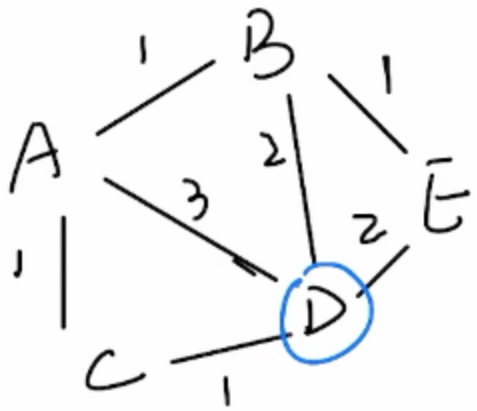
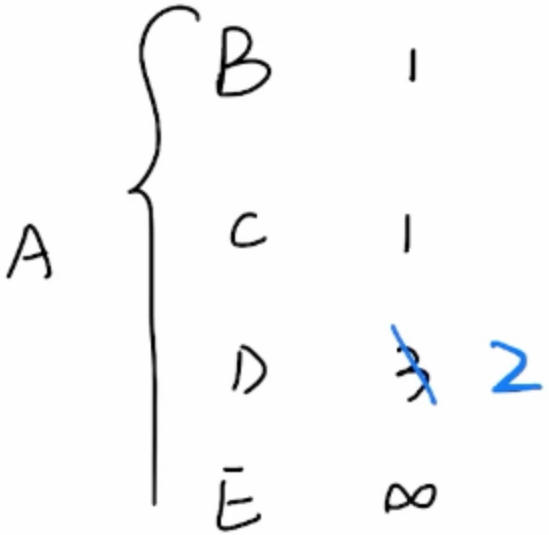
Dijkstra Algorithm

505. The Maze II
743. Network Delay Time
787. Cheapest Flights Within K Stops
1631. Path With Minimum Effort
1102. Path With Maximum Minimum Value
1514. Path with Maximum Probability
1368. Minimum Cost to Make at Least One Valid Path in a Grid

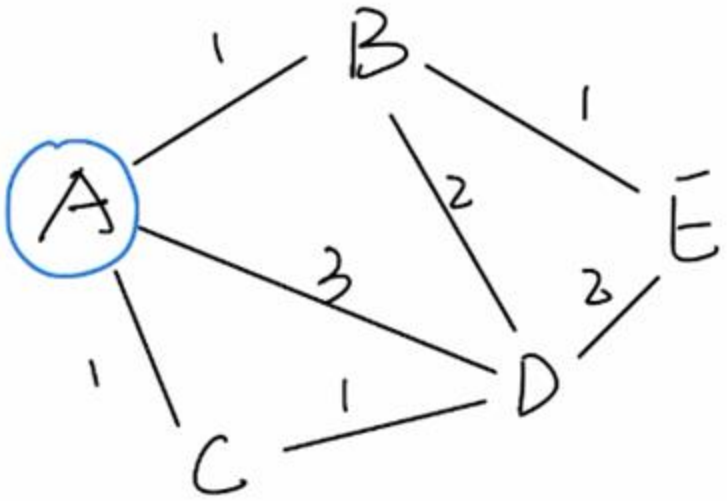
Edge Relaxation





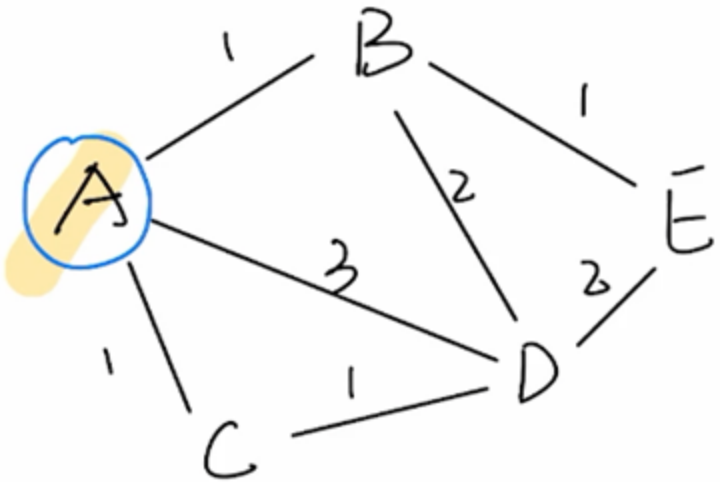
Start A, Find A to BCDE shortest distance
Init A to D distance to 3, but found A -> C -> D has 2 , So we change D as End Point shortest distance from 3 to 2 that's edge relaxation.

