2642. Design Graph With Shortest Path Calculator - Difícil 2

Problema:

> Há um grafo dirigido e ponderado que consiste em n nós numerados de 0 a n - 1.

> As arestas do grafo são inicialmente representadas pelo array fornecido edges, onde edges[i] = [from_i, to_i, edgeCost_i] significa que há uma aresta de from_i para to_i com custo edgeCost_i.

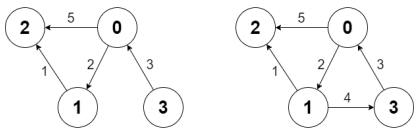
> Implemente a classe Graph:
>
> - Graph(int n, int[][] edges) inicializa o objeto com n nós e as arestas fornecidas.
> - addEdge(int[] edge) adiciona uma aresta à lista de arestas, onde edge = [from, to, edgeCost].
> - int shortestPath(int node1, int node2) retorna o custo mínimo de um caminho de node1 para node2. Se nenhum caminho existir, retorna -1. O custo de um caminho é a soma dos custos das arestas no caminho.

Restrições:

- É garantido que não existe uma aresta entre dois nós antes de adicionar usando addEdge.
- 1 <= n <= 100
- 0 <= edges.length <= n * (n - 1)
- edges[i].length == edge.length == 3
- 0 <= from_i, to_i, from, to, node1, node2 <= n - 1
- 1 <= edgeCost_i, edgeCost <= 10^6
- Não há arestas repetidas nem laços (self-loops) no grafo em nenhum momento.
- No máximo 100 chamadas serão feitas para addEdge.
- No máximo 100 chamadas serão feitas para shortestPath.

Exemplos:

Exemplo 1:



Entrada:

["Graph", "shortestPath", "shortestPath", "addEdge", "shortestPath"]

[[4, [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]]], [3, 2], [0, 3], [[1, 3, 4]], [0, 3]]

Saída:

[null, 6, -1, null, 6]

Explicação:

> - Após inicializar o grafo g com 4 nós (numerados de 0 a 3) e as arestas [[0, 2, 5], [0, 1, 2], [1, 2, 1], [3, 0, 3]], o primeiro comando g.shortestPath(3, 2) calcula o caminho mais curto do nó 3 para o nó 2. Nesse caso, o menor caminho é 3 → 0 → 1 → 2, com um custo total de 3 + 2 + 1 = 6, então o resultado retornado é 6. Em seguida, o comando g.shortestPath(0, 3) tenta encontrar um caminho do nó 0 para o nó 3, mas como não há um caminho disponível, o método retorna -1. Posteriormente, a chamada g.addEdge([1, 3, 4]) adiciona uma nova aresta ao grafo, conectando o nó 1 ao nó 3 com um custo de 4. Após essa adição, o comando g.shortestPath(0, 3) é executado novamente e, agora, o menor caminho do nó 0 para o nó 3 é 0 → 1 → 3, com um custo total de 2 + 4 = 6. Assim, o resultado retornado é 6.

Solução proposta:

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

```
class Graph:

    def __init__(self, n: int, edges: List[List[int]]):


    def addEdge(self, edge: List[int]) -> None:


    def shortestPath(self, node1: int, node2: int) -> int:



# Your Graph object will be instantiated and called as such:
# obj = Graph(n, edges)
# obj.addEdge(edge)
# param_2 = obj.shortestPath(node1,node2)
```

Para iniciar, começamos pelo contrutor do grafo:

```
def __init__(self, n: int, edges: List[List[int]]) -> None:
```

Define-se o n como o n recebido, simbolizando o número de nós:

```
self.n = n
```

Cria-se uma lista de adjacências para guardar as arestas e seus custos:

```
self.graph = [[] for _ in range(n)]  # Lista de adjacência
```

```
for from_node, to_node, cost in edges:
    self.graph[from_node].append((to_node, cost))
```

Em seguida, foi implementado o método para adicionar uma aresta:

```
def addEdge(self, edge: List[int]) -> None:
    from_node, to_node, cost = edge
    self.graph[from_node].append((to_node, cost))
```

Para criar o método shortestPath() foi utilizado o algoritmo de Dijkstra. Para maior eficiência, utilizamos uma fila de prioridades implementada com uma HEAP de mínimo:

```
djk_HEAP = [(0, no_inicio)]
```

Por fim, o algoritmo é aplicado normalmente e retorna -1 caso não exista caminho:

```
        djk_SET = [float('inf')] * self.n
        djk_SET[no_inicio] = 0

        # Enquanto a fila não estiver vazia
        while djk_HEAP:
            custo_no_atual, no_atual = heapq.heappop(djk_HEAP)

            if no_atual == no_dest:
                return custo_no_atual

            if custo_no_atual > djk_SET[no_atual]: # Existe opção melhor
                continue

            # Insere arestas vizinhas na HEAP
            for neighbor, cost in self.graph[no_atual]:
                new_dist = custo_no_atual + cost
                if new_dist < djk_SET[neighbor]:
                    djk_SET[neighbor] = new_dist
                    heapq.heappush(djk_HEAP, (new_dist, neighbor))

        return -1
```