Módulo 4 - Aula 1

Representação de Grafos e DFS

Representação de Grafos

Representação de Grafos

Matriz e lista adjacência

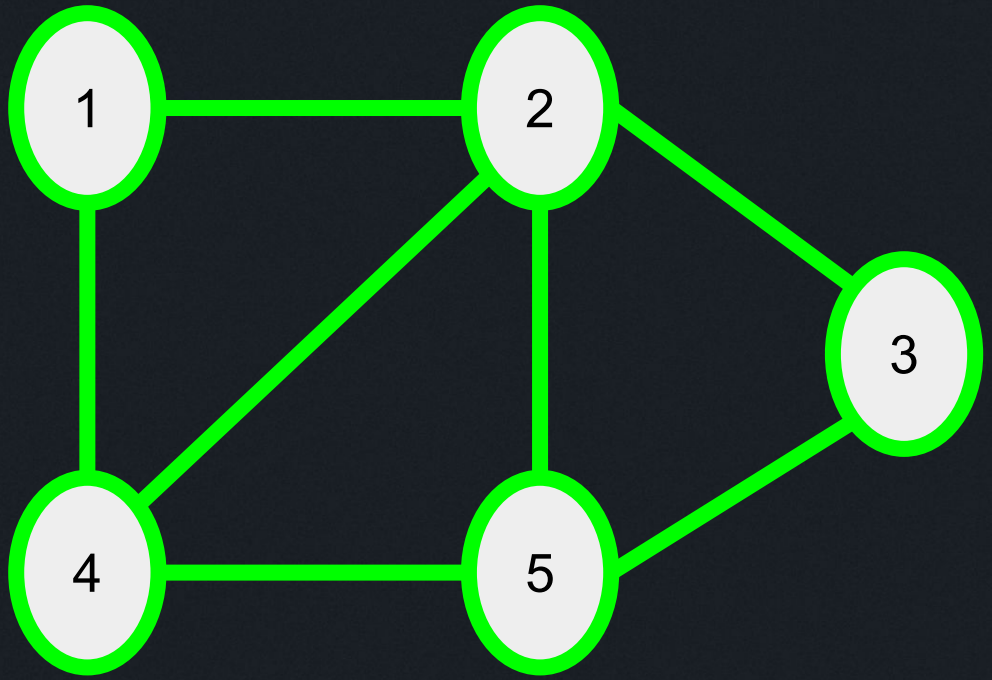
É uma estrutura de dados G, definida como G={V, E}, onde:

- V = conjunto de nós / vértices
  - Servem para modelar elementos de um problema

- E = conjunto de arestas
  - Cada aresta liga um par de nós, representando uma associação entre esses elementos do problema
  - Podem ser ponderadas ou não
  - Podem ser direcionadas (grafos dígrafos) ou não

