Chapter 11: Dijkstra's Algorithm

Section 11.1: Dijkstra's Shortest Path Algorithm

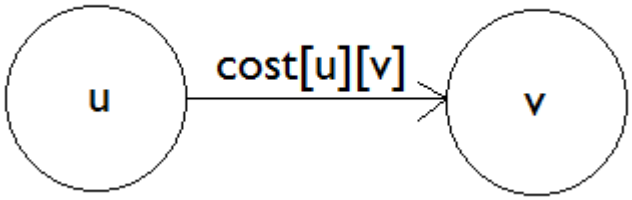
Before proceeding, it is recommended to have a brief idea about Adjacency Matrix and BFS

Dijkstra's algorithm is known as single-source shortest path algorithm. It is used for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by Edsger W. Dijkstra in 1956 and published three years later.

We can find shortest path using Breadth First Search (BFS) searching algorithm. This algorithm works fine, but the problem is, it assumes the cost of traversing each path is same, that means the cost of each edge is same. Dijkstra's algorithm helps us to find the shortest path where the cost of each path is not the same.

At first we will see, how to modify BFS to write Dijkstra's algorithm, then we will add priority queue to make it a complete Dijkstra's algorithm.

Let's say, the distance of each node from the source is kept in **d[]** array. As in, **d[3]** represents that **d[3]** time is taken to reach **node 3** from **source**. If we don't know the distance, we will store *infinity* in **d[3]**. Also, let **cost[u][v]** represent the cost of **u-v**. That means it takes **cost[u][v]** to go from **u** node to **v** node.



We need to understand Edge Relaxation. Let's say, from your house, that is **source**, it takes *10* minutes to go to place **A**. And it takes *25* minutes to go to place **B**. We have,

```
d[A] = 10
d[B] = 25
```

Now let's say it takes *7* minutes to go from place **A** to place **B**, that means:

```
cost[A][B] = 7
```

Then we can go to place **B** from **source** by going to place **A** from **source** and then from place **A**, going to place **B**, which will take *10 + 7 = 17* minutes, instead of *25* minutes. So,

```
d[A] + cost[A][B] < d[B]
```

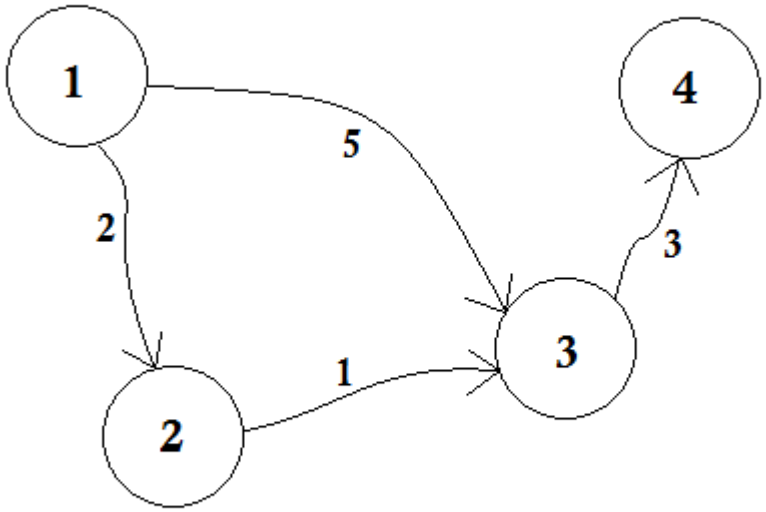
Then we update,

```
d[B] = d[A] + cost[A][B]
```

This is called relaxation. We will go from node **u** to node **v** and if **d[u] + cost[u][v] < d[v]** then we will update **d[v] = d[u] + cost[u][v]**.

