Módulo Grafos

Curso Estruturas de Dados e Algoritmos Expert

Prof. Dr. Nelio Alves
https://devsuperior.com.br

Lista de exercícios

Soluções:
https://github.com/devsuperior/curso-eda/tree/main/grafos

Problema "juiz_cidade" (Adaptado de LeetCode 997)
Empresas: Amazon, Apple, Adobe

Em uma cidade, há n pessoas rotuladas de 1 a n. Há um boato de que uma dessas pessoas é secretamente o juiz da cidade.

Se o juiz da cidade existir, então:

1. O juiz da cidade não confia em ninguém.
2. Todos (exceto o juiz da cidade) confiam no juiz da cidade.
3. Existe exatamente uma pessoa que satisfaz as propriedades 1 e 2.

Você recebe um array `trust` onde `trust[i] = [ai, bi]` representa que a pessoa rotulada como `ai` confia na pessoa rotulada como `bi`. Se uma relação de confiança não existir no array `trust`, então tal relação de confiança não existe.

Retorne o rótulo do juiz da cidade se ele existir e puder ser identificado, ou retorne -1 caso contrário.

Restrições

- 1 <= n <= 1000
- 0 <= trust.length <= 10^4
- trust[i].length == 2
- Todos os pares de trust são únicos
- ai != bi
- 1 <= ai, bi <= n

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "n": 2,`
`  "trust": [[1, 2]]`
	2

}	

Exemplo 2:

Entrada 2	**Saída 2**

{
  "n": 3,
  "trust": [[1, 3], [2, 3]]
}
	3

Exemplo 3:

Entrada 2	**Saída 2**

{
  "n": 3,
  "trust": [[1, 3], [2, 3], [3, 1]]
}
	-1

Assinaturas:

Javascript:
```
var findJudge = function(n, trust)
```

Java:
```
public int findJudge(int n, int[][] trust)
```

C#:
```
public int FindJudge(int n, int[][] trust)
```

Python:
```
def findJudge(n, trust):
```

Problema "caminho_valido” (Adaptado de Leetcode 1971)
Empresas: Facebook, Google, Microsoft, Amazon

Há um grafo bidirecional com n vértices, onde cada vértice é rotulado de 0 a n - 1 (inclusive). As arestas no grafo são representadas como um array 2D de inteiros chamado edges, onde cada `edges[i] = [ui, vi]` denota uma aresta bidirecional entre o vértice ui e o vértice vi. Cada par de vértices está conectado por no máximo uma aresta, e nenhum vértice possui uma aresta para si mesmo.

Você quer determinar se existe um caminho válido que vá do vértice source ao vértice destination.

Dados edges e os inteiros n, source, e destination, retorne true se existir um caminho válido de source para destination, ou false caso contrário.

Restrições

- 1 <= n <= 2 * 10^5
- 1 <= n <= 2 * 10^5
- 0 <= edges.length <= 2 * 10^5
- edges[i].length == 2
- 0 <= ui, vi <= n - 1
- ui != vi
- 0 <= source, destination <= n - 1
- Não há arestas duplicadas.
- Não há laços.

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "n": 3,`
`  "edges": [[0,1],[1,2],[2,0]],`
`  "source": 0,`
`  "destination": 2`
`}`
	`1`

Explicação: Existem dois caminhos do vértice 0 ao vértice 2.

- - 0 → 1 → 2
- - 0 → 2

Exemplo 2:

Entrada 2	**Saída 2**

{
  "n": 6,
  "edges": [[0,1],[0,2],[3,5],[5,4],[4,3]],
  "source": 0,
  "destination": 5
}
	3

Explicação: Não há caminho do vértice 0 ao vértice 5.

Assinaturas:

Javascript:
```
var validPath = function(n, edges, source, destination)
```

Java:
```
public boolean validPath(int n, int[][] edges, int source, int
destination)
```

C#:
```
public bool ValidPath(int n, int[][] edges, int source, int
destination)
```

Python:
```
def validPath(n, edges, source, destination):
```

Problema "provincias" (Adaptado de LeetCode 547)
Empresas: Amazon, Google, DoorDash, Apple, TikTok, etc.

