



Módulo 4 - Aula 1

Representação de Grafos e DFS

Representação de Grafos

Representação de Grafos

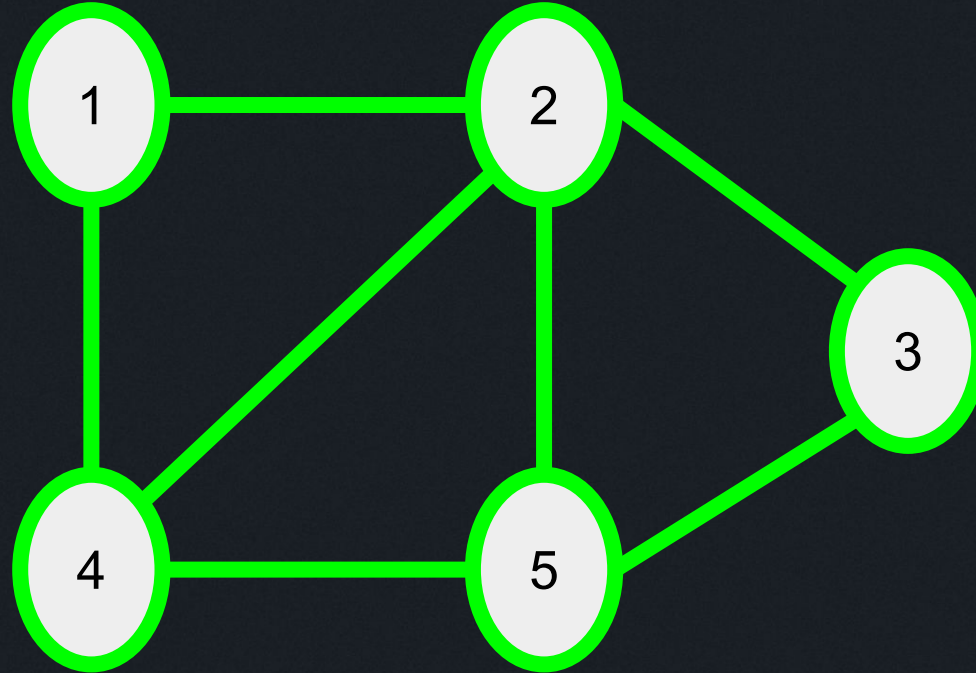
Matriz e lista adjacência

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

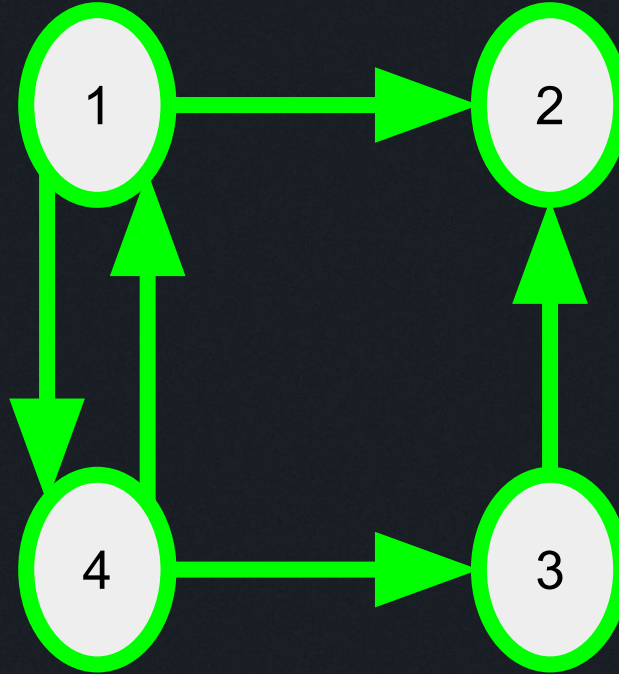
Matriz e lista adjacência



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

Representação de Grafos

Matriz e lista adjacência



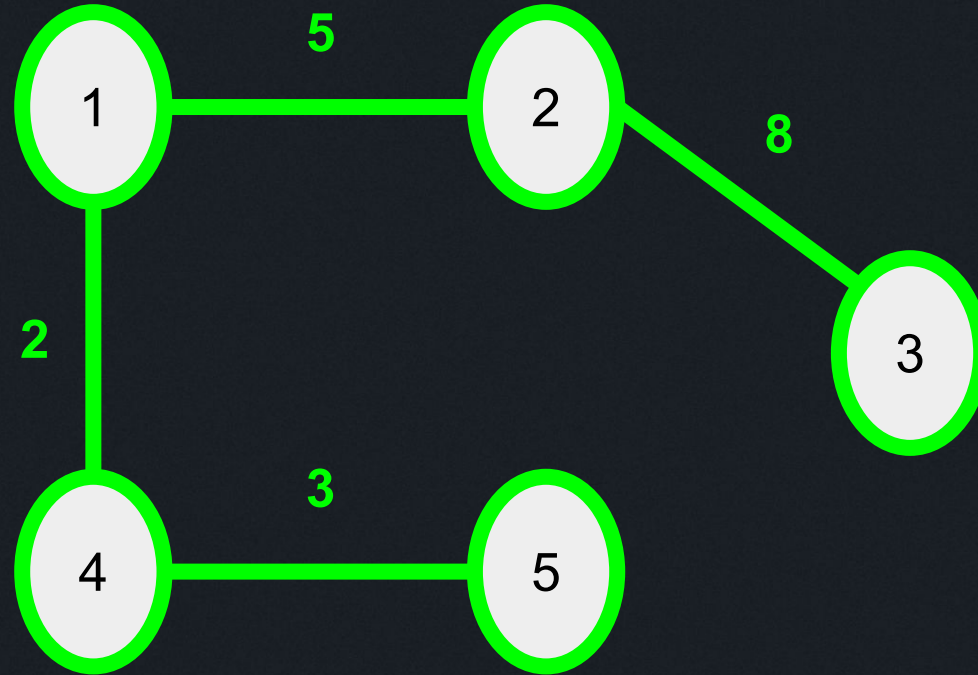
Grafo com 4 nós e 5 arestas

Arestas não ponderadas

Arestas direcionadas

Representação de Grafos

Matriz e lista adjacência



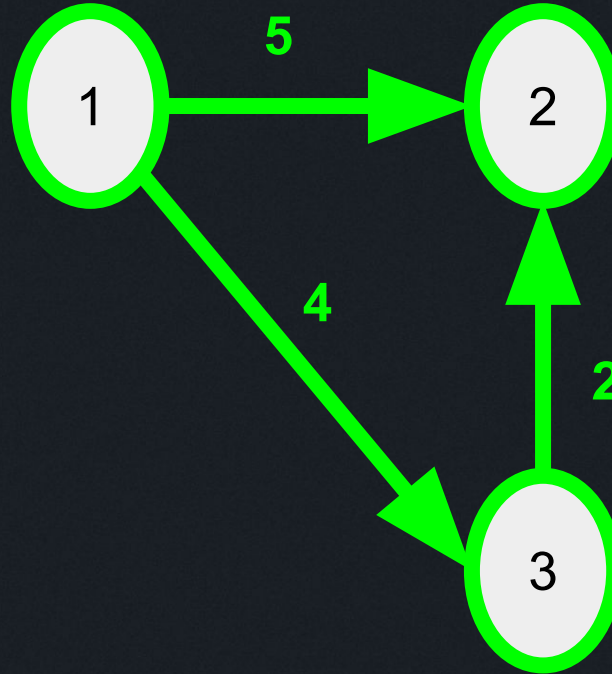
Grafo com 5 nós e 4 arestas

Arestas ponderadas

Arestas não direcionadas

Representação de Grafos

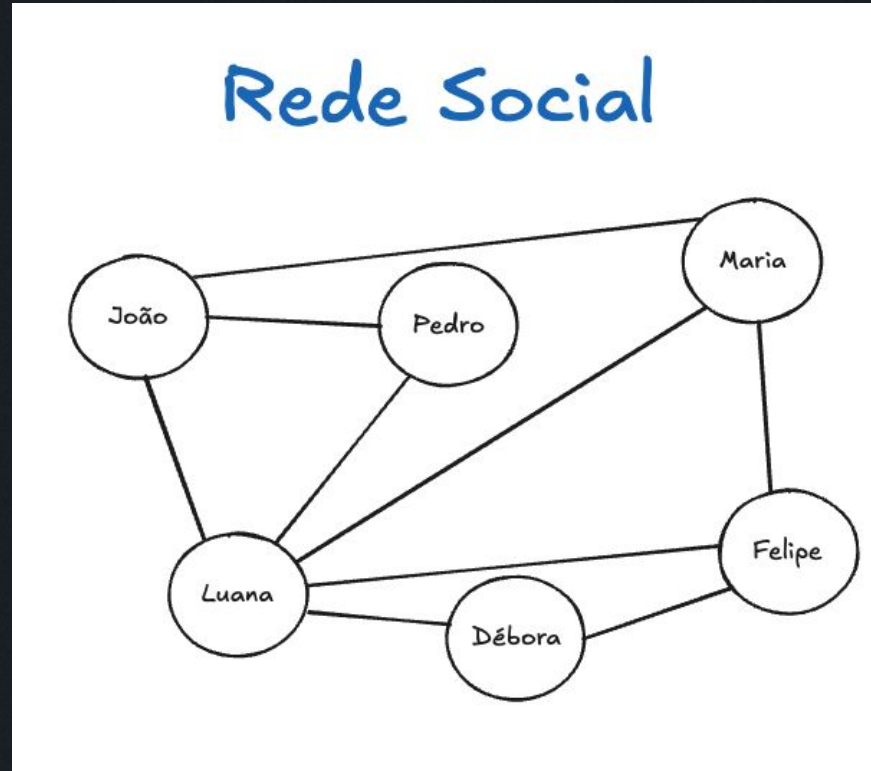
Matriz e lista adjacência



Grafo com 3 nós e 3 arestas
