Dijkstra’s Algorithm
Floyd’s Algorithm

(Practice makes perfect!)

CSC3100: Data Structures Tutorial

Jingjing Qian
jingjingqian@link.cuhk.edu.cn

Outline

- LeetCode P743. Network Delay Time
- Dijkstra’s Algorithm
- Java Implementation

P743. Network Delay Time

P743. Given n nodes, and a list of travel times between directed edges:

times[i] = [ui, vi, wi]

(wi is the time it takes for a signal to travel from ui to vi.)

Given a node k , our task is to return the minimum time it takes for all nodes to receive a signal sent from k.

(Return -1 if the it is impossible for the signal to reach every node.)

Abstraction:

Travel time	≫	Edge weight
Signal sent from k	≫	Single source k
Minimum time	≫	Maximum distance

Find the shortest path from A to every other vertex.
The longest one indicates the minimum time needed

PSEUDOCODE OF DIJKSTRA'S ALGORITHM

(Denote the starting vertex as s, visited set $S = \emptyset$)

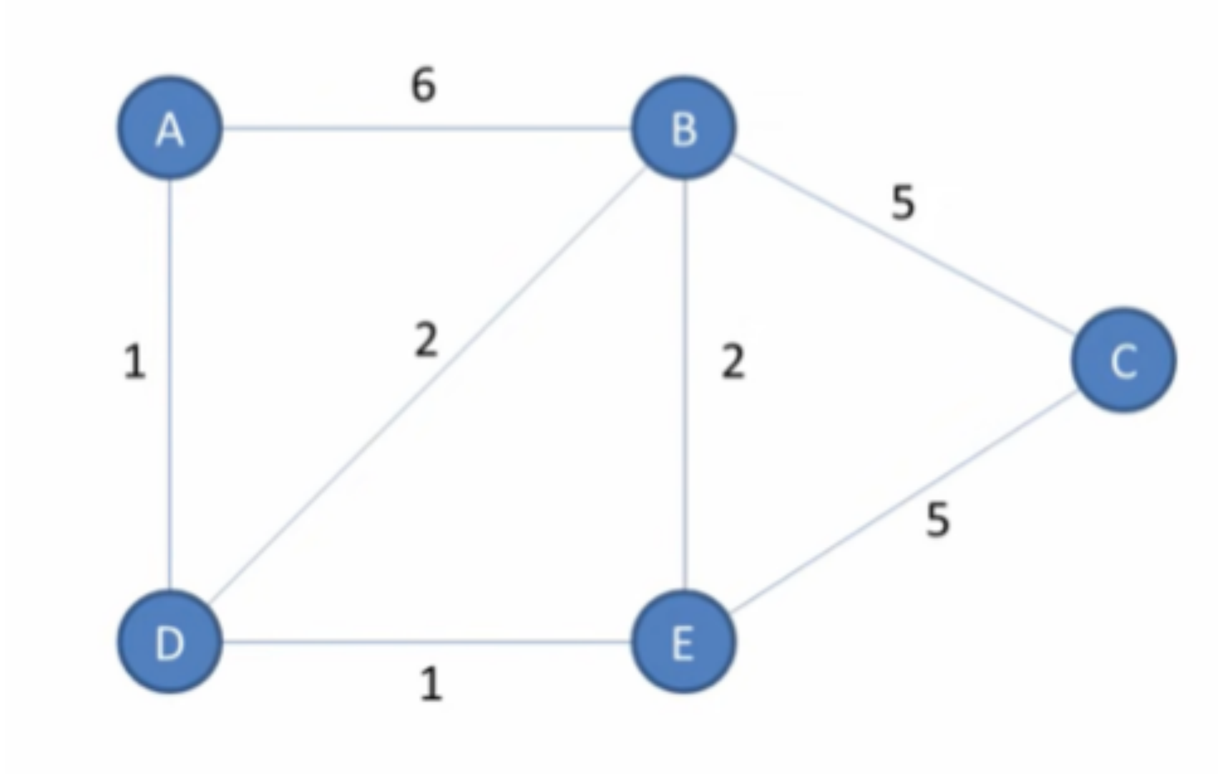
- Initialize all distances: $d(v) = \begin{cases} 0, & v = s \\ \infty, & v \neq s \end{cases}$
- Repeat:
  - Visit vertex $u \notin S$ s.t $d(u) = \min\{ d(v), v \notin S\}$
  - $S = S \cup \{u\}$, Evaluate and update all u's unvisited neighbors:

