2714. Find Shortest Path with K Hops

Hard Graph Shortest Path Heap (Priority Queue) Leetcode Link

Problem Description

You are working with a special type of graph, which is an undirected, weighted, and connected graph represented by a number of nodes and a list of edges. Each edge has a weight, establishing the cost to travel between two nodes. The unique challenge you face is to determine the shortest path from a start node `s` to a destination node `d`. The twist is the ability to "hop over" certain edges, making their weight effectively zero, but you can only do this for at most `k` edges. This "hop" capability allows you to ignore the weight of the selected edges, which can drastically change the result compared to the usual shortest path calculations.

The goal is to use this capability strategically, choosing up to `k` edges to skip, in order to minimize the total weight of the path from `s` to `d`.

Intuition

To tackle this problem, leverage Dijkstra's algorithm, which is commonly used to find the shortest paths between nodes in a weighted graph. Traditionally, Dijkstra's algorithm does not account for being able to bypass any edges. Because of this, you'll need to adapt the algorithm to accommodate the possibility of "hopping over" some edges.

To do this, create a modified graph representation that takes into account both the actual weight of the edges and the number of hops used thus far. Keep track of the shortest distances from the start node `s` to all other nodes with various numbers of hops, up to the maximum `k` hops allowed. This requires a two-dimensional array where one dimension is the node identifier and the second dimension is the number of hops used. Initialize the distances to infinity to ensure that any explored path will replace the placeholder value.

Use a priority queue to store and quickly access the current shortest path candidates, ordered by their distance. This queue will contain tuples with the current distance, the node identifier, and the number of hops used. Whenever a node is dequeued, examine its neighbors and attempt two types of updates: one where an additional hop is used (if you have hops left), and one where the edge's actual weight is considered in the usual manner.

By doing this at each stage, you are effectively exploring all combinations of used and unused hops, up to the limit `k`. After considering all nodes and paths, the shortest distance to the destination node `d` can be found by looking at the shortest distances recorded for reaching `d` with each possible number of hops, and then taking the minimum.

The underlying Dijkstra's algorithm uses a greedy strategy to guarantee the shortest path is found. By integrating it with the concept of hops, you can extend its utility to this unique scenario, providing an efficient and elegant solution.

Solution Approach

Let's break down the implementation of the solution as per the reference approach provided:

1. **Graph Construction**: We start by constructing a graph `g` that represents the given undirected weighted graph. This is a list of lists, where `g[u]` contains tuples `(v, w)` indicating there is an edge from the node `u` to node `v` with weight `w`. This step transforms the edge list into an adjacency list representing the same graph, which is a commonly used data structure for graph algorithms allowing efficient traversal of connected nodes.

2. **Distance Initialization**: Initially, we create a 2D list `dist` filled with infinity values. The dimensions are `[n][k + 1]`, where `n` is the number of nodes and `k` is the maximum number of hops allowed. `dist[u][t]` will store the shortest distance to reach node `u` using exactly `t` hops.

3. **Priority Queue**: A min-heap priority queue `pq` is used for efficient retrieval of the current shortest path candidate nodes to be evaluated. A tuple `(dis, u, t)` is pushed into the queue, where `dis` is the current shortest distance, `u` is the node, and `t` is the number of hops used to reach node `u`.

4. **Dijkstra's Algorithm with Modifications**: We adapt Dijkstra's algorithm to deal with hops. We pop a node from the priority queue and look at its neighbors. For each neighbor `v`, we consider two scenarios:

    - If we have remaining hops (i.e., `t + 1 <= k`), we consider what happens if we "hop over" the edge to `v`. If `dist[v][t + 1]` is greater than the current distance `dis` without adding the weight of the edge `w`, we found a shorter path to `v` with one more hop. We update `dist[v][t + 1]` and push `(dis, v, t + 1)` into `pq`.
    - We also consider the case where we don't use a hop. If the sum of `dis + w` is less than `dist[v][t]`, then we found a shorter path without using a hop, and we update `dist[v][t]` and push `(dis + w, v, t)` into `pq`.
5. **Finding the Shortest Path**: After we have processed all possible paths, the shortest path from the start node `s` to the destination node `d` can be deduced by finding the minimum distance from all the distances recorded in `dist[d][0...k]`.