Representação de Grafos

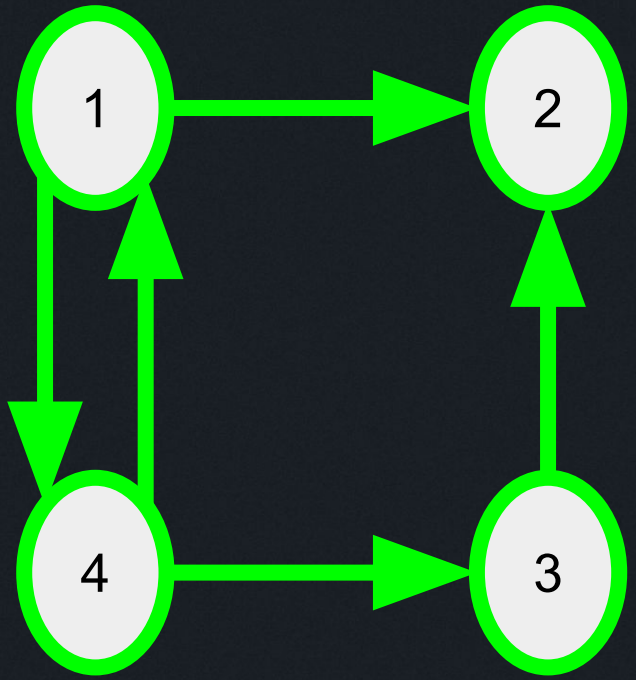
Matriz e lista adjacência



Grafo com 5 nós e 7 arestas
Arestas não ponderadas
Arestas não direcionadas

Representação de Grafos

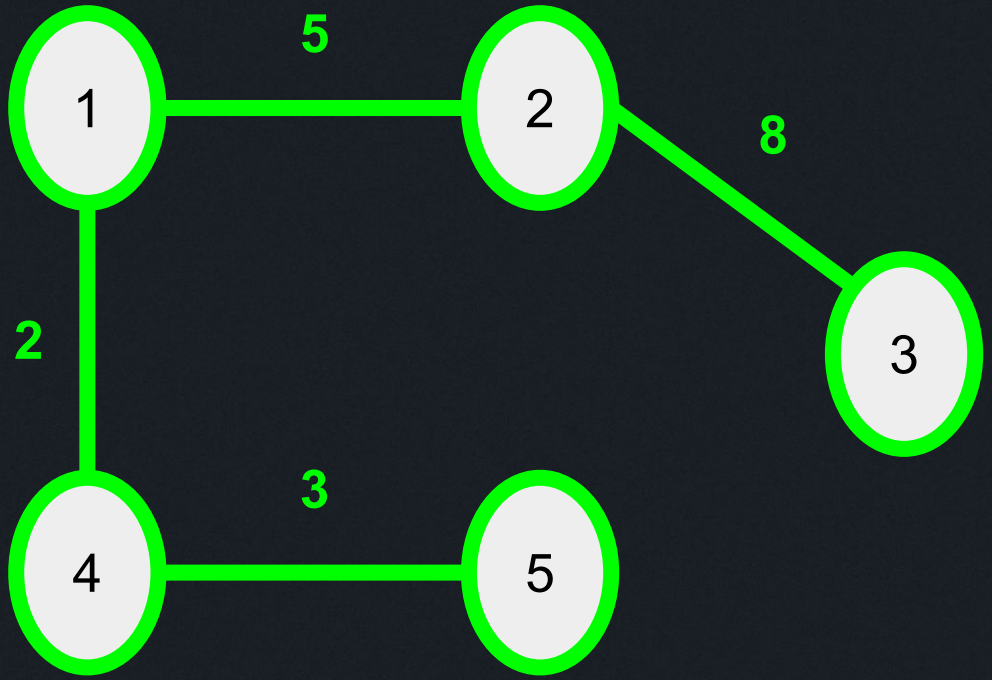
Matriz e lista adjacência



Grafo com 4 nós e 5 arestas
Arestas não ponderadas
Arestas direcionadas

Representação de Grafos

