AI - Path Finding

Introduction

So far, we've looked at how an AI agent within our games can make fundamental decisions on what cause of action to take next, using variations on finite state machines. At some point in most games, though, an AI agent is going to want to move somewhere in the world - chasing the player, finding health packs, or being directed to move from one base to another. Such behaviour requires additional knowledge about the world within which the AI is operating, so that the AI knows which regions of the world can be moved in, and which are impassible (or perhaps undesirable) to move in. Usually, we also want an AI agent to move in the most efficient manner possible - if there's a lift from the top floor to the bottom, why take the stairs?

In this tutorial, we're going to take a look at the process of *pathfinding*, and how game levels can be represented as nodes in a graph, and how the $A*$ path searching algorithm can return accurate paths in an efficient manner.

Pathfinding

There are a variety of reasons why we might need to calculate a path from point A to point B in our game environments:

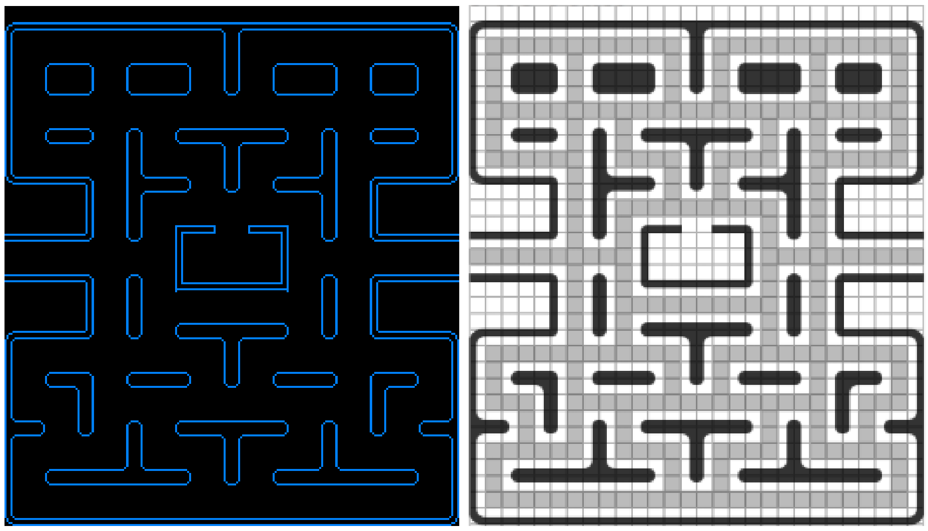
- The ghosts eyes heading back to base after *Pac-Man* has eaten them.
- The bad guys in *Deus Ex* or *Metal Gear Solid* heading toward the place where an alarm has been tripped, or the player has been spotted.
- Yorda running to the player when called for by *Ico*.
- The golden thread showing where to go next in *Fable*.
- The route plotted on the map to the desired destination in *Red Dead Redemption*.

In order to perform any of these tasks in our game code, we are going to have to implement some sort of pathfinding component, so that we can ask the game world what the best possible path from one position to another is. This process consists of three main components:

- Represent the environment as a graph of small navigable units or nodes.
- Find a route connecting a series of these nodes from the starting point to the target location.
- Move the AI agent along that route convincingly.

Representing the environment

We can represent a game world as a series of interconnected areas - rooms connect to corridors and hallways, which may then connect to an outside area, and so on. Perhaps even within a single room there could be multiple areas - the desks in the computer lab present obstacles that must be moved around to get from one end to another. To represent either of these concepts in our game world, we need to think of a way of subdividing a world into areas that are traversable, and those that are not. One such method could be to subdivide the entier play area up into a grid of regular shapes, like squares or hexagons. Here's an example of doing this, for the play area of Pac-Man:



On the left we have the classic Pac-Man maze, as it was in the original arcade game. On the right we have the same maze, but as a grid - black regions are the impassable walls, grey are the paths that Pac-Man and the ghosts can take, and white are areas completely outside of the play region. In this example, the idea of a grid of squares maps very well onto the original game maze, as it was actually drawn on the screen as a set of square tiles. This doesn't mean that pathfinding can only be done on tile based-grids - the roads in a racing game can be represented as a series of square areas in a completely polygonal world.