    If $d'(v) < d(v)$: Update its shortest distance

    Update its previous neighbor
- Procedure ends when all vertices are visited

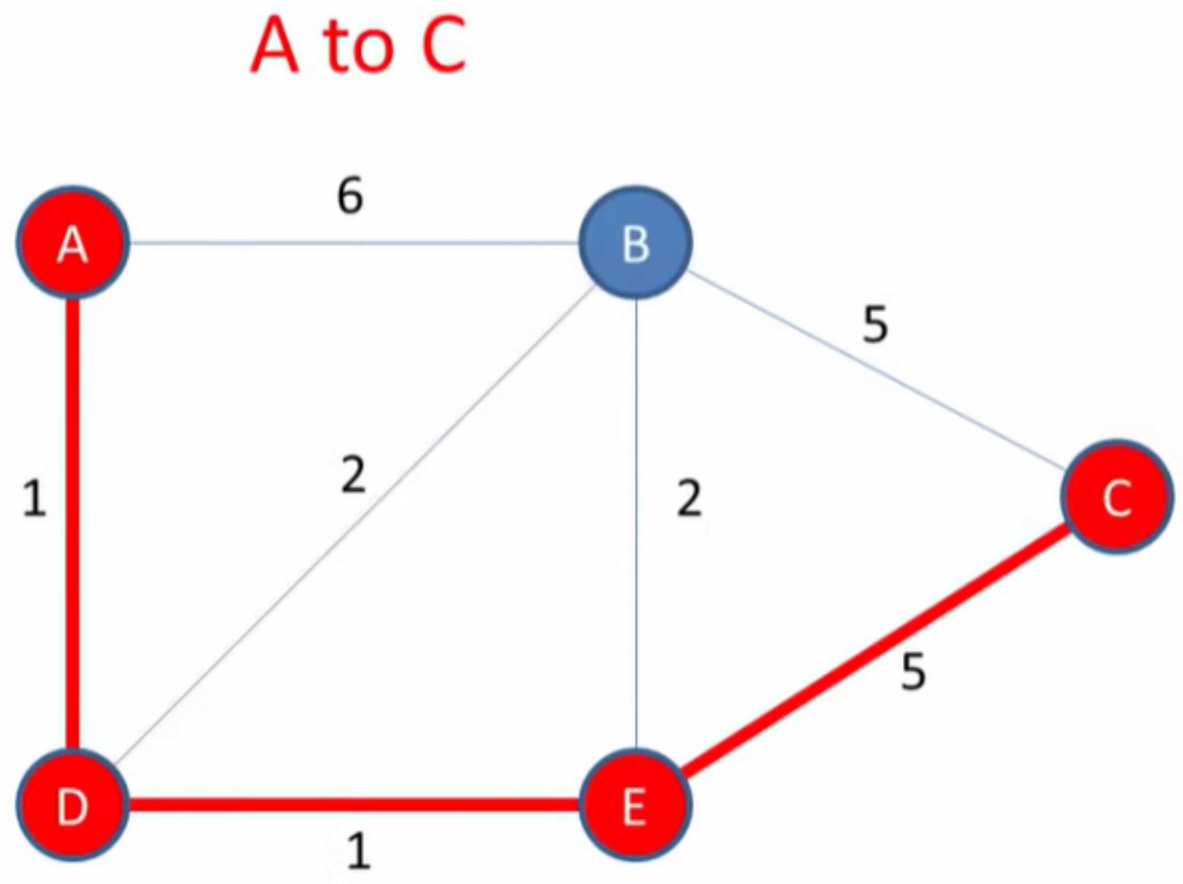
EXAMPLE

Find the shortest distance from vertex A to every other vertex



Vertex	Shortest Distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

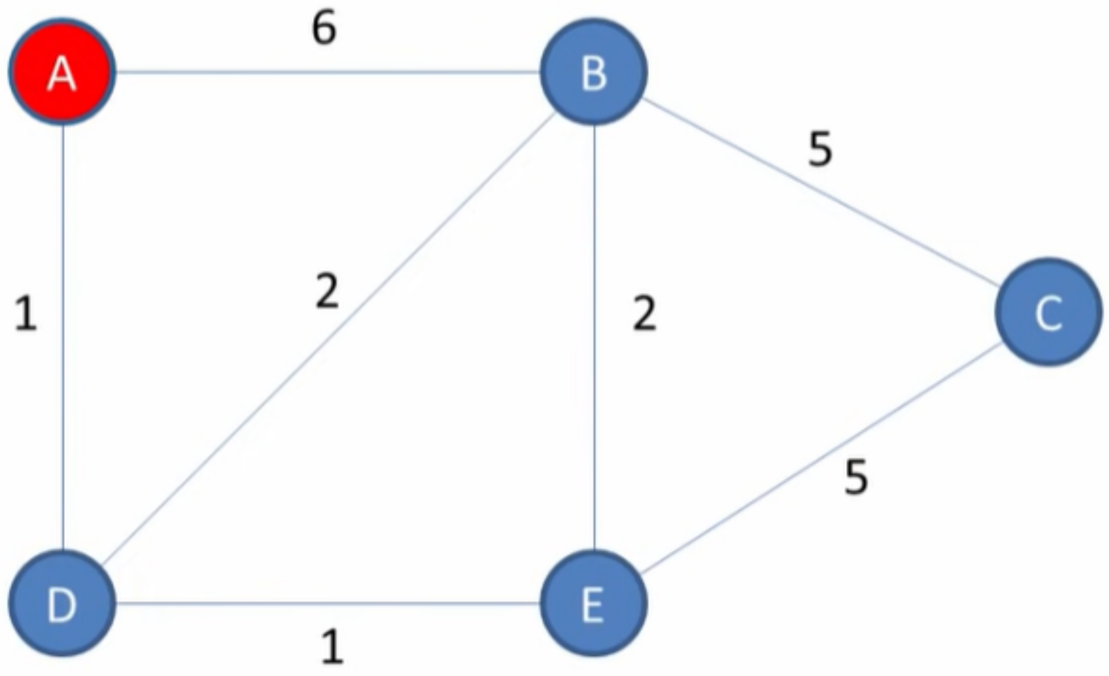
EXAMPLE



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

EXAMPLE

Consider the start vertex, A
Distance to A from A = 0
Distances to all other vertices from A are unknown, therefore ∞ (infinity)