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



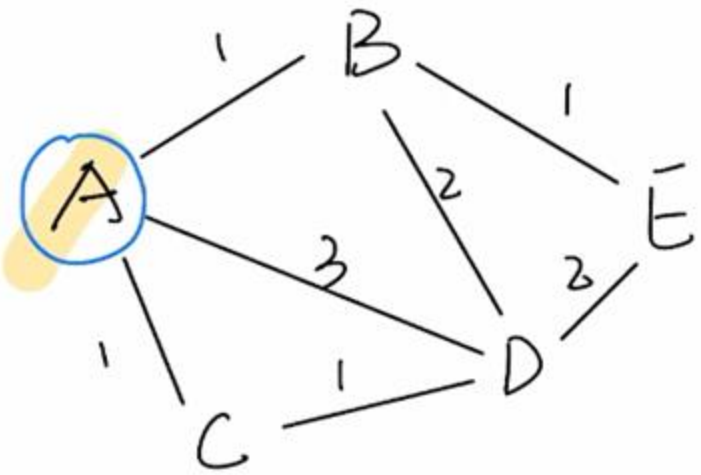
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	∞	
C	∞	
D	∞	
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



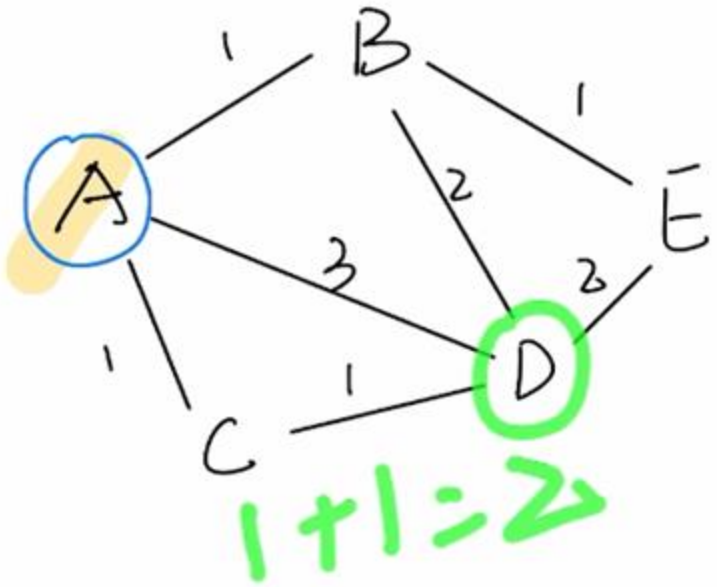
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	3	A
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



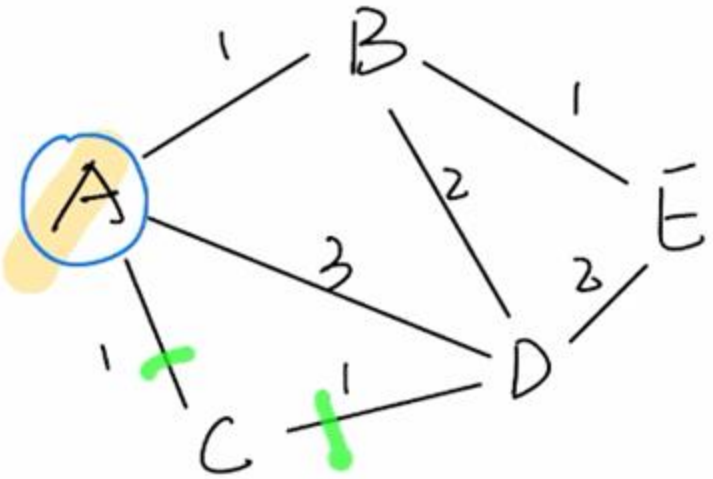
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	3	A
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



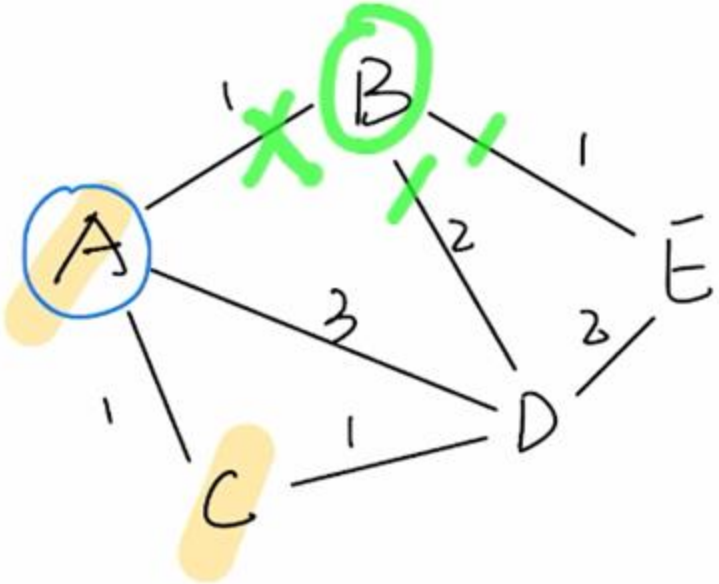
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	3	A
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