While the Pac-Man maze above is easily thought of as a grid, its also clear that there are impassable regions, and that there is therefore not necessarily free movement from one node to any of its neighbours. This *could* be represented as a 2D array of booleans, with 'true' for passable areas and 'false' for walls, but even in a game as simple as Pac-Man this wouldn't work properly all of the time. Why is this? At the left and right hand side of the Pac-Man maze there's a tunnel - if Pac-Man goes down the left hand side tunnel, he immediately reappears on the right, and vice-versa.

To represent the maze in Pac-Man, tunnel and all, we can instead model the whole thing as a *graph*; each square of the maze is a node, and if Pac-Man can travel from that node to another, we can say that there's an *edge* between the nodes. By doing this, we can free up the spatial position of the nodes from whether it is connected to the node or not, so now the left tunnel and right tunnel can be connected with an edge, even though they're on opposite sides of the maze. Perhaps the edges in a grid representation aren't necessarily bi-directional - you can only really travel in one direction on an escalator, so perhaps there's an edge indicating that its possible to go from bottom to top, but not back down again.

The A* Algorithm

In order for a ghost to find and eat Pac-Man, there needs to be some way in which the graph of the maze can be traversed, from the ghost's node, to Pac-Man's node. In the Pac-Man maze, there's always a path from point A to point B (there's no doors or escalators here!), so there's a multitude of different ways to get from the ghost's position to the position of the player. Ideally, the hungry ghost would find the fastest path from their node to Pac-Man's, so our pathfinding algorithm needs to find not just *a* path, but the *best* path.

The algorithm we will be looking at to determine the best path in a graph is the A* ('A star') algorithm, which has long been used within the games industry for its ability to quickly find good paths in a game world, and for its ease of adaption to different gameplay scenarios. A* is a *best-first* search, which means that when trying to find the best path from A to B, it will find the neighbouring node with the best score according to some metric that determines that node's suitability for the given path.

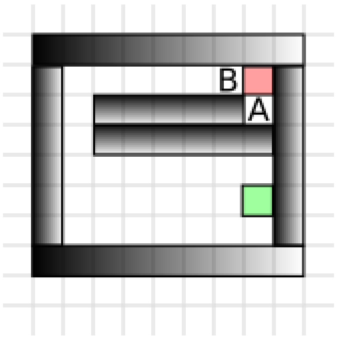
The metric by which A* determines which node to travel to next as it traverses the graph in the search for the best path is determined by a simple calculation:

$$f = g + h$$

The value g represents the cost of moving from the starting point to the considered node, following the currently assumed 'best' path. The h value represents the *heuristic*, and is a simple measurement of how far away the considered node is from the goal. It can only be a heuristic (meaning a guess!), as there could be obstacles or winding routes through the world to get from one point to another, but something as simple as the Euclidean distance from a considered node to the destination is a good indicator of which node should be examined next by the A* algorithm.

Characteristics of A*

At the most abstract level, the algorithm works by, for a given node, expanding to one of its neighbouring nodes (meaning there is an edge in the graph from one to the other) based on which node has the best f score, as described above, and then repeating this step until the destination is reached. Sounds logical, and perhaps even too easy! The algorithm gets more complicated, and to see why, let's look at the following basic maze:



Here, grey nodes are impassible walls, the red node is our starting point, and the green node is the destination. By visual inspection, we can see that of the two nodes that could be travelled to from the red starting node, that point A is closer to the green destination than point B (it has a better heuristic score), and so will be chosen by A* to move to next. However from node A there's actually no way of getting to the destination without travelling through point B, making point A actually point*less*, and so shouldn't be considered as a part of the 'best' path from red to green.

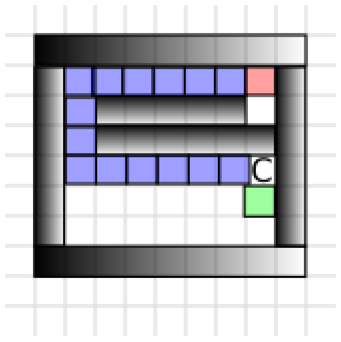
The open list and closed list

To represent this idea of trying out nodes that may not be part of the final path from one point to another, A* has the concept of an *open list* and a *closed list*, bot containing graph nodes that have at some point either been visited, or which may be visited. When the A* algoroithm reaches a node for the first time, and wants to try and determine which way to go, it places all of the neighbours of that node in the *open list*, calculates f for them all, and then picks the node with the best f value out of that list. So after one iteration of trying to solve the path above, nodes A and B would **both** be added to the open list, and then node A would be selected for further consideration, as it appears to be closer to the destination.