Arestas ponderadas
Arestas direcionadas

Representação de Grafos

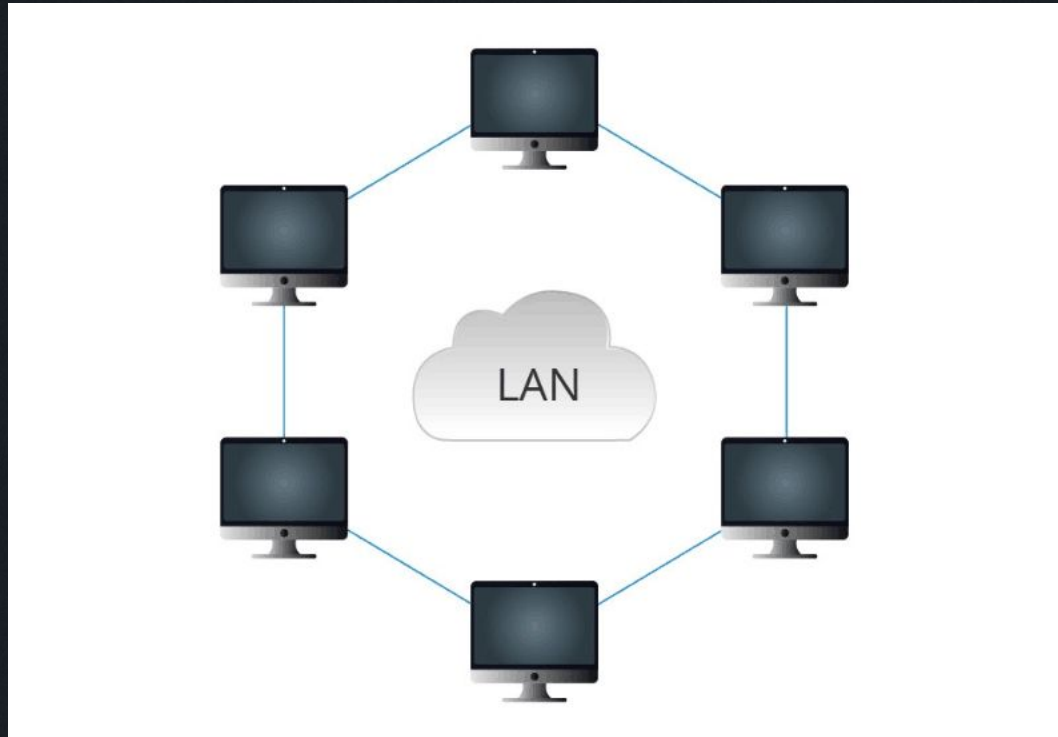
Matriz e lista adjacência



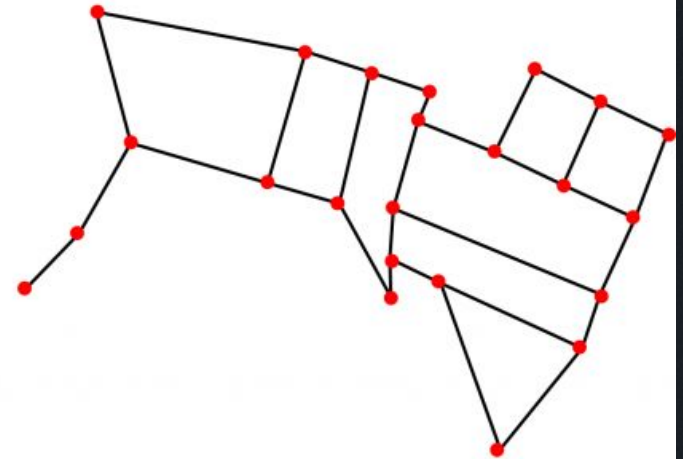
Representação de Grafos

Matriz e lista adjacência

Redes de Computadores



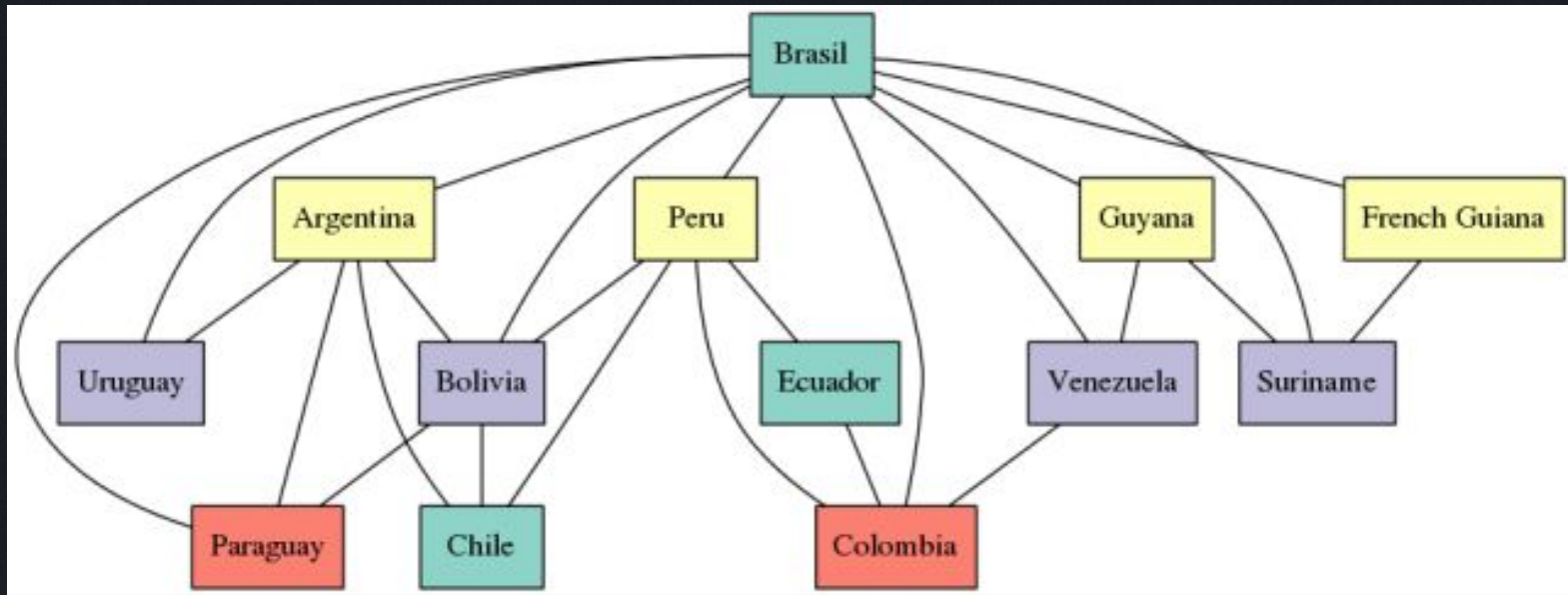
Matriz e lista adjacência

[illegible]

Representação de Grafos

Matriz e lista adjacência

Colorindo um Mapa Geográfico



Representação de Grafos

Matriz e lista adjacência

Há duas alternativas para representar em memória um grafo:

Matriz de Adjacência

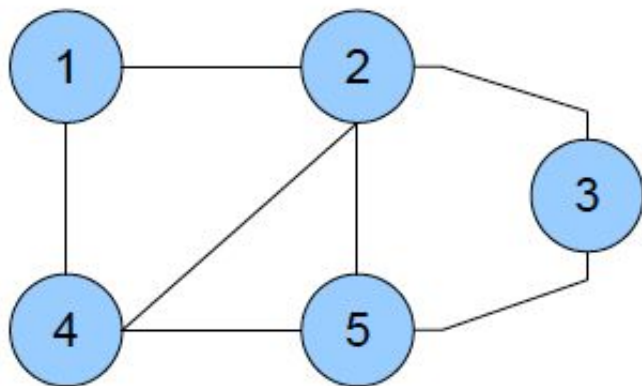
- *Para grafos não-ponderados*
Uma matriz booleana com $|V| \times |V|$ elementos indica se há uma aresta entre qualquer par u, v (u e v são nós do grafo)
- *Para grafos ponderados*
A matriz armazena o peso de cada aresta

