Pathfinding using A* (A-Star)

Rajiv Eranki, 2002

Traversing graphs and finding paths

Let's take a look at the diagram below:

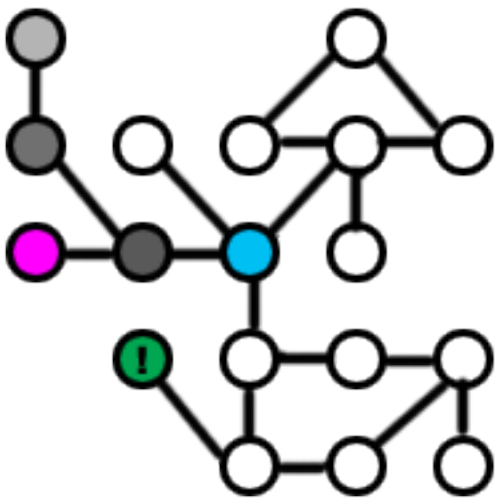
This is a pretty typical problem, no? We have a starting point, and ending point, and a bunch of roads we can take to get there. We can even abstract this so that the roads are "moves" (of, say, a little sliding puzzle) and the points are states of the puzzle. Our goal is to find the shortest path possible to the end node in the fewest moves — and not take a long time to find it!

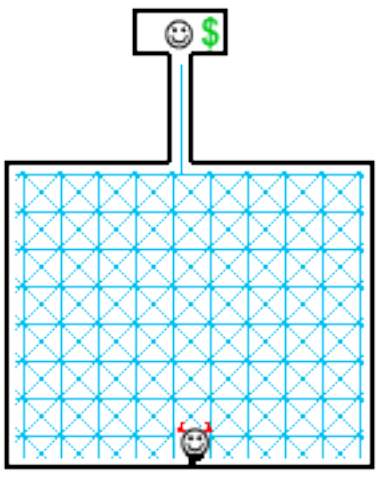
Depth First Search (dfs)

The simplest (and least efficient) method of traversing a graph is the Depth First Search (dfs). Recursion suits this method well. Examine the pseudocode for a simple dfs:

```
      // Simple dfs
1:    function dfs(node position)
2:        color(position)
3:        for each successor adjacent to node "position"
4:            if successor is colored, skip it
5:            if next is the goal node, stop the search
6:            else, dfs(successor)
7:        end
8:    end
```

"Coloring" a node means marking it to show we've gone there. This prevents a search from searching the same node more than once. This algorithm works -- oh, it works. In fact, this method will always find you *a* path to the solution. Not that it's special or anything -- the next two methods will always find the solution as well, only they won't take until Hell freezes over to find it. Take, for example, the poor sod below. He looks happy, but that's simply because he's an idiot. Looking for treasure in this vast cavern, this guy decided to systematically use dfs. He will eventually find the gold for sure. He may traverse the entire graph, though, when he could have simply looked to his left! This kind of blind search, not taking into account the location of the goal, is likely to take much more time than an intelligent search. But before improving on our guessing skills, let's examine another algorithm.