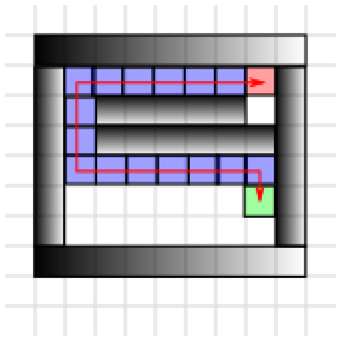
As A actually has no neighbours (other than the starting node...), the algorithm actually cannot continue down the path of A. Once a node has been processed (whether it is continued on with or not), it is placed on the *closed list*, which is essentially a history of considered nodes. However, there's still a node in the open list, B, so the algorithm can actually continue, by removing B from the open list and processing it - the node to the left of B will be added to the open list, and will then be considered the 'best' node as there's no other nodes.

Node parents

So far, we know that the closed list contains every node the algorithm has processed, and so also that eventually, some combination of some of those closed list nodes will form a path. Assume that in our example the algorithm has iterated several times, and has generated the following set of nodes:

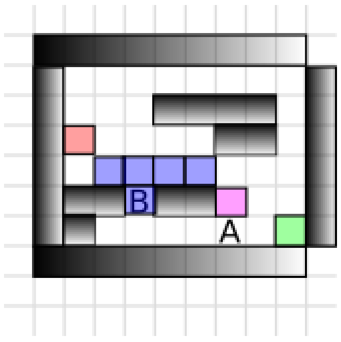


When node C is 'expanded', the destination will be added to the open list, as it is connected to C. If the destination node is ever found via expansion, then there *must* be a path! To actually calculate the path, a node in the graph has a concept of a *parent* node, as well as neighbours. This parent starts off as the node which was processed when the node was added to the open list. By following the chain of parent nodes from the destination, eventually the nodes which form the best path will be found:

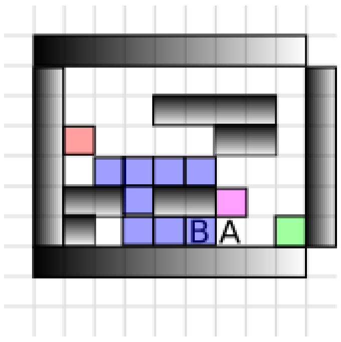


Updating node parents

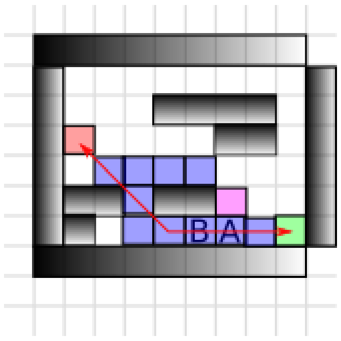
Sometimes, when expanding nodes through the graph, it could be that a node is visited again. Consider the following example maze :



The blue nodes represent nodes that have been added to the open list, and the purple node is the node currently about to be processed by the A* algorithm. As node A is a neighbour of the purple node, it should be expanded into the open list. In the next iteration of the A* algorithm, it picks the best node in the list. Perhaps, due to a combination of diagonal costs or some other metric, instead of picking *A* to be the best node in the list, node *B* is instead picked to be the best next step. The algorithm will then continue on, until eventually we get to this situation:



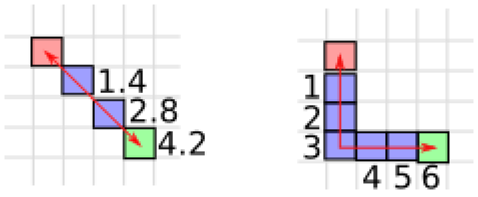
Following the path has reached the new point B, meaning that A is now to be added to the open list - but it's already *in* the open list! However, while its *h* score is still the same (it's still the same distance away from the destination that it was when it was first found!), the *g* score could be less, as the path to get to it via the bottom path is slightly shorter than the top path. In this situation, the 'parent' of node A should change from being the purple node, to being node B, and the node's *g* score updated, so that when the path is finally completed, the path is routed through B:



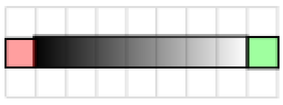
By updating the g score and parent whenever possible, we ensure that no redundant loops or wrong turns are included in the path - following the parents will *always* lead to the source node in the fastest route possible.