The two main adaptations to the standard Dijkstra's algorithm are:

- The use of a `t` dimension in the `dist` array and priority queue tuples to keep track of the number of hops.
- Adjusting the path relaxation step to consider both "hopping over" an edge and the actual weight of the edge.

By applying these changes, we maintain the greedy nature of the standard Dijkstra's algorithm, ensuring that the shortest path is found, while also incorporating the additional rules about hopping over edges in an efficient manner.

Example Walkthrough

Let's use a small graph example to illustrate the solution approach. Our task is to find the shortest path from the start node `s` to the destination node `d`, with the ability to hop over at most `k` edges.

Given Graph Structure:

Let's consider a graph with 4 nodes and some edges with weights between them. Our nodes are `0` to `3`, where `0` is our start node `s` and `3` is our destination node `d`. Let's use `k = 1`, which means we can skip the weight of one edge.

The graph is represented by the following set of edges with weights:

- `(0, 1)` with weight `4`
- `(0, 2)` with weight `1`
- `(1, 3)` with weight `1`
- `(2, 3)` with weight `5`

Therefore, our adjacency list representation of the graph, `g`, after graph construction would be:

- `g[0]`: `[(1, 4), (2, 1)]`
- `g[1]`: `[(0, 4), (3, 1)]`
- `g[2]`: `[(0, 1), (3, 5)]`
- `g[3]`: `[(1, 1), (2, 5)]`

Step-by-Step Walkthrough:

1. **Graph Construction**: We have already constructed the adjacency list `g`.

2. **Distance Initialization**: We initialize `dist` as a 2D list with dimensions `[4][k + 1]`, filled with infinity:

    - `dist = [[inf, inf], [inf, inf], [inf, inf], [inf, inf]]`
3. **Priority Queue**: We start with a priority queue `pq` and push the start node `0` with a distance `0` and `0` hops used: `pq = [(0, 0, 0)]`.

4. **Dijkstra's Algorithm with Modifications**: Now, we start the modified Dijkstra's algorithm:

    - We pop `(0, 0, 0)` from `pq`. We update `dist[0][0]` to `0` as it's the starting node.

    - Checking neighbors of `0`, we have `1` and `2`. For `1` with edge weight `4`:

        - If we hop: since we haven't used any hops yet, we update `dist[1][1]` to `0` (`0` distance + `0` weight because we hopped), and add `(0, 1, 1)` to `pq`.
        - If we don't hop: we update `dist[1][0]` to `4` (`0` distance + `4` weight), and add `(4, 1, 0)` to `pq`.
    - For `2` with edge weight `1`:

        - If we hop: we update `dist[2][1]` to `0` (`0` distance + `0` weight), and add `(0, 2, 1)` to `pq`.
        - If we don't hop: we update `dist[2][0]` to `1` (`0` distance + `1` weight), and add `(1, 2, 0)` to `pq`.
    - Next, `pq` has `[(0, 1, 1), (0, 2, 1), (4, 1, 0), (1, 2, 0)]`, sorted by distance.

    - Processing continues, evaluating each node and its neighbors following the steps outlined, while always selecting the next closest node from the priority queue and updating `dist` considering both hopping and not hopping.

5. **Finding the Shortest Path**: After all possible paths are processed, we check `dist[3]`. The shortest path to `d` is the minimum of `dist[3][0]` and `dist[3][1]`.

In this example, if we hop from `0` to `2`, and then move from `2` to `3` without hopping, the total distance is `1` (edge `(0, 2)` weight) + `5` (edge `(2, 3)` weight) = `6`. Without hopping, the path `0 -> 1 -> 3` would have a weight of `5` which is longer than the path using a hop. Therefore, the shortest path using at most `k=1` hops is from `0` to `2` using a hop and then to `3` without a hop, yielding a minimum distance of `6`.

