787. Cheapest Flights Within K Stops

Medium 5350 246 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`.

```
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int
dst, int k)
    {
        // build the graph as adjacent list format
        typedef pair<int, int> PII;
        vector<vector<PII>> graph(n, vector<PII>());
        for (auto flight: flights)
        {
            int f = flight[0], t = flight[1], w = flight[2];
            graph[f].push_back({t, w});
        }

        // shortest path to all vertices from src
        int dist[n];

        // initilize to infinity
        memset(dist, INTMAX, sizeof dist);
        dist[src] = 0;

        // early termination flag
        bool any_updated = false;
        bool vertices_updated[n];
        memset(vertices_updated, false, sizeof vertices_updated);

        // src is updated as base case
        vertices_updated[src] = true;

        // tmp holders;
        int tmp_dist[n];
        bool tmp_vertices_updated[n];
```

```
        for (int i = 0; i < k + 1; i++)
        {
            // initialize all vertices as not updated for i stops
            memset(tmp_vertices_updated, false, sizeof tmp_vertices_updated);
            // store i - 1 stops dist to a tmp holder
            memcpy(tmp_dist, dist, sizeof tmp_dist);
            // enumerate all vertices, as from
            for (int f = 0; f < n; f++)
            {
                // jump over vertices not updated at i - 1 stops iteration
                if (!vertices_updated[f]) continue;
                // for every adjacent nodes, as to
                for (auto [t, w] : graph[f]) {
                    // update dist[t], we need to use tmp_dist, because for
we need to make sure dist[f] is from i - 1 stops
                    // and dist[f] might be updated at i stops
                    dist[t] = min(dist[t], tmp_dist[f] + w);
                    if (dist[t] == tmp_dist[f] + w)
                    {
                        // update early termination flags
                        any_updated = true;
                        tmp_vertices_updated[t] = true;
                    }
                }
            }
            // if no updates, early termination
            if (!any_updated) break;
            any_updated = false;
            // replace vertices_updated with tmp holders data
            memcpy(vertices_updated, tmp_vertices_updated, sizeof
vertices_updated);
        }
        // if dst cannot reached return -1, otherwise return dist[dst]
        if (dist[dst] < INTMAX) return dist[dst];
        return -1;
    }
};
```

Algorithm:

Sub Problem

We could raise our main problem: calculate `dist[v] (k stops)` shortest path from src to every vertices `v` with at most k stops. Once the main problem is solved we could just return `dist[dst]` as it is the shortest path from src to dst within k stops.

Then we could reduce the main problem to our sub problem: calculate `dist[v] (i stops)` shortest path from src to every vertices with at most i stops (i < k)

Recurrence Equation

We need to get the recurrence equation between sub problems `dist[v] (i stops)` and `dist[v] (i - 1) stops`

For shortest path from `src` to `v` with at most `i` stops , they can be divided to two cases:

Case1: the path has at most `i - 1` stops. Which is the definition of `dist[v] (i - 1 stops)`

Case2: the path has k stops. Which means every edges point to v could be the last edge in the path, therefore `dist[v] (i stops) = min{ dist[w] (i - 1 stops) + weight(w -> v) for every adjacent w }`

The minimum value among these two cases should be the shortest path from `src` to `v` with at most `i` stops. Therefore it can be written as :

```
dist[v] (i stops) = min{
        dist[v] (i - 1) stops,
        min {
                dist[w] (i - 1 stops) + weight(w -> v) for every adjacent w
        }
}
```

Base case

If we do not take any flight, we could only reach `src`, and the price is 0. Other vertices shoulde be initialize to infinity.

Merge Sub Problems

We need to derive the results of sub problems from base case. So we start from 0-stops, to k stops. As the following pseudo code shows, the outer loop indicates current stops number. Then we enumerate all vertices, update the `dist` according to recurrence equation we represented above.

```
for i = 0 to k {
        foreach vertex f in all cities {
                if f is not updated at last iteration {
                        continue
```

```
                }

                dist[v] = dist[v] (i - 1) stops

                for vertex t in all cities such that (f, t) in all flights {
                        dist[v] (i stops) = min( dist[v] (i stops), dist[f] (i - 
1 stops) + weight (f -> t)
                }
        }
        break if no dist[v] updated in current iteration
}
```

Note: if a node is not updated at i - 1 stops, it could be passed at i stops. Because as our recurrence equation shows, for case 2: only path has i stops could possibly update `dist[v] (i stops)`, therefore if a node is not updated at i - 1 stops, the path should always lower than or equal to i - 1 stops. which could be covered by `dist[v] (i - 1 stops)`. Therefore, we could jump over any vertices does not get updated in previous iteration.

And if all vertices do not updated, which means case 2 will never happens, so we could just stop the algorithm, because `dist[v] (k stops) = dist[v] (k - 1) stops = ... = dist[v] (i stops)`

Time Complexity

At worst case, we need to loop K times if there's no early termination, and in the inner loop, in worst case, we will enumerate all edges by one time.

The recurrence relation is `T(K) = T(K - 1) + O(E) = O(KE)`

Therefore, the time complexity for is O(KE), where K is the number of stops, E is the number of edges.