Edge traversal cost

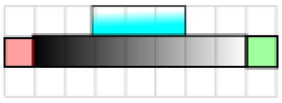
So far, although we've introduced the idea of their being a traversal cost to go from node to node, which gets summed into the *g* cost, we've not really elaborated on what this cost might be. In the Pac-Man maze, Pac-Man can only move in 4 directions - up, down, left, and right. So maybe the cost is always the same, maybe '1'. But what about games where diagonal movement is allowed? Perhaps the cost to move diagonally is about 1.4 (approximately the square root of 2). This means that it is more beneficial to move diagonally in the graph search:



We can therefore encode the difference between moving in straight lines and moving diagonally in each node by adding a cost to each edge. We can extend this concept further by considering additional costs for traversing a particular edge - maybe the path from node to node requires the AI character to swim, and is therefore slower and more dangerous than walking on ground? Consider the following simple grid area:



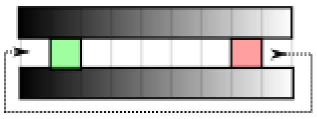
Seems that either going along the top path, or the bottom path will be exactly the same, and it doesn't matter which path is generated. But what if we added some 'water' tiles in?



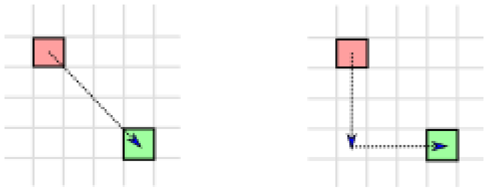
If we make each edge going into a 'water' node have a cost of 10 instead of 1, then the A* algorithm will *always* then generate the lower path. The pathfinding code doesn't 'know' what a water tile is, or any other details about the map, but as long as the edge cost has been encoded in the graph, the 'correct' result will always be generated by the A* algorithm.

Heuristic Difficulties

Now that we know more of the broad details of how the A* algorithm determines a path, let's think back to the Pac-Man game, where there was a tunnel that allowed Pac-Man to quickly loop around to the other side of the maze. We can reduce this down to a simple little test maze:



If we assume that the cost of going through the tunnel is '1', as with going to any other node, then clearly to get from the red node to the green node, the tunnel is the best path to take. But which path will A* actually take? Due to the A* f calculation, A* will actually go left through the maze, rather than right and through the tunnel. The heuristic calculation usually used in A* is either Euclidean distance (calculating the length of the vector from source to destination), or sometimes the Manhatten distance:



Unfortunately, the tunnel is a little bit 'non-Euclidean' in that going right actually makes you go left. This can be solved in a couple of different ways. One way could be to preprocess the grid, so that the h cost is stored per node, rather than calculated when the node is added to the open list, allowing it to be loaded up with the rest of the level data. To store the h cost to every node from every node can get quite expensive, though! An alternative would be to tag nodes as being 'special' and needing extra processing - perhaps if a node knows it is connected to a 'non-Euclidean' node, the calculated heuristic is insatead the distance from the *end point* to the goal, rather than the distance form the node itself. This only works if the node ever gets to be considered in the open list - if a path is expanded straight into the opposite direction, this wouldn't work.

If there is to be only one such 'special' node in the game world, the heuristic calculation could always take this into consideration, with the distance being the minimum of either the Euclidean distance to the goal, or the sum of the Euclidean distance to the 'special' node, plus the distance from the end point of the 'special' node to the destination. This would allow any AI character in the game world take into consideration whether it is worthwhile heading towards the magic tunnel in Pac-Man, or instead just go the more 'realistic' route.

