Pathfinding with A*

Pathfinding is all about finding a valid path from A to B, where A and B are simply states. If more than one path exists, then we want the "best" one. When we talk about pathfinding, we generally mean finding a path from one position to another, like on a map. It is more general than that however. The states A and B could just as well be states of a game of chess, or more simple states describing the position of the tiles in a 9-puzzle, and then the path would consist of the steps needed to go from one state to the other. A generally being the current state of the puzzle and B being the final solved state.

The fact that pathfinding is more than walking around on a geographic map is what makes it so useful, but the algorithm itself is the same no matter what the states are and it is generally easier to understand if we can think of it as the path on a map, so my examples will use that approach.

Quick overview

The general idea of the algorithm is to move into the first state called A, and from there look at all valid moves. Those moves are added to a list of open moves – moves not yet taken, but possible to take in the future. Such a move in the list we call a node. Following that, you take the "best" of the nodes in the open list out and put it into a list of closed/traveled nodes and repeat the process. In the end you will take out an open node which is the target state B.

1. Add a node containing the position of A, the cost of getting to it and the node that led to it to list of open positions. For the origin the cost and the parent node is nonexistent.
2. Take the "best" node from the open list. This is generally the one with the lowest cost. If this is the target B, then you are done
3. For each possible move from the current position, add a node containing the move position, the cost of traveling it plus the cost of traveling to its parent node and finally the parent node, to the open list
4. Goto 2

I will mention the heuristics later on.

To make this simplistic description a little bit more concrete, I have made up an example.

Quick example

The cost map table contains the cost of traveling a position. A1 has a value of 3 which means that if you are to travel to A1, it will cost you 3. 3 is just a number, but could represent anything. In a terrain it would be terrain type, if it is flat land, forest, mountains or a road. If it is a puzzle then every position/state would probably have a cost of one. Note that diagonal moves are allowed in this example.

Cost map	A	B	C	D	E
1	3	3	3	100	
2	100	100	3	100	
3	100	100	3	100	
4		100	100	100	
5					
6					

We will start in A1 and the destination is C3. I only filled out some of the positions since this example will never reach the empty nodes anyway.

The terrain consists of cheap positions with a cost of 3 and very expensive ones with a cost of 100.

1. Add [A1,0,-] to the open list
   open = [A1,0,-]
   closed = empty
2. Take out A1 from the open list and add it to the closed list
   open = empty
   closed = [A1,0,-]
3. For each neighbor of A1, add their nodes to the open list, IF it is not in any of the lists already
   open = [B1,0+3=3,A1] [A2,0+100=100,A1] [B2,0+100=100,A1]
   closed [A1,0,-]
4. Take the best node, the one with lowest cost, from the open list, add it to the closed list and make the node the current one. This node is [B1,3,A1], which is not the destination.
   open = [A2,0+100=100,A1] [B2,0+100=100,A1]
   closed [A1,0,-] [B1,3,A1]
5. For each neighbor of B1, add its node to the open list, IF it is not in any of the lists already
   open = [A2, 100,A1] [B2, 100,A1] [C1,3+3=6,B1] [C2,3+3=6,B1]
   closed [A1,0,-] [B1,3,A1]
6. Take the best node from the open list, add it to the closed list and make the node the current one. This node is either C2 or C1 each having a cost of 6. For now just select one at random. We select [C2,6,B1] to make this a shorter example. Later we will use a heuristic.
   open = [A2, 100,A1] [B2, 100,A1] [C1,6,B1]
   closed [A1,0,-] [B1,3,A1] [C2,6,B1]
7. For each neighbor of C2, add its node to the open list, IF it is not in any of the lists already
   open = [A2, 100,A1] [B2, 100,A1] [C1, 6,B1] [D1,6+100=106,C2] [D2,6+100=106,C2] [D3,6+100=106,C2] [C3,6+3=9,C2] [B3,6+100=106,C2]
   closed [A1,0,-] [B1,3,A1] [C2, 6,B1]

