Things that were not immediately obvious to me

Making mistakes so you don't have to

Bloxorz/Bloxors

Posted on August 15, 2008 by MXH

A few months ago I came across a cute little puzzle game called Bloxorz. Inspired by Norvig's Sudoku solver, I hacked together a JS-based pathfinder for Bloxorz problems. Although the proper spelling of the game's name continues to frustrate me, the puzzle turns out to be easily solved with breadth-first search.

Simplifications

The rules of Bloxorz are best explained by the game itself, but, in principle, the goal is to flip a 1×2 rectangular block around an irregular grid of 1×1 squares such that no part of the block is ever unsupported. There are a number of wrinkles on this concept: Some squares can support the block when it is lying horizontally, but not when it is standing vertically, some squares contain switches which alter the board when they are tripped by the block, etc. I divided these complications into 'static' and 'dynamic' parts of the problem, and threw away all the dynamic ones. Specifically, I:

- Do not handle the effects of switches on the map. Switches may be marked with an "avoid" flag, in which case (for the purposes of pathfinding) "hard plates" are treated like weak cells, and "soft plates" are treated like empty space. The maps before and after a switch's effects are treated as two separate pathfinding problems.
- Do not handle the effects of "splitters". As with "hard plates", "splitters" may be marked with an "avoid" flag, in which case they are treated like weak cells. I do not solve pathfinding problems for the two-cube case.

These simplifications reduce the pathfinding problem to one of navigating the block around a board on which some squares may not be touched at all, and other squares may not support the block vertically.

Code

Here is the core pathfinding function from my solver:

```
p.Pathfind = function (startPos, endPos) {
        var queue       = [];
        var prevLUT     = {};
        var i, a, k, p, np;

        // Build tree
        queue.push(endPos);
        prevLUT[endPos.MakeKey()] = null;
        while (queue.length)
        {
                p = queue.shift();
                a = this.FilterPositions(p.NextPositions());
                for (i = 0; i < a.length; i++)
                {
                        np = a[i]; k = np.MakeKey();
                        if (typeof(prevLUT[k]) != "undefined")
                        {
                                continue;
                        }

                        queue.push(np);
                        prevLUT[k] = p;

                        if (np.EqualP(startPos))
                        {
                                queue.length = 0;
                                break;
                        }
                }
        }

        // Walk tree backwards
        var rv = [];
        for (p = startPos, k = p.MakeKey(), rv.push(p); np = prevLUT[k]; p = np, k = p.MakeKey(), rv.push(p));
        if (rv[rv.length-1].EqualP(endPos))
        {
                return rv;
        }
};
```

General Approach

The code performs a breadth-first search on the graph of legal block positions. For performance reasons, the graph is generated only as needed. Information is recorded as the search progresses, and this information is later used to generate a minimum-length series of legal block positions between the initial and goal positions.

Block positions may be represented as a pair of (x, y) pairs, of one of the following three forms:

((x, y), (x+1, y)	East-West aligned horizontal block
((x, y), (x, y+1)	North-South aligned horizontal block
((x, y), (x, y))	Vertical block

The network of *all* block positions may be represented as an undirected graph in which:

- Nodes are block positions
- Edges connect positions separated by exactly one legal move

Note that each node has exactly four edges, and that this graph contains an infinite number of nodes. (The latter point isn't as big a problem as you might at first suppose.)

A map may be used to create a filter function F(), which takes a list of block positions and returns the *legal* subset of that list, i.e. those positions which a block could legally occupy (but not necessarily reach) on that map. If F() is applied to the nodes of the block position network, and any nodes not in the result set of F() are pruned from the graph (along with any edges connected to them) then the nodes of the resulting graph G represent all legal positions, and the edges of G represent all legal moves.

Any legal sequence of moves between any two legal positions will be represented by a path through G. In particular, a minimum-length series of moves will be represented by a shortest-path in G. Since G is an unweighted graph, we can use breadth-first search (BFS) to find a shortest-path between any two nodes of G, assuming that one exists.

Please note that a path is not guaranteed to exist between any two nodes of G. If G is a disconnected graph there are guaranteed to be nodes between which there is no path; such graphs represent maps in which some legal positions are not reachable from others. (This is the normal case in most of the later levels before switches are tripped.)

Path Generation

BFS, as its name implies, is a search algorithm. We can use it directly to determine if the goal node is connected to the initial node, but we must also use the results of the search to produce a shortest-path. Two simple tricks turn BFS into a path-generation algorithm.

To explain path generation, two terms need to be defined:

- A node is “discovered” when it is added to the BFS algorithm’s queue
- A node is “processed” when it is removed from the BFS algorithm’s queue

Since nodes are processed in monotonically increasing order of their distance from the origin node (this is essentially the definition of BFS), the node being processed is always on a shortest-path between any of its previously undiscovered neighbors and the node from which the BFS started. (If node A is being processed, all nodes closer to the origin than A have already been processed. Therefore, any unprocessed node is not closer to the origin than node A. All neighbors of an undiscovered node are unprocessed, and therefore no closer to the origin than node A. No undiscovered neighbor of node A has a neighbor closer to the origin than node A.) When processing a node, therefore, we store that node as the value associated with the key(s) of its newly discovered neighbors(s) in an associative array. We can later use that array to find the first step of a shortest-path between any discovered node and the origin node. By chaining many such first steps (each of which leads to another discovered node) we can create an entire shortest-path between any discovered node and the origin node.

Our second trick is simply to begin the search from the goal node, and search for the initial node. If and when BFS finds the initial node, we can use the aforementioned associative array to find a shortest-path between it and the goal node.

Practical Considerations

To recap: Once we have G, we can use the BFS algorithm to find a shortest-path between the two nodes representing the initial and goal positions, and then translate that shortest-path into a minimum-length series of legal moves between the positions. (If no path can be found, the goal position cannot be reached from the initial position.) The only remaining major difficulty involves the generation of G, which, in theory, was generated by feeding an infinite-size graph of all block positions through F(), which is clearly an impracticable approach. Fortunately, G can be iteratively generated.

At any point in the BFS algorithm, we only need to know a node's immediate neighbors. Each node of G has exactly 4 possible neighbors, each of which can be computed by applying one of the 4 legal transformations to the node's position. This set of 4 nodes can then be filtered by F(), yielding the set of actual neighbors, which can be used by the BFS algorithm. In the code above, NextPositions() computes the set of all possible neighbors of a node (i.e. the neighbors of that node in the graph of *all* block positions) and FilterPositions() eliminates illegal positions from this set. These functions supply all the information about G than BFS requires.

G can be seen to be relatively small: all game maps have far less than 300 squares, and there are far less that 5*num_squares possible positions on any map. This yields an upper limit on the size of G of 1500 nodes, and as BFS visits each node at most once, we will have no performance problems.

Share and Enjoy:

This entry was posted in Projects. Bookmark the permalink.

One Response to *Bloxorz/Bloxors*

Pingback: *Recent Faves Tagged With "pathfinding" : MyNetFaves*