743. Network Delay Time - Média 2

Problema:

> Seja uma rede de n nós (rotulados de 1 a n) e uma lista de tempos, que são os tempos de viagem representados como arestas direcionadas, onde times[i] = (ui, vi, wi), sendo ui o nó de origem, vi o nó de destino e wi o tempo que leva para um sinal viajar do nó de origem para o nó de destino.

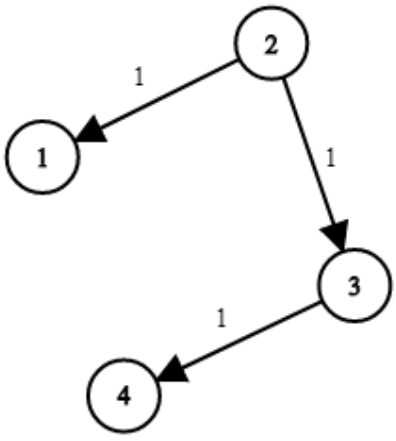
> É enviado um sinal de um nó específico k. Retorne o tempo mínimo necessário para que todos os n nós recebam o sinal. Se for impossível para todos os n nós receberem o sinal, retorne -1.

Restrições:

- 1 <= k <= n <= 100
- 1 <= times.length <= 6000
- times[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 0 <= wi <= 100
- Todos os pares (ui, vi) são únicos (ou seja, sem arestas múltiplas).

Exemplos:

Exemplo 1:



Entrada:

times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

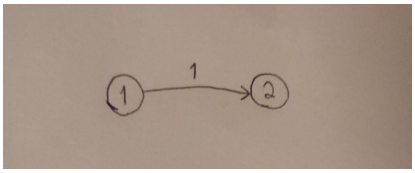
Saída:

2

Explicação:

> - Saindo do nó 2, o sinal atingiria os nós {1, 3} em tempo 1 e chegaria ao nó quatro em tempo 2.

Exemplo 2:



Entrada:

times = [[1,2,1]], n = 2, k = 1

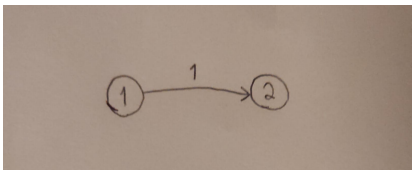
Saída:

1

Explicação:

> - Saindo do nó 1, o sinal atingiria os nós {2} em tempo 1.

Exemplo 3:



Entrada:

Input: times = [[1,2,1]], n = 2, k = 2

Saída:

-1

Explicação:

> - Saindo do nó 2, o sinal não chegaria a nenhum outro vértice pois não existem arestas saindo de 2.

Solução proposta:

Para soluções em Python3, o LeetCode oferece o seguinte ponto de partida:

```
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) ->
int:
```

O algoritmo de Dijkstra foi escolhido para implementar a solução, uma vez que interessa encontrar o caminho de menor custo para o percurso que o sinal desempenha entre os nós.

Primeiramente, define-se o grafo a ser utilizado:

```
graph = defaultdict(list)
for u, v, w in times:
    graph[u].append((v, w))
```

Em seguida, definem-se as estruturas auxiliares para a execução do algoritmo, sendo elas uma fila de prioridade (para diminuir o custo do algoritmo) e um dicionário para armazenar o caminho de menor custo para os nós que já entraram na "mancha".

```
djk_HEAP = [(0, k)]  # (custo, nó atual)
djk_SET = {}
```

O loop principal do algoritmo é responsável por processar a fila de prioridade até que todos os nós sejam visitados ou a fila se esvazie, como o nó é desenfileirado de um HEAP de mínimo, é garantido que ele será o nó de menor custo:

```
while djk_HEAP:
    custo, no_atual = heapq.heappop(djk_HEAP)

    if no_atual in djk_SET:
        continue

    djk_SET[no_atual] = custo
```

Ainda no loop, para cada nó processado, as arestas vizinhas são inseridas na heap, atualizando o custo acumulado. Se um nó vizinho ainda não foi visitado, ele é adicionado à heap com o custo atualizado:

```
for neighbor, weight in graph[no_atual]:
    if neighbor not in djk_SET:
        heapq.heappush(djk_HEAP, (custo + weight, neighbor))
```

Por fim, se algum nó não foi alcançado a solução retorna -1, caso contrário, o custo mínimo da travessia:

```
if len(djk_SET) == n:
    return max(djk_SET.values())
return -1
```