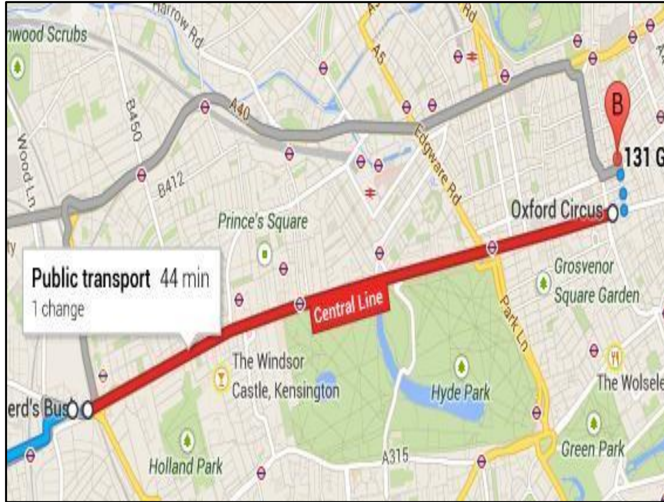
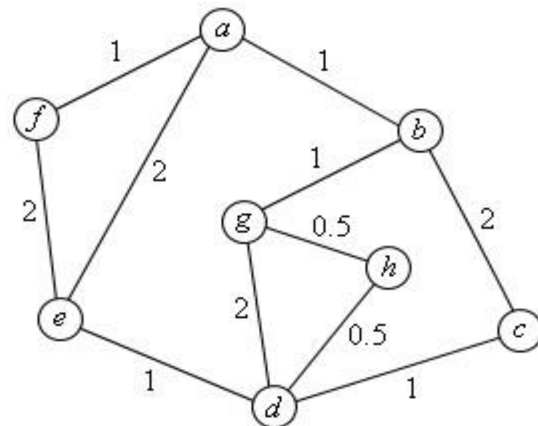
Dijkstra's Algorithm



Let's consider a scenario where Harry has to travel from Oxford Circus to the Bus stop, Google makes his work easier by finding the shortest path between two places amidst n number of available path. Have you ever wondered how Google Maps are so efficient in determining the shortest path between two places and how maze robot plan their path?

Well, Dijksrtra's algorithm forms the basis. Dijkstra's algorithm is a greedy algorithm used to find the single source shortest path between a given point to any other point. Before diving into the algorithm, lets understand few basic terminologies,



- Edges: Edges refer to the path (ef, fa, ab, bg, gd…).
- Node: Node refers to the intersection of two edges (a, b, c, d,…).
- Cost: Cost refers to any quantity. It can be a distance, traffic jam factor, road risk, or a combination of all these. With reference to the above diagram the cost between f and e is 2.
- Movement Cost: The movement cost represents a total cost sum from a starting point to any other node.

- Closed List: Closed List contains a set of paths leading to the shortest path.
- Open List: Open List contains a set of all possible path from a given parent node.

Note: Both the open and closed list contain {possible node, cost, parent node}

Let us consider the above diagram for reference to find the shortest path from node g to node f:

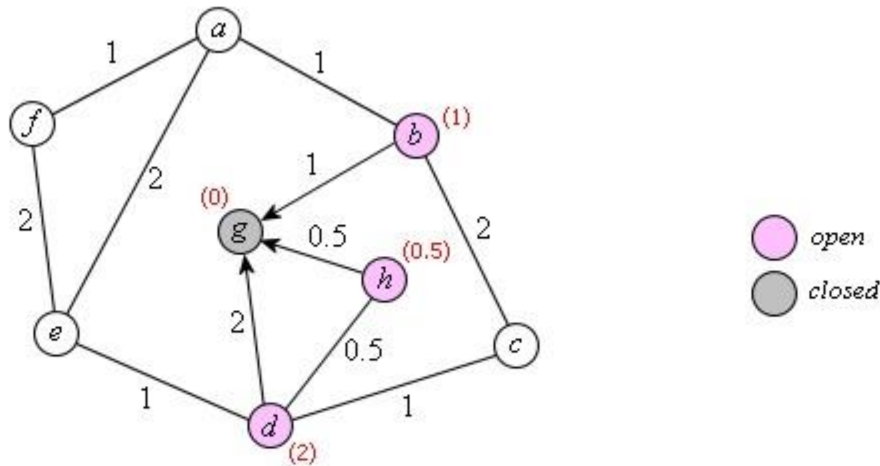
Step 1:

The open and closed list are declared as an empty array. The movement cost is set to null. The current node, i.e, g is assigned to a default value zero.

