LeetCode 787

https://leetcode.com/problems/cheapest-flights-within-k-stops/description/

Yifeng Zeng

Description

787. Cheapest Flights Within K Stops

There are n cities connected by m flights. Each fight starts from city u and arrives at v with a price w.

Now given all the cities and fights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with up to k stops. If there is no such route, output -1.

Example 1:

```
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1
Output: 200
Explanation:
The graph looks like this:
0 --(100)--> 1 -----(100)------|
|                              v
 ----------(500)------------->2
```

The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

Example 2:

```
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0
Output: 500
```

```
Explanation:
The graph looks like this:
0 --(100)--> 1 -----(100)------|
|                              v
 -----------(500)------------->2
```

The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

Note:

- The number of nodes n will be in range [1, 100], with nodes labeled from 0 to n - 1.
- The size of flights will be in range [0, n * (n - 1) / 2].
- The format of each flight will be (src, dst, price).
- The price of each flight will be in the range [1, 10000].
- k is in the range of [0, n - 1].
- There will not be any duplicated flights or self cycles.

Idea Report

We can think of each node as a state so we can write a state transfer matrix. For the example input: n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1. We will have the following matrix:

	0	**1**	**2**
0	0	0	0
1	max	100	100
2	max	500	200

The rows are the city IDs, the columns are the number of flights, so if we allow k = 1, then there can be 2 flights. Each cell in the matrix means we can travel from the src city (0) to the current city (row) with k flights (column) and the value is the lowest cost. Note that k is no longer stops, k is now we fly how many flights. That '500' means we travel from src (0) to the current city (2, row), and with maximum 1 flight (column) the lowest cost is 500, which is the answer of our example 2. This matrix can be defined as int[][] prices = new int[n][K + 1]; where n is the number of cities, and

K is the numnber of flights allowed (which is the orginal K stops allowed plus 1). So for each arrival city flight[1], the cost is of i stops is the the cost at from city flight[0] of i - 1 stops, plus the ticket price flight[2]. The base case is that the if the from and to city are both src city, we need 0 cost because we are already here, and if 0 flights is allowed (new k == 0), we need a max cost to indicated it's not possible. With this dynamic programming functional equation and base cases, we can write the DP solution:

Code:

```
class Solution {
    // DP AC
    public int findCheapestPrice(int n, int[][] flights,
                                 int src, int dst, int K) {
        K++;
        int[][] prices = new int[n][K + 1];
        for (int[] price : prices) {
            Arrays.fill(price, Integer.MAX_VALUE);
        }

        prices[src][0] = 0;
        for (int i = 1; i <= K; i++) {
            for (int j = 0; j < n; j++) {
                prices[j][i] = prices[j][i - 1];
                for (int[] flight : flights) {
                    if (prices[flight[0]][i - 1] != Integer.MAX_VALUE) {
                        prices[flight[1]][i] = Math.min(prices[flight[1]][i],
                                                        prices[flight[0]][i - 1]
                                                        + flight[2]);
                    }
                }
            }
        }

        // for (int[] p : prices) {
        //     System.out.println(Arrays.toString(p));
        // }

        return prices[dst][K] == Integer.MAX_VALUE ? -1 : prices[dst][K];
    }
}
```

We can also do a DFS search, directly seach from the src to dst and find the minimum cost within k stops. This may take too much time because we are trying all the possible combinations.

Code:

```java
class Solution {
    // DFS time limit exceeded
    public int findCheapestPrice(int n, int[][] flights, int src, int dst,
                                 int K) {
        Map<Integer, Map<Integer, Integer>> map = new HashMap<>();
        // map from, to, price
        for (int[] flight : flights) {
            if (map.containsKey(flight[0])) {
                map.get(flight[0]).put(flight[1], flight[2]);
            } else {
                Map<Integer, Integer> temp = new HashMap<>();
                temp.put(flight[1], flight[2]);
                map.put(flight[0], temp);
            }
        }

        int[] cheapest = new int[]{Integer.MAX_VALUE};
        helper(n, flights, src, dst, K, cheapest, map, 0);

        return cheapest[0] == Integer.MAX_VALUE ? -1 : cheapest[0];
    }

    private void helper(int n, int[][] flights, int cur, int dst,
                        int k, int[] cheapest,
                        Map<Integer, Map<Integer, Integer>> map, int cost) {
        if (cur == dst) {
            cheapest[0] = Math.min(cheapest[0], cost);
            return;
        }
        if (k < 0) {
            return;
        }

        Map<Integer, Integer> nei = map.getOrDefault(cur, new HashMap<>());
        for (Map.Entry<Integer, Integer> entry : nei.entrySet()) {
            helper(n, flights, entry.getKey(), dst, k - 1, cheapest, map,
                   cost + entry.getValue());
        }
    }
}
```

We can do a BFS search, and find out for each city what is the lowest cost. If the cost from current city the next city is greater than another path to that next city, we stop the search, this way trims some of the necessary searching branch and speeds up.

Code:

```java
class Solution {
    // BFS AC
    public int findCheapestPrice(int n, int[][] flights, int src, int dst,
                                 int K) {
        int[] next = new int[n];
        int[] prev = new int[n];
        Arrays.fill(next, -1);
        Arrays.fill(prev, -1);
        prev[src] = 0;
        next[src] = 0;

        Map<Integer, Map<Integer, Integer>> map = new HashMap<>();
        // map from, to, price
        for (int[] flight : flights) {
            if (map.containsKey(flight[0])) {
                map.get(flight[0]).put(flight[1], flight[2]);
            } else {
                Map<Integer, Integer> temp = new HashMap<>();
                temp.put(flight[1], flight[2]);
                map.put(flight[0], temp);
            }
        }


        Deque<Integer> q = new LinkedList<>();
        q.offer(src);
        int level = -1;

        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                int cur = q.poll();
                Map<Integer, Integer> nei = map.get(cur);
                if (nei == null) {
                    continue;
                }
                for (int to : nei.keySet()) {
                    if (next[to] == -1 || nei.get(to) + prev[cur] < next[to]) {
                        next[to] = nei.get(to) + prev[cur];
                        q.offer(to);
                    }
                }
            }
            System.arraycopy(next, 0, prev, 0, n);
            level++;
            if (level == K) {
                break;
            }
        }

        return prev[dst];
```

```
        }
    }
```

Summary

- 3-D method use iteration/recurtion, 2-D solution space.
- Define the dynamic programming functional equation and base cases is hard, the coding part is relatively easy.