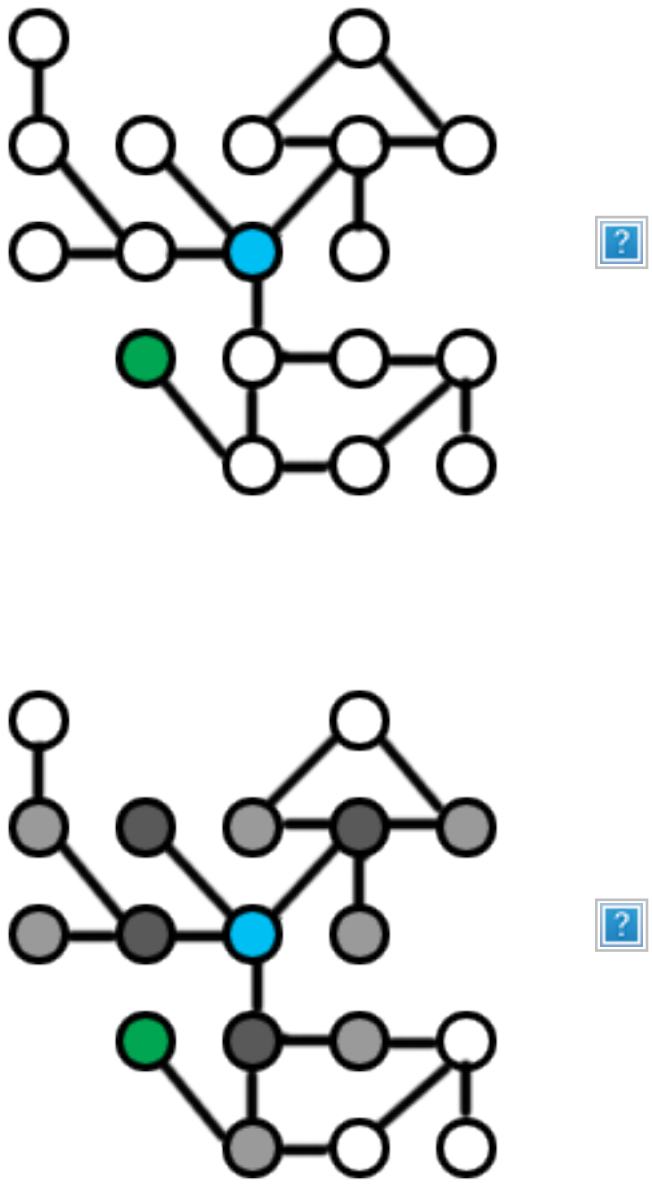
Breadth First Search (bfs)

Yep, you saw it coming. This algorithm is much nicer than dfs because not only are you guaranteed a path to the goal, but you're guaranteed the shortest path. An improvement, indeed. So in this method, we examine each of the nodes next to the node in question before we go deeper. This has the effect of forming concentric "circles" around the node with each ply of the search. Since we're slowly moving out from the center we're going to reach the goal as quickly as possible. This method is marginally harder to implement, but it's still easy. Take a gander at the pseudocode. Note that this is *not* recursive (since we have to look at all the options before going deeper).

```
   // Simple bfs
1: structure node
2:     position pos
3:     node *parent
4: end
-
5: function bfs(node start_position)
6:     add start_position to the queue
7:     while the queue is not empty
8:         pop a node off the queue, call it "item"
9:         color item on the graph // make sure we don't search it again
10:        generate the 8 successors to item
11:        set the parent of each successor to "item" // this is so we can backtrack our final solution
12:        for each successor
```

```
13:             if the successor is the goal node, end the search
14:             else, push it to the back of the queue // So we can search this node
15:         end
16:     end
-
17:     if we have a goal node, look at its ancestry to find the path (node->parent->parent->parent..., etc)
18:     if not, the queue was empty and we didn't find a path :^\
19:end
20:
```

Take a look at the progression of a typical bfs down below. Note that the path found by the computer is south, south, northwest -- the shortest path. It could well have been south, east, east, southwest, west, northwest by dfs.



A* (A-star)

Ah... Now to the heart of pathfinding, A*! You've probably heard of this algorithm, because it's the leading pathfinding algorithm (plus, that's the name of this tutorial and I'll be a monkey's uncle if you weren't searching for an A* tutorial). This is the algorithm typically used in games like Warcraft III. A* is not bfs, nor is it dfs. In fact, it's a combination of Dijkstra's algorithm (which I haven't added to the this tutorial... yet) and best-first. Don't worry, though, the algorithm is very easy t o understand.

The question arises while programming bfs and dfs: What if instead of blindly guessing the next node to traverse like in a simple dfs we pick the node which looks most promising? A* searches exactly like that: in a nutshell, we generate our possibilities and pick the one with the least projected cost. Once a possibility is generated and its cost is calculated, it stays in the list of possibilities until all the better nodes have been searched before it.