Listas de Adjacência

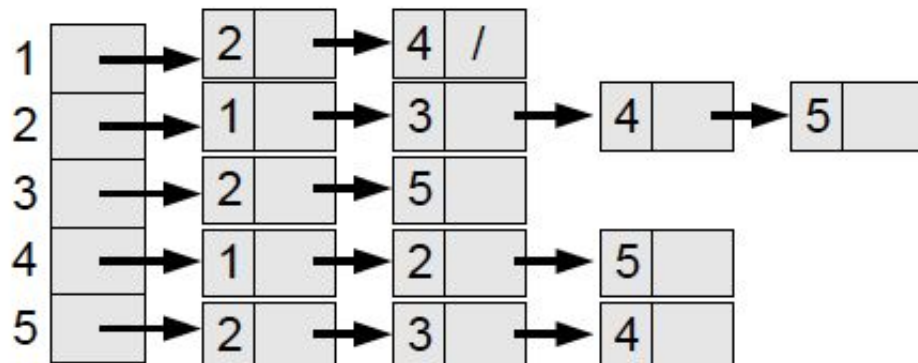
- Para cada nó do grafo, há uma lista indicando seus vizinhos

Representação de Grafos

Matriz e lista adjacência



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0



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
```

Representação de Grafos

Matriz e lista adjacência

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
```


Representação de Grafos

Matriz e lista adjacência

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
```

Representação de Grafos

Matriz e lista adjacência

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] = [ai, bi]` representing that the person labeled `ai` trusts the person labeled `bi`. 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
```