Vertex	Shortest distance from A	Previous vertex
A		
B		
C		
D		
E		

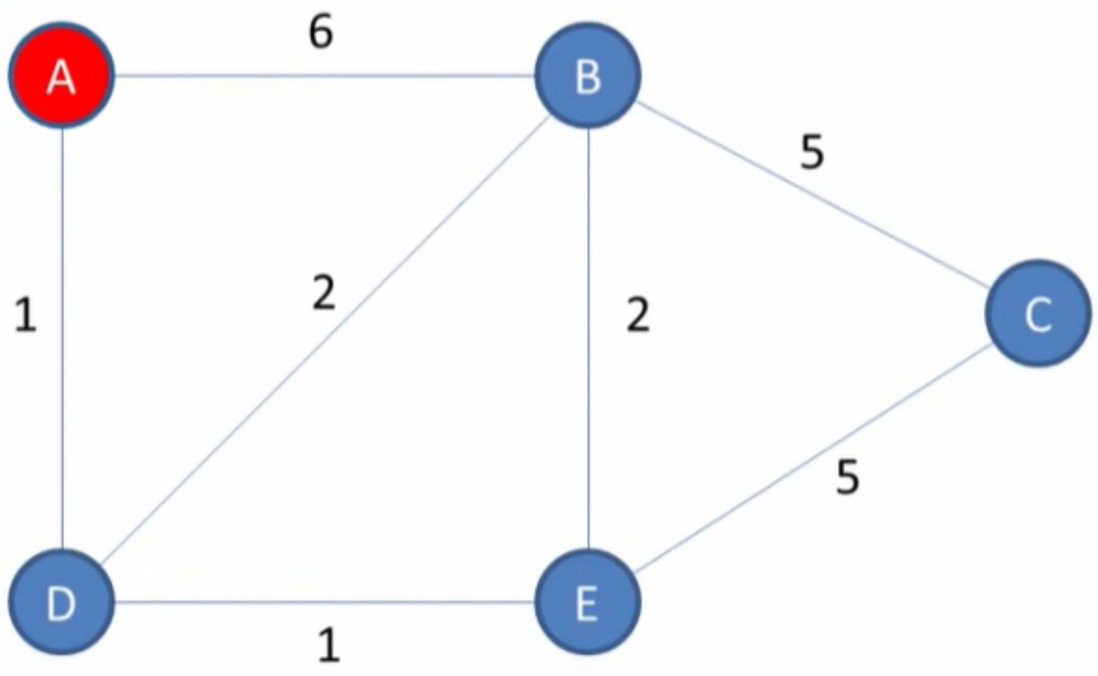
Visited = [ ]

Unvisited = [A, B, C, D, E]

EXAMPLE

Visit the unvisited vertex with the smallest known distance from the start vertex

First time around, this is the start vertex itself, A



Vertex	Shortest distance from A	Previous vertex
A	$\boxed{0}$	
B	∞	
C	∞	
D	∞	
E	∞	

