sun.iwu.edu/~sander/CS255/Notes/AdjLists.html

http://www.geeksforgeeks.org/graph-and-its-representations/