Step 2:

Now, put the starting node g into open list, so the open list becomes, **open = { { g, 0, null } }.** It simply means open is an array with one element in it, that one element is a node with name g, has a movement cost of 0, and has a parent of null. Find all immediate neighbors for the node. g has three neighbors, b, h, and d. Set parent node for all node b, h, and d to g, and calculate movement cost for node b, h, and d to 0+1=1, 0+0.5=0.5, and 0+2=2 respectively. The open list should now contain,

- open = { { b, 1, g }, { h, 0.5, g }, { d, 2, g } } closed = { { g, 0, null } }.

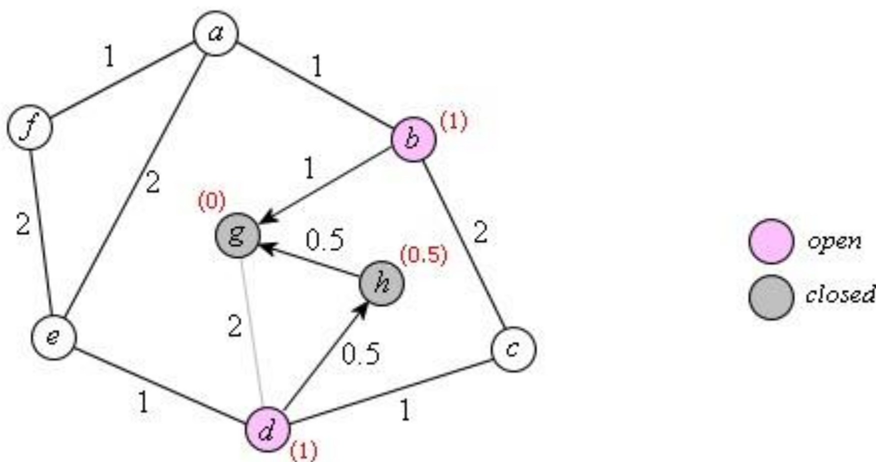


Step 3:

Choose the node with the lowest movement cost from open list. Node h has the lowest movement cost of (0.5), compared to b with (1), or d with (2). Now, h is the picked node and g and d are its neighbors. Node g is in the closed list, so just ignore it. The other neighbor node d is already a member of open list, meaning d has a parent (node g) and a current movement cost (2). Because of this we can't just change the parent and movement cost like we did previously. What to do now is to check which path is shorter. The current movement cost for d is (2), this is due to d's parent is g, and the travel cost from g to d is (0)+2 = 2. The current movement cost for h however is (0.5), and to travel from h to d only an additional cost of 0.5 is needed, making the path g->h->d to cover only 1 unit of movement cost. Clearly path with less movement cost is to be favored. To reflect this, reassign d's parent to h and recalculate its movement cost to (0.5)+0.5 = 1. Node h can now be removed from open list and closed. At this point open list should be, open = { { b, 1, g}, { d, 1,

h } }, and closed list is, closed = { { g, 0, null }, { h, 0.5, g } }. Now, node d in open list has changed, movement cost is now 1 and parent is node h.

- open = { { b, 1, g}, { d, 1, h } } □ closed={{g,0,null},{h,0.5,g}}.

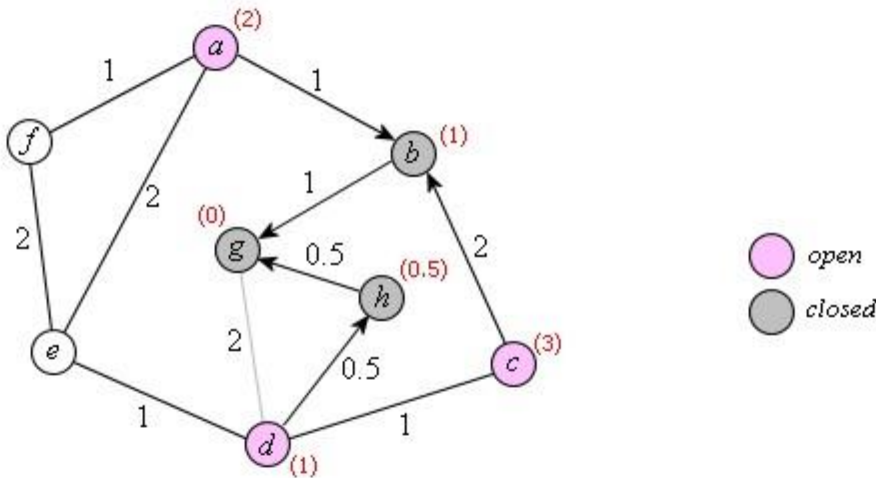


Step 4:

Pick a node with lowest movement cost from open list. This time we have a tie, both node b and d has equal movement cost of (1). In this case, pick b because it comes first. From step two, find all of b's neighbors, and we get two nodes, a and c. Both a and c are not members of open list so set their parent to b, and calculate their movement cost with (1)+1 and (1)+2 respectively. (1)+1 is due to (b)+a, and (1)+2 is (b)+c, movement cost wise. Put both node a and c to open list and close node b by removing it from open list and transferring to closed list. At this time open list should be,

- open = { { d, 1, h }, { a, 2, b }, { c, 3, b } }

closed = { { g, 0, null }, { h, 0.5, g }, { b, 1, g } }.

