Path Finder For Creating A .NET Component For Grid Based Graph

R. ANBUSELVI↑
Lecturer in Computer Science,
Bishop Heber College, Trichy – 17.

R.S. BHUVANESWARAN,
Asst.Prof.Anna University, Chennai.

SUMMARY

Path finding is one of those required elements of the Computer network. Any path finding algorithm will work as long as there are no obstacles or distractions along the way. If there is an obstacle, then the character needs to figure out a way to move around and still reach the goal.

Most of the path finding algorithms found in the literature [1], [2], [3] were designed with arbitrary graphs which is not realistic.

To achieve the best path there are many algorithms, which are more or less effective, depending on the particular case.

Efficiency depends not only on the time needed for calculation but also on the reliability of the result.

In this research work we have attempted with Grid Based graph, since the ssearch area is simplified into a manageable number of nodes.

The next step is to conduct a search to find the shortest path.

The efficiency can be obtained in our proposed system when number of nodes is increased with less memory and less time.

1. INTRODUCTION

"**PATH FINDER**", is developed as a .NET component that allows to build graphs and perform certain operations on these structures. The main component of the path finding algorithm is search area, which is discussed in detail in next section. Section 2 dealt with beginning of search and path scoring. Section 3 presented the related works, Section 4 describe the implementation particulars and conclude with section 5.

1.1 THE SEARCH AREA:

Let us assume that we have someone who wants to get from point A to point B. Let us assume that a wall separates the points A, A1,B1,…: Let A be the starting point and B be the ending point. This is illustrated below, with green square being the starting point A, and red square being the ending point B, and the blue squares being the wall in between.

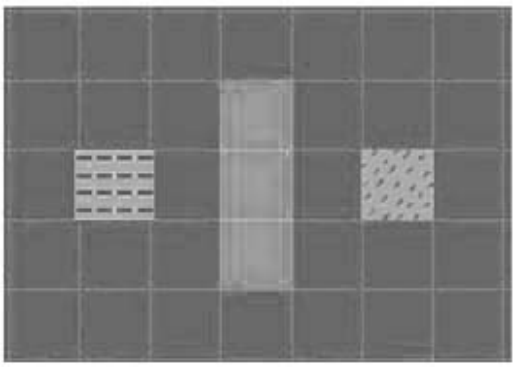

Fig: 1 search area representing the square grids.

The first thing we should notice is that the search area is divided into a square grid. Here the first step is the pathfinding. This particular method reduces the search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, its status is recorded as walkable or unwalkable. The path is found by figuring out which squares should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached.

These center points are called "nodes". When we read about path finding elsewhere, we will often see people discussing nodes. Squares can otherwise be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. [2],[4].

2. STARTING THE SEARCH

We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

- Begin at the starting point A and add it to an "open list" of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path we want to take, but maybe not. Basically, this is a list of squares that need to be checked out.

Manuscript received April 5, 2009
Manuscript revised April 20, 2009

- Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its "parent square". This parent square stuff is important when we want to trace our path. It will be explained more later.
- Drop the starting square A from our open list, and add it to a "closed list" of squares that we don't need to look at again for now.

At this point, we should have something like the following illustration. In this illustration, the dark green square in the center is our starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green.

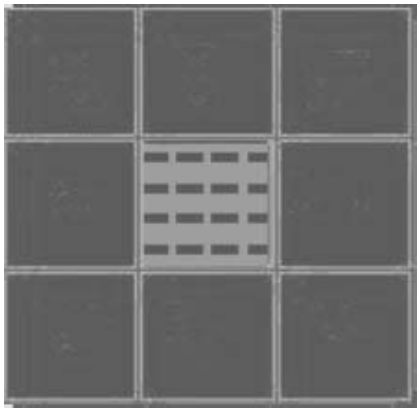

Fig: 2 Open list of squares

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

2.1 PATH SCORING:

The key to find the squares to use when figuring out the path is the following equation:

F = G + H

Where F is calculated by adding G and H.

- G is the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
- H is the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. The actual distance cannot be known until we find the path, due to an obstacle. We have a way to calculate H in this paper, but there are many others that we can find in other articles on the web [2], [3], [4].

The path is generated by repeatedly going through an open list and choosing the square with the lowest F score. This process is described as follows.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, [1],[2],[4]. where we calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically.