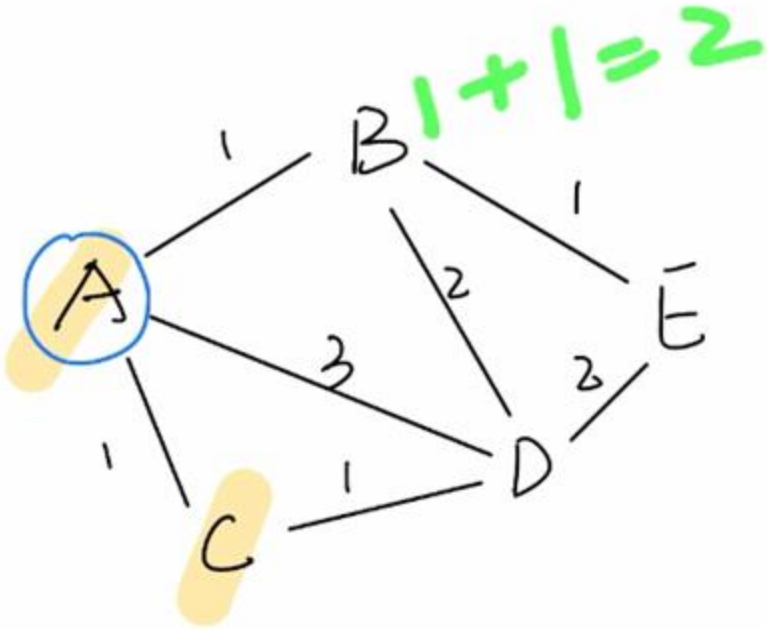
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	2	C
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



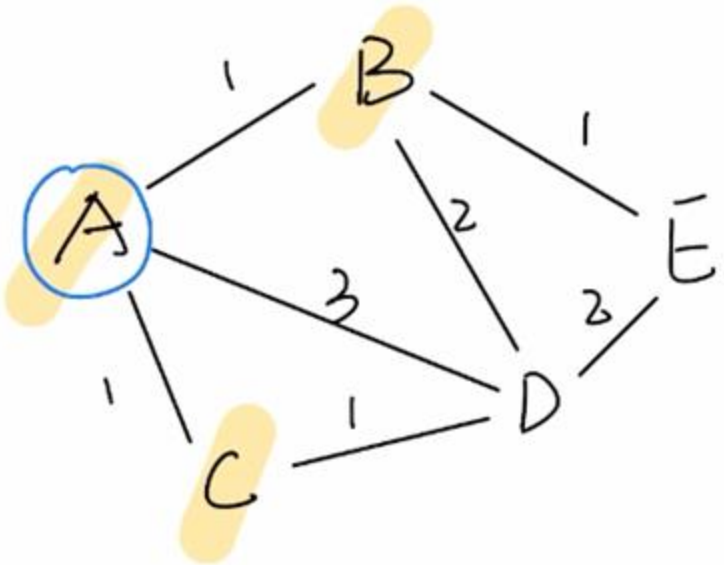
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	2	C
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



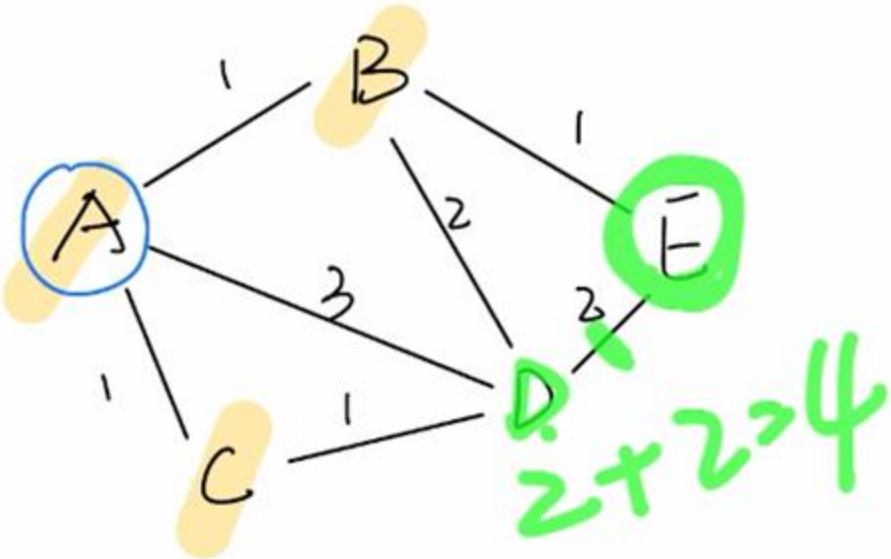
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	2	C
E	∞	