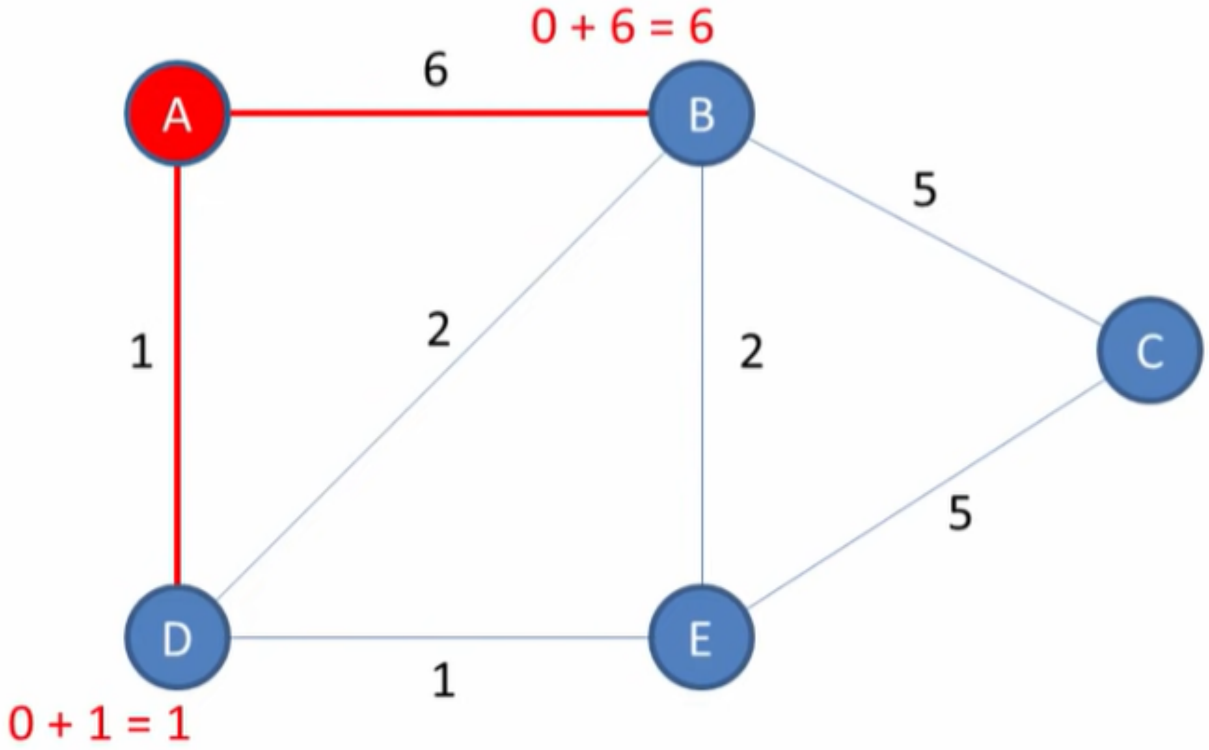
Visited = [ ]

Unvisited = [A, B, C, D, E]

EXAMPLE

Update the previous vertex for each of the updated distances

In this case we visited B and D via A



Visited = [ ]

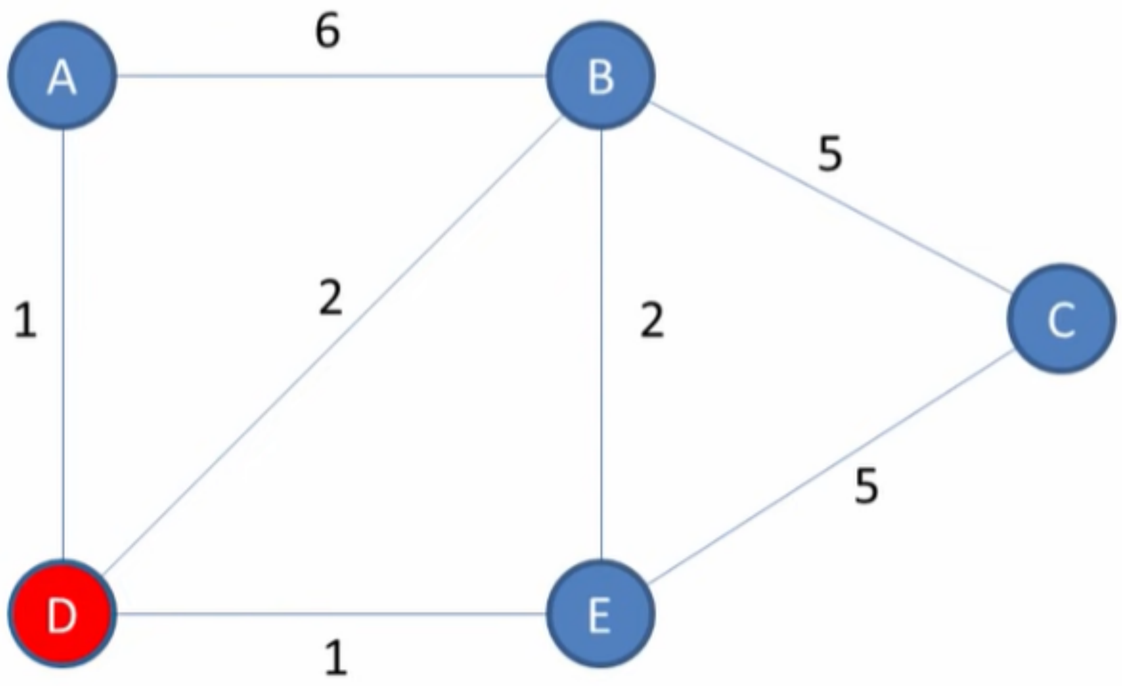
Unvisited = [A, B, C, D, E]

Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

EXAMPLE

Visit the unvisited vertex with the smallest known distance from the start vertex