Python Solution

```python
from typing import List
from heapq import heappush, heappop
from math import inf

class Solution:
    def shortestPathWithHops(self, num_nodes: int, edges: List[List[int]],
                             source: int, destination: int, max_hops: int) -> int:
        # Create an adjacency list to store the graph
        graph = [[] for _ in range(num_nodes)]
        for start, end, weight in edges:
            graph[start].append((end, weight))
            graph[end].append((start, weight))

        # Initialize the distances to infinity, for all nodes and for each number of hops
        distances = [[inf] * (max_hops + 1) for _ in range(num_nodes)]
        # The distance to the source node with 0 hops is 0
        distances[source][0] = 0

        # Priority queue will store tuples of (distance, node, hops)
        priority_queue = [(0, source, 0)]

        # Continue processing until the priority queue is empty
        while priority_queue:
            # Get the node with the minimum distance
            cur_dist, cur_node, hops = heappop(priority_queue)

            # Explore all adjacent nodes
            for neighbor, weight in graph[cur_node]:
                # If we can reach the neighbor with an additional hop and it's beneficial
                if hops + 1 <= max_hops and distances[neighbor][hops + 1] > cur_dist:
                    distances[neighbor][hops + 1] = cur_dist
                    heappush(priority_queue, (cur_dist, neighbor, hops + 1))

                # If we can reach the neighbor without an additional hop and it offers a shorter path
                if distances[neighbor][hops] > cur_dist + weight:
                    distances[neighbor][hops] = cur_dist + weight
                    heappush(priority_queue, (cur_dist + weight, neighbor, hops))

        # Calculate the shortest path to the destination allowing for up to max_hops hops,
        # and return it if possible; return -1 if there is no path.
        shortest_path = min(distances[destination])
        return int(shortest_path) if shortest_path != inf else -1

```

Java Solution

```java
class Solution {
    public int shortestPathWithHops(int nodes, int[][] edges, int start, int destination, int maxHops) {
        List<int[]>[] graph = new List[nodes];
        Arrays.setAll(graph, i -> new ArrayList<>());

        // Construct an adjacency list from the edge list
        for (int[] edge : edges) {
            int from = edge[0], to = edge[1], weight = edge[2];
            graph[from].add(new int[] {to, weight});
            graph[to].add(new int[] {from, weight});
        }

        // Priority queue will be used to process nodes in order of distance
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // Starting with the start node, distance of 0 and 0 hops
        pq.offer(new int[] {0, start, 0});

        // Initialize distance array holding minimum distances for each hop count
        int[][] distances = new int[nodes][maxHops + 1];
        final int infinity = 1 << 30;
        for (int[] row : distances) {
            Arrays.fill(row, infinity);
        }
        distances[start][0] = 0;

        // Process nodes until priority queue is empty
        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int currentDistance = current[0], currentNode = current[1], currentHops = current[2];

            // Check each neighbour of the current node
            for (int[] edge : graph[currentNode]) {
                int nextNode = edge[0], edgeWeight = edge[1];

                // If hopping to the next node without increasing distance is possible and beneficial
                if (currentHops + 1 <= maxHops && distances[nextNode][currentHops + 1] > currentDistance) {
                    distances[nextNode][currentHops + 1] = currentDistance;
                    pq.offer(new int[] {currentDistance, nextNode, currentHops + 1});
                }

                // If going to the next node and increasing the distance is beneficial
                if (distances[nextNode][currentHops] > currentDistance + edgeWeight) {
                    distances[nextNode][currentHops] = currentDistance + edgeWeight;
                    pq.offer(new int[] {currentDistance + edgeWeight, nextNode, currentHops});
                }
            }
        }

        // Find the minimum distance to the destination within the allowed number of hops
        int result = infinity;
        for (int i = 0; i <= maxHops; ++i) {
            result = Math.min(result, distances[destination][i]);
        }

        // Return inf if no path satisfies the conditions
        return result == infinity ? -1 : result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <cstring>
#include <tuple>
#include <algorithm>
using namespace std;

class Solution {
public:
    int shortestPathWithHops(int n, vector<vector<int>>& edges, int start, int destination, int k) {
        // Create a graph representation with adjacency lists
        vector<vector<pair<int, int>>> graph(n);
        for (const auto& edge : edges) {
            int u = edge[0], v = edge[1], weight = edge[2];
            graph[u].emplace_back(v, weight);
            graph[v].emplace_back(u, weight);
        }

        // Declare a min-heap priority queue to maintain (distance, node, hops) tuples
        priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<tuple<int, int, int>>> minHeap;
        // Add the starting node to the queue with distance 0 and 0 hops
        minHeap.emplace(0, start, 0);

        // Initialize the distance array, setting all distances to a large value
        int distances[n][k + 1];
        memset(distances, 0x3f, sizeof(distances)); // Using 0x3f to fill the array with a high number
        distances[start][0] = 0;

        // Process the nodes in the queue
        while (!minHeap.empty()) {
            auto [currentDistance, currentNode, hops] = minHeap.top();
            minHeap.pop();

            // Iterate through all neighbors of the current node
            for (auto &[neighbor, weight] : graph[currentNode]) {
                // If within hops limit and the current path has a better distance, update and enqueue
                if (hops + 1 <= k && distances[neighbor][hops + 1] > currentDistance) {
                    distances[neighbor][hops + 1] = currentDistance;
                    minHeap.emplace(currentDistance, neighbor, hops + 1);
                }
                // If taking the edge leads to a better distance, update and enqueue
                if (distances[neighbor][hops] > currentDistance + weight) {
                    distances[neighbor][hops] = currentDistance + weight;
                    minHeap.emplace(currentDistance + weight, neighbor, hops);
                }
            }
        }

        // Calculate the minimum distance to the destination node within k hops
        int minDistance = *min_element(distances[destination], distances[destination] + k + 1);
        // If the minimum distance is still the initialized high value, return -1 (path not found)
        return (minDistance == 0x3f3f3f3f) ? -1 : minDistance;
    }
};