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



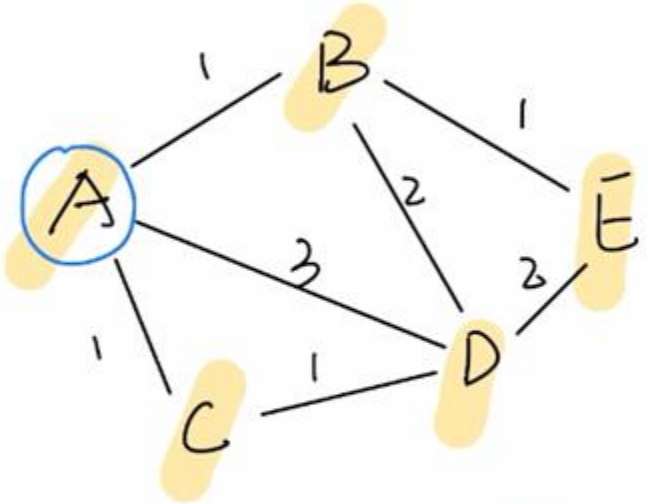
Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	2	C
E	2	B

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
D	2	C
E	2	B

Dijkstra's Algorithm - A Single Source Shortest Path Algorithm



Target Vertex	Shortest Distance from source vertex	Previous Vertex
A	0	—
B	1	A
C	1	A
→ D	(2)	C
→ E	2	B

A → C → D

A → B → E

505. The Maze II

Medium 929 41 Add to List Share

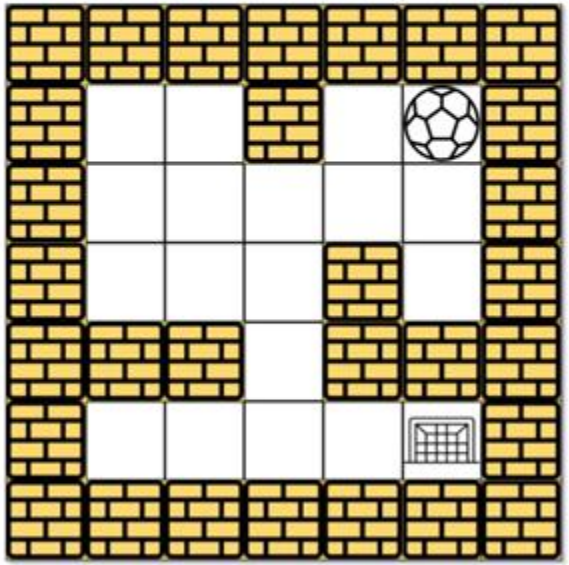
There is a ball in a `maze` with empty spaces (represented as `0`) and walls (represented as `1`). The ball can go through the empty spaces by rolling **up, down, left or right**, but it won't stop rolling until hitting a wall. When the b stops, it could choose the next direction.

Given the `m x n` `maze`, the ball's `start` position and the `destination`, where `start = [start_row, start_col]` and `destination = [destination_row, destination_col]`, return *the shortest* ***distance*** *for the ball to stop at the destination*. If the ball cannot stop at `destination`, return `-1`.

The **distance** is the number of **empty spaces** traveled by the ball from the start position (excluded) to the destina (included).

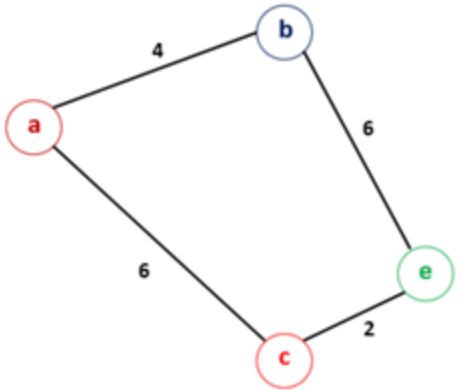
You may assume that **the borders of the maze are all walls** (see examples).

Example 1:



Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]
Output: 12
Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.
The length of the path is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.

```java
class Solution {
    int[][] dirs={{0, 1},{0, -1},{-1, 0},{1, 0}};
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        int[][] distance = new int[maze.length][maze[0].length];
        for (int[] row: distance) Arrays.fill(row, Integer.MAX_VALUE);
        distance[start[0]][start[1]] = 0;
        dijkstra(maze, start, distance);
        return distance[destination[0]][destination[1]] == Integer.MAX_VALUE ? -1 : distance[destination[0]][destination[1]];
    }

    public void dijkstra(int[][] maze, int[] start, int[][] distance) {
        PriorityQueue<int[]> q = new PriorityQueue<>((a, b) -> (a[2] - b[2]));
        q.offer(new int[]{start[0], start[1], 0});
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            for (int[] dir: dirs) {
                int x = cur[0] + dir[0], y = cur[1] + dir[1], count = 0;
                while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 0) {
                    x += dir[0];
                    y += dir[1];
                    count++;
                }
                x -= dir[0];
                y -= dir[1];
                if (distance[cur[0]][cur[1]] + count < distance[x][y]) {
                    distance[x][y] = distance[cur[0]][cur[1]] + count;
                    q.add(new int[]{x, y, distance[x][y]});
                }
            }
        }
    }
}
```