This time around, it is vertex D



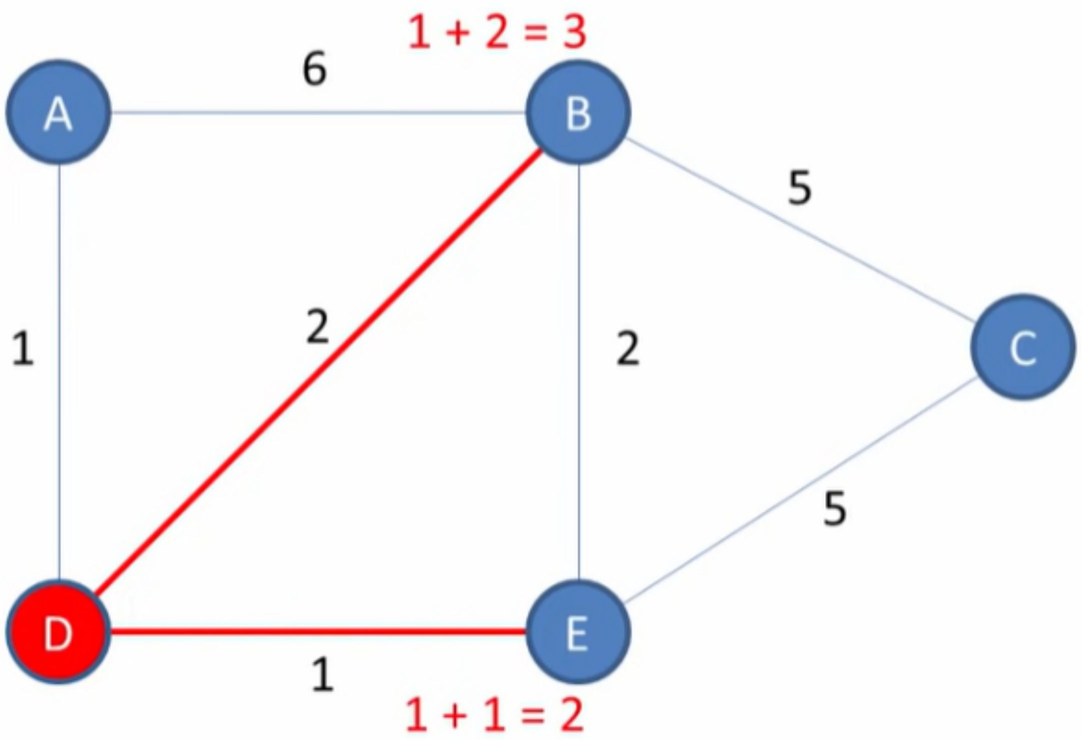
Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

Visited = [A]

Unvisited = [B, C, D, E]

EXAMPLE

For the current vertex, calculate the distance of each neighbour from the start vertex



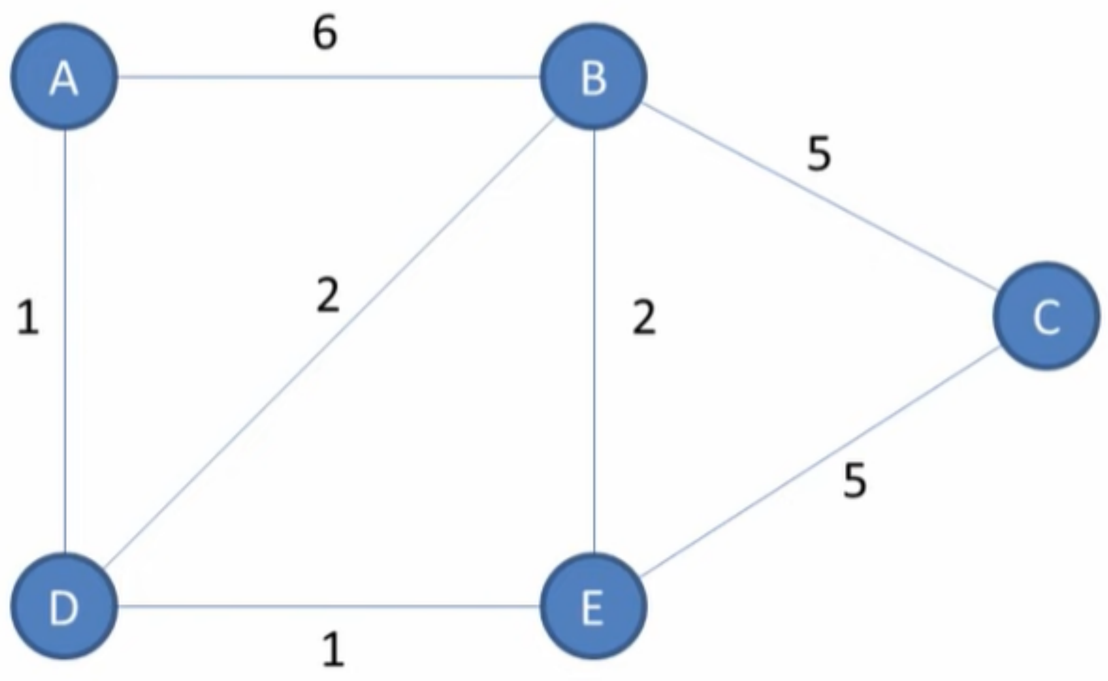
Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

Visited = [A]

Unvisited = [B, C, D, E]

