Pathfinding Algorithms

A path-finding algorithm is a means of finding a way from A to B. This way is not necessarily the fastest, or the shortest, but is simply one of the methods to get to the end. These algorithms typically rely on a numeric "weight" in order to determine how to approach the system they need to examine. In terms of navigation the "weight" could refer to the time it takes to get between two points, or simply just the distance. Changing the inspected variable in such a way would allow a Sat Nav to differentiate between a short, traffic-clogged route, and a longer, traffic-free route.

Dijkstra's Algorithm

How it works

Dijkstra's algorithm is one of the simplest path-finding solutions available. Its goal is to find the shortest path from one point to another. To do so it takes into account a set of factors for each node: what nodes it can connect to, the weight between itself and those nodes, whether those nodes have already been visited, and whether it can improve the best route to one of its neighbouring nodes. Within my program, Dijkstra is the algorithm utilised, and the method of weighting the links is the Euclidean distance between the two nodes.

When Dijkstra's algorithm begins it visits the starting node, and looks at all the nodes that it can connect to. Next it will save those nodes into a queue, ordered from the shortest distance, to the longest distance. The nodes in the queue will save information about: what the shortest distance to them is, and what node can connect to them with that value. The starting node is marked as visited, and the algorithm moves on.

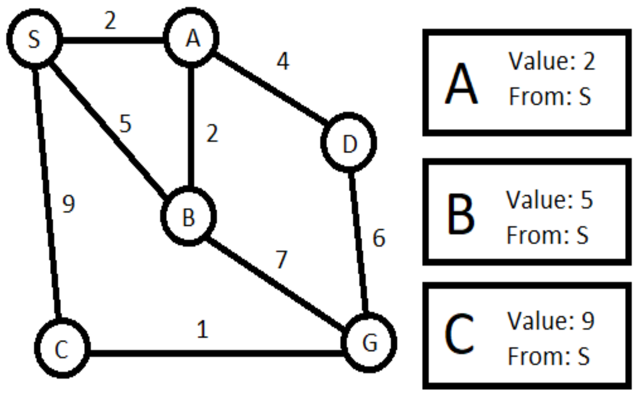


Illustration 1: The queue created by visiting the starting node, S.

The first item in the queue (the one that has the shortest total distance) is the next one the algorithm visits. When the algorithm gets to node A it does the same thing as before: inspect the possible neighbours, and find the distance to them. Previously visited neighbours (S) are not inspected. It will then compare the shortest route that can be found through A, with the shortest routes already stored in its neighbours. If the route to a neighbour is shorter through A then the value originally stored will be overwritten. This can be seen in the examples, as the route SAB is shorter than the route SB. Once all changes are made (node B) and new nodes have been added (node D) the queue order is updated, to maintain the distance based sorting.

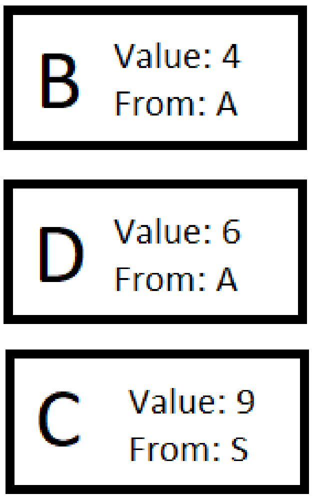


Illustration 2: The queue following the changes made to it by visiting the fist node in the queue, A.

The algorithm will continue this way until the goal (G) is first in the queue. With this graph, after B, the goal would feature at the end of the queue with a value of 11. Then D would be inspected and return a value of 12, so the shortest route to G would be unchanged.

At this point the list consists of C at 9, and G at 11 so C would be visited. From C, G can be reached in just 10 units instead of 11, so the value becomes 10, and the from becomes C. With G now the first item in the queue, all that's left to do is for the algorithm to return the optimal path to the solution. It does this by examining which nodes gave the shortest path to a given node. By starting at the goal, it can see that C was the node that gave the shortest path. Then, by going to C, it can be seen that the node it was visited from was S. This process repeats until the start node is reached and, from that, the shortest route is found to be SCG.

