Dijkstra's Algorithm for Shortest Path in a Weighted Graph

Given a graph and a source vertex in the graph, find the shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is a variation of the BFS algorithm. In Dijkstra's Algorithm, a SPT*(shortest path tree)* is generated with given source as root. Each node at this SPT stores the value of the shortest path from the source vertex to the current vertex. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

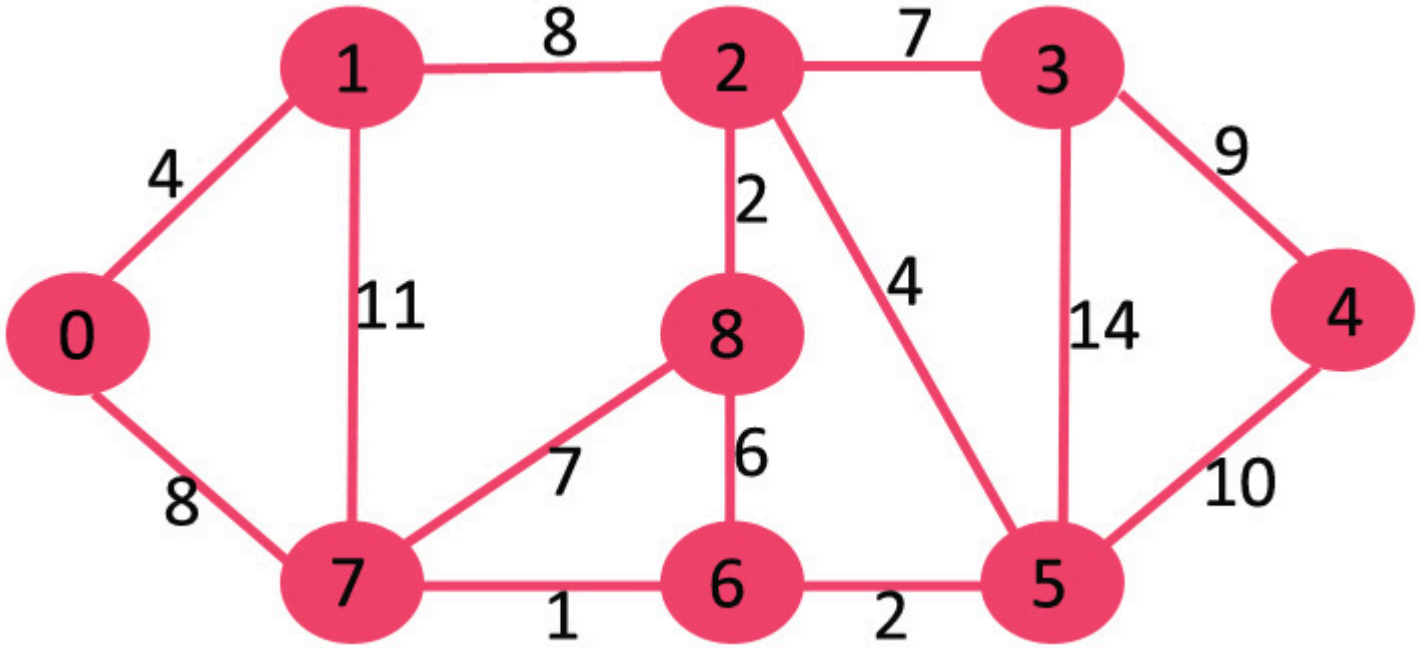
Below is the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given weighted graph.

Algorithm:

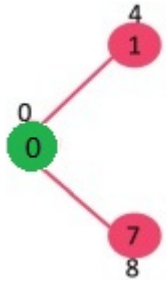
1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* doesn't include all vertices:
    - Pick a vertex u which is not there in *sptSet* and has minimum distance value.
    - Include u to *sptSet*.
    - Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of

edge u-v, is less than the distance value of v, then update the distance value of v.

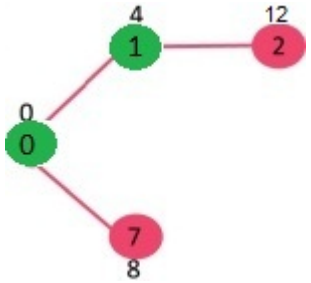
Let us understand the above algorithm with the help of an example. Consider the below given graph:



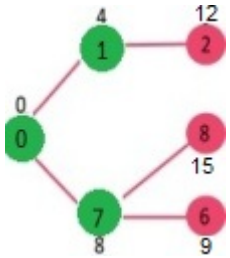
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



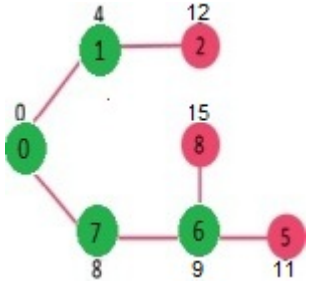
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

Implementation:
Since at every step we need to find the vertex with minimum distance from the source vertex from the set of vertices currently not added to the SPT, so we can use a min heap for easier and efficient implementation. Below is the complete algorithm using priority_queue(min heap) to implement Dijkstra's Algorithm:

```
1) Initialize distances of all vertices as infinite.

2) Create an empty priority_queue pq. Every item
   of pq is a pair (weight, vertex). Weight (or
   distance) is used as the first item of pair
   as the first item is by default used to compare
   two pairs

3) Insert source vertex into pq and make its
   distance as 0.

4) While either pq doesn't become empty
    a) Extract minimum distance vertex from pq.
       Let the extracted vertex be u.
    b) Loop through all adjacent of u and do
       following for every vertex v.

           // If there is a shorter path to v
           // through u.
           If dist[v] > dist[u] + weight(u, v)

               (i) Update distance of v, i.e., do
                     dist[v] = dist[u] + weight(u, v)
               (ii) Insert v into the pq (Even if v is
                    already there)
```

Dash
All
Articles
Videos
Problems
Quiz
Contest
Get 90% Refund!
Courses Tutorials Jobs Practice Contests

```
   paths.
```

C++ Java


```java
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath {
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static final int V = 9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex   Distance from Source\n");
        for (int i = 0; i < V; i++)
            System.out.println(i + "                " + dist[i]+"\n");
    }

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src)
```

```
{
    int dist[] = new int[V]; // The output array. dist[i] will hold
    // the shortest distance from src to i

    // sptSet[i] will true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized
    Boolean sptSet[] = new Boolean[V];

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++) {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices
        // not yet processed. u is always equal to src in first
        // iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
        // picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an
            // edge from u to v, and total weight of path from src to
            // v through u is smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] != 0 &&
               dist[u] != Integer.MAX_VALUE && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
```

```
        }

        // Driver method
        public static void main(String[] args)
        {
            /* Let us create the example graph discussed above */
            int graph[][] = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                          { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                          { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                          { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                          { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                          { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                          { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                          { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                          { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
            ShortestPath t = new ShortestPath();
            t.dijkstra(graph, 0);
        }
    }
```

Output:

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14
```

Dash
All
Articles
Videos
Problems
Quiz
Contest

Time Complexity: The time complexity of the Dijkstra's Algorithm when implemented using a min heap is O(E * logV), where E is the number of Edges and V is the number of vertices.

Note: The Dijkstra's Algorithm **doesn't work** in the case when the Graph has negative edge weight.

Mark as Read

Report An Issue

If you are facing any issue on this page. Please let us know.