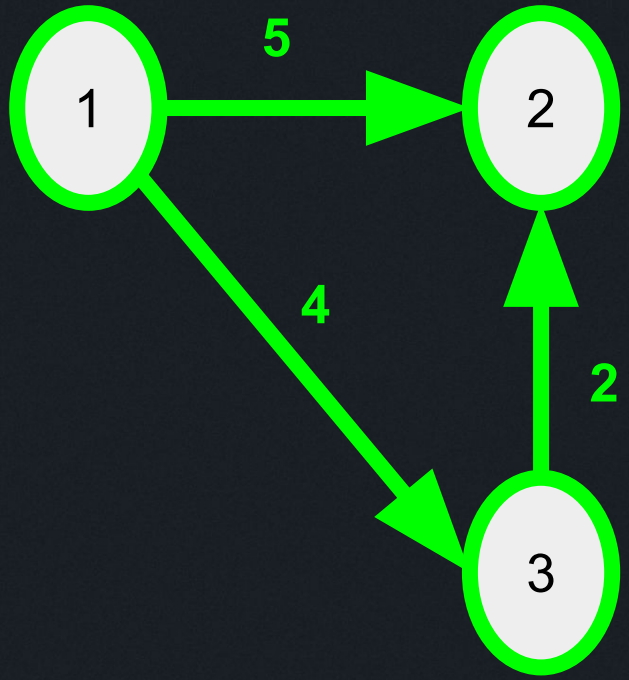
Matriz e lista adjacência



Grafo com 5 nós e 4 arestas
Arestas ponderadas
Arestas não direcionadas

Representação de Grafos

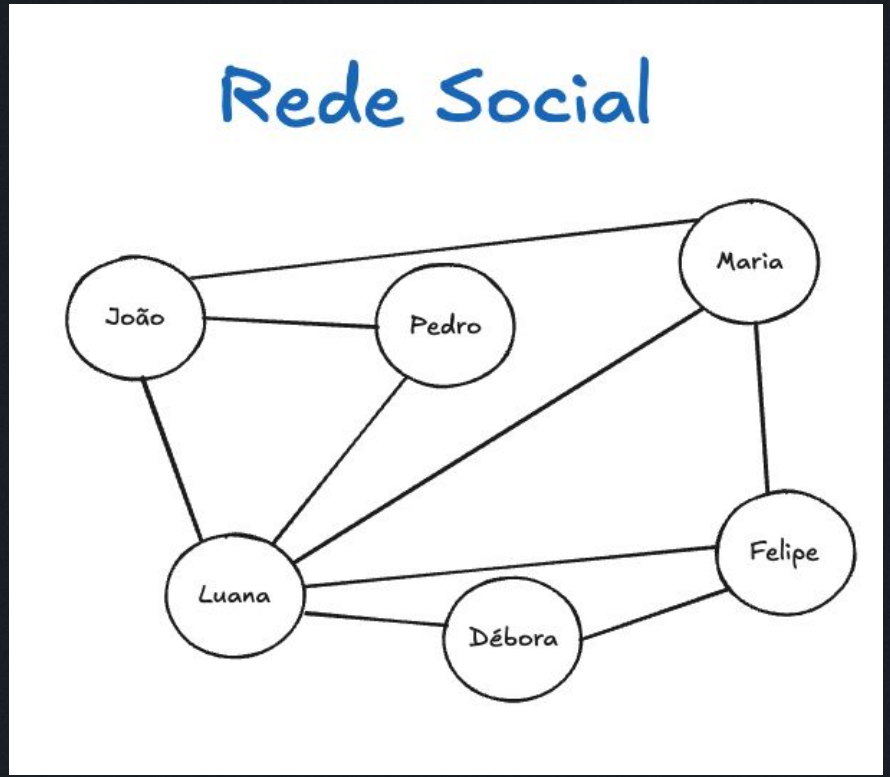
Matriz e lista adjacência



Grafo com 3 nós e 3 arestas
Arestas ponderadas
Arestas direcionadas

Representação de Grafos