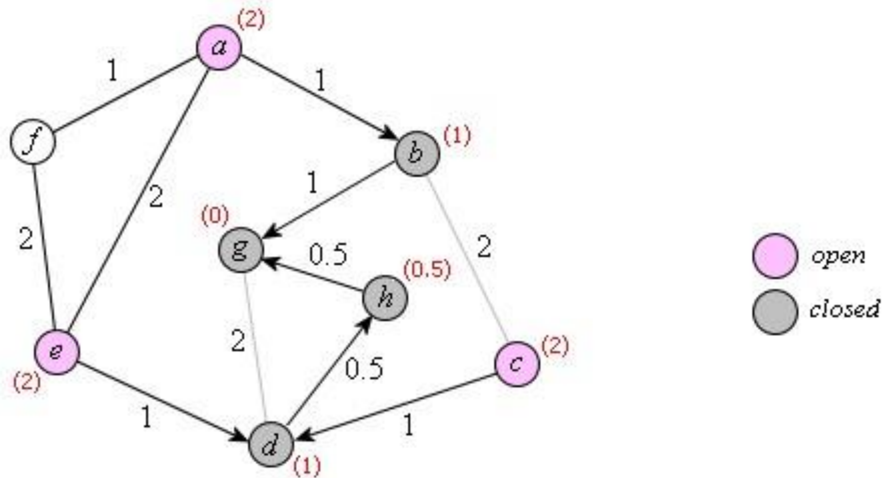


Step 5:

Node with lowest movement cost in open list is d, and its neighbors are g, h, c, and e. Node g and h are already closed so ignore them. Node c however is already a member of open list. Currently node c is a child of b, and the movement cost of c from b is (3). However, since movement cost for node d is (1), it takes only 2 movement cost to get to c from d. Clearly moving to c from d is much better than from b. Because of this reassign c's parent node to d, and recalculate its movement cost to (2). Now we check the last neighbor e. Node e is not in open list, so just assign its parent to d, calculate its movement cost to (1)+1, and put it into the open list. Close node d. Confirm that open list is now,

- open = { { a, 2, b }, { c, 2, d }, { e, 2, d } }

closed = { { g, 0, null } , { h, 0.5, g }, { b, 1, g }, { d, 1, h }}.

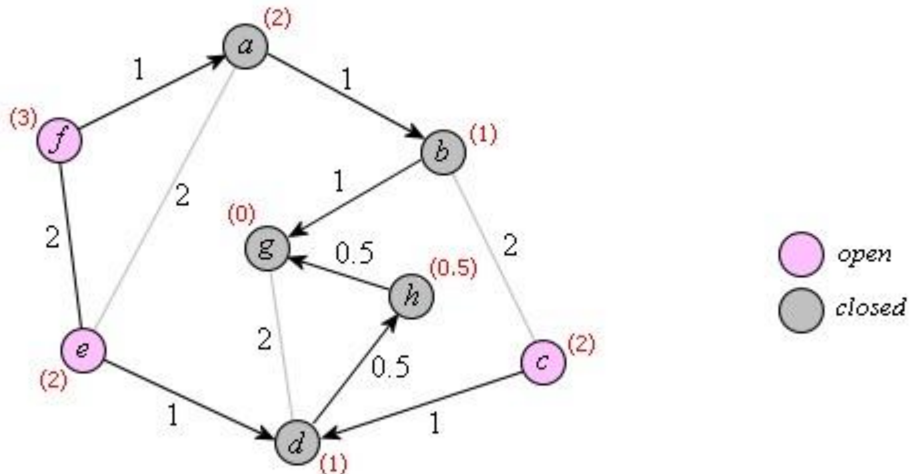


Step 6:

Pick node with lowest movement cost from open list, at this time all a, c, and e ties with movement cost of (2). Just pick node a because it comes first in the list. Find all of its neighbors to get node b, e, and f. Node b is closed ignore it. Node e is in the open list and currently has a movement cost of (2). To get to e from a, total movement cost of 4 is needed, twice the current movement cost of e, so just leave node e be. Node f is not in open list, so assign node a as its parent, calculate its movement cost to (2)+1, and put it into the open list. Close node a. Right now open list should be,

 open = { { c, 2, d }, { e, 2, d }, { f, 3, a } }

closed = { { g, 0 , null } , { h, 0.5 , g }, { b, 1, g }, { d, 1, h }, { a, 2, b }}.

