787. Cheapest Flights Within K Stops

Medium Depth-First Search | Breadth-First Search | Graph | Dynamic Programming | Shortest Path | Heap (Priority Queue)

Leetcode Link

Problem Description

This LeetCode problem requires finding the cheapest flight route from one city to another with certain constraints. We have `n` cities and a list of flights where each flight is represented by `[from_i, to_i, price_i]`. This indicates a flight from city `from_i` to city `to_i` at the cost `price_i`. Given additional parameters `src`, `dst`, and `k`, the goal is to find the minimum cost to travel from the source city `src` to the destination city `dst` with at most `k` stops in between. If no such route exists, the function should return `-1`.

Intuition

To solve this problem, we use a variation of the Bellman-Ford algorithm which is suitable for finding shortest paths in graphs with edge weights that could be negative (not the case here) and can handle queries for paths with a limited number of edges.

We initialize a list `dist` of size `n` with a high value representing infinity since we are looking for the minimum cost. This list will keep track of the minimum cost to reach each city. The entry for the source city `src` is set to 0 since it costs nothing to stay at the same city.

The algorithm will go through the flights at most `k+1` times (the number of stops plus the initial city). On each iteration, we make a copy of the `dist` list called `backup`. This is important to ensure that updates in the current round do not affect the other updates in the same round since we are only allowed to use `k` stops.

For each flight `[f, t, p]` in `flights`, we check if the current known cost to city `t` is greater than the cost to city `f` plus the price `p` of the flight from `f` to `t`. If it is, we update the cost for city `t` in the `dist` list with the new lower cost. This represents a relaxation step where we relax the cost to get to `t` via `f` if it is possible to get there more cheaply.

After `k+1` iterations, the `dist` list contains the minimum costs of reaching each city from the source city with up to `k` stops. The value in `dist[dst]` is the cheapest price to get to the destination `dst`. However, if the value at `dist[dst]` is still infinity (represented by `INF`), it means there was no possible route within the given number of stops, hence we return `-1`.

Solution Approach

The implementation of the solution uses dynamic programming to iteratively update the cheapest prices for reaching each city within the constraint of at most `k` stops. Here's a step-by-step walkthrough of the algorithm:

1. **Initialization**: An array `dist` is initialized to store the cheapest prices to each city. It is filled with `INF` (a large number indicating infinity) to represent that initially, all destinations are unreachable. The price to reach the `src` city is set to `0` because it costs nothing to start from there.

2. **Algorithm**: The core part of the algorithm runs for `k+1` iterations, where `k` is the maximum number of stops allowed. The extra iteration is because a journey from `src` to `dst` with `k` stops consists of `k+1` flights.

3. **Backup Array**: In each iteration, a backup of the `dist` array is created. This is crucial because updates to the `dist` array within a single iteration must be based on the state of the array from the previous iteration, not the current one where some values have already been updated.

4. **Relaxation**: For each flight `[f, t, p]`, where `f` is the starting city, `t` is the target city, and `p` is the price of the flight, the algorithm checks if the current known price to city `t` can be improved by taking the flight from city `f`. The relaxation condition is:

   ```
   if dist[t] > backup[f] + p:
       dist[t] = backup[f] + p
   ```

   If the price to reach city `t` (`dist[t]`) is higher than the price to reach city `f` (`backup[f]`) plus the price of the flight `p`, the cheaper price is updated in `dist[t]`.

5. **Returning the result**: After `k+1` iterations, the algorithm looks at the price recorded in `dist[dst]`, which is the minimum cost to reach destination `dst` from source `src` within `k` stops. If this price is still `INF`, it means no such route was found within the limit of stops, and `-1` is returned. Otherwise, the cheapest price found is returned.

This approach efficiently utilizes the Bellman-Ford algorithm's concept of relaxing edges but modifies it to account for the number of stops constraint by limiting the number of relaxation iterations.

Example Walkthrough

Let's use a small example to illustrate the solution approach described above. Consider the following scenario where `n = 4` cities, `src = 0` (source city), `dst = 3` (destination city), `k = 1` (maximum 1 stop allowed), and the flights are represented by a list of `[from_i, to_i, price_i]` as follows: `flights = [[0, 1, 100], [1, 3, 100], [0, 2, 500], [2, 3, 100]]`.

