****

**Dijkstra's algorithm**, is a graph search algorithm that solves the single-source shortest path problem for a graph with non negative edge path costs.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

**Algorithm**

Dijkstra's Algorithm finds the shortest path from a starting vertex to all other vertices in a graph. It should be noted that distance between nodes can also be referred to as weight.

1. Create a distance list, a previous vertex list, a visited list, and a current vertex.
2. All the values in the distance list are set to infinity except the starting vertex which is set to zero.
3. All values in visited list are set to false.
4. All values in the previous list are set to -1.
5. Current node is set as the starting vertex.
6. Mark the current vertex as visited.
7. Update distance and previous lists based on those vertices which can be immediately reached from the current vertex.
8. Update the current vertex to the unvisited vertex that can be reached by the shortest path from the starting vertex.
9. Repeat (from step 6) until all nodes are visited.

## Pseudocode

In the following algorithm, u := extract\_min(Q) searches for the vertex u in the vertex set Q that has the least dist[u] value. That vertex is removed from the set Q and returned to the user. length(u, v) calculates the length between the two neighbor-nodes u and v. alt on line 10 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

1 **function** Dijkstra(*Graph*, *source*):

2 **for each** vertex *v* in *Graph*: *// Initializations*

3 dist[*v*] := infinity *// Unknown distance function from source to v*

4 previous[*v*] := undefined

5 dist[*source*] := 0 *// Distance from source to source*

6 *Q* := copy(*Graph*) *// All nodes in the graph are unoptimized - thus are in Q*

7 **while** *Q* **is not** empty: *// The main loop*

8 *u* := extract\_min(*Q*) *// Remove best vertex from priority queue; returns source on first iteration*

9 **for each** neighbor *v* of *u*: *// where v has not yet been considered*

1. *alt* = dist[*u*] + length(*u*, *v*)
2. **if** *alt* < dist[*v*] *// Relax (u,v)*
3. dist[*v*] := *alt*
4. previous[*v*] := *u*

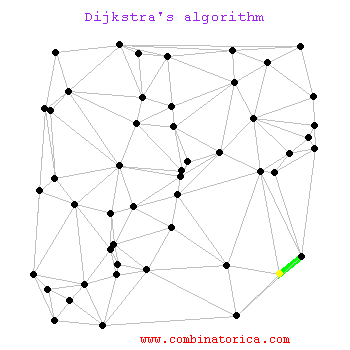
14 **return** previous[]

If we are only interested in a shortest path between vertices source and target, we can terminate the search at line 9 if u = target. Now we can read the shortest path from source to target by iteration:

*S := empty sequence  
 u := target* ***while*** *defined previous[u]  
 insert u at the beginning of S  
 u := previous[u]*  
Now sequence S is the list of vertices constituting one of the shortest paths from source to target, or the empty sequence if no path exists.  
A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of previous[] we would store all nodes satisfying the relaxation condition. For example, if both r and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both r and source to previous[target]. When the algorithm completes, previous[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these short paths between two given nodes we would use path finding algorithm on the new graph, such as depth first search







REFERENCES

<http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml>  
<http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm>  
<http://students.ceid.upatras.gr/~papagel/english/java_docs/minDijk.htm>

Sample Outputs

