743. Network Delay Time

Aayush Adhikari

June 23, 2024

Problem Statement

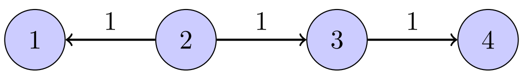
Problem Statement

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.
We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Examples

Example 1

- **Input**: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
- **Output**: 2



Iteration Table - Bellman-Ford Algorithm

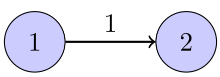
Iteration	dist array
Initial	$[\infty, 0, \infty, \infty, \infty]$
1	$[\infty, 0, 1, \infty, \infty]$
2	$[\infty, 0, 1, 2, \infty]$
3	$[\infty, 0, 1, 2, 3]$

Iteration Table - Dijkstra's Algorithm

Iteration	dist array
Initial	$[\infty, 0, \infty, \infty, \infty]$
Step 1	$[\infty, 0, 1, \infty, \infty]$
Step 2	$[\infty, 0, 1, 2, \infty]$
Step 3	$[\infty, 0, 1, 2, 3]$

Example 2

- **Input**: times = [[1,2,1]], n = 2, k = 1
- **Output**: 1



Iteration Table - Bellman-Ford Algorithm

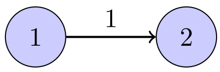
Iteration	dist array
Initial	$[\infty, \infty, 0]$
1	$[\infty, 0, 1]$

Iteration Table - Dijkstra's Algorithm

Iteration	dist array
Initial	$[\infty, \infty, 0]$
Step 1	$[\infty, 0, 1]$

Example 3

- **Input**: times = [[1,2,1]], n = 2, k = 2
- **Output**: -1



Iteration Table - Bellman-Ford Algorithm

Iteration	dist array
Initial	$[\infty, \infty, 0]$
1	$[\infty, \infty, 1]$

Iteration Table - Dijkstra's Algorithm

Iteration	dist array
Initial	$[\infty, \infty, 0]$
Step 1	$[\infty, \infty, 1]$

Approach for Dijkstra's Algorithm

To solve this problem, we can use Dijkstra's algorithm, which is suitable for finding the shortest paths from a single source node to all other nodes in a weighted graph.

Approach

1. **Graph Representation**: Represent the network using an adjacency list where each node points to a list of its neighbors and the corresponding travel times.
2. **Dijkstra's Algorithm**: Initialize a distance array 'dist' where 'dist[k] = 0' (distance to itself is zero) and 'dist[i] = infinity' for all other nodes initially. Use a priority queue (min-heap) to continually extract the node with the smallest known distance from 'k' and relax (update) distances to its neighbors if a shorter path is found.
3. **Result Calculation**:
   - After running Dijkstra's algorithm, check the 'dist' array:
     - If any node still has distance 'infinity', it means that node is unreachable from 'k', and hence it's impossible for all nodes to receive the signal. In such cases, return -1.
     - Otherwise, return the maximum value in the 'dist' array, which represents the minimum time it takes for the signal to reach all nodes.

Algorithm for Dijkstra's Algorithm

Algorithm

Input: times: List[List[int]], n: int, k: int
Output: Minimum time it takes for all n nodes to receive the signal, or -1 if impossible

Build the graph from ‘times‘ using an adjacency list;
Initialize ‘dist‘ array with infinity values, except ‘dist[k] = 0‘;
Use a priority queue (min-heap) to implement Dijkstra's algorithm starting from node k;

while *priority queue is not empty* **do**
    Extract the node u with the smallest distance from the priority queue;
    Relax all edges from u to its neighbors in the graph;
end

if *any node still has distance infinity* **then**
    return-1;
end
else
    return the maximum value in ‘dist‘ array;
end

Algorithm 1: Network Delay Time using Dijkstra's Algorithm

Solution Code for Dijkstra's Algorithm

```
import heapq
from collections import defaultdict

def networkDelayTime(times, n, k):
    # Step 1: Build the graph representation
    graph = defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))

    # Step 2: Dijkstra's algorithm to find the shortest paths from node k
    dist = {node: float('\infty') for node in range(1, n+1)}
    dist[k] = 0
    min_heap = [(0, k)]  # (distance, node)

    while min_heap:
        current_dist, u = heapq.heappop(min_heap)

        if current_dist > dist[u]:
            continue

        for v, weight in graph[u]:
            distance = current_dist + weight
            if distance < dist[v]:
                dist[v] = distance
                heapq.heappush(min_heap, (distance, v))

    # Step 3: Determine the maximum distance in dist array
    max_dist = max(dist.values())

    if max_dist == float('\infty'):
        return -1  # Some nodes are unreachable
    else:
        return max_dist
```

Approach for Bellman-Ford Algorithm

To solve this problem, we can use the Bellman-Ford algorithm, which is suitable for finding shortest paths from a single source node to all other nodes in a graph with negative edge weights.

Approach

1. **Initialize**: Create an array 'dist' with size $n+1$ where $\text{dist}[k] = 0$ and $\text{dist}[i] = \infty$ for all other nodes initially. This array will hold the minimum distance from node k to all other nodes.
2. **Relaxation**: Relax edges repeatedly. For each edge (u, v, w) in 'times', if $\text{dist}[u] + w < \text{dist}[v]$, update $\text{dist}[v]$ to $\text{dist}[u] + w$.
3. **Detect Negative Cycles**: After $n-1$ iterations, the 'dist' array will contain the minimum distances from k to all nodes reachable from k. If there's any further improvement in the n-th iteration, it indicates the presence of a negative cycle.
4. **Check Unreachable Nodes**: If there's no negative cycle and any node still has distance 'infinity', return -1 (some nodes are unreachable). Otherwise, return the maximum value in the 'dist' array, which represents the minimum time it takes for the signal to reach all nodes.

Algorithm for Bellman-Ford Algorithm

Algorithm

Input: times: List[List[int]], n: int, k: int
Output: Minimum time it takes for all n nodes to receive the signal, or -1 if impossible

Initialize 'dist' array with size $n+1$ with all elements set to ∞, except 'dist[k] = 0';

for *i from 1 to* $n-1$ **do**
  for *each edge* (u, v, w) *in 'times'* **do**
    if $dist[u] + w < dist[v]$ **then**
      $\text{dist}[v] = \text{dist}[u] + w$;
    end
  end
end

if *any node still has distance* ∞ **then**
  return-1;
end
else
  return the maximum value in 'dist' array;
end

Algorithm 2: Network Delay Time using Bellman-Ford Algorithm

Solution Code using Bellman-Ford Algorithm

```
import sys

def networkDelayTime(times, n, k):
    # Step 1: Initialize distances
    \infty = sys.maxsize
    dist = [\infty] * (n + 1)
    dist[k] = 0

    # Step 2: Relax edges repeatedly
    for _ in range(n - 1):
```

```
        for u, v, w in times:
            if dist[u] != \infty and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Step 3: Check for negative cycles
    for u, v, w in times:
        if dist[u] != \infty and dist[u] + w < dist[v]:
            return -1

    # Step 4: Return the maximum value in dist array
    max_time = max(dist[1:])
    return max_time if max_time != \infty else -1
```