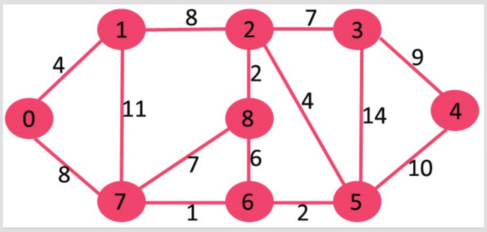
Dijkstra’s Shortest Path Algorithm using priority_queue of STL

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



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

We have discussed Dijkstra’s shortest Path implementations.

- Dijkstra’s Algorithm for Adjacency Matrix Representation (In C/C++ with time complexity O(v^2)
- Dijkstra’s Algorithm for Adjacency List Representation (In C with Time Complexity O(ELogV))
- Dijkstra’s shortest path algorithm using set in STL (In C++ with Time Complexity O(ELogV))

The second implementation is time complexity wise better, but is really complex as we have implemented our own priority queue.

The Third implementation is simpler as it uses STL. The issue with third implementation is, it uses set which in turn uses Self-Balancing Binary Search Trees. 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 speciality of heap (or priority queue). However, the problem is, priority_queue doesn’t support decrease key. To resolve this problem, do not update a key, but insert one more copy of it. So we allow multiple instances of same vertex in priority queue. This approach doesn’t require decrease key operation and has below important properties.

- Whenever distance of a vertex is reduced, we add one more instance of 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(ELogV)) as there will be at most O(E) vertices in priority queue and O(Log E) is same as O(Log V)

Below is algorithm based on above idea.

```
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 used as first item  of pair
   as 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)

5) Print distance array dist[] to print all shortest
   paths.
```

Below is C++ implementation of above idea.

```
// 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
    // http://geeksquiz.com/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 program to test methods of graph class
int main()
{
    // create the graph given in above fugure
    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);

    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
```