Tutorial Code

To investigate the A* algorithm, we're going to expand the codebase to use it for some pathfinding. The code already has some classes relating to pathfinding; notably the **NavigationMap** class, which is an abstract class designed to represent anything that can provide a **NavigationPath** - a series of **Vector3s** representing waypoints that an AI can move through to get from one position to another in the game world:

```cpp
#pragma once
#include "../../Common/Vector3.h"
#include "NavigationPath.h"
namespace NCL {
   using namespace NCL::Maths;
   namespace CSC8503 {
      class NavigationMap  {
      public:
         NavigationMap()  {}
         ~NavigationMap() {}

         virtual bool FindPath(const Vector3& from, const Vector3& to,
                               NavigationPath& outPath) = 0;
      };
   }
}
```

NavigationMap class header

NavigationGrid Class

There's also a concrete implementation of a grid-based navigable area - **NavigationGrid**. As with the other classes we've looked at in this module, it's already defined for you, but several of its functions are currently empty. Here's its class declaration, from the **NavigationGrid.h** file:

```cpp
#pragma once
#include "NavigationMap.h"
#include <string>
namespace NCL {
   namespace CSC8503 {
      class NavigationGrid : public NavigationMap  {
      public:
         NavigationGrid();
         NavigationGrid(const std::string&filename);
         ~NavigationGrid();

         bool FindPath(const Vector3& from, const Vector3& to,
                       NavigationPath& outPath) override;
      protected:
         bool NodeInList(GridNode* n,
            std::vector<GridNode*>& list) const;
         GridNode* RemoveBestNode(std::vector<GridNode*>& list) const;
         float    Heuristic(GridNode* hNode, GridNode* endNode) const;
         int nodeSize;
         int gridWidth;
         int gridHeight;

         GridNode* allNodes;
      };
   } //end of CSC8503 namespace
} //end of NCL namespace
```

NavigationGrid class header

It stores a number of objects designed to represent a node in a grid-based graph, along with some currently empty functions - their usage in the A* algorithm should be fairly intuitive based on just their names.

GridNode Struct

You have been provided with a struct to represent a node in a graph suitable for pathfinding with. It's defined in the **NavigationGrid.h** file, and looks like this:

```cpp
struct GridNode {
   GridNode* parent;

   GridNode* connected[4];
   int         costs[4];

   Vector3      position;

   float f;
   float g;

   int type;

   GridNode() {
      for (int i = 0; i < 4; ++i) {
         connected[i] = nullptr;
         costs[i] = 0;
      }
      f = 0;
      g = 0;
      type = 0;
      parent = nullptr;
   }
};
```

NavigationGrid class header

Implementing A*

Each GridNode has a set of pointers to nodes it might be connected to, and the associated costs in navigating from one node to another. Along with that, we also have the per-node variables we need for the A* algorithm to work - we store a f and g value, along with a 'parent' node for storing the best path from one node to another, and a *position* to represent the centre of that node in the world coordinates of our virtual environment.

The first of our empty methods is *NodeInList*, which will take an **std::vector** of **GridNodes**, and return whether the passed in **GridNode** parameter is in the list or not. Add the following code, to replace the simple 'return false' that's currently in there:

```cpp
bool NavigationGrid::NodeInList(GridNode* n,
        std::vector<GridNode*>& list) const {
   std::vector<GridNode*>::iterator i =
        std::find(list.begin(), list.end(), n);
   return i == list.end() ? false : true;
}
```

NavigationGrid::NodeInList method

The *NodeInList* method is a pretty big indicator that we might be using an **std::vector** to represent our open and closed lists! We'll see a bit more evidence in our next 'empty' method, *RemoveBestNode*. Recall from the theory outlined earlier that in A* we maintain an open list, and iteratively try and find the best node within the open list to try and expand towards our goal node. To implement the algorithm, then, we'll need a method that can search through a **std::vector**, and find the GridNode with the lowest f cost - recall that f is the sum of g and h, and thus consists of the distance travelled to get to a node, and a prediction of how far away that node is from the destination. Add the following code to the empty *RemoveBestNode* method in the **NavigationGrid.cpp** file:

```
1  NavigationGrid::GridNode*  NavigationGrid::RemoveBestNode(
2           std::vector<GridNode*>& list) const {
3     std::vector<GridNode*>::iterator bestI = list.begin();
4  
5     GridNode* bestNode = *list.begin();
6  
7     for (auto i = list.begin(); i != list.end(); ++i) {
8        if ((*i)->f < bestNode->f) {
9           bestNode = (*i);
10          bestI    = i;
11       }
12    }
13    list.erase(bestI);
14    return bestNode;
15 }
```

NavigationGrid::RemoveBestNode method

We simply iterate through the passed in list, and keep track of which **GridNode** currently has the best *f* score, such that after the for loop has completed, bestI will *always* contain the best node - we can then remove it from the vector (line 13) and return the **GridNode** (line 14). Remember, *erasing* a pointer from a vector doesn't delete the object the pointer is pointing at!