Há n cidades. Algumas delas estão conectadas, enquanto outras não. Se a cidade a está conectada diretamente com a cidade b, e a cidade b está conectada diretamente com a cidade c, então a cidade a está conectada indiretamente com a cidade c.

Uma província é um grupo de cidades conectadas diretamente ou indiretamente e sem outras cidades fora desse grupo.

Você recebe uma matriz n x n chamada `isConnected`, onde `isConnected[i][j] = 1` se a cidade i e a cidade j estiverem conectadas diretamente, e `isConnected[i][j] = 0` caso contrário.

Retorne o número total de províncias.

Restrições

- 1 <= n <= 200
- n == isConnected.length
- n == isConnected[i].length
- isConnected[i][j] is 1 or 0.
- isConnected[i][i] == 1
- isConnected[i][j] == isConnected[j][i]

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "isConnected": [`
`  [1, 1, 0],`
`  [1, 1, 0],`
`  [0, 0, 1]`
`  ]`
`}`
	`2`

Exemplo 2:

Entrada 2	**Saída 2**

`{`
`  "isConnected": [`
`  [1, 0, 0],`
`  [0, 1, 0],`
`  [0, 0, 1]`
`  ]`
`}`
	`3`

`Assinaturas:`

Javascript:
```
var findCircleNum = function(isConnected)
```

Java:
```
public int findCircleNum(int[][] isConnected)
```

C#:
```
public int FindCircleNum(int[][] isConnected)
```

Python:
```
def findCircleNum(isConnected):
```

Problema "chaves_e_salas" (Adaptado de LeetCode 841)
Empresas: Tinkoff, Walmart Labs, Amazon, Uber, Google

Há n salas rotuladas de 0 a n - 1 e todas as salas estão trancadas, exceto a sala 0. Seu objetivo é visitar todas as salas. No entanto, você não pode entrar em uma sala trancada sem ter sua chave.

Quando você visita uma sala, pode encontrar um conjunto de chaves distintas nela. Cada chave tem um número, indicando qual sala ela abre, e você pode pegar todas elas para desbloquear as outras salas.

Dado um array `rooms` onde `rooms[i]` é o conjunto de chaves que você pode obter se visitasse a sala i, retorne verdadeiro se você pode visitar todas as salas, ou falso caso contrário.

Constraints:

- n == rooms.length
- 2 <= n <= 1000
- 0 <= rooms[i].length <= 1000
- 1 <= sum(rooms[i].length) <= 3000
- 0 <= rooms[i][j] < n
- Todos os valores de rooms[i] são únicos.

Exemplo 1:

Entrada 1	**Saída 1**
`{ "rooms": [[1], [2], [3], []] }`	`true`

Exemplo 2:

Entrada 2	**Saída 2**
`{ "rooms": [[1, 3], [3, 0, 2], [2], [0]] }`	`false`

Explicação: Não conseguimos entrar na sala de número 2 pois a única chave que a desbloqueia está nessa mesma sala.

Assinaturas:

Javascript:
```
var canVisitAllRooms = function(rooms)
```

Java:

```
public boolean canVisitAllRooms(List<List<Integer>> rooms)

C#:
public bool CanVisitAllRooms(IList<IList<int>> rooms)

Python:
def canVisitAllRooms(rooms):
```

Problema "requisitos" (Adaptado de Leetcode 207)
Empresas: Facebook, Amazon, Microsoft, Apple, Yahoo, TikTok, Google, Oracle, etc.

Há um total de `numCourses` cursos que você precisa fazer, rotulados de 0 a numCourses - 1. Você recebe um array `prerequisites` onde prerequisites[i] = [ai, bi] indica que você deve fazer o curso bi primeiro se quiser fazer o curso ai.

Por exemplo, o par [0, 1] indica que, para fazer o curso 0, você deve primeiro fazer o curso 1.

Retorne verdadeiro se você conseguir concluir todos os cursos. Caso contrário, retorne falso.

Restrições:

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= ai, bi < numCourses
- All the pairs prerequisites[i] are unique.

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "numCourses": 2,`
`  "prerequisites": [[1, 0]]`
`}`
	`true`

Explicação: Você precisa fazer 2 cursos. Para fazer o curso 1 você precisa ter finalizado o curso 0, então é possível.