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

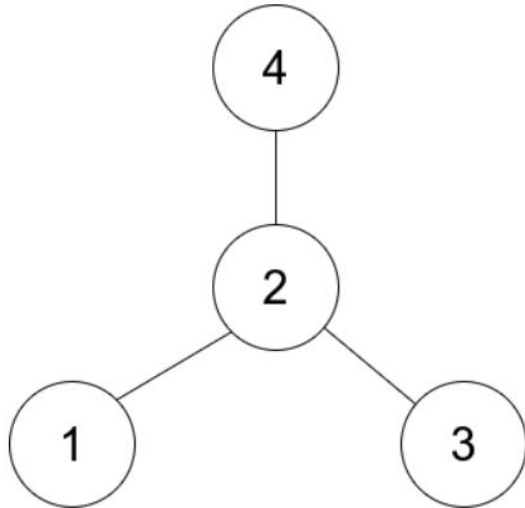
Companies

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] = [ui, vi]` indicates that there is an edge between the nodes `ui` and `vi`. 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
```



Busca em Profundidade

DFS

Busca em Profundidade

Percorre todos os vértices de um **grafo G** a partir de um **vértice de origem s** até descobrir cada vértice acessível a partir de s

Visita primeiro os vértices **mais profundos de s**, depois retorna visitando os demais vértices do caminho

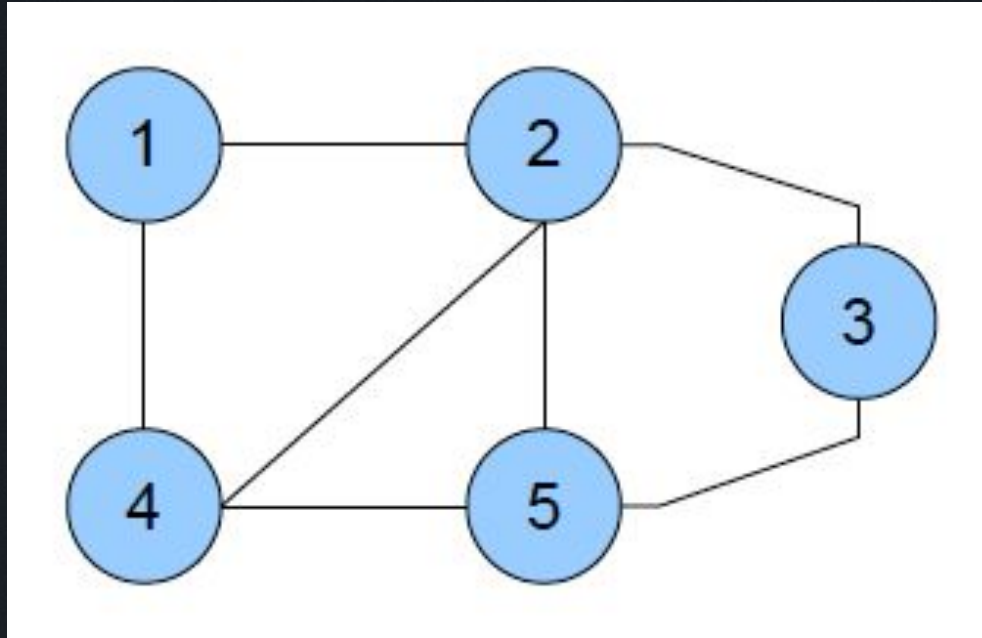
Em inglês: **DFS (depth-first search)**

Útil em entrevistas porque é a forma mais rápida de escrever um código recursivo que percorre todo um grafo a partir de um nó de origem

DFS

Busca em Profundidade

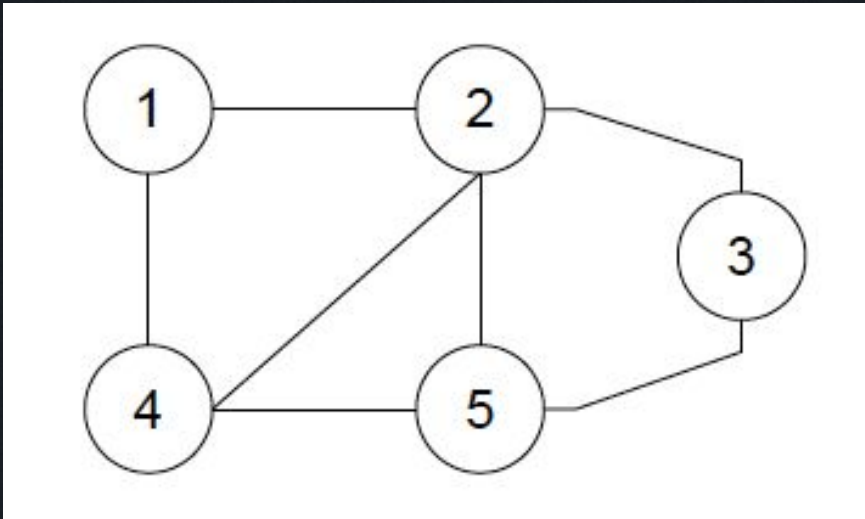
Vamos simular o DFS começando do nó 1 no grafo a seguir:



DFS

Busca em Profundidade

Inicia marcando todos os nós como não visitados (em branco na figura)
Chama inicialmente o DFS para o nó de origem 1
Todo DFS começa sempre por um nó de origem

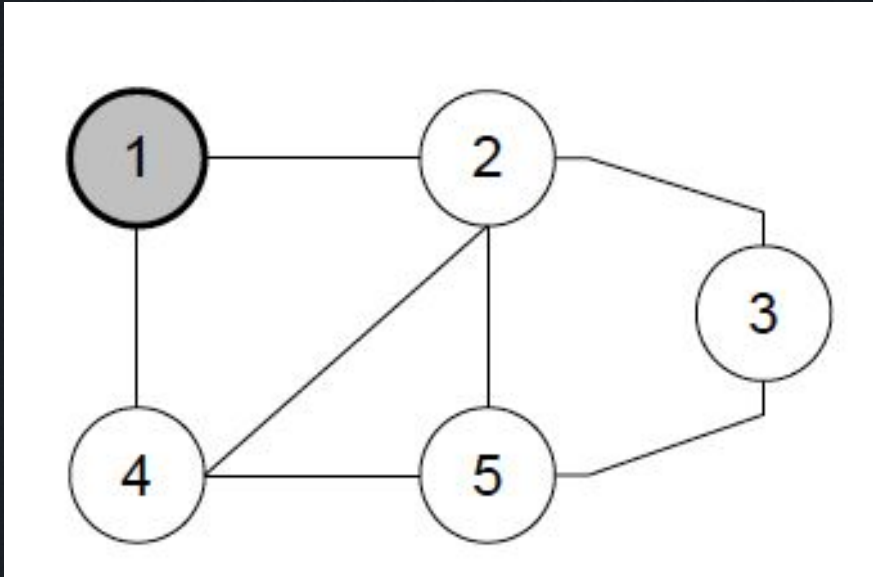


Chamadas recursivas
DFS(1)

DFS

Busca em Profundidade

DFS(1) vai marcar 1 como visitado
Depois, iterar nos vizinhos 2 e 4, chamando DFS para eles



Chamadas recursivas

DFS(1)