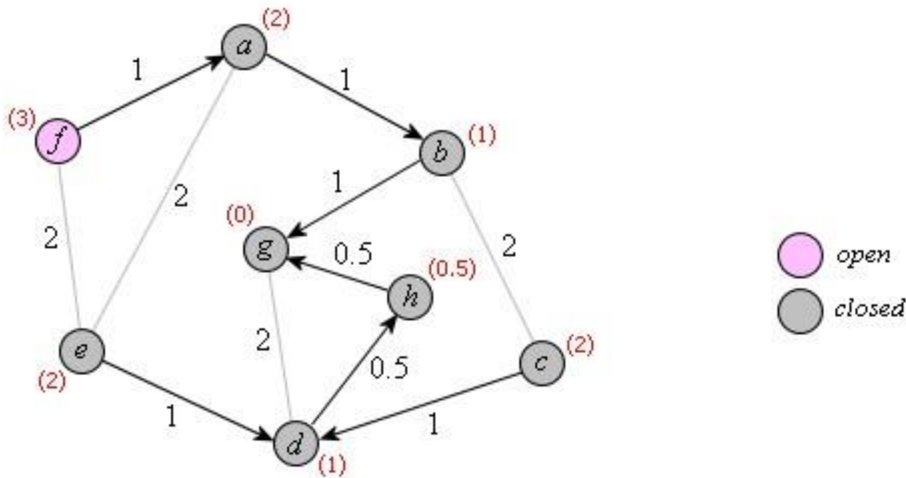


Step 7:

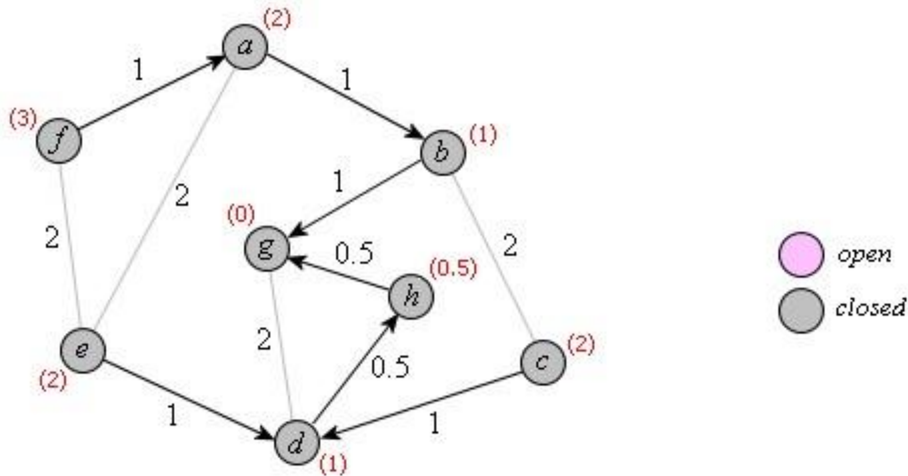
Pick node c from open list and we find that all of its neighbors, node b and d, are already closed. Since there is nothing else to do, close node c. Now open list should be, open = { { e, 2, d }, { f, 3, a } }, and closed = { { g, 0 , null } , { h, 0.5 , g }, { b, 1, g }, { d, 1, h }, { a, 2, b }, { c, 2, d } }. Next pick node e from open list and we find that it has three neighbors, node a, d, and f. Node a and d are closed just ignore them. Node f however is already a member of open list and currently has a movement cost of (3). Moving from e to f takes (2)+2 = 4, larger than (3) so don't do any modification to node f. Close node e.

- open = { { f, 3, a } }

- closed = { { g, 0, null }, { h, 0.5, g }, { b, 1, g }, { d, 1, h }, { a, 2, b }, { c, 2, d }, { e, 2, d } }.



Only one node remaining in our open list. So pick it and do the routine neighbor check. It has node a and e as neighbors however both node a and e are already closed. Finally we close node f by putting it into the closed list. Open list then becomes, open = { }, and closed list becomes, closed = { { g, 0, null }, { h, 0.5, g }, { b, 1, g
}, { d, 1, h }, { a, 2, b }, { c, 2, d }, { e, 2, d }, { f, 3, a } }. That's it. The search is done, open list is now empty and target node f is in the closed list. In fact not just between two nodes, the algorithm actually found the shortest path to ALL other nodes (yes not just to f) from node g. The end result of Dijkstra’s search can be described by the figure below. Black arrow lines represent child/parent relationship between nodes. A child node always points to its parent, arrow wise. The gray lines are paths that will never be used.



Looking at the figure, if any node is chosen other than g, the path from that node to node g can be traced by simply following the arrow, and one can be sure it is the shortest possible path to g.

This is how the shortest path between two points is found and hence this algorithm finds its application in various domains like transportation planning and packet routing in communication networks, including the Internet. Multitudes of less obvious applications include finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling. In the world of entertainment, one can mention path finding in video games and finding best solutions Top puzzles using their state-space graphs.