EXAMPLE

Add the current vertex to the list of visited vertices



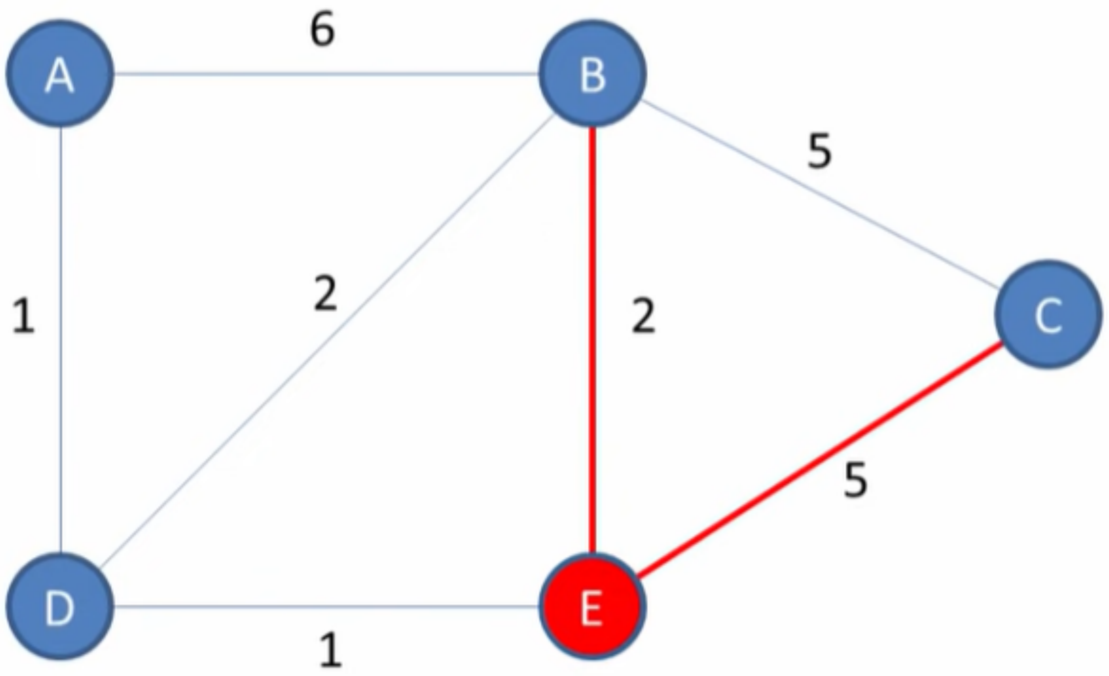
Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	∞	
D	1	A
E	2	D

Visited = [A, D]   Unvisited = [B, C, E]

EXAMPLE

For the current vertex, examine its unvisited neighbours

We are currently visiting E and its unvisited neighbours are B and C



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	∞	
D	1	A
E	2	D

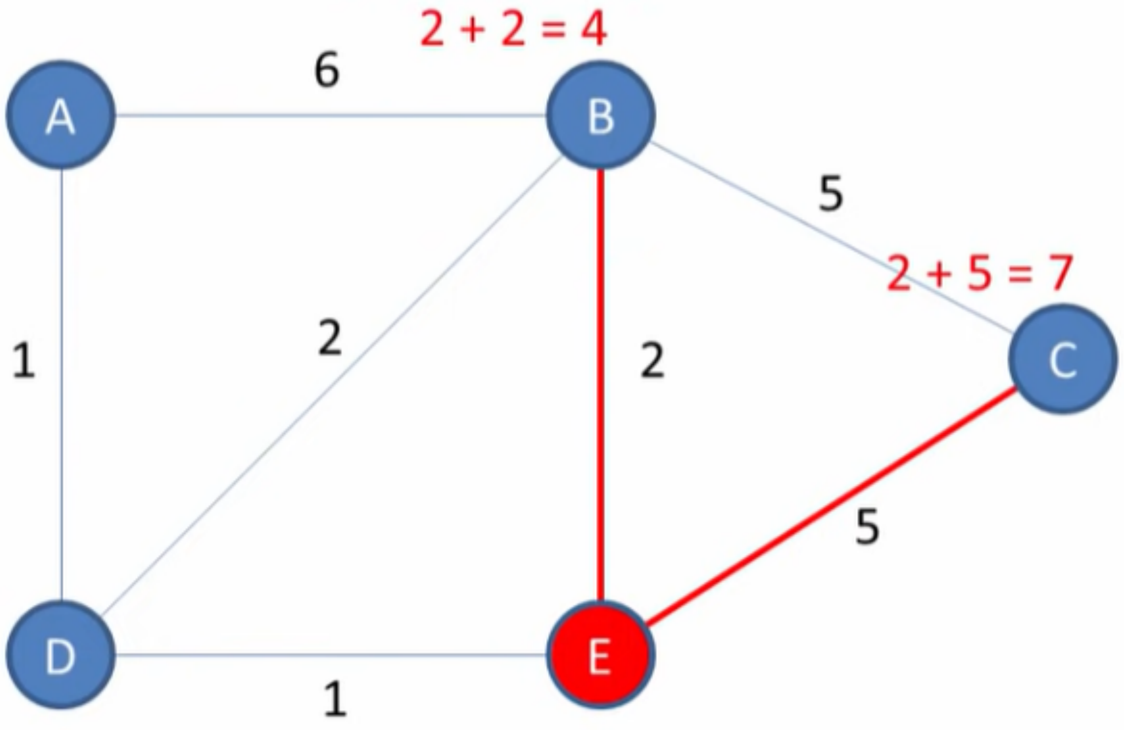
Visited = [A, D]

