How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm

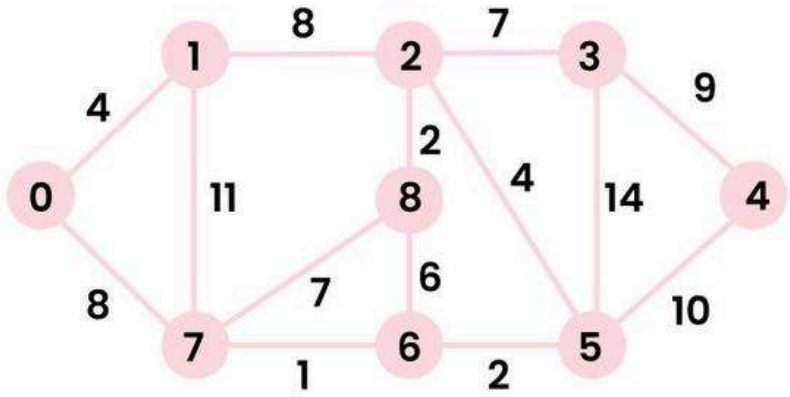
Last Updated : 04 Jun, 2024

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

Note: The given graph does not contain any negative edge.

Examples:

Input: *src = 0, the graph is shown below.*



Working of Dijkstra's Algorithm

Output: *0 4 12 19 21 11 9 8 14*

Explanation: *The distance from 0 to 1 = 4.*

The minimum distance from 0 to 2 = 12. 0->1->2

The minimum distance from 0 to 3 = 19. 0->1->2->3

The minimum distance from 0 to 4 = 21. 0->7->6->5->4

The minimum distance from 0 to 5 = 11. 0->7->6->5

The minimum distance from 0 to 6 = 9. 0->7->6

The minimum distance from 0 to 7 = 8. 0->7

The minimum distance from 0 to 8 = 14. 0->1->2->8

Dijkstra's Algorithm using Adjacency Matrix:

The idea is to generate a ***SPT (shortest path tree)*** *with a given source as a root. Maintain an Adjacency Matrix with two sets,*

- *one set contains vertices included in the shortest-path tree,*
- *other set includes vertices not yet included in the shortest-path tree.*

Algorithm:

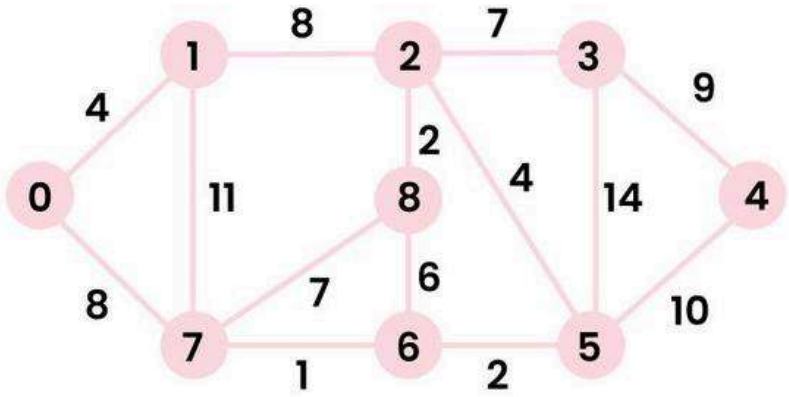
- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance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

Note: We use a boolean array **sptSet[]** to represent the set of vertices included in **SPT**. If a value **sptSet[v]** is true, then vertex v is included in **SPT**, otherwise not. Array **dist[]** is used to store the shortest distance values of all vertices.

Illustration of Dijkstra Algorithm:

To understand the Dijkstra's Algorithm lets take a graph and find the shortest path from source to all nodes.