Advantages and Disadvantages

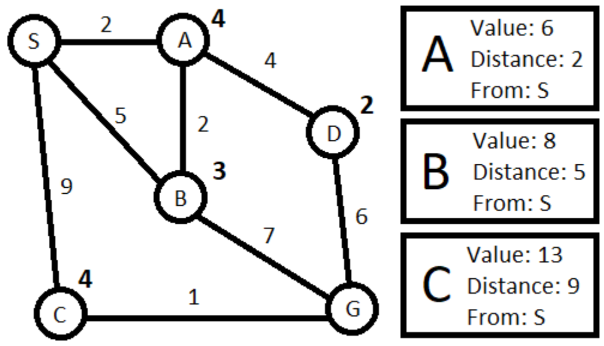
The big advantage of Dijkstra's algorithm is that it finds the absolute shortest route possible. However, that is also its greatest weakness. The algorithm will search through almost every possible node to find a solution. The only nodes it won't search are ones that have their shortest distance being further than the shortest distance to the goal. This results in a comparatively slow search time, compared to other methods.

A Star (A*)

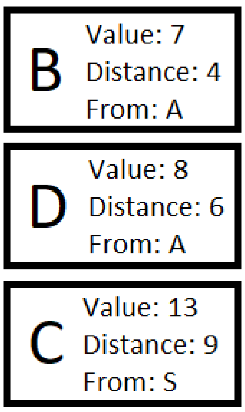
How it works

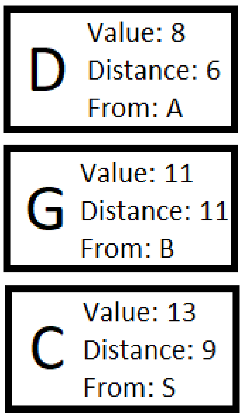
This algorithm is relatively similar to Dijkstra's in the way that it works. Except it has one big difference in purpose. It does not aim to find the shortest route, but it aims to find a route quickly. This difference is achieved by adding an additional criteria to the way the queue is ordered. Whereas in Dijkstra only the distance to the node is accounted for, A* also factors in the distance from an inspected node to the goal.

The initial stage of A* on the example graph results in a queue that looks very similar to the Dijkstra equivalent. However, this one tracks the distance separately from the value which the queue is ordered by. This time the value is the link weight added to the distance from a node, to the goal. Keeping a track of the distance separately is important though, because we don't want to have the node-to-goal distance accumulating throughout the graph. When the algorithm visits A, the value for D should be 8, and not 12. The correct value is found by adding the values of the two links (2 + 4) and the node-to-goal distance of D (2) together, which results in 2+4+2 = 8.


S
A
B
C
D
G
2
4
5
2
9
3
7
6
4
1
2
4
A Value: 6 Distance: 2 From: S
B Value: 8 Distance: 5 From: S
C Value: 13 Distance: 9 From: S
Illustration 3: An updated graph for A, where the distance from node to goal is listed beside the nodes.*

Again, the next queue order (ill. 4) doesn't look much different, but it's after node B (ill. 5) that the difference really shows. The queue in illustration 5 lists G as coming before C. With the prior knowledge given from examining the results of Dijkstra on the this graph, it is known that the shortest route to the goal actually lies through C. However, the heuristics of A* will avoid C entirely, in favour of finding a quicker solution.


B Value: 7 Distance: 4 From: A
D Value: 8 Distance: 6 From: A
C Value: 13 Distance: 9 From: S
Illustration 4: The queue after A visits node A.*


D Value: 8 Distance: 6 From: A
G Value: 11 Distance: 11 From: B
C Value: 13 Distance: 9 From: S
Illustration 5: The queue after A visits node B*

Advantages and Disadvantages

A* finds its place in being an efficient algorithm. By doing a non-exhaustive search on the graph it spends much less time looking for a solution. However, it's still possible, with the correct graph layout, to force A* to look at all possible nodes, just as Dijkstra's algorithm would. A* is also not as good at finding a short path, and would be a bad choice if the shortest route was being tested for (without any time constraints).