8. Take the best node from the open list, add it to the closed list and make the node the current one. This node is now [C1,6,B1]
   open = [A2, 100,A1] [B2, 100,A1] [D1, 106,C2] [D2, 106,C2] [D3, 106,C2] [C3, 9,C2] [B3,106,C2]
   closed [A1,0,-] [B1,3,A1] [C2, 6,B1] [C1,6,B1]
9. For each neighbor of C1, add its node to the open list, IF it is not in any of the lists already. All neighbors are however already in the lists, so nothing is changed
   open = [A2, 100,A1] [B2, 100,A1] [D1, 106,C2] [D2, 106,C2] [D3, 106,C2] [C3, 9,C2] [B3,106,C2]
   closed [A1,0,-] [B1,3,A1] [C2, 6,B1] [C1,6,B1]
10. Take the best node from the open list… that is [C3,9,C2] which is the destination, so we are there now. To find the path traveled we simply see that C3 points to C2 which points to B1 which points to A1 which is the origin.

Ok, when you type every little step out like that, it is not a quick example after all.

Heuristics

Remember when we should find the node with the lowest cost, and two nodes had the same cost. One was picked at random, but obviously you should always pick the one closest to the destination. The question is then which one is closest. Perhaps the one which is closer on the map is not closer on the path. If the shortest path is a winding one, then you cannot really tell which step brings you closer to the target… that is, not before you have traveled the entire path and can look back.

If you are inside a house and want to go into the garden then you often have to step away from the garden to get closer to the door of the house which lets you get outside where you can go around the house to the garden. Only after trying the direct path, bumping your head into a wall and taking the seemingly longer path, will you know the best option. In real life you will generally take the best path, but occasionally you are in an unfamiliar house and you take the path leading to a dead end or perhaps just a longer path.

This is a problem which there is no general solution. It is all about heuristics. How good you are at guessing. If you are path finding in a landscape and see wet lands ahead on the direct path towards the target, then your experience may allow you to anticipate trouble and let you change your route slightly to the left or right to steer clear of the wet. That is probably a good heuristics to avoid that terrain, but then again… perhaps there is a road leading straight through. It is anyone's guess.

Using heuristics

You use a heuristic by adding an extra value to your nodes. That is the expected cost of going from it to the target. A node then contains its position, the cost of getting to it, the parent node leading to it and an expected cost of getting from it to the target. That way the expected cost of a path going through this node is the true and well known cost of getting to the node plus the expected cost of reaching the target.

Now we consider the "best" node to be the one with a lowest total cost, and not the one with the lowest actual cost.

If the heuristic is perfect, then the expected cost is equal to the actual cost and every single step we take will lead directly to the destination.

Another quick, less detailed, example to show how a simple heuristics can help, and how the lack of one will waste time path finding.

Here is another cost map. The first number is the travel cost as before, the second is the expected travel cost to reach the destination from that position. For now we will expect the cost to be 3 for every move required when following the most direct path.

Cost map	A	B	C	D	E	F	G
1	1-6	1-6	1-6	1-9	100-12	100-15	
2	4-3	100-3	100-6	100-9	100-12	100-15	
3	1-0	100-3	100-6	100-9			
4	100-3	100-3	100-6	100-9			
5							

Again we start at A1 but this time the destination is A3.

First without heuristics. Just take the cheapest node based on the cost of reaching it. We will go from A1 to B1 to C1 to D1 . Reaching D1 will cost 3. That is the cost of traveling B1+C1+D1. This didn't bring us closer to the destination at all. Now we have to consider A2 which then brings us to A3 at a total cost of 6 from A2+A3.

Now with the heuristics which expect every unknown node on the direct path to the destination to have a travel cost of 3. We add B1,B2 and A2 to the open list and expect the cost of the entire path to the destination through those nodes to be B1=1+6=7, B2=100+3=103, A2=4+3=7.