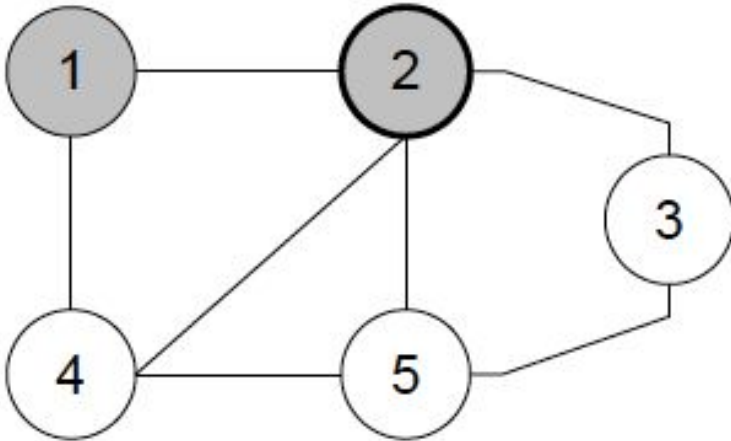
DFS(2)

DFS(4)

DFS

Busca em Profundidade

DFS(2) marca 2 como visitado
Depois, itera nos vizinhos 1, 3, 4, 5, chamando DFS para eles



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(4)

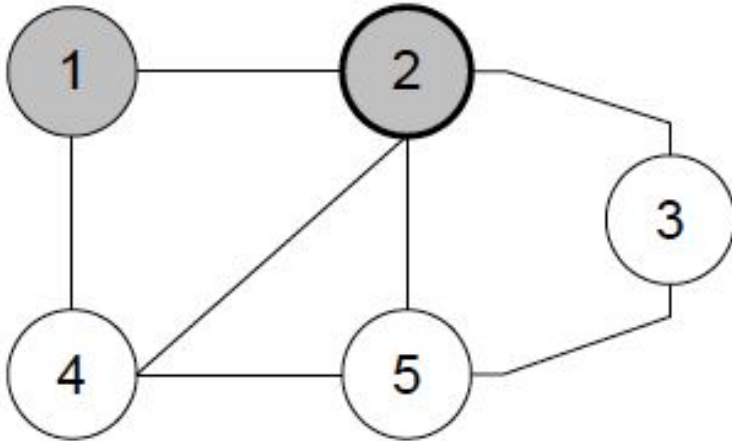
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(1) sendo chamado novamente
Dessa vez, o nó 1 já foi visitado, então não faz mais nada
Retrocede a recursão para DFS(2), que chama o próximo vizinho 3



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(4)

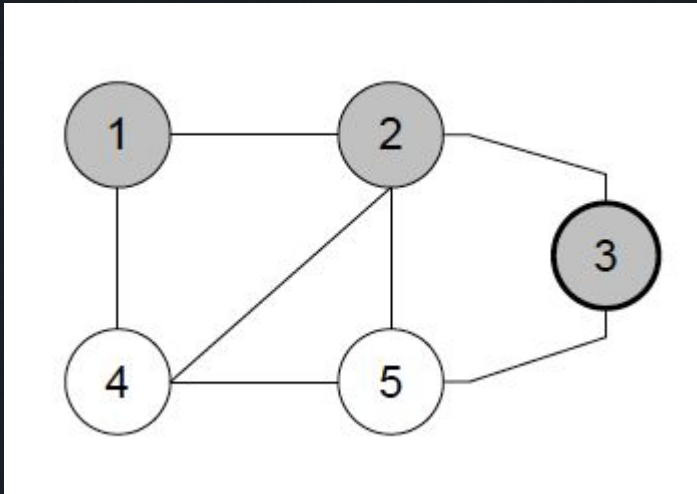
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(3) marca o nó 3 como visitado
Depois, itera sobre os vizinhos 2 e 5, chamando DFS para eles



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(4)

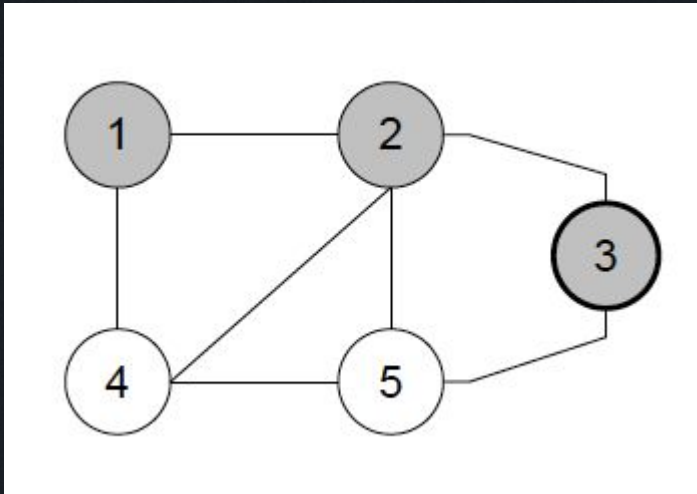
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(2) sendo chamado novamente
Dessa vez, o nó 2 já foi visitado, então não faz mais nada
Retrocede a recursão para DFS(3), que chama o próximo vizinho 5



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(4)

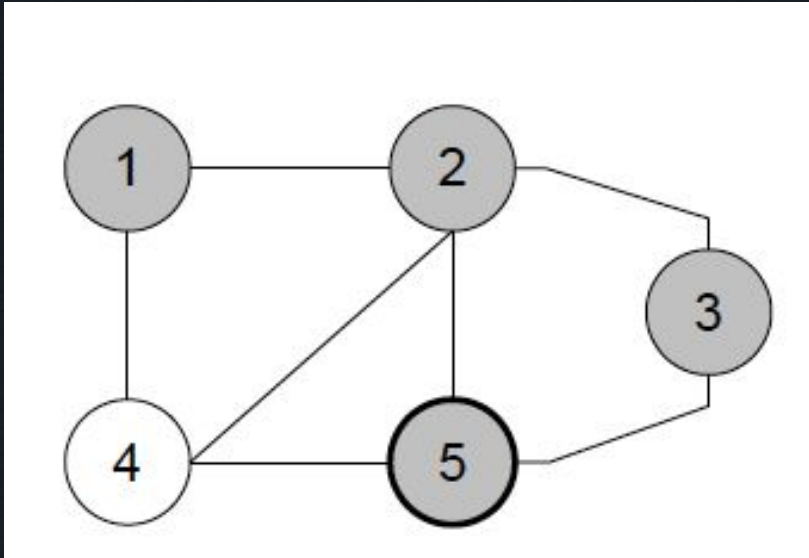
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(5) marca o nó 5 como visitado
Depois, itera sobre os vizinhos 2, 3, 4, chamando DFS para eles



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(2)

DFS(3)

DFS(4)

DFS(4)