For our example heuristic calculation, we're going to simply use Euclidean distance, so add in the following code to the *Heuristic* method, the usage of which we'll see shortly:

```
float NavigationGrid::Heuristic(GridNode* n1, GridNode* n2) const {
   return (n1->position - n2->position).Length();
}
```

NavigationGrid::Heuristic method

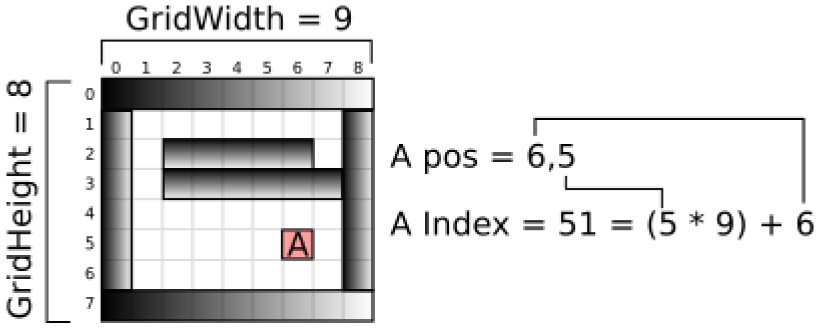
The real implementation of A* is within the *FindPath* method, which will taken in two world positions, and fill in a **NavigationPath**, returning true if a path can be found, and false if not. Start off the *FindPath* method by adding the following code:

```
bool NavigationGrid::FindPath(const Vector3& from,
         const Vector3& to, NavigationPath& outPath) {
   //need to work out which node 'from' sits in, and 'to' sits in
   int fromX = (from.x / nodeSize);
   int fromZ = (from.z / nodeSize);

   int toX = (to.x / nodeSize);
   int toZ = (to.z / nodeSize);

   if (fromX < 0 || fromX > gridWidth - 1 ||
      fromZ < 0 || fromZ > gridHeight - 1) {
      return false; //outside of map region!
   }

   if (toX < 0 || toX > gridWidth - 1 ||
      toZ < 0 || toZ > gridHeight - 1) {
      return false; //outside of map region!
   }
```

NavigationGrid::FindPath method

The *FindPath* method takes in world space positions, but internally operates on a grid of **GridNodes**; so we'll have to in some way work out which node equates to which position. To do so, we can divide the positions we receive by how many units in size each node will be - doing this for the x and z axes gets us two integers for each position (lines 4+5, and lines 7+8), which we can use as indices into an array of **GridNodes**. Before we continue on to the algorithm properly, we need to make sure these integers are actually valid for indices - if an AI is somehow placed outside of the world map, the quick calculations for from and to could end up being less than 0, or greater than how many nodes we have, which would mean we'd end up accessing invalid memory!

With the two indices per position started, we can then determine the actual start and end nodes, making *startNode* and *endNode*. The NavigationGrid class stores a flat list of GridNodes, we we need to turn our 2 coordinates into 1 index, by multiplying the Z value by how many nodes we have on each row, and then adding on how many into that row we want. This is pretty much exactly the same as what we did to determine indices back in the index buffering tutorial:



Now that the starting node and end node of the graph have been found, we can start to actually calculate an A* path between them. To represent the open and closed list outlined earlier, we're going to use std::vectors - these will work fine for demonstrating the algorithm, but you should start to think about which C++ containers would be a better fit for this task.

```
19    GridNode* startNode = &allNodes[(fromZ * gridWidth) + fromX];
20    GridNode* endNode   = &allNodes[(toZ * gridWidth) + toX];
21
22    std::vector<GridNode*>  openList;
23    std::vector<GridNode*>  closedList;
24
25    openList.emplace_back(startNode);
26
27    startNode->f = 0;
28    startNode->g = 0;
29    startNode->parent = nullptr;
30
31    GridNode* currentBestNode = nullptr;
```

NavigationGrid::FindPath method

We prime the open list with our starting node - the A* algorithm will then begin, by finding the best node current in the open list, and then operating on it and its neighbours. Each **GridNode** searched will have an f and g score, which may change over time if better paths to a node ar ever found. So, to start the algorithm off, we give our starting node an f and g score of 0 each - we didn't have to travel anywhere yet!

