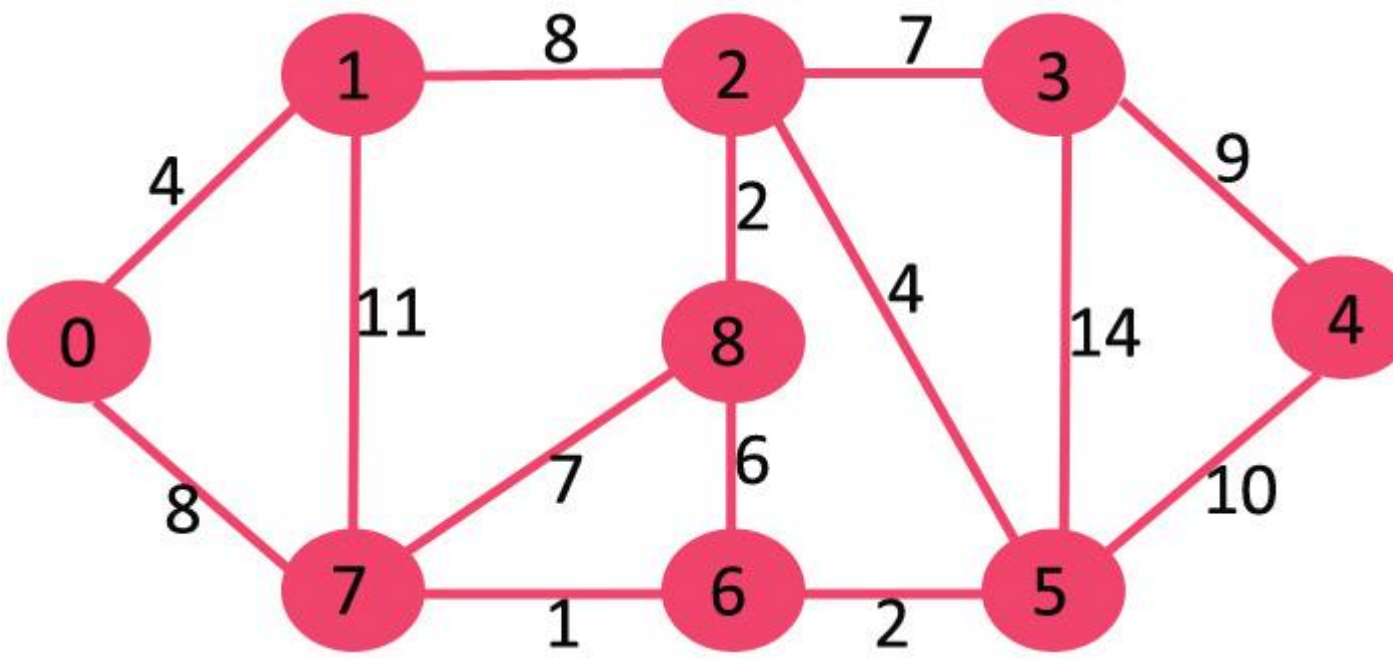
How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm

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

Examples:

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



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 shortest path algorithm using Prim's Algorithm in $O(V^2)$:

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree.

Like Prim's MST, generate a *SPT (shortest path tree)* with a given source as a root. Maintain two sets, one set contains vertices included in the shortest-path tree, other set includes

vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Follow the steps below to solve the problem:

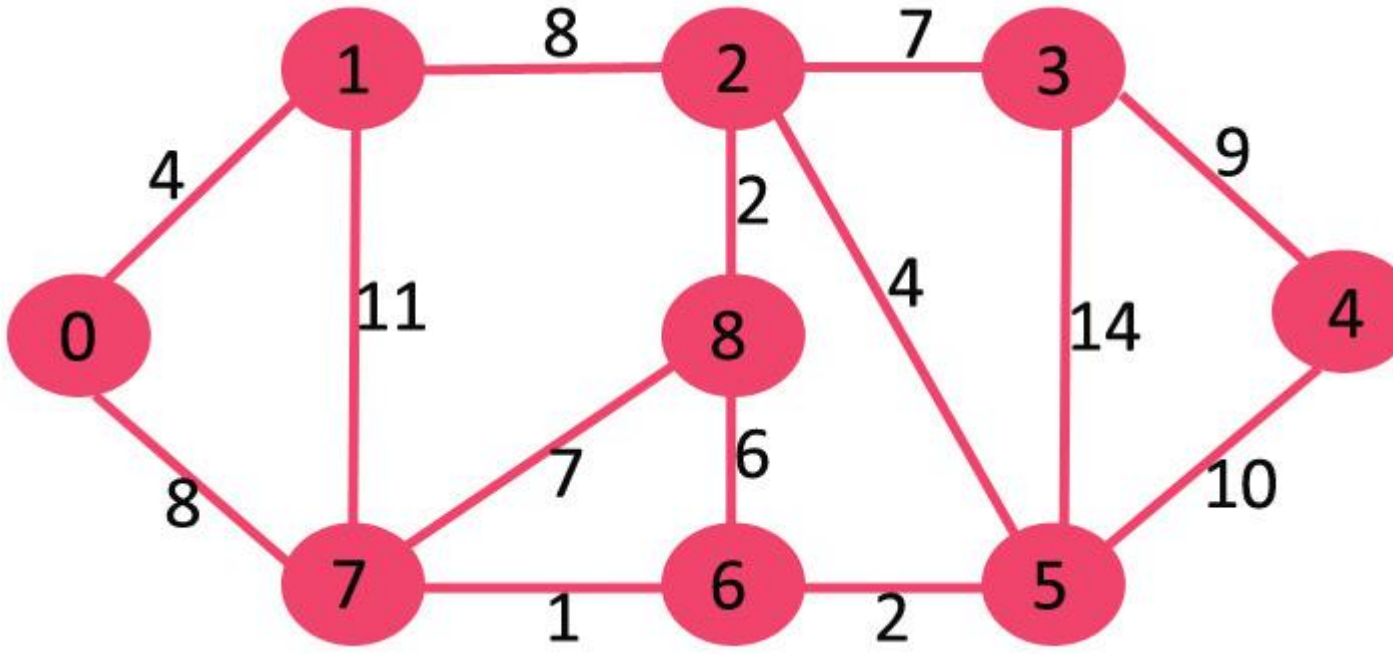
- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance 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.