In this example, there are direct flights from city 0 to city 1 and city 2, and from city 1 and city 2 to city 3 with the respective prices of 100 and 500.

1. **Initialization**: We start by initializing the `dist` array with `INF` values for all cities except the `src` city: `dist = [0, INF, INF, INF]`.

2. **Algorithm**: We will iterate `k+1` times, which in this case is 2 times (`1 stop + 1 initial trip`).

3. **Backup Array**: In each iteration, a backup of `dist` is made.

4. **Relaxation**:

   - First iteration:
     - Flight `[0, 1, 100]` (from city 0 to city 1): Checks if `dist[1] > backup[0] + 100`. Since `INF > 0 + 100`, update `dist[1]` to `100`.
     - Flight `[1, 3, 100]` (from city 1 to city 3): Checks if `dist[3] > backup[1] + 100`. Since `INF > INF + 100`, no update is made.
     - Flight `[0, 2, 500]` (from city 0 to city 2): Checks if `dist[2] > backup[0] + 500`. Since `INF > 0 + 500`, update `dist[2]` to `500`.
     - Flight `[2, 3, 100]` (from city 2 to city 3): Checks if `dist[3] > backup[2] + 100`. Since `INF > INF + 100`, no update is made.
   - Second iteration (with updated `dist` from first iteration; new backup is made):
     - Flight `[0, 1, 100]`: Since `dist[1]` is already at its lowest from the first iteration, no changes occur.
     - Flight `[1, 3, 100]`: Checks if `dist[3] > backup[1] + 100`. Since `INF > 100 + 100`, update `dist[3]` to `200`.
     - Flight `[0, 2, 500]`: Since `dist[2]` is already at its lowest from the first iteration, no changes occur.
     - Flight `[2, 3, 100]`: Checks if `dist[3] > backup[2] + 100`. Since `200 > 500 + 100`, no update is made. (Showing that we found a cheaper way to get to city 3 during this iteration)

5. **Returning the result**: After 2 iterations, we look at `dist[dst]`, which is `dist[3]`. The value is `200`, which is the cheapest price to reach city `dst` from `src` within 1 stop. This value is returned as the result. If `dist[dst]` had still been `INF`, then we would return `-1`, indicating no possible route within the given stop constraint.

Python Solution

```python
from typing import List

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:
        # Use a high value to represent 'infinity', since there is no direct representation in Python
        INF = float('inf')

        # Initialize distance array, with 'infinity' for all nodes except the source node
        distance = [INF] * n
        distance[src] = 0

        # Perform (K+1) iterations of Bellman-Ford algorithm to find the cheapest path with at most K stops
        for _ in range(k + 1):
            # Make a copy of distance array to prevent using updated values within the same iteration
            distance_backup = distance.copy()

            # Update the distance for each edge (flight) if a cheaper price is found
            for from_node, to_node, price in flights:
                # The new price is considered only if the previous node was reached within K stops
                distance[to_node] = min(distance[to_node], distance_backup[from_node] + price)

        # If the destination is still at 'infinity', it means it's unreachable within K stops
        return -1 if distance[dst] == INF else distance[dst]