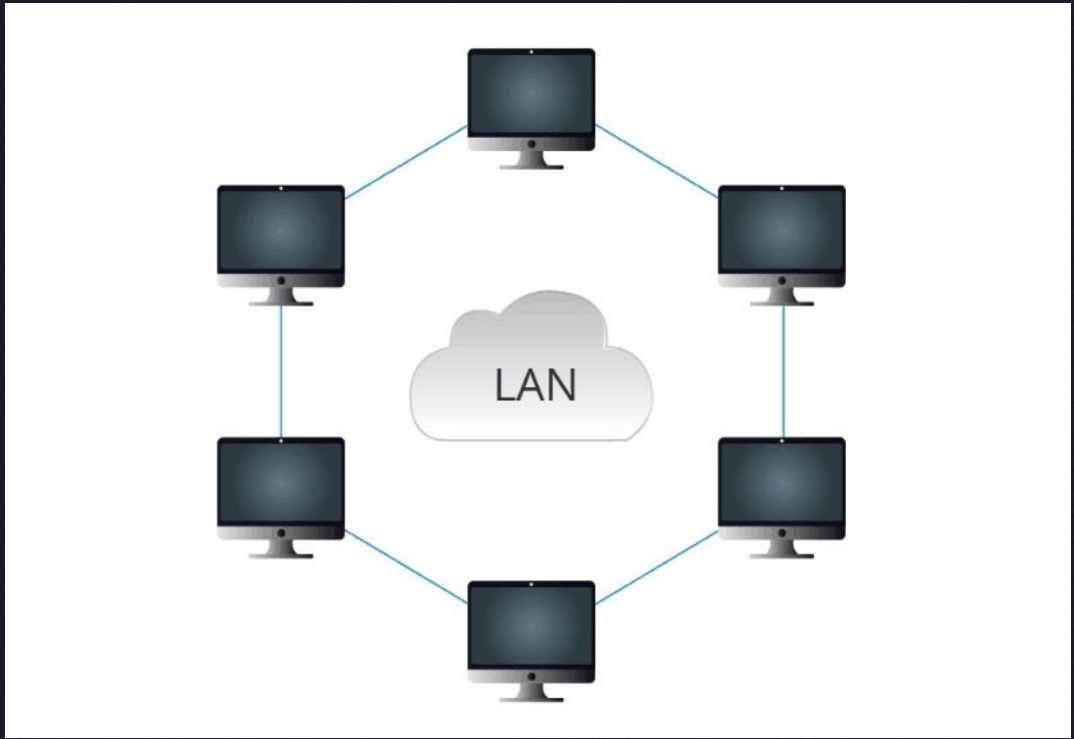
Matriz e lista adjacência



Representação de Grafos

Matriz e lista adjacência

Redes de Computadores

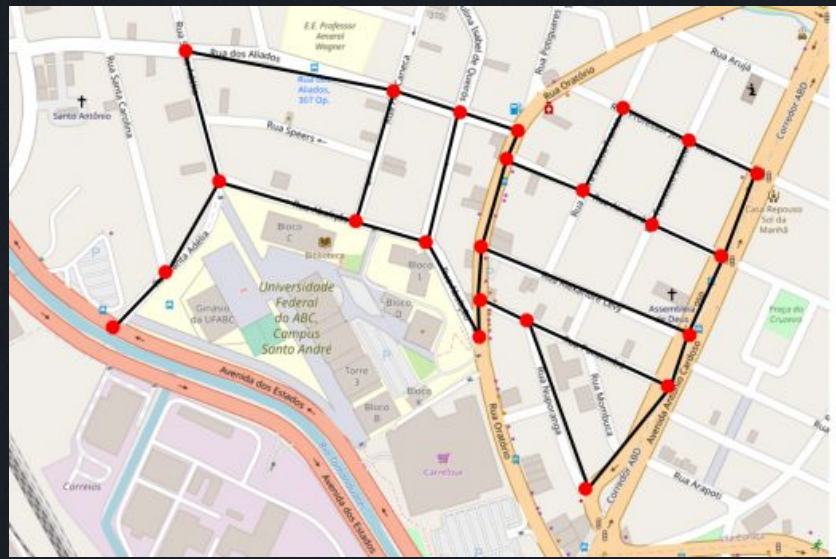


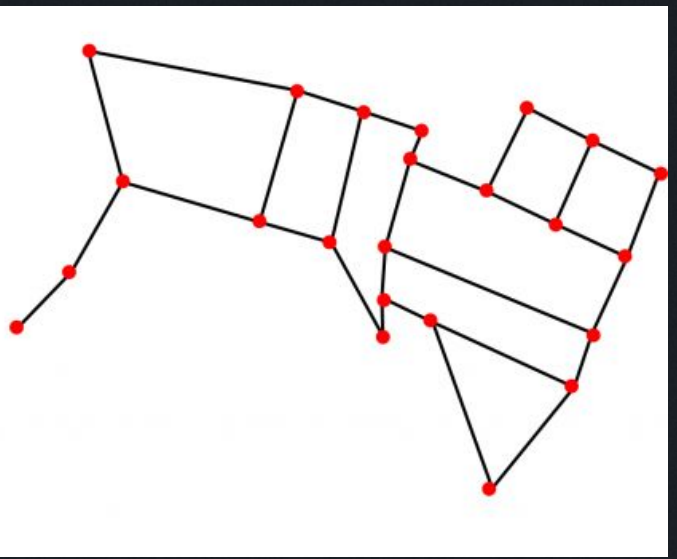
ada

Representação de Grafos

Matriz e lista adjacência

Mapa de Ruas como um Grafo

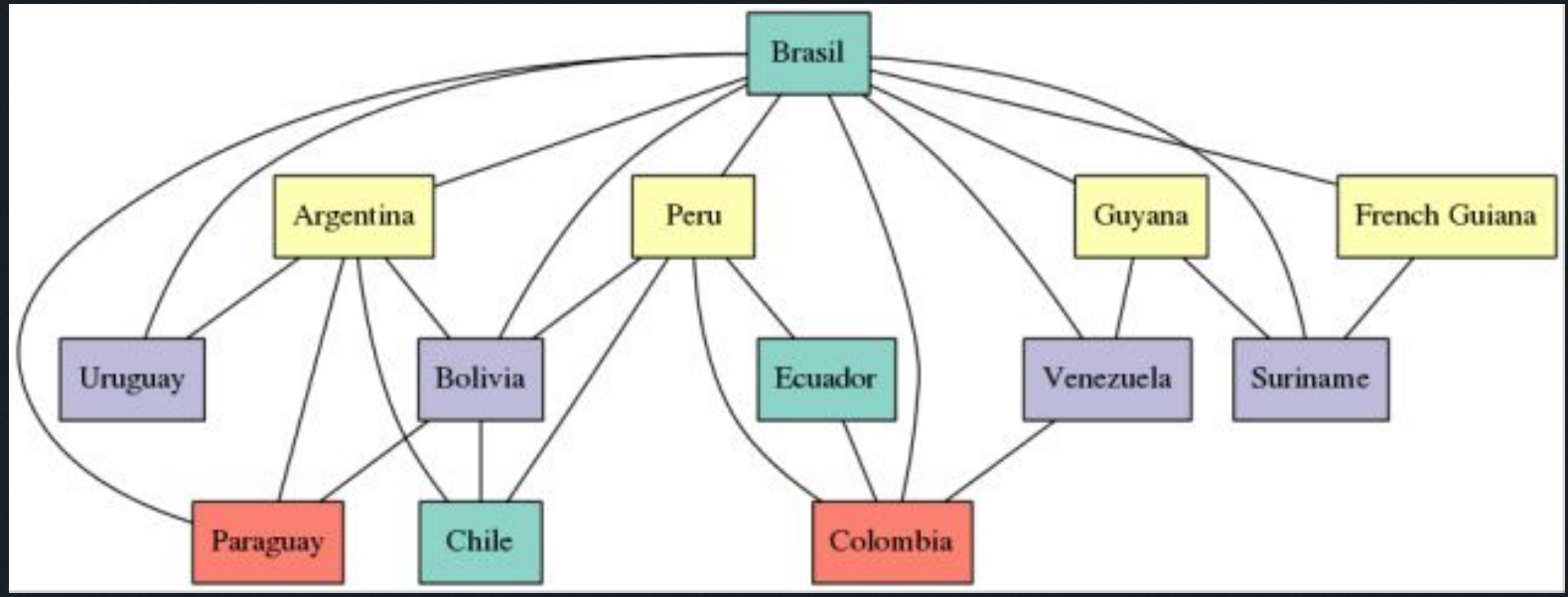




Representação de Grafos

Matriz e lista adjacência

Colorindo um Mapa Geográfico



ada

Representação de Grafos

Matriz e lista adjacência

Há duas alternativas para representar em memória um grafo:

Matriz de Adjacência

- *Para grafos não-ponderados*
  Uma matriz booleana com |V|x|V| elementos indica se há uma aresta entre qualquer par u, v (u e v são nós do grafo)