Unvisited = [B, C, E]

EXAMPLE

Update the previous vertex for each of the updated distances

In this case we visited C via E



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

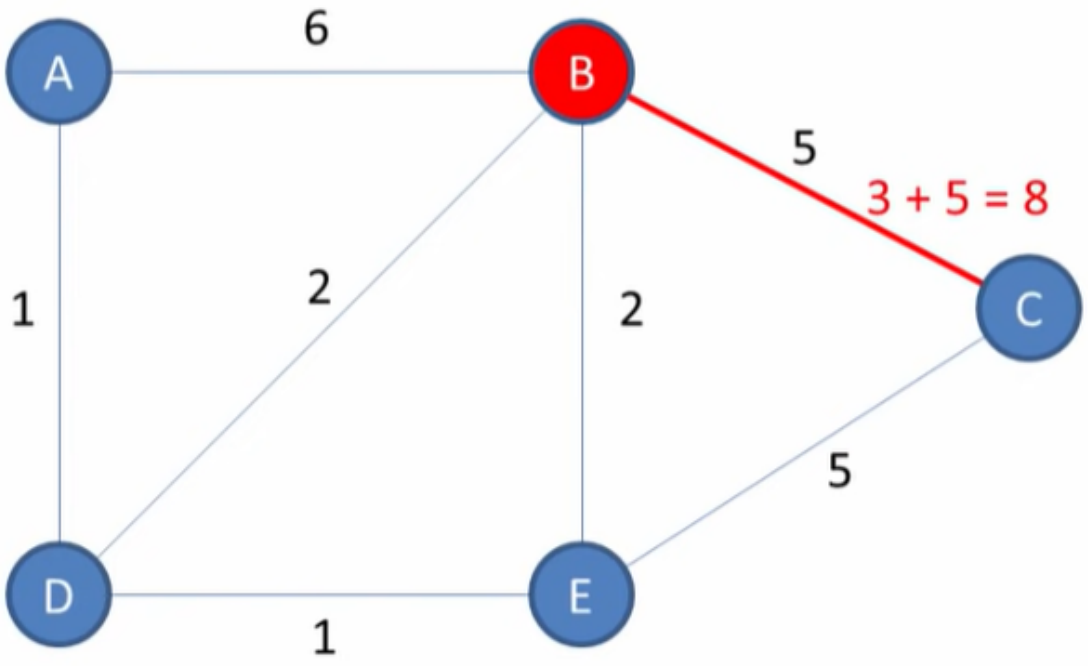
Visited = [A, D]

Unvisited = [B, C, E]

EXAMPLE

Update the previous vertex for each of the updated distances

No distances were updated, so we don't need to do this either



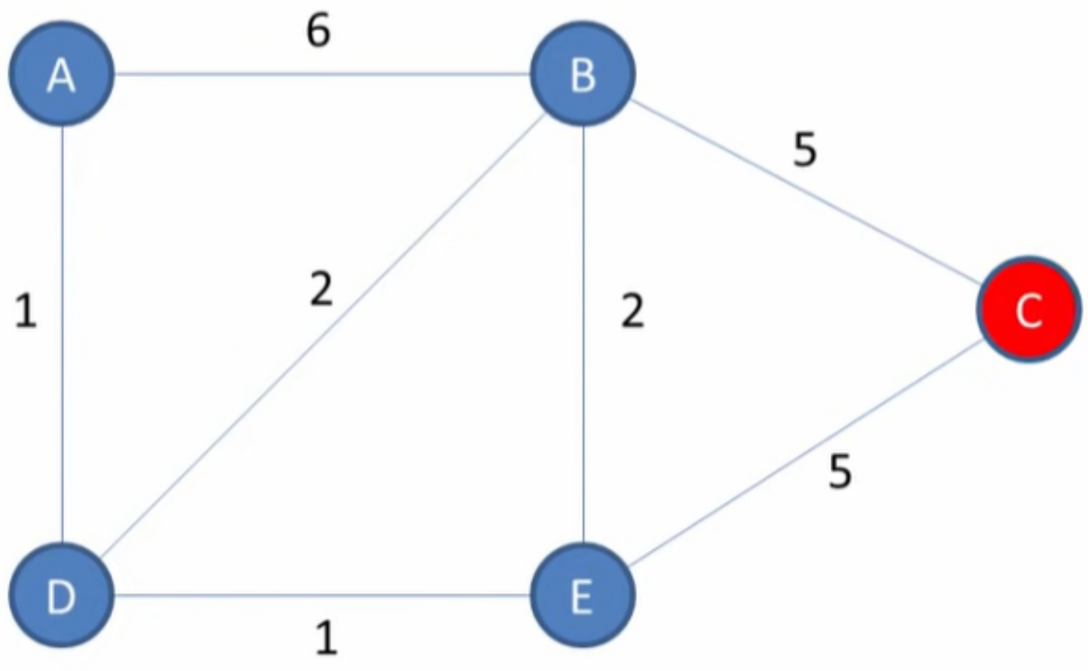
Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E]   Unvisited = [B, C]

EXAMPLE

Visit the unvisited vertex with the smallest known distance from the start vertex

This time around, it is vertex C



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	$\boxed{7}$	E
D	1	A
E	2	D

Visited = [A, D, E, B] Unvisited = [C]

Why does the greedy approach work in Dijkstra ?