This is (probably) called as Manhattan method since it is like calculating the number of city blocks from one place to another, where we can't cut across the block diagonally. F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

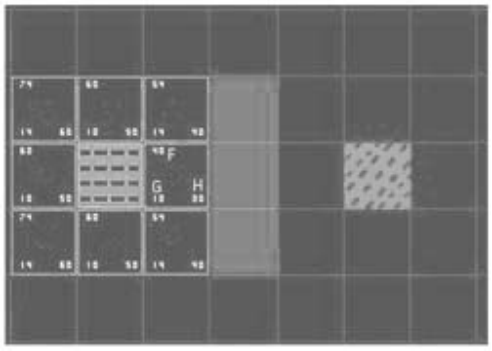

Fig: 3 List of squares on the open list

Create a search graph G, consisting solely of the start node, n_o. Put n_o on a list called OPEN.

Create a list called CLOSED that is initially empty.

If OPEN is empty, exit with failure.

Select the first node on OPEN, remove it from

OPEN, and put it on CLOSED. Called this node n.

If n_0 is the starting node, n is the goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to n_o in graph G. (The pointers define a search tree and are established in Step 7.)

Expand node n, generating the set M, of its successors that are not already ancestors of n in G. Install these members of M as successors of n in G.

Establish a pointer to n from each of those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member, m, of M that was already on OPEN or CLOSED, redirect its pointer to n if the best path to m found so far is through n. For each member of M already on CLOSED, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.

The F score for each square, again, is simply calculated by adding G and H together.[1],[5].

2.2 CONTINUING THE SEARCH

To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

- Drop it from the open list and add it to the closed list.
- Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the "parent" of the new squares.
- If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don't do anything. On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square.

3. RELATED WORKS

Specifically open and closed lists and path scoring using F, G, and H. There are lots of other pathfinding algorithms, but those other methods are not A*, which is generally considered to be the best of the lot. Marco Pinter [4] discusses many of them in the article referenced at the end of this article, including some of their pros and cons. Sometimes alternatives are better under certain circumstances, but we should understand what we are getting into.

4. NOTES ON IMPLEMENTATION

Now that we understand the basic method, here are some additional things to think about when we are writing our own program. Some of the following materials reference the program I wrote in C++ and Blitz Basic, but the points are equally valid in other languages.

4.1. OTHER UNITS (COLLISION AVOIDANCE)

If we happen to look closely at my example code, we will notice that it completely ignores other units on the screen. The units pass right through each other. Depending on the game, this may be acceptable or it may not. If we want to consider other units in the pathfinding algorithm and have them move around one another, I suggest that we only consider units that are either stopped or adjacent to the pathfinding unit at the time the path is calculated, treating their current locations as unwalkable. For adjacent units that are moving, we can discourage collisions by penalizing nodes that lie along their respective paths, thereby encouraging the pathfinding unit to find an alternate route (described more under #2).

If we choose to consider other units that are moving and not adjacent to the pathfinding unit, we will need to develop a method for predicting where they will be at any given point in time so that they can be dodged properly. Otherwise we will probably end up with strange paths where units zig-zag to avoid other units that aren't there anymore.

4.2 EXPERIMENT AND ANALYSIS

Consider the following RPG situation, and a swordsman who wants to path finding around a nearby wall:

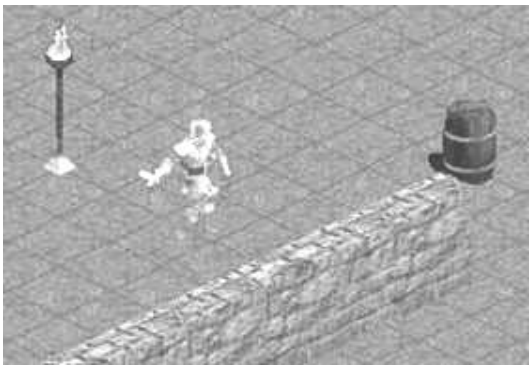

Fig: 4 Path finding around a near by wall.

Given this kind of map, we could place nodes in a variety of ways, and use a variety of densities. In this example, let's use a high-density node network, as is shown below.