*Consider below graph and **src = 0***

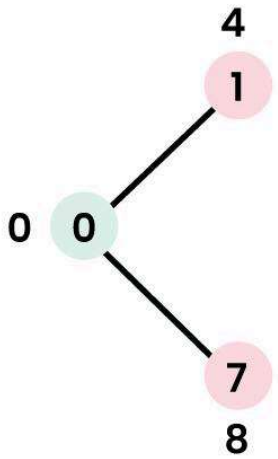


Working of Dijkstra's Algorithm

Step 1:

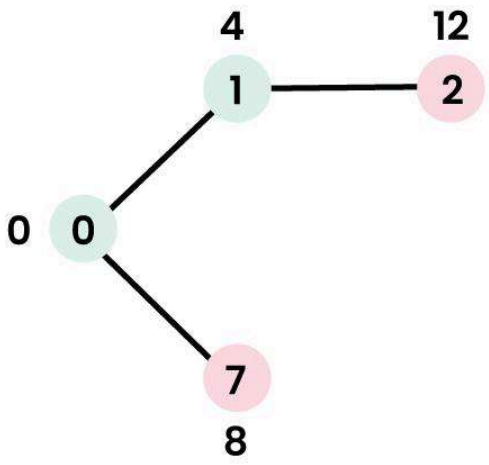
- *The set **sptSet** is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where **INF** indicates infinite.*
- *Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in **sptSet**. So **sptSet** becomes {0}. After including 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.*

*The 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.*



Working of Dijkstra's Algorithm

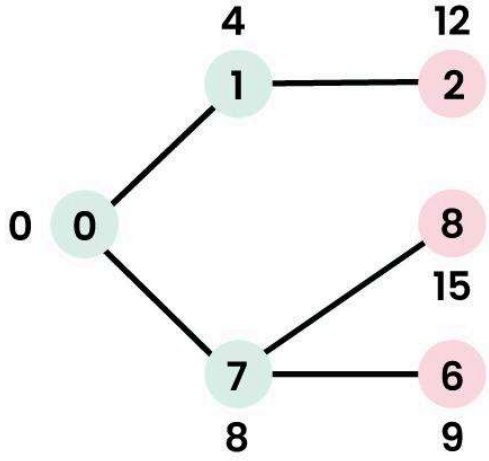
- *Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). The vertex 1 is picked and added to **sptSet**.*
- *So **sptSet** now becomes {0, 1}. Update the distance values of adjacent vertices of 1.*
- *The distance value of vertex 2 becomes **12**.*



Working of Dijkstra's Algorithm

Step 3:

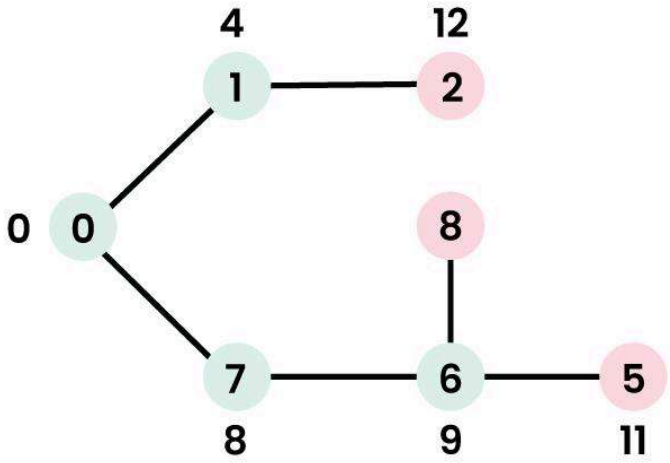
- *Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). Vertex 7 is picked. So **sptSet** now becomes **{0, 1, 7}**.*
- *Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (**15 and 9** respectively).*



Working of Dijkstra's Algorithm

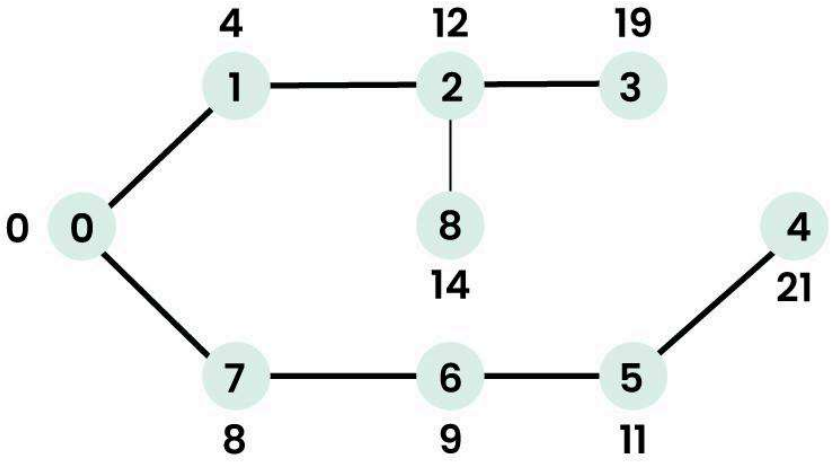
Step 4:

- *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.*



Working of Dijkstra's Algorithm

*We repeat the above steps until **sptSet** includes all vertices of the given graph. Finally, we get the following S**hortest Path Tree (SPT).***



Working of Dijkstra's Algorithm

C++ C Java Python C# Javascript

```c
// C program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    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[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
                 // shortest
    // distance from src to i

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

    // Initialize all distances as INFINITE and stpSet[] as
    // 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 the 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]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver's code
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 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 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}
```

Output

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

Time Complexity: $O(V^2)$
Auxiliary Space: O(V)

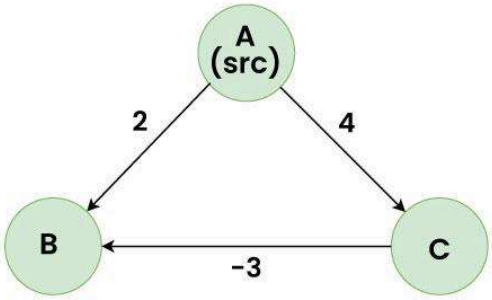
Notes:

- The code calculates the shortest distance but doesn't calculate the path information. Create a parent array, update the parent array when distance is updated and use it to show the shortest path from source to different vertices.
- The time Complexity of the implementation is **$O(V^2)$**. If the input graph is represented using adjacency list, it can be reduced to O(E * log V) with the help of a binary heap. Please see Dijkstra's Algorithm for Adjacency List Representation for more details.
- Dijkstra's algorithm doesn't work for graphs with negative weight cycles.

Why Dijkstra's Algorithms fails for the Graphs having Negative Edges ?

The problem with negative weights arises from the fact that Dijkstra's algorithm assumes that once a node is added to the set of visited nodes, its distance is finalized and will not change. However, in the presence of negative weights, this assumption can lead to incorrect results.

Consider the following graph for the example:



Shortest Distance A to B using Dijkstra = 2
Actual Shortest Distance A to B = 4 - 3 = 1

Failure of Dijkstra in case of negative edges.

In the above graph, A is the source node, among the edges **A** to **B** and **A** to **C** , **A** to **B** is the smaller weight and Dijkstra assigns the shortest distance of **B** as 2, but because of existence of a negative edge from **C** to **B**, the actual shortest distance reduces to 1 which Dijkstra fails to detect.

Note: We use Bellman Ford's Shortest path algorithm in case we have negative edges in the graph.

Dijkstra's Algorithm using Adjacency List in O(E logV):

*For Dijkstra's algorithm, it is always recommended to use **Heap** (or **priority queue**) as the required operations (extract minimum and decrease key) match with the speciality of the heap (or priority queue). However, the problem is, that priority_queue doesn't support the decrease key. To resolve this problem, do not update a key, but insert one more copy of it. So we allow multiple instances of the same vertex in the priority queue. This approach doesn't require decreasing key operations and has below important properties.*

- *Whenever the distance of a vertex is reduced, we add one more instance of a vertex in priority_queue. Even if there are multiple instances, we only consider the instance with minimum distance and ignore other instances.*

- *The time complexity remains **O(E * LogV)** as there will be at most O(E) vertices in the priority queue and O(logE) is the same as O(logV)*

Below is the implementation of the above approach:

C++ Java Python C# Javascript

```cpp
// C++ Program to find Dijkstra's shortest path using
// priority_queue in STL
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// This class represents a directed graph using
// adjacency list representation
class Graph {
    int V; // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list<pair<int, int> >* adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair>[V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}
```