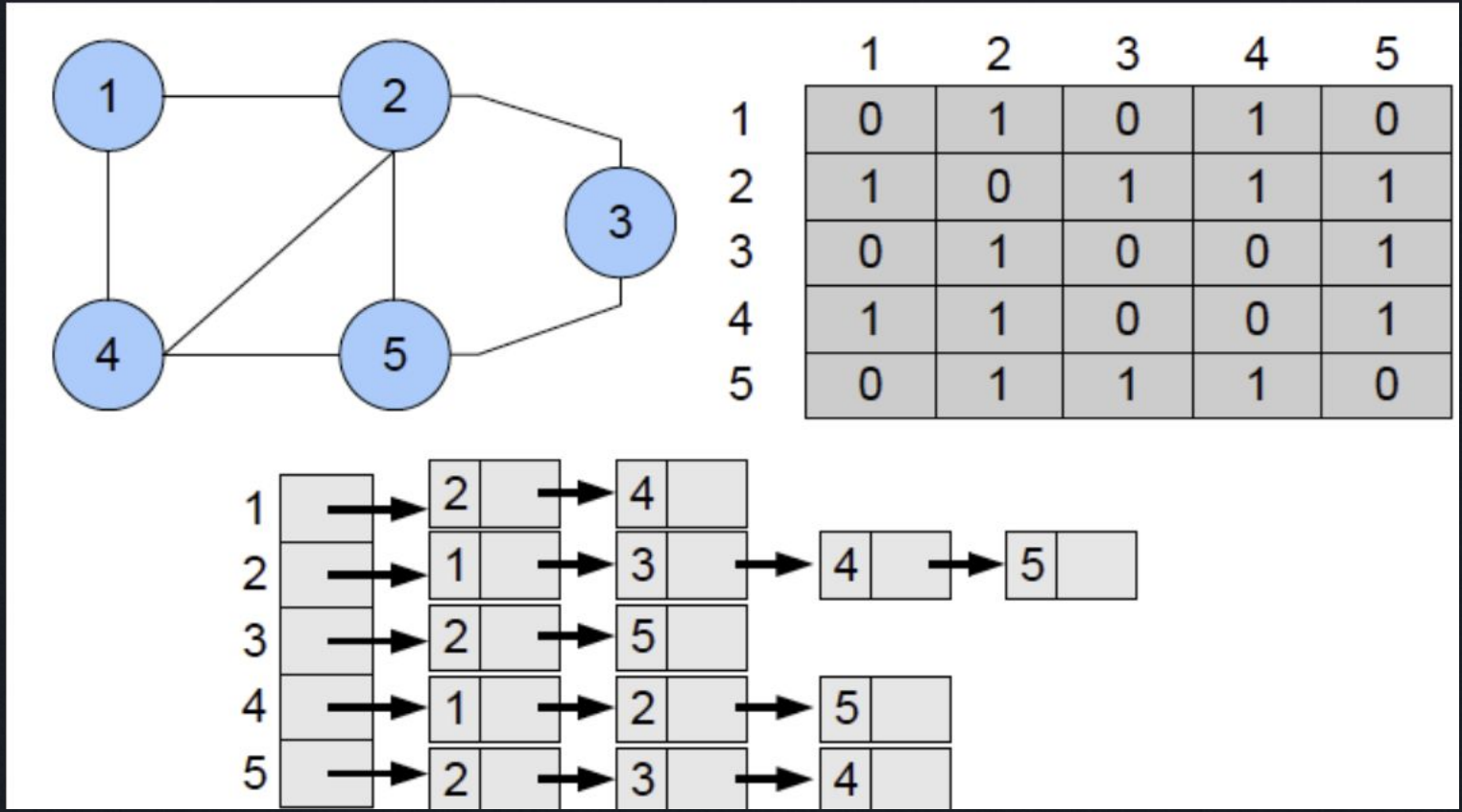
- *Para grafos ponderados*
  A matriz armazena o peso de cada aresta

Listas de Adjacência

- Para cada nó do grafo, há uma lista indicando seus vizinhos

Representação de Grafos

Matriz e lista adjacência



Matriz Adjacência

```
class Graph:
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [ [False]*V for _ in range(V)]

    def addEdge(self, u, v):
        self.adj[u][v] = True
        self.E += 1
```

Lista Adjacência

```
class Graph:
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [ [] for _ in range(V) ]

    def addEdge(self, u, v):
        self.adj[u].append(v)
        self.E += 1
```

Representação de Grafos

Matriz e lista adjacência

Lista Adjacência com Grafo Ponderado - tupla para vizinho e peso

```
class Graph:
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [ [] for _ in range(V) ]

    def addEdge(self, u, v, w):
        self.adj[u].append( (v, w) ) #saving tuple
        self.E += 1
```