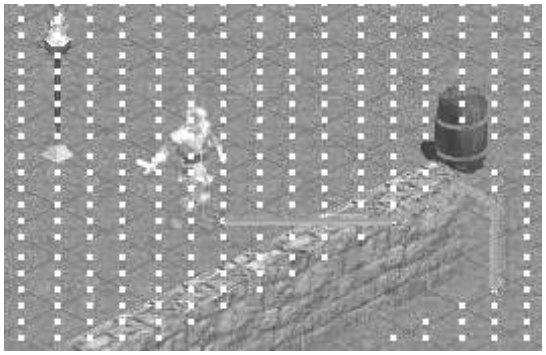

Fig: 5 High- Density Node Network

Let us say we were adding an item with an F score of 17 to our existing heap. We currently have 7 items in the heap, so the new item would be placed in position number 8. This is what the heap looks like. The new item is underlined.

10 30 20 34 38 30 24 17

We would then compare this item it to its parent, which is in position 8/2 = position 4. The F value of the item currently in position 4 is 34. Since 17 is lower than 34, we swap them. Now our heap looks like this:

10 30 20 17 38 30 24 34

Then we compare it with its new parent. Since we are in position 4 we compare it to the item in position number 4/2 = 2. That item has an F score of 30. Since 17 is lower than 30, we swap them, and now our heap looks like this:

10 17 20 30 38 30 24 34

We then compare it to its new parent. Since we are now in position #2, we compare it with the item in position number 2/2 = 1, which is the top of the heap. In this case, 17 is not lower than 10, so we stop and leave the heap the way it is.

Removing Items from the Heap

Removing items from the heap involves a similar process, but sort of in reverse. First, we remove the item in slot #1, which is now empty. Then we take the last item in the heap, and move it up to slot #1. In our heap above, this is what we

would end up with. The previously last item in the heap is underlined.

34 17 20 30 38 30 24

Next we compare the item to each of its two children, which are at locations (current position * 2) and (current position * 2 + 1). If it has a lower F score than both of its two children, it stays where it is. If not, we swap it with the lower of the two children. So, in this case, the two children of the item in slot #1 are in position 1*2 = 2 and 1*2+1 = 3. It turns out that 34 is not lower than both children, so we swap it with the lower of the two, which is 17. This is what we end up with:

17 34 20 30 38 30 24

Next we compare the item with its two new children, which are in positions 2*2 = 4, and 2*2+1 = 5. It turns out that it is not lower than both of its children, so we swap it with the lower of the two children (which is 30 in slot 4). Now we have this:

17 30 20 34 38 30 24

Finally we compare the item with its new children. As usual, these children would be in positions 4*2 = 8, and 4*2 +1 = 9. But there are not any children in those positions because the list is not that big. The process terminates when it reaches the bottom of the level of the heap.

5. CONCLUSION

Actually, we started with some basic path-finding routines on a tile-based map, such as a randomized search and the popular “right hand on the wall” trick. Then we went to more advanced environments based on graphs and implemented a generic A* algorithm that could be used for graph- or tile-based environments.

Finally, we merged A* searching with the BSP tree we have been working with for the past couple of chapters, and we created a generic path that follows any type of path, whether it was created by an A* search or not.

REFERENCES

[1] Craig Reynold’s, (1999) **“**Sterring Behavior for Autonomous Characters:” 1st Feb 2000 [Reyn87] Gamasutra, cited by 68 – Related articles – all 12 versions.

[2] Dave Pottinger, “Coordinated Unit Movement:” 22nd Jan 1999. Gamasutra Vol.3: Issue 3. First in a two-part series of articles on formation and group-based movement by Age of Empires designer.

[3] DavePottinger’s “Implementing Coordinated Movement:” 29th Jan 1999. Game Developer PP 48-58. Second in two-part series.

[4] Marco Pinter, “More Realistic Part Finding Article:” 14th Mar 2001. Gamasutra.

[5] Eric Marchesin, “A simple c# Genetic alogorithm Article:” 22nd June 2003, .NET 1.0, 4.72

[6] http://www.gamasutra.com

R.Anbuselvi received M.Sc in computer science from Bharathidasan University, India in 1992. She received M.Phil from Mother Teresa women's University kodaikanal in 2001 .Her research include Artificial Intelligence. She is working as a lecturer in Bishop Heber 's college, Trichy.

R.S. Bhuvaneswaran received Bachelor of Science in Mathematics with Gold Medal in 1987 from Madras University, Master of Technology in Computer Science and Engineering from Pondicherry University, in 1996 and Ph.D in Computer Science and Engineering from Anna University in 2003. He is a Post Doctoral Fellow of JSPS, Japan (2004-2006).Presently, he is with Anna University as Assistant Professor. His research interests include design of algorithms, distributed systems, wireless networks and fault tolerant systems.