JAVA IMPLEMENTATION – O(V^2 + E)

***Without using the priority queue**

```
class Solution {
    public int networkDelayTime(int[][] times, int n, int k) {
        final int INF = Integer.MAX_VALUE / 2;
        int[][] g = new int[n][n];
        for (int i = 0; i < n; ++i)
            Arrays.fill(g[i], INF);
        for (int[] t : times) {
            int x = t[0] - 1, y = t[1] - 1;
            g[x][y] = t[2];
        }

        int[] dist = new int[n];
        Arrays.fill(dist, INF);
        dist[k - 1] = 0;
        boolean[] used = new boolean[n];
        for (int i = 0; i < n; ++i) {
            int x = -1;
            for (int y = 0; y < n; ++y) {
                if (!used[y] && (x == -1 || dist[y] < dist[x])) {
                    x = y;
                }
            }
            used[x] = true;
            for (int y = 0; y < n; ++y) {
                dist[y] = Math.min(dist[y], dist[x] + g[x][y]);
            }
        }
```

JAVA IMPLEMENTATION – O(ElogV)

***With the priority queue**

```
class Solution {
    int N = 110, M = 6010;
    // Adjacent linked list
    int[] he = new int[N], e = new int[M], ne = new int[M], w = new int[M];
    // dist[x] = y is the distance
    int[] dist = new int[N];
    //
    boolean[] vis = new boolean[N];
    int n, k, idx;
    int INF = 0x3f3f3f3f;
    void add(int a, int b, int c) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx;
        w[idx] = c;
        idx++;
    }}
```

JAVA IMPLEMENTATION


```
class Solution {
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        // Initializes the linked list
        Arrays.fill(he, -1);
        // input data
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            add(u, v, c);
        }
        // run
        dijkstra();
        // search answer
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
}
```

JAVA IMPLEMENTATION


```java
class Solution {
    void dijkstra() {
        Arrays.fill(vis, false);
        Arrays.fill(dist, INF);
        dist[k] = 0;
        // Use pq to store all possible vertex to be update
        // {index, distance}, return the vertex with small distance
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[1]-b[1]);
        q.add(new int[]{k, 0});
        while (!q.isEmpty()) {
            // pop from pq
            int[] poll = q.poll();
            int id = poll[0], step = poll[1];
            // if the pop vertex is visited continue loop
            if (vis[id]) continue;
            // else mark the vertex as updated(visited), and update the distances of other vertex
            vis[id] = true;
            for (int i = he[id]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dist[id] + w[i]) {
                    dist[j] = dist[id] + w[i];
                    q.add(new int[]{j, dist[j]});
                }
            }}}}
```

Floyd's algorithm

To compute the shortest path between every pair of vertices in a graph, we might intuitively come up the idea of performing Dijkstra's algorithm $|V|$ times. The overall time complexity would be $\mathcal{O}(|V||E| \log |V|)$.

Floyd and Warshall propose an algorithm that solves the above problem in $\mathcal{O}(|V|^3)$, which is quicker than running Dijkstra's algorithm multiple times when $|E| > \frac{|V|^2}{\log |V|}$, e.g., when the graph is dense.

Floyd's algorithm

The algorithm uses Dynamic Programming-like technique that solves a problem by considering subproblems.

In the *k*-th state of our dynamic programming, we only consider additional vertices 1 to *k* as shortcuts. Then, $\mathtt{DP}_k$ can utilize the information given by $\mathtt{DP}_{k-1}$ (of course, $\mathtt{DP}_k$ is expected to give no worse shortest path solutions than $\mathtt{DP}_{k-1}$).

When considering the shortest path from *i* to *j*, a path that go to *k* from *i* and another path that go to *j* from *k* can be considered. Being implied, $\mathtt{DP}_k[i][j] = \min(\mathtt{DP}_{k-1}[i][j], \mathtt{DP}_{k-1}[i][k] + \mathtt{DP}_{k-1}[k][j])$.

Floyd's algorithm

The pseudocode for Floyd-Warshall's Algorithm is as follows:

```
Function FLOYD-WARSHALL(Graph G):
    initialize dist as the adjacency matrix of G
    for Node k = 1 to |V|:
        for Node i = 1 to |V|:
            for Node j = 1 to |V|:
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j])
```

The time complexity is $\mathcal{O}(|V|^3)$.