**Problem statement**

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.

**Example**

* **Input:** n=3, flights=[[0,1,100],[1,2,100],[0,2,500]], src=0, dst=2, k=1
* **Output:** 200

1

100 100

500

**Iteration table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Step** | **Current city** | **Path taken** | **Currernt cost** | **Current stops** |
| 1 | 0 | start | 0 | -1 |
| 2 | 1 | 0 🡪 1 | 100 | 0 |
| 3 | 2 | 0 🡪 2 | 500 | 0 |

**Algorithm**

To solve this problem, we can use a modified version of Dijkstra’s algorithm, which is typically used to find the shortest path in terms of distance. Here, we need to consider the number of stops as an additional constraint.

**Approach**

1. **Graph Representation**: Represent the flights using an adjacency list where each city points to its neighboring cities along with the cost of the flight.
2. **Priority Queue**: Use a priority queue to explore paths in order of increasing cost. This ensures that once we reach the destination city, it’s done with the minimum possible cost.
3. **Algorithm Steps**:

• Initialize the priority queue with the source city, a cost of 0 (starting cost), and -1 stops (initially).

• Dequeue from the priority queue and explore each city along with its current accumulated cost and stops taken.

• If the dequeued city is the destination city, return the accumulated cost as the cheapest price found.

• If the number of stops taken is less than K, explore all neighboring cities: – Calculate the cost to reach each neighboring city by adding the cost of the flight to the current accumulated cost. – Enqueue these neighbors with updated costs and increment the number of stops by 1.

• Continue this process until either the priority queue is empty (indicating no more paths to explore) or the destination city is reached.

1. **Termination**: If no path is found within K stops, return −1 indicating that no valid route exists.

**Algorithm**

**Input**: Number of cities n, flights represented as flights, source city src, destination city dst, maximum stops k

**Output**: Cheapest price from src to dst with at most k stops, or −1 if no such route exists **Function** FindCheapestPrice(flights, src, dst, k):

Initialize a graph represented as an adjacency list;

Initialize a priority queue to store the current path cost and city;

Enqueue (src, 0, −1) into the priority queue ; // Start with source city, cost 0, and -1 stops

**while** priority queue is not empty **do**

Dequeue (city, cost,stops) from the priority queue;

**if** city is dst **then**

**return** cost;

**end**

**if** stops < k **then**

**for** each neighbor of city in the graph **do**

Calculate newCost as cost + price of flight;

Enqueue (neighbor, newCost,stops + 1) into the priority queue;

**end**

**end**

**end**

**return −1** ; // No valid path found

**Algorithm 1**: Find Cheapest Price with at most K Stops

**Solution code**

class Solution:

def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:

graph = defaultdict(list)

for u, v, price in flights:

graph[u].append((v, price))

pq = [(0, src, 0)]

min\_cost = defaultdict(lambda: float(’inf’))

min\_cost[(src, 0)] = 0

while pq:

cost, current\_city, stops = heappop(pq)

if current\_city == dst:

return cost

if stops <= k:

for neighbor, price in graph[current\_city]:

new\_cost = cost + price

if new\_cost < min\_cost[(neighbor, stops + 1)]:

min\_cost[(neighbor, stops + 1)] = new\_cost

heappush(pq, (new\_cost, neighbor, stops + 1))

return -1

**Bellman-Ford Algorithm for Cheapest Flights with K Stops**

**Algorithm**

1. Initialize an array dist with size n to store the minimum cost to reach each city from src.

Initialize dist[src] to 0 and all other entries to ∞.

2. Relax all edges n − 1 times:

• For each flight [u, v, w] in flights, if dist[u] + w < dist[v], update dist[v] to dist[u] + w.

3. After n − 1 iterations, dist[dst] contains the minimum cost to reach dst from src with at most k stops, or −1 if no such path exists.

**Source code**

class Solution:

def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:

# Step 1: Initialize distances array with infinity and set source distance to 0

inf = float(’inf’)

dist = [inf] \* n

dist[src] = 0

# Step 2: Relax edges for k + 1 times

for \_ in range(k + 1):

# Create a copy of dist array for the current iteration

current\_dist = dist[:]

# Relax all edges (u, v, w)

for u, v, w in flights:

if dist[u] != inf and dist[u] + w < current\_dist[v]:

current\_dist[v] = dist[u] + w

# Update dist array for the next iteration

dist = current\_dist

# Step 3: Return the shortest distance to dst, or -1 if not reachable

return dist[dst] if dist[dst] != inf else -1