Below is the illustration of the above approach:

Illustration:

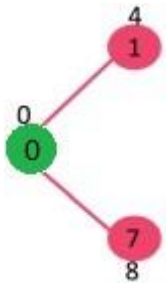
To understand the Dijkstra's Algorithm lets take a graph and find the shortest path from source to all nodes.
Consider below graph and **src = 0**



Step 1:

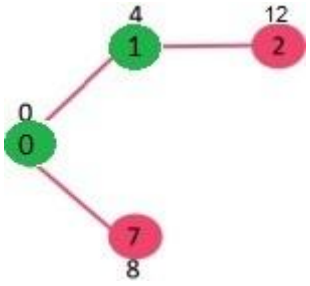
- The set ***sptSet*** is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where **INF** indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to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



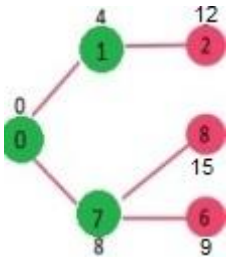
Step 2:

- 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**.



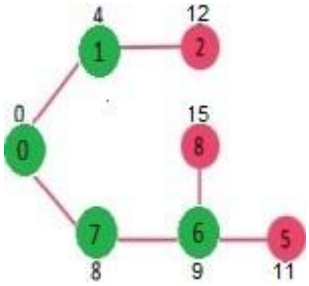
Step 3:

- 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).

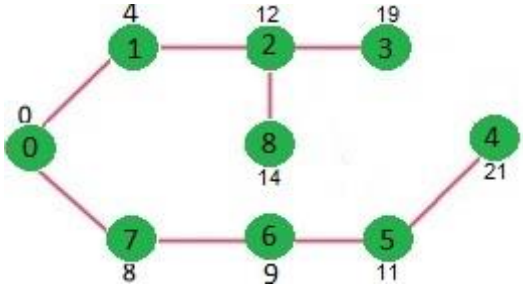


Step 4:

- 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.



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



Below is the implementation of the above approach:

```
// 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
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, 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 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 (like prim's implementation), and use it to show the shortest path from source to different vertices.
- The code is for undirected graphs, the same Dijkstra function can be used for directed graphs also.

- The code finds the shortest distances from the source to all vertices. If we are interested only in the shortest distance from the source to a single target, break them for a loop when the picked minimum distance vertex is equal to the target.
- 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. It may give correct results for a graph with negative edges but you must allow a vertex can be visited multiple times and that version will lose its fast time complexity. For graphs with negative weight edges and cycles, the Bellman-Ford algorithm can be used, we will soon be discussing it as a separate post.

Dijkstra's shortest path algorithm using Heap 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 specialty 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++ 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));
}

// Prints shortest paths from src to all other vertices
void Graph::shortestPath(int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // https://www.geeksforgeeks.org/implement-min-heap-using-stl/
    priority_queue<iPair, vector<iPair>, greater<iPair> >
        pq;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    /* Looping till priority queue becomes empty (or all
    distances are not finalized) */
    while (!pq.empty()) {
        // The first vertex in pair is the minimum distance
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted distance (distance must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

        // 'i' is used to get all adjacent vertices of a
        // vertex
        list<pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i) {
            // Get vertex label and weight of current
            // adjacent of u.
            int v = (*i).first;
            int weight = (*i).second;
```

```
            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight) {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Driver's code
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    // Function call
    g.shortestPath(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(E * logV), Where E is the number of edges and V is the number of vertices.
Auxiliary Space: O(V)