In BFS, we didn't need to visit any node twice. We only checked if a node is visited or not. If it was not visited, we pushed the node in queue, marked it as visited and incremented the distance by 1. In Dijkstra, we can push a node

in queue and instead of updating it with visited, we *relax* or update the new edge. Let's look at one example:



Let's assume, **Node 1** is the **Source**. Then,

```
d[1] = 0
d[2] = d[3] = d[4] = infinity (or a large value)
```

We set, **d[2], d[3]** and **d[4]** to *infinity* because we don't know the distance yet. And the distance of **source** is of course *0*. Now, we go to other nodes from **source** and if we can update them, then we'll push them in the queue. Say for example, we'll traverse **edge 1-2**. As **d[1] + 2 < d[2]** which will make **d[2] = 2**. Similarly, we'll traverse **edge 1-3** which makes **d[3] = 5**.

We can clearly see that *5* is not the shortest distance we can cross to go to **node 3**. So traversing a node only once, like BFS, doesn't work here. If we go from **node 2** to **node 3** using **edge 2-3**, we can update **d[3] = d[2] + 1 = 3**. So we can see that one node can be updated many times. How many times you ask? The maximum number of times a node can be updated is the number of in-degree of a node.

Let's see the pseudo-code for visiting any node multiple times. We will simply modify BFS:

```
procedure BFSmodified(G, source):
Q = queue()
distance[] = infinity
Q.enqueue(source)
distance[source]=0
while Q is not empty
    u <- Q.pop()
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
        end if
    end for
end while
Return distance
```

This can be used to find the shortest path of all node from the source. The complexity of this code is not so good. Here's why,

In BFS, when we go from **node 1** to all other nodes, we follow *first come, first serve* method. For example, we went to **node 3** from **source** before processing **node 2**. If we go to **node 3** from **source**, we update **node 4** as **5 + 3 = 8**. When we again update **node 3** from **node 2**, we need to update **node 4** as **3 + 3 = 6** again! So **node 4** is updated twice.

Dijkstra proposed, instead of going for *First come, first serve* method, if we update the nearest nodes first, then it'll take less updates. If we processed **node 2** before, then **node 3** would have been updated before, and after updating **node 4** accordingly, we'd easily get the shortest distance! The idea is to choose from the queue, the node, that is closest to the **source**. So we will use *Priority Queue* here so that when we pop the queue, it will bring us the closest node **u** from **source**. How will it do that? It'll check the value of **d[u]** with it.

Let's see the pseudo-code:

```
procedure dijkstra(G, source):
Q = priority_queue()
distance[] = infinity
Q.enqueue(source)
distance[source] = 0
while Q is not empty
    u <- nodes in Q with minimum distance[]
    remove u from the Q
    for all edges from u to v in G.adjacentEdges(v) do
        if distance[u] + cost[u][v] < distance[v]
            distance[v] = distance[u] + cost[u][v]
            Q.enqueue(v)
        end if
    end for
end while
Return distance
```

The pseudo-code returns distance of all other nodes from the **source**. If we want to know distance of a single node **v**, we can simply return the value when **v** is popped from the queue.

Now, does Dijkstra's Algorithm work when there's a negative edge? If there's a negative cycle, then infinity loop will occur, as it will keep reducing the cost every time. Even if there is a negative edge, Dijkstra won't work, unless we return right after the target is popped. But then, it won't be a Dijkstra algorithm. We'll need Bellman–Ford algorithm for processing negative edge/cycle.

Complexity:

The complexity of BFS is **O(log(V+E))** where **V** is the number of nodes and **E** is the number of edges. For Dijkstra, the complexity is similar, but sorting of *Priority Queue* takes **O(logV)**. So the total complexity is: **O(Vlog(V)+E)**

Below is a Java example to solve Dijkstra's Shortest Path Algorithm using Adjacency Matrix

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
```

```
        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;
    }

    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    void dijkstra(int graph[][], int src)
    {

        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        dist[src] = 0;

        for (int count = 0; count < V-1; count++)
        {
            int u = minDistance(dist, sptSet);

            sptSet[u] = true;

            for (int v = 0; v < V; 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];
        }

        printSolution(dist, V);
    }

    public static void main (String[] args)
    {
    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);
    }
}
```

Expected output of the program is

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