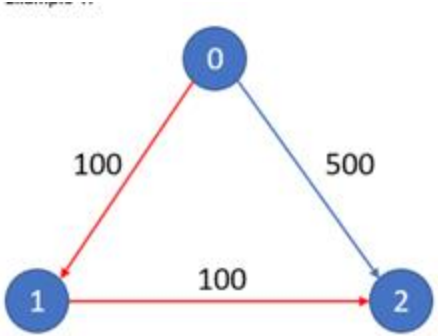

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

787. Cheapest Flights Within K Stops

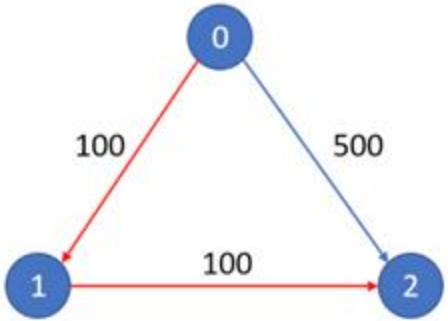
Medium 4016 162 Add to List Share

There are `n` cities connected by some number of flights. You are given an array `flights` where `flights[i] = [from_i, to_i, price_i]` indicates that there is a flight from city `from_i` to city `to_i` with cost `price_i`.

You are also given three integers `src`, `dst`, and `k`, return ***the cheapest price*** *from* `src` *to* `dst` *with at most* `k` *stops.* If there is no such route, return `-1`.



Input: n = 3, flights = [[0,1,100], [1,2,100],[0,2,500]], src = 0, dst = 2, k = 1
Output: 200
Explanation: The graph is shown.
The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