The next code is the A* implementation properly. While we have nodes left to consider in the open list, we find the 'best' node to remove from it using the *RemoveBestNode* method that we filled out earlier. At some point, if a path can be found between the start and end, then the destination node will be found within the open list - so on line 35 we check for this case. In such cases, we can start building up a route from the destination node back to the starting node by following the **GridNode** parent variables - they will always point to the best way to get to the next node in the path. By putting the node positions inside the *outPath* variable, we can return a set of 'waypoints' that an AI can aim torwards on its way to its destination.

```
32    while (!openList.empty()) {
33       currentBestNode = RemoveBestNode(openList);
34
35       if (currentBestNode == endNode) {//we've found the path!
36          GridNode* node = endNode;
37          while (node != nullptr) {
38             outPath.PushWaypoint(node->position);
39             node = node->parent; //Build up the waypoints
40          }
41          return true;
42       }
43       else {
44          for (int i = 0; i < 4; ++i) {
45             GridNode* neighbour = currentBestNode->connected[i];
46             if (!neighbour) { //might not be connected...
47                continue;
48             }
49             bool inClosed  = NodeInList(neighbour, closedList);
50             if (inClosed) {
51                continue; //already discarded this neighbour...
52             }
53
54             float h = Heuristic(neighbour, endNode);
55             float g = currentBestNode->g + currentBestNode->costs[i];
56             float f = h + g;
57
58             bool inOpen    = NodeInList(neighbour, openList);
59
60             if (!inOpen) { //first time we've seen this neighbour
61                openList.emplace_back(neighbour);
62             }
63             //might be a better route to this node!
64             if (!inOpen || f < neighbour->f) {
65                neighbour->parent = currentBestNode;
66                neighbour->f = f;
67                neighbour->g = g;
68             }
69          }
70          closedList.emplace_back(currentBestNode);
71       }
72    }
73    return false; //open list emptied out with no path!
74 }
```

NavigationGrid::FindPath method

The more likely case is that the best node in our open list is *not* the destination node, and so on line 44 we start the process of expanding the algorithm through the graph, by iterating through every possible neighbour (note that in cases where there's *not* a connection between adjacent **GridNodes**, we must skip over them in line 47). As part of the A* algorithm, we add processed nodes to a 'closed list' indicating that we no longer need to consider them, even if we expand into them again later on, so on line 49 we work out if we've already seen this node, and if so, just move on to the next one.

If this is a new node, we need to calulate the h, g, and f scores for it. The h value is always calculated using the *Heuristic* method we filled out earlier, while g is the sum of the best node's current g cost, plus the cost to move to this neighbouring node (remember, different node traversals might have different costs associated with them, such as rocky terrain, or a dangerous area of the map that should be avoided), with f being the sum of h and g.

If a node has been seen before, the f score calculated on line 56 might still be better than the node's current score (perhaps we had to traverse rocky terrain to get to it the first time, but now a path over open ground has been discovered), so on line 64 we check for this case, and if necessary update the node's scores, and it's 'parent' node, which should always be the 'least cost' path to the source, as over the course of the algorithm this will build up to being the 'least cost' from the destination to the source. Once we've finished with the currently selected node and expanded the algorithm to all of its neighbours, we can place the node in the closed list (line 70), and restart the process of trying to guess the best node.