We select the cheapest one which is either B1 or A2. We now select based on the expected remaining cost which is 3 for A2 and 6 for B1. Therefore we select A2.

With A2 as the current node, we add A3 and B3 with their expected costs and their actual costs (travel cost plus cost to reach parent) A3=1+0+4=5 and B3=100+3+4=107.

Again we select the node with lowest expected total cost. It is A3 which is the destination.

The point of this example was to show how a simple heuristics can help the algorithm get some common sense. If it cost the same to go right and to go left, but the destination is to the right, then it is probably a good idea to go that way and see what happens. If you go right and then reach a dead-end, then you can consider going left.

The most common heuristics is to assume a constant move cost from a node to the destination in a direct line, but as hinted previously, you can make up any good heuristics and just add its guess at a remaining cost (to reach the target from that node) to the actual cost of reaching the node.

Implementation

When this algorithm is implemented in practice, you generally want the open list to be able to quickly return the node with the lowest cost. Running through an ordinary list or array looking for the cheapest node will be slow when the list is long, so this should be optimized. A common container to use is a priority queue implemented as a heap. I will not go into details about that here, but just mention it so it can be looked up elsewhere. Let me just briefly state that inserting, in order, into a priority queue is generally O(log n) and so is extracting, and reordering, the queue. A list on the other hand would be O(n) for sorted insertion and O(1) for extraction.

When you reach the destination node, you have a node pointing towards its parent. While that "pointing" could be done by naming the parent like "B4" it makes a lot more sense to use an actual pointer/reference to the node of the parent. That way you do not have to search through any lists looking for the parent – you have a direct shortcut to it.

The final little point is that if you have a reasonable sized "map" with the states/positions arranged in a regular grid, then you could store the nodes in the grid itself. Instead of a grid where each position holds the cost of traveling that node, it could hold that cost along with the cost of reaching it during the current pathfinding as well as the name of the position which led to this position. This way it is very easy to see if a positions neighbors have already been put in the closed list – then the neighbor positions will have a cost written into them. You will still have to maintain an open list though, since you need to find the best new node to explore and you would probably rather not search through every grid position looking for it.

Final details

There are a few details, perhaps more than a few, which I have left out until now.

Optimistic heuristic

One important thing to note is that the heuristics need to be realistic or optimistic. It should never estimate a longer path to the target that the true path. If it does, then you can end in a situation where you arrive at a position from the wrong side so to speak. Imagine that you want to go into the garden and that you are inside the house. You can go away from the garden out through the door and around the house or you can go down into the basement and dig a tunnel to the neighbor' house and from there you enter the garden. If you falsely assume that the cost of using the door is one million, while the real cost is ten, and you assume, correctly, that the tunnel path has a cost of a hundred thousand, then you go through the tunnel and end in the garden through a very non-optimal path.

The example that I created to demonstrate heuristics was actually faulty in that it assumed a higher cost than the real one. I did that because I knew it would not create trouble (in that scenario) and because it made it easier to make a simple example.

The problem here is that you stop calculating paths when you arrive at the destination, so you never realize that the door would have been a better option. If you ensure that your estimate is accurate or at least always lower than the actual cost, then you are ensured that when you reach the destination, it is by following a path which is optimal. There may exist other equally good paths – but none that are better.

Never have negative travel cost

As with the non optimistic heuristic, you run into trouble if you have a state which subtracts from your cost. If you have a path which leads you into the wilderness for a year and then lets you step into a time machine which transports you to the destination and one year back in time, then you have such a negative cost.

A more realistic example could be a cost measure which is your physical well being while you walk along the path. Walking will tire you, but if you arrive at an oasis you can drink, eat and relax before moving on. Then the cost – your fatigue – will be subtracted from.

Obviously the only way to ensure that you find that strange, but yet optimal, path is to search the entire domain. This is not something that this algorithm will do for you. The whole point of it is to search as little as possible.