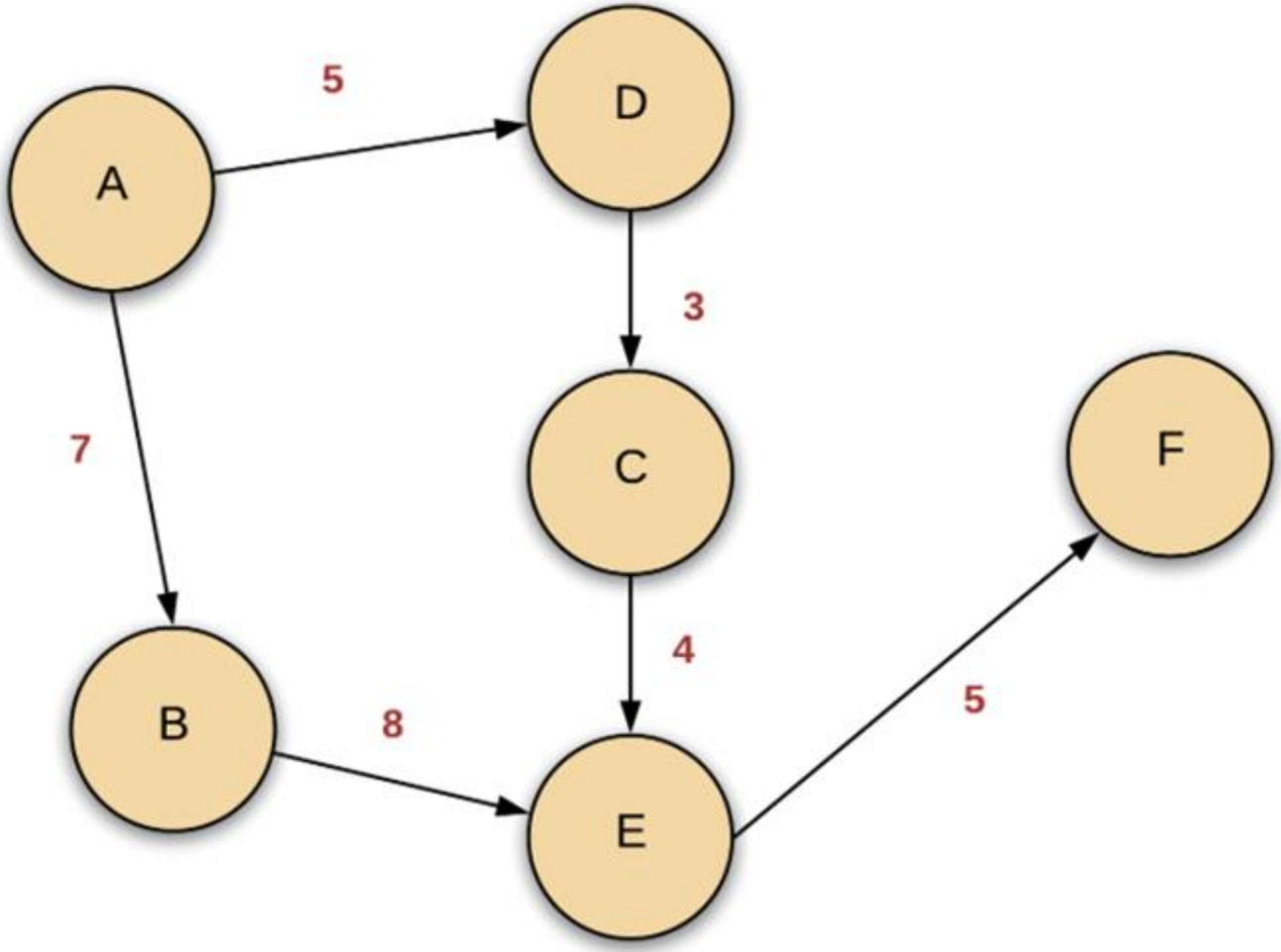


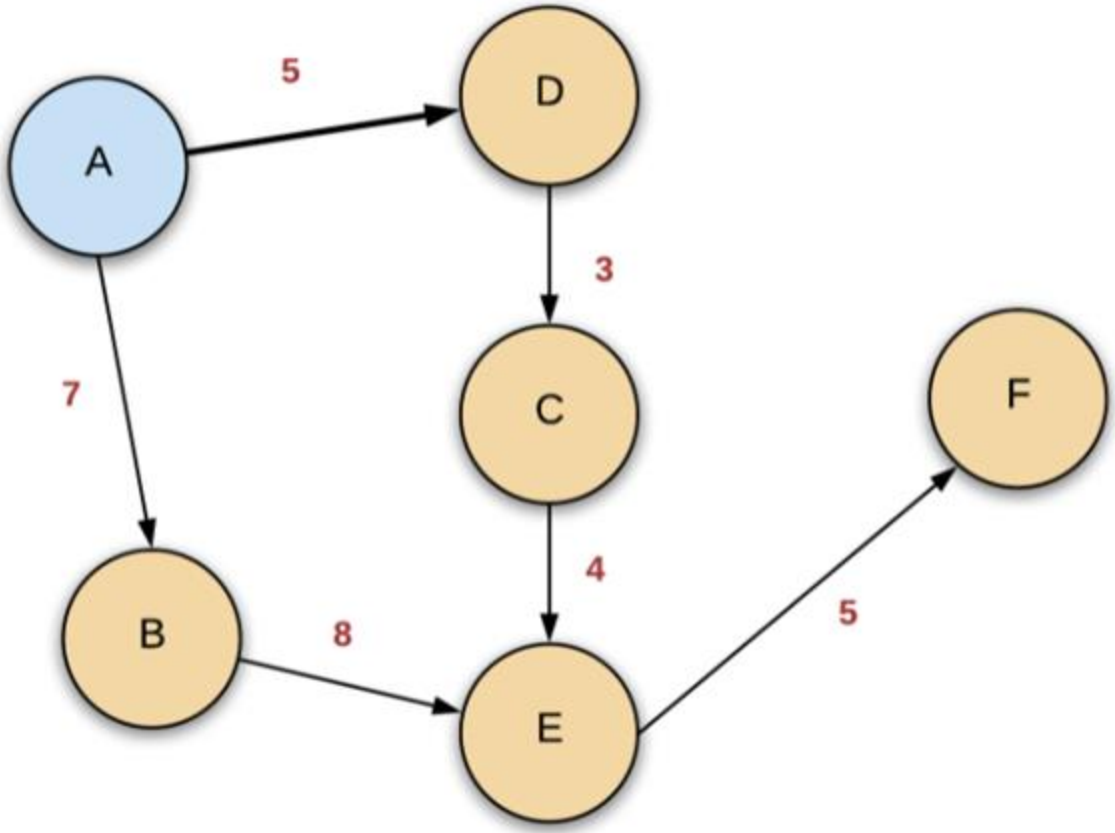
Input: n = 3, flights = [[0,1,100], [1,2,100],[0,2,500]], src = 0, dst = 2, k = 0
Output: 500
Explanation: The graph is shown.
The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

Dijikstra with variation, need to add stop checking.