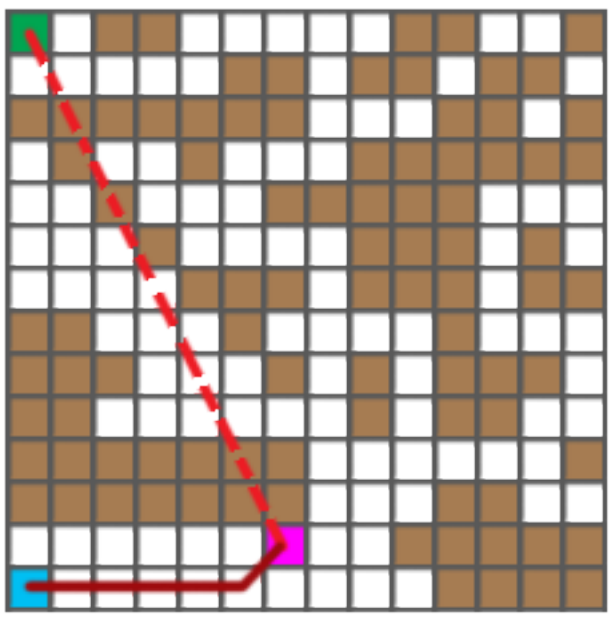
First, let's define the cost function. The cost of a node, f, is given by the following partial differential equation (just kidding ;^)):

$$f = g + h$$

"Clever", you say, "but what are **g** and **h**?". Good question. **g** is the cost it took to get to the node, most likely the number of squares we passed by from the start. **h** is our guess as to how much it'll cost to reach the goal from that node. It's our heuristic (a heuristic, informally, is something which is not a definite series of steps to a solution (like an algorithm) but it helps us determine our answers is a rough way). A* will find you the best path in a very short time provided your h is perfect. In this case, we obviously can't determine h perfectly without doing some other pathfinding so we just use an approximation. Very rarely (if ever) is your h perfect. Take a look at the diagram below, a square grid such as one you'd find in an RTS. The brown spots mark spaces that are blocked (by buildings, people, or natural barriers). Blue and green are the starting point and goal, respectively.

In searching, say we generate the purple square. Its **g**, the maroon solid line, is the distance from the starting point. Since we moved 5 squares east (at a distance cost of 1) and one square northeast (at a cost of √2), our **g** is 5+&rad ic;2, about 6.414. We don't have to recalculate **g** entirely each node. We can just add the distance to the parent node plus the parent's **g**. With maps like these, we can define **h** as the linear distance to the goal. I'm not going to coun t those squares, so this is left as, er, an "exercise to the reader". ;^)

Tip: Do we really need to store the euclidean distance in **g** and **h**? Think about it. If we used the *square* of the distances, our comparisons between nodes will still be the same.



We're going to have to modify our node structures to hold the new information. Our nodes are going to look something like this when we're done (same as before, but we snuck in an f, g, and h. Technically we don't need the 'f' because it's simply g+h, but we want to save our poor processors the extra add operation):

```
struct node {
    node *parent;
    int x, y;
    float f, g, h;
}
```

Alright, so now that we can calculate the cost, let's actually do A*, shall we? First, we create lists open and closed. open is similar to bfs's queue, and closed is similar to the coloring technique we used before. However, we have to make a list because coloring doesn't store cost information. So we pop the element with the least **f** off the open list, and generate its successors. If one of its successors is the goal, we're golden and can stop the search. For each successor, calculate its cost and add it to the open list. At the end of this iteration, put the node we popped off the open list on the closed list. This is best illustrated in pseudocode:

```
       // A*
1:     initialize the open list
2:     initialize the closed list
3:     put the starting node on the open list (you can leave its f at zero)
-
4:     while the open list is not empty
5:         find the node with the least f on the open list, call it "q"
6:         pop q off the open list
7:         generate q's 8 successors and set their parents to q
8:         for each successor
9:             if successor is the goal, stop the search
10:            successor.g = q.g + distance between successor and q
11:            successor.h = distance from goal to successor
12:            successor.f = successor.g + successor.h
-
13:            if a node with the same position as successor is in the OPEN list \
-                  which has a lower f than successor, skip this successor
14:            if a node with the same position as successor is in the CLOSED list \
-                  which has a lower f than successor, skip this successor
15:            otherwise, add the node to the open list
16:        end
17:        push q on the closed list
18:    end
```

It's important to note that we may search the same point a few times — but only if the new path is more promising than the last time we searched it!