```

Typescript Solution

```typescript
type Edge = [number, number, number]; // Define a type for edges, represented as tuple [source, destination, weight]
type Graph = Map<number, Array<[number, number]>>; // Graph type with an adjacency list

const buildGraph = (n: number, edges: Edge[]): Graph => {
    const graph = new Map<number, Array<[number, number]>>();
    for (const [u, v, weight] of edges) {
        if (!graph.has(u)) graph.set(u, []);
        if (!graph.has(v)) graph.set(v, []);
        graph.get(u)?.push([v, weight]);
        graph.get(v)?.push([u, weight]);
    }
    return graph;
};

const shortestPathWithHops = (n: number, edges: Edge[], start: number, destination: number, k: number): number => {
    // Create a graph representation with adjacency lists
    const graph = buildGraph(n, edges);

    // Define a type for priority queue elements: [distance, node, hops]
    type PriorityQueueElement = [number, number, number];
    // Create a min-heap priority queue to maintain the nodes
    const minHeap: PriorityQueueElement[] = [];
    const enqueue = (element: PriorityQueueElement) => {
        minHeap.push(element);
        minHeap.sort(([distanceA], [distanceB]) => distanceA - distanceB);
    };
    const dequeue = () => minHeap.shift();

    // Add the starting node to the queue with distance 0 and 0 hops
    enqueue([0, start, 0]);

    // Initialize the distance array, setting all distances to a large value
    const distances: number[][] = Array.from({ length: n }, () => Array(k + 1).fill(Infinity));
    distances[start][0] = 0;

    // Process the nodes in the queue
    while (minHeap.length) {
        const [currentDistance, currentNode, hops] = dequeue()!;

        // Iterate through all neighbors of the current node
        graph.get(currentNode)?.forEach(([neighbor, weight]) => {
            // If within hops limit and the current path has a better distance, update and enqueue
            if (hops + 1 <= k && distances[neighbor][hops + 1] > currentDistance) {
                distances[neighbor][hops + 1] = currentDistance;
                enqueue([currentDistance, neighbor, hops + 1]);
            }

            // If taking the edge leads to a better distance, update and enqueue
            if (distances[neighbor][hops] > currentDistance + weight) {
                distances[neighbor][hops] = currentDistance + weight;
                enqueue([currentDistance + weight, neighbor, hops]);
            }
        });
    }

    // Calculate the minimum distance to the destination node within k hops
    const minDistance = Math.min(...distances[destination]);
    // If the minimum distance is still the initialized high value, return -1 (path not found)
    return isFinite(minDistance) ? minDistance : -1;
};

// You can now call the function 'shortestPathWithHops' with the parameters as required.

```

Time and Space Complexity

The time complexity of the given code is `O(E + n * k * log(n))`, where `E` represents the edges in the given graph, `n` is the number of nodes, and `k` is the maximum number of hops. The `E` term comes from the initial edge iteration to construct the adjacency list, and `n * k * log(n)` comes from the while loop where we consider each hop for each node and the priority queue (min-heap) operations which have `O(log n)` complexity. Specifically, if all nodes are connected to all other nodes the edges number would be close to `n^2`, making the overall time complexity look like `O(n^2 * log n)`.

The space complexity of the code is `O(n * k)`, which is used to store distances for every node at every possible hop from 0 to `k`.