Lista Adjacência com Grafo Ponderado - usando adj e weight

```
class Graph:
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [ [] for _ in range(V)]
        self.weight = [ [] for _ in range(V) ]

    def addEdge(self, u, v, w=1):
        self.adj[u].append(v)
        self.weight[u].append(w)
        self.E += 1
```

Conceitos importantes em um grafo:

Grau de saída/emissão de um nó
Quantidade de arestas que saem de um nó

Grau de entrada/recepção de um nó
Quantidade de arestas que apontam para um nó

Ciclo
Caminho de nós e arestas que retorna a um nó inicial

Prática no LeetCode

https://leetcode.com/problems/find-the-town-judge

997. Find the Town Judge

Solved

Easy | Topics | Companies

In a town, there are `n` people labeled from `1` to `n`. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties **1** and **2**.

You are given an array `trust` where `trust[i] = [a_i, b_i]` representing that the person labeled `a_i` trusts the person labeled `b_i`. If a trust relationship does not exist in `trust` array, then such a trust relationship does not exist.

Return *the label of the town judge if the town judge exists and can be identified, or return* `-1` *otherwise*.

Example 1:

```
Input: n = 2, trust = [[1,2]]
Output: 2
```

Example 2:

```
Input: n = 3, trust = [[1,3],[2,3]]
Output: 3
```

ada

Prática no LeetCode

https://leetcode.com/problems/find-the-town-judge

```
class Graph:
 def __init__(self, V):
   self.V = V
   self.E = 0
   self.inDegree = [0]*V
   self.outDegree = [0]*V
 def addEdge(self, source:int , target:int, weight):
   self.E += 1
   self.inDegree[target]+=1
   self.outDegree[source]+=1
 def townJudge(self) :
     for i in range(self.V):
         if self.inDegree[i]==self.V-1 and self.outDegree[i]==0:
             return i+1
     return -1
class Solution:
   def findJudge(self, n: int, trust: List[List[int]]) -> int:
       g = Graph(n)
       for [source, target] in trust:
           g.addEdge(source-1, target-1, 1)
       return g.townJudge()
```

Prática no LeetCode

https://leetcode.com/problems/find-center-of-star-graph

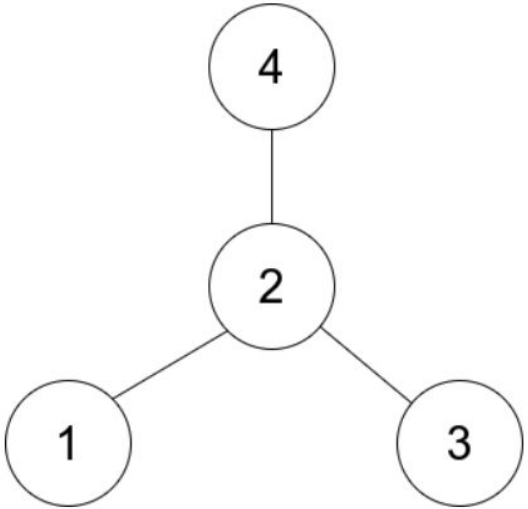
1791. Find Center of Star Graph

Easy | Topics | Companies | Hint

There is an undirected **star** graph consisting of `n` nodes labeled from `1` to `n`. A star graph is a graph where there is one **center** node and **exactly** `n - 1` edges that connect the center node with every other node.

You are given a 2D integer array `edges` where each `edges[i] = [u_i, v_i]` indicates that there is an edge between the nodes `u_i` and `v_i`. Return the center of the given star graph.

Example 1:



Prática no LeetCode

https://leetcode.com/problems/find-center-of-star-graph

```
def findCenter(self, edges: List[List[int]]) -> int:
    degree = dict()
    for source, target in edges:
        if target not in degree:
            degree[target] = 0
        if source not in degree:
            degree[source] = 0
        degree[target] += 1
        degree[source] += 1

    for key, value in degree.items():
        if value == len(degree)-1:
            return key
    return -1
```