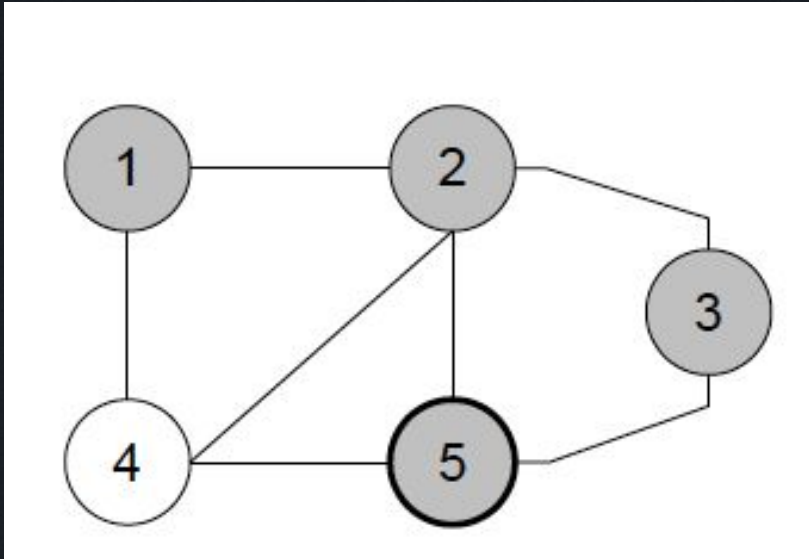
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(2) sendo chamado novamente
Dessa vez, o nó 2 já foi visitado, então não faz mais nada
Retrocede a recursão para DFS(5), que chama o próximo vizinho 3



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(2)

DFS(3)

DFS(4)

DFS(4)

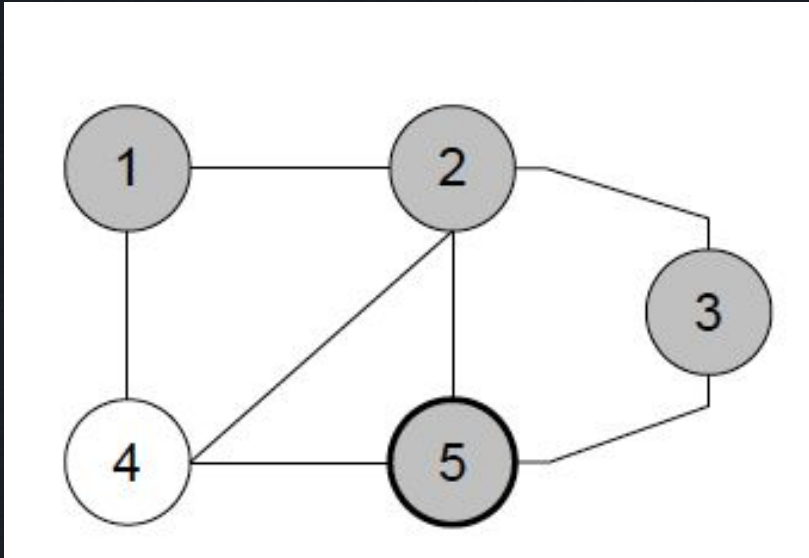
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(3) sendo chamado novamente
Dessa vez, o nó 3 já foi visitado, então não faz mais nada
Retrocede a recursão para DFS(5), que chama o próximo vizinho 4



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(2)

DFS(3)

DFS(4)

DFS(4)

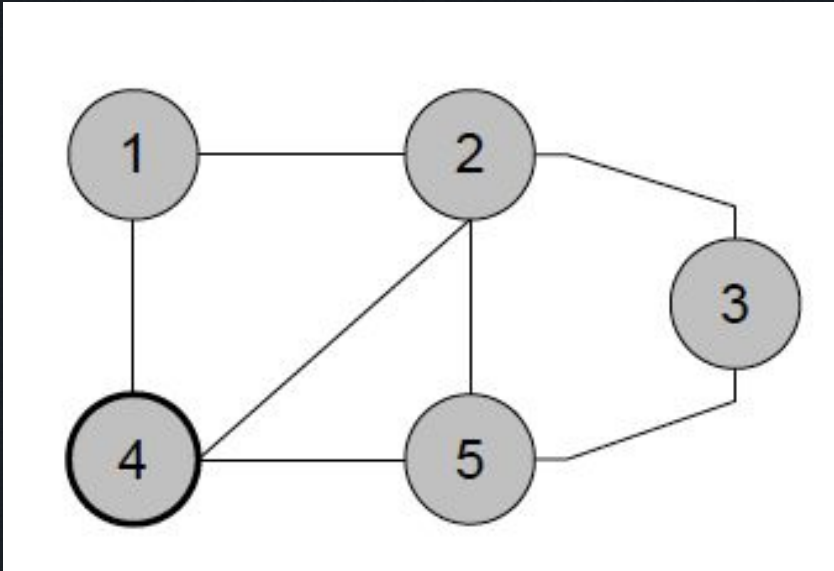
DFS(5)

DFS(4)

DFS

Busca em Profundidade

DFS(4) marca o nó 4 como visitado
Depois, itera sobre os vizinhos 1, 2, 5, chamando DFS para eles



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(2)

DFS(3)

DFS(4)

DFS(1)

DFS(2)

DFS(5)

DFS(4)

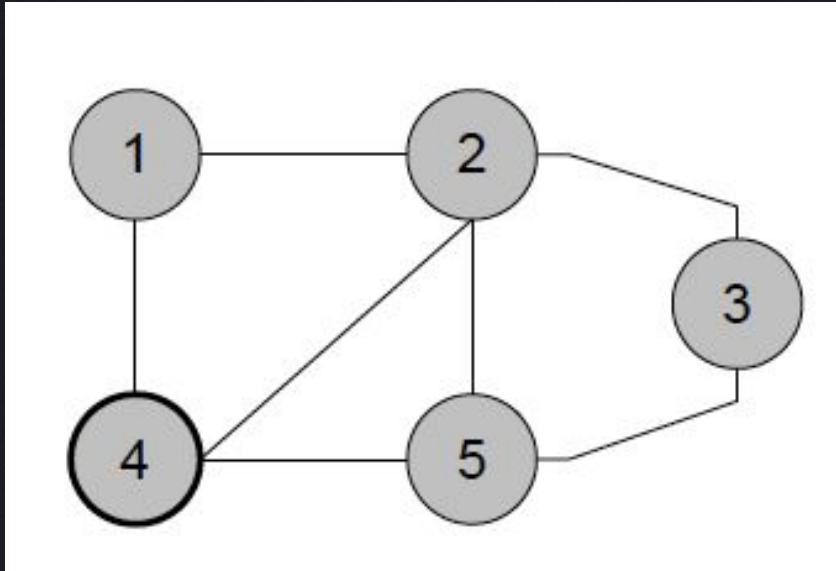
DFS(5)

DFS(4)

DFS

Busca em Profundidade

Não temos mais nós não-visitados, então sempre vai cair no caso base
As chamadas abaixo de DFS(4) **(em negrito)** ainda serão feitas



Chamadas recursivas

DFS(1)

DFS(2)

DFS(1)

DFS(3)

DFS(2)

DFS(5)

DFS(2)

DFS(3)

DFS(4)

DFS(1)

DFS(2)

DFS(5)

DFS(4)

DFS(5)

DFS(4)

Vamos levar em conta essa implementação de grafo já discutida:

```
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
```

DFS

Busca em Profundidade

DFS básico, que só visita os nós sem fazer nada

```
class Graph:

    def dfs(self, node, visited = set()):
        if node not in visited:
            visited.add(node)
            for neigh in self.adj[node]:
                self.dfs(neigh, visited)
```

DFS

Busca em Profundidade

DFS retornando uma lista na ordem que foram visitados os nós

```
class Graph:
```

```
    def dfsVisitedList(self, node, visited = set()):
```

```
        if node in visited:
```

```
            return []
```

```
        visited.add(node)
```

```
        listOfVisitedNodes = [node]
```

```
        for neigh in self.adj[node]:
```

```
            listOfVisitedNodes += self.dfsVisitedList(neigh, visited)
```

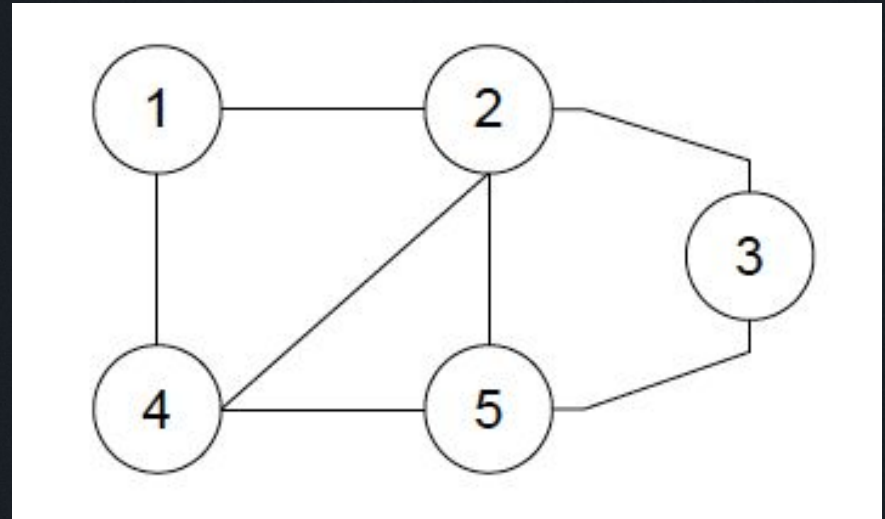
```
        return listOfVisitedNodes
```

DFS

Busca em Profundidade

Testando para nosso grafo

