Pathfinding - Fifteen minute talk

By: Richard Howes

Pathfinding is used in gaming, mapping, GPS systems, and other applications so that the best route from one point to another can be determined.

The best route is often subjective to the specific application. Aircraft have to consider the curvature of the earth and weather. Ships and submarines must determine currents, storm systems, and three dimensional geographical features. Land vehicles consider highways, surface streets, pavement/dirt roads, and construction delays. Pedestrians, skiers, equestrians, off-roaders must consider paths, tracks, walkways or going cross-country through woods or fields, etc. Most of us have played games with bots that get lost or trapped in a “U” shaped dead end and cannot get out. These all present interesting problems for the programmer.

Let’s consider several solutions, and see examples of their efficiencies and deficiencies.

**Setup:**

A 2D grid 20x20 squares. Start point is approximately 25 spaces from the end point. This grid contains terrain: hills, mountains, roads, swamps, woods, plains, etc. Each terrain type contains a negative value indicating difficulty. For example, Mountains are more difficult than hills and roads are least difficult compare to everything else.

**Consideration:**

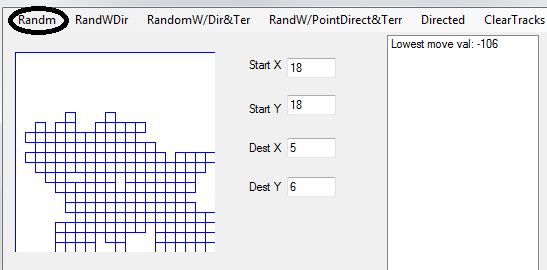
Games and the like require responsiveness. The fewer cycles needed to determine the best path, and the shorter the calculation times required, are critical.

How do we get an object from the Start Point to the End Point with a movement difficulty closest to zero?

Without using a pre-planned path, the Movement Object “MO” should move either randomly, or look around at all adjacent paths for the best terrain path to take.

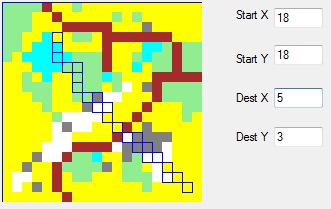
1. **Pure Random Path:**

This solution follows the laws of molecular physics in which paths are completely random. One random path may be in a straight line but hundreds of thousands of paths are spirals, circles, loops, switchbacks, etc. A simulation of this path over 500,000 cycles, written in C#, takes several ten minutes to find possible results to move an object 20 spaces. After 500,000 cycles the best number of steps found is often in the thousands.



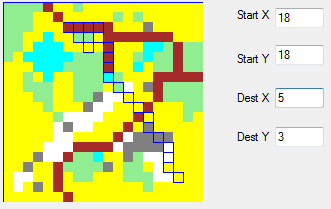
1. **Random with direction:**

This solution determines in what direction the object should move towards its goal and randomly picks the first direction that reduces the distance. This solution loops through 50,000 tries and finds a result in 50-70 terrain value steps.



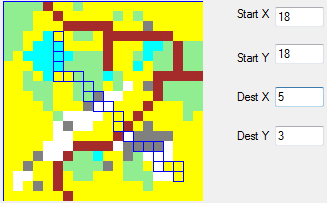
1. **Random with Direction and Terrain Analysis:**

This solution determines in what direction the object should move towards its goal and randomly chooses Forward or Forward-to-the-right. (I never added forward-to-the-left because better things were coming along…) Reducing the number of repetitions to about 50,000 it reaches the goal in 40-50 terrain value steps in less than 1 second. It still did not produce the best path. The first couple experiments above didn’t include terrain calculations.



1. **Random with Pointed Direction and Terrain Analysis:**

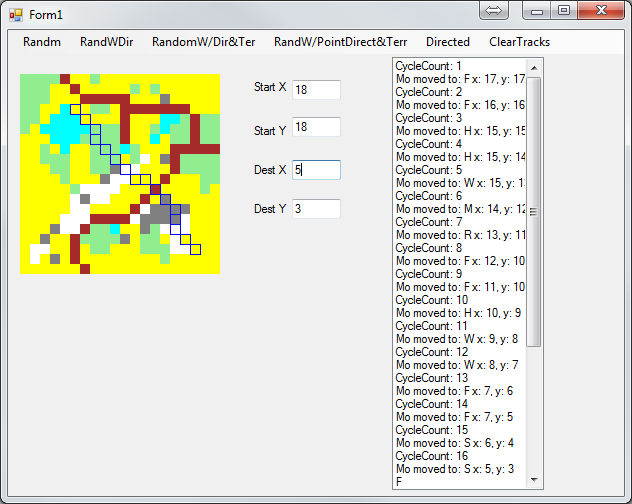
This picks two unique grid positions that will move the object closer to the goal. Attempt to reduce the number of path attempts by checking for direction towards the destination. This finds a move value of 55-60 terrain value in 5000 tries in less than 1 second to run the search loop.



1. **Directed - Best Version So Far:**

“Directed” has no loops. It executes very quickly by finding a compass direction using an angle in degrees determined from an ArcTangent calculation (and people say programmers never have to do math!) The solution then looks at the three grids facing the destination and determines which one of those three has the lowest terrain value and it picks that grid to move into. That repeats until the destination is reached. Adding hard-left and hard-right helped the movement-object go around mountains.

An interesting behavior occurred when the MO got stuck on the corner of a road and shifted back and forth Hard-left to Forward-Right and back. Another time the MO got stuck in hills with a mountain blocking the path forward and the same oscillation occurred. Code was added to store the last previous grid location and disallow moving back into a position the MO previous occupied. (This can be bad for “U” shaped dead ends and better logic will need to be used – such as “allowing going backwards once,” using a “Change direction” or a “change path” flag, etc., or preplanning the entire path.)



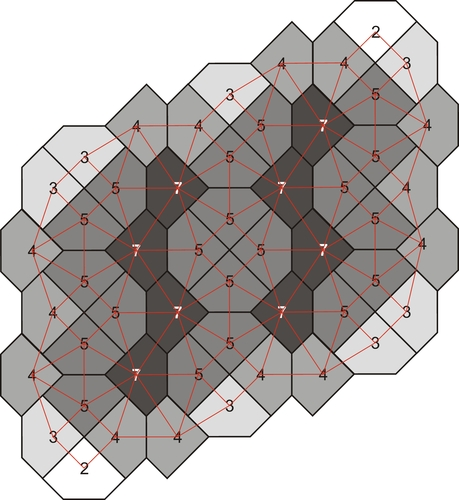
**Not the Final Solution**:

Another interesting point is that moving any direction other than “forward” should logically contain a movement penalty. Having used a square grid, moving 45 degrees from North-South or East-West is a cost of 41.4% longer distance that is “given away.” The example code briefly experimented with that, in addition to adding penalties for moving Left, Right, hard-left, or hard right from forward. (This is different from North-South.)

A Hex grid would solve some of those losses proving 6 adjacent spaces to move into.



A five sided “diamond” grid gives 6 or 7 adjacent spaces to move into, forming straighter lines from one point to another.



Additionally, the final version can result in an object reaching a dead-end that it cannot get out of (if cliffs are added). Adding calculations for terrain in all grids surrounding the object will allow the object to back out and go around if it enters a box-canyon or approaches an impassable river. Additional coding to handle a condition of “follow the edge” would prevent the object from going back into the canyon.