```

Java Solution

```java
class Solution {
    // Represent an unreachable cost with a high value
    private static final int INFINITY = 0x3f3f3f3f;

    // Finds the cheapest price for a flight with up to k stops
    public int findCheapestPrice(int n, int[][] flights, int source, int destination, int k) {
        // Initialize distance array with infinity to represent no flights booked yet
        int[] currentDist = new int[n];
        // Backup array used during relaxation to avoid overwriting information prematurely
        int[] prevDist = new int[n];
        // Fill the distance array with infinity
        Arrays.fill(currentDist, INFINITY);
        // The cost to get to the source from the source is 0
        currentDist[source] = 0;

        // Perform relaxation k+1 times (since 0 stops means just 1 flight)
        for (int i = 0; i < k + 1; ++i) {
            // Copy current distances to the backup array
            System.arraycopy(currentDist, 0, prevDist, 0, n);
            // Iterate through each flight
            for (int[] flight : flights) {
                int from = flight[0], to = flight[1], cost = flight[2];
                // Relax the distance if a shorter path is found
                // Only update the distance using values from previous iteration (prevDist)
                currentDist[to] = Math.min(currentDist[to], prevDist[from] + cost);
            }
        }
        // If destination is reachable, return the cost, otherwise return -1
        return currentDist[destination] == INFINITY ? -1 : currentDist[destination];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {
        const int INF = 0x3f3f3f3f; // Define an "infinity" value for initial distances.
        vector<int> distances(n, INF); // Initialize all distances to "infinity" except the source.
        vector<int> previousIterationDistances; // Used to store distances from the previous iteration.
        distances[src] = 0; // The distance from the source to itself is always 0.

        // Run the Bellman-Ford algorithm for K+1 iterations because you can have at most K stops in between,
        // which translates to K+1 edges in the shortest path.
        for (int i = 0; i < K + 1; ++i) {
            // Make a copy of the current state of distances before this iteration.
            previousIterationDistances = distances;

            // For each edge in the graph, try to relax the edge and update the distance to the destination node
            for (const auto& flight : flights) {
                int from = flight[0], to = flight[1], price = flight[2];

                // Relaxation step: If the current known distance to 'from' plus the edge weight
                // to 'to' is less than the currently known distance to 'to', update it.
                if (previousIterationDistances[from] < INF) {
                    distances[to] = min(distances[to], previousIterationDistances[from] + price);
                }
            }
        }

        // After K+1 iterations, if the distance to the destination is still "infinity", no such path exists;
        // otherwise, return the shortest distance to the destination.
        return distances[dst] == INF ? -1 : distances[dst];
    }
};

```

Typescript Solution

```typescript
function findCheapestPrice(n: number, flights: number[][], src: number, dst: number, K: number): number {
    const INF = Number.POSITIVE_INFINITY; // Define an "infinity" value for initial distances.
    let distances: number[] = new Array(n).fill(INF); // Initialize all distances to "infinity" except the source.
    let previousIterationDistances: number[]; // Used to store distances from the previous iteration.
    distances[src] = 0; // The distance from the source to itself is always 0.

    // Run the Bellman-Ford algorithm for K+1 iterations because you can have at most K stops in between,
    // which translates to K+1 edges in the shortest path.
    for (let i = 0; i <= K; ++i) {
        // Make a copy of the current state of distances before this iteration.
        previousIterationDistances = distances.slice();

        // For each edge in the graph, try to relax the edge and update the distance to the destination node
        for (const flight of flights) {
            const [from, to, price] = flight;

            // Relaxation step: if the current known distance to 'from' plus the edge weight
            // to 'to' is less than the currently known distance to 'to', update it.
            if (previousIterationDistances[from] < INF) {
                distances[to] = Math.min(distances[to], previousIterationDistances[from] + price);
            }
        }
    }

    // After K+1 iterations, if the distance to the destination is still "infinity", no such path exists;
    // otherwise, return the shortest distance to the destination.
    return distances[dst] === INF ? -1 : distances[dst];
}

```

Time and Space Complexity

Time Complexity

The given code defines a function that finds the cheapest price for a flight from a source to a destination with at most `k` stops. The main operations in this code are as follows:

1. We initialize the `dist` array of size `n` with `INF` to represent the minimum cost to reach each node. This operation has a time complexity of `O(n)`.

2. We then run a loop `k+1` times to account for the fact that we can make at most `k` stops, which means we are considering paths with at most `k+1` edges. The loop operation involves iterating through the list of `flights`.

3. Inside the loop, a `.copy()` operation is performed on the `dist` array, which again takes `O(n)` time.

4. Then, in the nested loop, for each flight `(f, t, p)`, we update the `dist` of the destination `t` if a cheaper price is found by considering the flight from `f` to `t` with price `p`. With `m` being the number of flights, this nested loop has a time complexity of `O(m)` per iteration of the outer loop.

Putting these together, the total time complexity of the algorithm becomes the combined complexity of the loop executed `k+1` times and the nested iteration over all the flights, resulting in `O((k+1) * (n + m)) = O(k * (n + m))`.

In the worst case, we might have to consider every flight for every iteration, so the worst-case time complexity is `O(k * m)` when `m` is significantly larger than `n`.

Space Complexity

The space complexity of the algorithm is determined by:

1. The `dist` array, which consumes `O(n)` space.

2. The `backup` array, which also consumes `O(n)` space and is created anew in each iteration of the loop but does not add to the space complexity asymptotically as only one extra array exists at any given time.

Thus, the total space complexity is `O(n)`, due to the space required for the `dist` and `backup` arrays to store the cost of reaching each node.