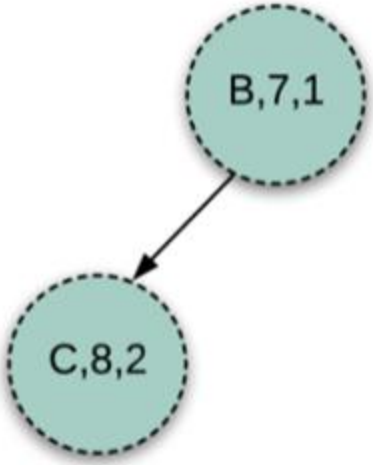
```java
public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
    Map<Integer, Map<Integer, Integer>> prices = new HashMap<>();
    Map<Integer, Integer> visited = new HashMap<>();
    for (int[] f : flights) prices.computeIfAbsent(f[0], value -> new HashMap<>()).put(f[1], f[2]);
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
    pq.add(new int[] {0, src, k + 1}); //{cost, city, step}
    while (!pq.isEmpty()) {
        int[] cur = pq.poll();
        int price = cur[0], city = cur[1], stops = cur[2];
        visited.put(city, stops);
        if (city == dst) return price;
        if (stops > 0) {
            Map<Integer, Integer> neighborsPrice = prices.getOrDefault(city, new HashMap<>());
            for (int nei : neighborsPrice.keySet())
                if (!visited.containsKey(nei) || stops > visited.get(nei))
                    pq.add(new int[] {price + neighborsPrice.get(nei), nei, stops - 1});
        }
    }
    return -1;
}
```

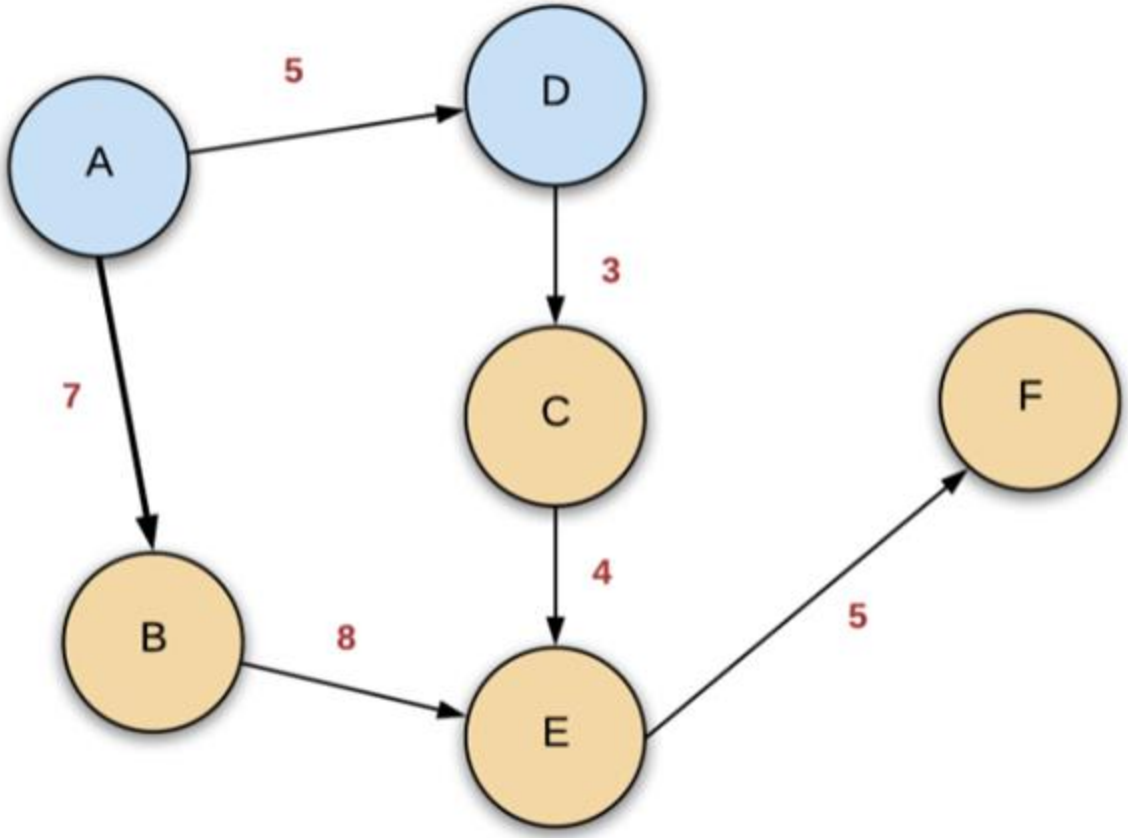
In Dijkstra we store the min cost taken to reach that node whereas we can't do that here as if number of stops exceed then that min cost would not be the same in case we took the different path. This problem can be called more as a BFS problem instead of Dijkstra.