```
g = Graph(6) # ignoring the 0 node
g.addEdge(1, 2)
g.addEdge(2, 1)
g.addEdge(1, 4)
g.addEdge(4, 1)
g.addEdge(2, 3)
g.addEdge(3, 2)
g.addEdge(2, 4)
g.addEdge(4, 2)
g.addEdge(2, 5)
g.addEdge(5, 2)
g.addEdge(3, 5)
g.addEdge(5, 3)
g.addEdge(4, 5)
g.addEdge(5, 4)
print( g.dfsVisitedList(1) )
```



Saída [1, 2, 3, 5, 4]

Prática no LeetCode

<https://leetcode.com/problems/is-graph-bipartite>

785. Is Graph Bipartite?

Solved 

Medium  Topics  Companies

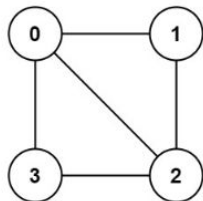
There is an **undirected** graph with n nodes, where each node is numbered between 0 and $n - 1$. You are given a 2D array `graph`, where `graph[u]` is an array of nodes that node `u` is adjacent to. More formally, for each `v` in `graph[u]`, there is an undirected edge between node `u` and node `v`. The graph has the following properties:

- There are no self-edges (`graph[u]` does not contain `u`).
- There are no parallel edges (`graph[u]` does not contain duplicate values).
- If `v` is in `graph[u]`, then `u` is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes `u` and `v` such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets `A` and `B` such that **every** edge in the graph connects a node in set `A` and a node in set `B`.

Return `true` if and only if it is **bipartite**.

Example 1:

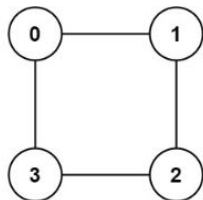


Input: `graph = [[1,2,3],[0,2],[0,1,3],[0,2]]`

Output: `false`

Explanation: There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

Example 2:



Input: `graph = [[1,3],[0,2],[1,3],[0,2]]`

Output: `true`

Prática no LeetCode

<https://leetcode.com/problems/is-graph-bipartite>



```
def isBipartite(self, graph: List[List[int]]) → bool:
    color = [-1]*len(graph)
    def dfs(node, setColor, otherColor):
        if color[node]≠-1:
            return color[node]==setColor
        color[node] = setColor
        for neigh in graph[node]:
            if color[neigh]==-1 and not dfs(neigh, otherColor, setColor):
                return False
            elif color[neigh]≠otherColor:
                return False
        return True
    for node in range(len(graph)):
        if color[node]==-1 and not dfs(node, 0, 1):
            return False
    return True
```

Prática no LeetCode

<https://leetcode.com/problems/number-of-islands/description>

200. Number of Islands

Solved 

Medium  Topics  Companies

Given an $m \times n$ 2D binary grid `grid` which represents a map of '1' s (land) and '0' s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is '0' or '1'.

Prática no LeetCode

<https://leetcode.com/problems/number-of-islands/description>



```
def dfs(self, grid, lin, col):
    if grid[lin][col]=='1':
        grid[lin][col] = '*'
        if lin-1>=0:
            self.dfs(grid, lin-1, col)
        if lin+1<len(grid):
            self.dfs(grid, lin+1, col)
        if col-1>=0:
            self.dfs(grid, lin, col-1)
        if col+1<len(grid[lin]):
            self.dfs(grid, lin, col+1)
def numIslands(self, grid: List[List[str]]) -> int:
    islands = 0
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            if grid[i][j]=='1':
                islands += 1
                self.dfs(grid, i, j)
    return islands
```


Prática no LeetCode

<https://leetcode.com/problems/island-perimeter/>

463. Island Perimeter

Solved 

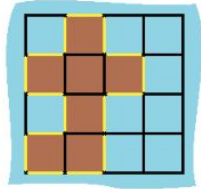
Easy  Topics  Companies

You are given `row x col grid` representing a map where `grid[i][j] = 1` represents land and `grid[i][j] = 0` represents water.

Grid cells are connected **horizontally/vertically** (not diagonally). The `grid` is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Example 1:



Input: `grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]`

Output: 16

Explanation: The perimeter is the 16 yellow stripes in the image above.

Example 2:

Input: `grid = [[1]]`

Output: 4

Example 3:

Input: `grid = [[1,0]]`

Output: 4

Constraints:

- `row == grid.length`
- `col == grid[i].length`
- `1 <= row, col <= 100`
- `grid[i][j]` is `0` or `1`.
- There is exactly one island in `grid`.

Prática no LeetCode

<https://leetcode.com/problems/island-perimeter/>

```
def dfs(self, grid, row, col):
    perimeter = 0
    grid[row][col] = 2
    #.          up          down          left          right
    moves = [ (row-1, col), (row+1, col), (row, col-1), (row, col+1) ]
    for move in moves:
        if move[0]<0 or move[0]>=len(grid): # border
            perimeter += 1
        elif move[1]<0 or move[1]>=len(grid[0]): # border
            perimeter += 1
        elif grid[move[0]][move[1]]==0:
            perimeter += 1
        elif grid[move[0]][move[1]]==1: #land
            perimeter += self.dfs(grid, move[0], move[1])
    return perimeter

def islandPerimeter(self, grid: List[List[int]]) -> int:
    for i in range( len(grid) ):
        for j in range( len(grid[i]) ):
            if grid[i][j]==1:
                return self.dfs(grid, i, j)
```

Obrigada