Main File

Now that we've added the new grid based pathfinding ability, we should probably test it, too. In the **CSC3223.cpp** file, let's add some code to the *TestPathfinding* and *DisplayPathfinding* functions:

```cpp
vector<Vector3> testNodes;
void TestPathfinding() {
   NavigationGrid grid("TestGrid1.txt");

   NavigationPath outPath;

   Vector3 startPos(80,0,10);
   Vector3 endPos(80, 0, 80);

   bool found = grid.FindPath(startPos, endPos, outPath);

   Vector3 pos;
   while (outPath.PopWaypoint(pos)) {
      testNodes.push_back(pos);
   }
}
void DisplayPathfinding() {
   for (int i = 1; i < testNodes.size(); ++i) {
      Vector3 a = testNodes[i - 1];
      Vector3 b = testNodes[i];

      Debug::DrawLine(a, b, Vector4(0, 1, 0, 1));
   }
}
```

TestPathfinding function

This lets us see how to use our new class, and get an idea of how we could service AI requests to find paths in an area. In *TestPathfinding*, we load up a text file, containing the following:

```
20 //how many units on each axis is each grid
10 //how many grid nodes across
10 //how many grid nodes down
xxxxxxxxxx
x........x //Can we find a path from here
x........x
x....xxxxx
x........x
xxxxxx...x
x......xxx
x..xxxxxxx
x........x //To here??
xxxxxxxxxx
```

TestGrid1.txt

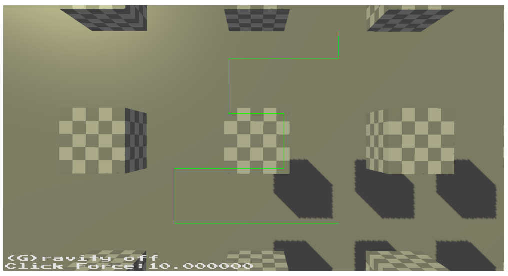
This gives us a little maze, where 'x' represents a wall, and '.' represents a passable area. In the constructor of a **NavigationGrid**, this file will be read in and parsed to form a grid of **GridNode** objects, connected according to whether their neighbour is a wall or a floor. The first value in the file defines how many units across each grid node is - in this case, each node is 10 units across, making for a 100 by 100 unit world, subdivided into 10 by 10 sections.

To actually find and get a path from some position to another, we need to make a **Navigation-Path** (line 5) - if a path is found, this will get filled up with **Vector3s** representing the centre of each node to travel through. On line 10, we actually try and get a new path, from *startPos*, to *endPos*, using the *FindPath* method we filled in earlier. Assuming the path can be found (the example values shown above should generate a path from the top right corner to the bottom right corner), we can then fill up a **std::vector** of **Vector3s**, representing the waypoints in out path (line 14). For an AI character, you would instead only pop a waypoint if the AI moved close enough to a node that it could continue on to the next one, using a distance check of the AIs current position.

For now, rather than making a full AI character, we'll just make a little test function that will display the generated path in our world - the **DisplayPathfinding** method simply builds up pairs of positions, and uses the **Debug::DrawLine** method to draw lines between them. Being able to see the paths being generated in your world is a useful ability, and something you should consider adding to any of the pathfinding systems you create for the coursework project.

Conclusion

If you've managed to do everything correctly, you should have an image similar to the following on screen:



```
(G)ravity off
Click Force:10.000000
```

It doesn't look like much, but if we trace that path out, it'll complete the maze outlined in **Test-Grid1.txt**, giving us not just *a* path from the top to the bottom, but the *best* path from the top to the bottom. This is the core benefit of the A* algorithm - if a path can be found, it's going to be the best way to get from A to B. Nearly all dynamic pathfinding in games uses A* in some way, although it may be augmented with additional calculations. In the example code we've used a fixed cost of 1 to go from one node to another (check the constructor of the **NavigationGrid** class to see this), but it could be much more complex value - different terrains could have different costs to traverse, and that cost could even change over time (perhaps in the game environment the temperature drops, causing a lake to freeze, unlocking a path that was not there before, and so on). This pathfinding algorithm can form the main method by which to get an AI character moving about in the world in your coursework, and using it will allow for more interesting gameplay beyond just 'simple' moving in straight lines.

Further Work

1) Consider the operation of the open list - what would be a better container for a structure that must be kept in an order where the 'best' node is always first? What about the closed list? Is a container object even required for this?
2) In large game environments, many 100s of AI agents will require new paths to be generated periodically; something that is best achieved by having threads dedicated to servicing pathfinding requests. Investigate what changes would need to be made to modify the current pahtfinding classes to support multiple pathfinding requests simultaneously.