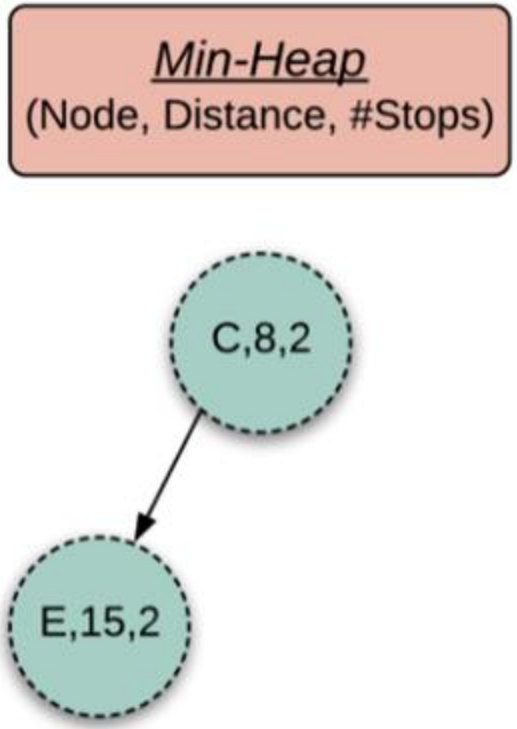

A
D
C
F
B
E
5
3
7
4
8
5

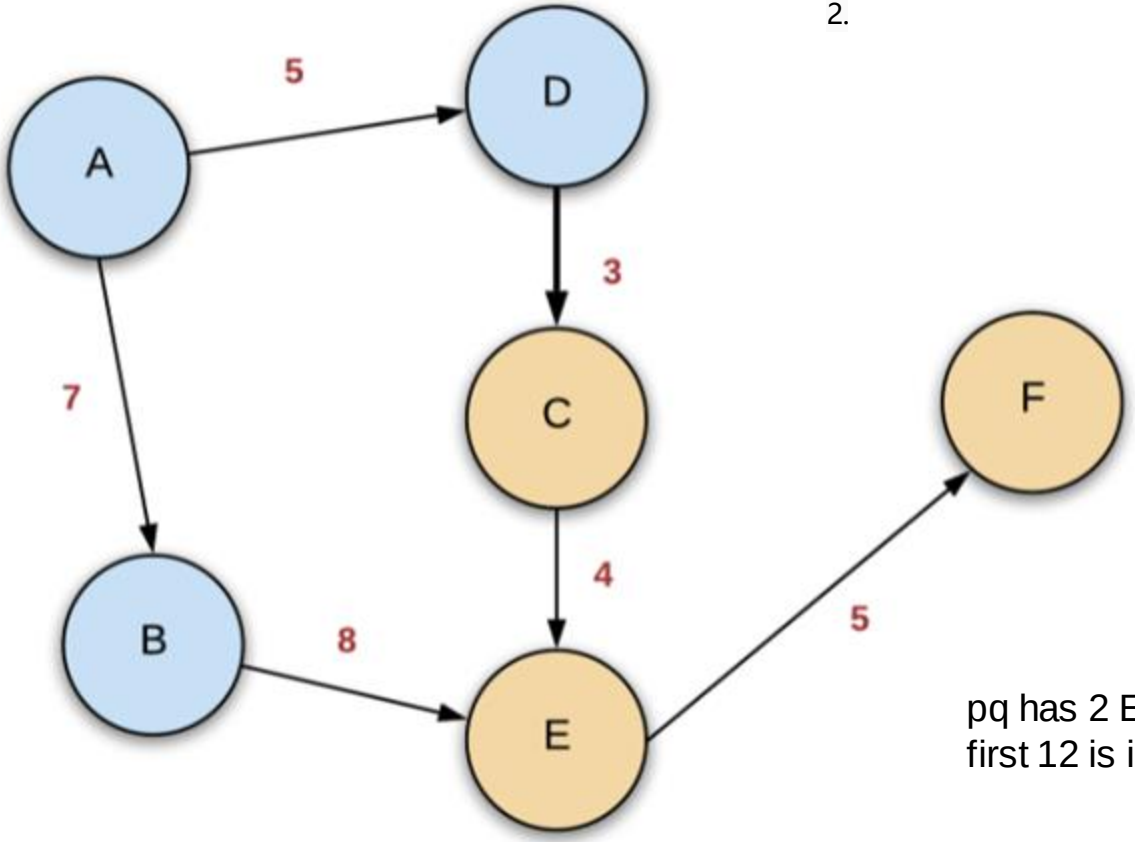


Min-Heap
(Node, Distance, #Stops)




A
D
C
F
B
E
5
3
7
4
8
5


Min-Heap
(Node, Distance, #Stops)
C,8,2
E,15,2



We should consider the valid E 15 2.

Min-Heap
(Node, Distance, #Stops)

pq has 2 E, but the first 12 is invalid.