Exemplo 2:

Entrada 2	**Saída 2**

`{`
`  "numCourses": 2,`
`  "prerequisites": [[1, 0], [0, 1]]`
`}`
	`false`

Explicação: Você precisa fazer 2 cursos. Para fazer o curso 1 você precisa ter finalizado o curso 0, mas para fazer o curso 0 você precisa ter finalizado o curso 1, então é impossível.

```
Assinaturas:

Javascript:
var canFinish = function(numCourses, prerequisites)

Java:
public boolean canFinish(int numCourses, int[][] prerequisites)

C#:
```

```
public bool CanFinish(int numCourses, int[][] prerequisites)

Python:
def canFinish(numCourses, prerequisites):
```

Problema "latencia_rede" (Adaptado de LeetCode 743)
Empresas: Amazon, Google, TikTok, Adobe, etc.

Você recebe uma rede de n nós, rotulados de 1 a n. Você também recebe `times`, uma lista de tempos de viagem como arestas direcionadas `times[i] = (ui, vi, wi)`, onde `ui` é o nó de origem, `vi` é o nó de destino e `wi` é o tempo que leva para um sinal viajar da origem ao destino.

Enviaremos um sinal a partir de um nó dado k. Retorne o tempo mínimo que leva para todos os n nós receberem o sinal. Se for impossível que todos os n nós recebam o sinal, retorne -1.

Constraints:

- 1 <= k <= n <= 100
- 1 <= times.length <= 6000
- times[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 0 <= wi <= 100
- All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "times": [[2, 1, 1], [2, 3, 1], [3, 4, 1]],`
`  "n": 4,`
`  "k": 2`
`}`
	`2`

Exemplo 2:

Entrada 2	**Saída 2**

`{`
`  "times": [[1, 2, 1]],`
`  "n": 2,`
`  "k": 1`
`}`
	`1`

Exemplo 3:

Entrada 3	**Saída 3**

`{`
`  "times": [[1, 2, 1]],`
`  "n": 2,`
`  "k": 2`
`}`
	`-1`

`Assinaturas:`

Javascript:
```
var networkDelayTime = function(times, n, k)
```

Java:
```
public int networkDelayTime(int[][] times, int n, int k)
```

C#:
```
public int NetworkDelayTime(int[][] times, int n, int k)
```

Python:
```
def networkDelayTime(self, times, n, k):
```

Problema "conectando_cidades" (Adaptado de LeetCode 1135)
Empresas: Amazon

Existem N cidades numeradas de 1 a N.

Você recebe conexões, onde cada conexão `connections[i] = [city1, city2, cost]` representa o custo para conectar `city1` e `city2` juntas. (Uma *conexão* é bidirecional: conectar `city1` e `city2` é o mesmo que conectar `city1` e `city2`.)

Retorne o custo mínimo para que, para cada par de cidades, exista um caminho de conexões (possivelmente de comprimento 1) que conecta essas duas cidades juntas. O custo é a soma dos custos de conexão utilizados. Se a tarefa for impossível, retorne -1.

O **custo** é a soma do custo das conexões utilizadas.

Restrições:

- 1 <= n <= 10^4
- 1 <= connections.length <= 10^4
- connections[i].length == 3
- 1 <= xi, yi <= n
- xi != yi
- 0 <= costi <= 10^5

Exemplo 1:

Entrada 1	**Saída 1**

`{`
`  "n": 3,`
`  "connections": [[1, 2, 5], [1, 3, 6], [2, 3, 1]]`
`}`
	`6`

Explicação: Escolher quaisquer duas arestas conectará todas as cidades, então escolhemos as de custo mínimo, 5 e 1.

Exemplo 2:

Entrada 2	**Saída 2**

`{`
`  "n": 3,`
`  "connections": [[1, 2, 3], [3, 4, 4]]`
`}`
	`-1`

Explicação: Não há como conectar todas as cidades mesmo que usemos todas as arestas.

```
Assinaturas:

Javascript:
var minimumCost = function(n, connections)

Java:
public int minimumCost(int n, int[][] connections)
```

C#:
```
public int MinimumCost(int n, int[][] connections)
```

Python:
```
def minimumCost(n, connections):
```