Módulo 4 - Aula 3

BFS

BFS

BFS

Busca em Largura

- 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 próximos de s
  - Começa por s
  - Depois, os vizinhos de s
  - Depois, os vizinhos dos vizinhos de s (2 de distância)
  - Depois, quem está a 3 de distância e assim por diante

- Em inglês: BFS (breadth first search)

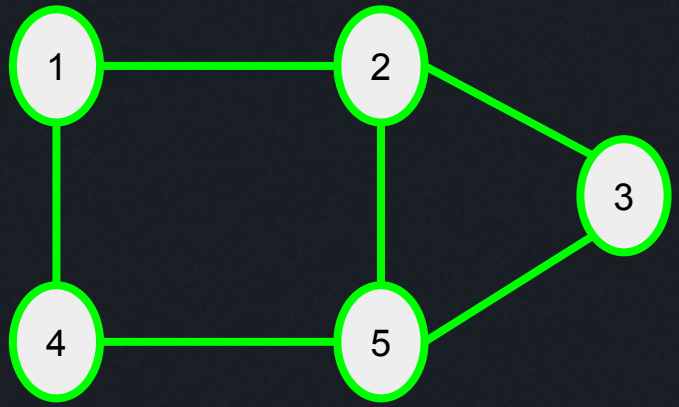
- Útil para calcular o menor número de arestas entre nós de um grafo

- Em um grafo não ponderado, o menor número de arestas também é o menor caminho entre dois nós

BFS

Busca em Largura

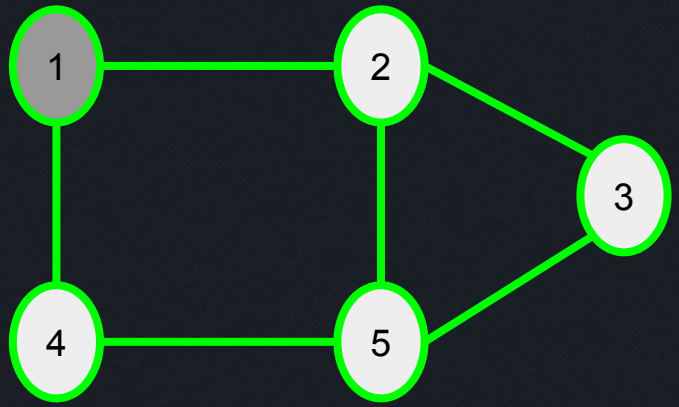
Simulando o BFS começando do **nó de origem 1**
Todos os nós estão inicialmente como não visitados (brancos na figura)



BFS

Busca em Largura

Inicia marcando o **nó de origem 1** como visitado (cinza na figura)
Insere inicialmente o **nó de origem 1** na fila

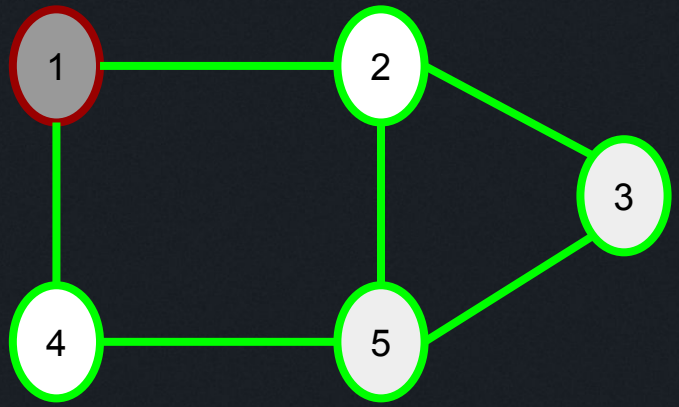


FILA: 1

BFS

Busca em Largura

Loop principal remove o **nó de trabalho 1** (em vermelho) da fila

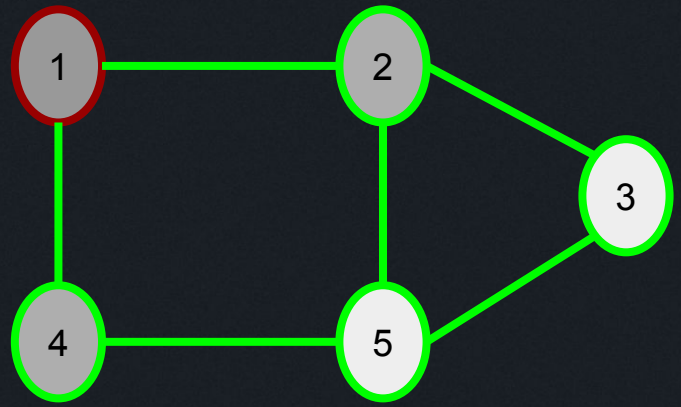


FILA:

BFS

Busca em Largura

Itera sobre os vizinhos do **nó de trabalho 1**
Os **vizinhos não-visitados 2 e 4** são marcados e inseridos na fila

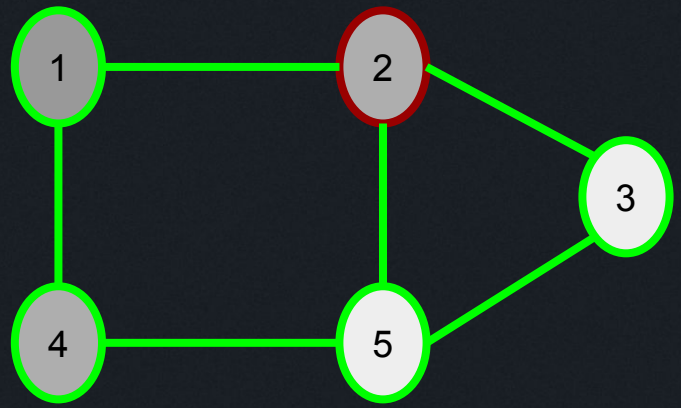


FILA: 2, 4

BFS

Busca em Largura

Loop principal, retirando o **nó de trabalho 2** (em vermelho) da fila

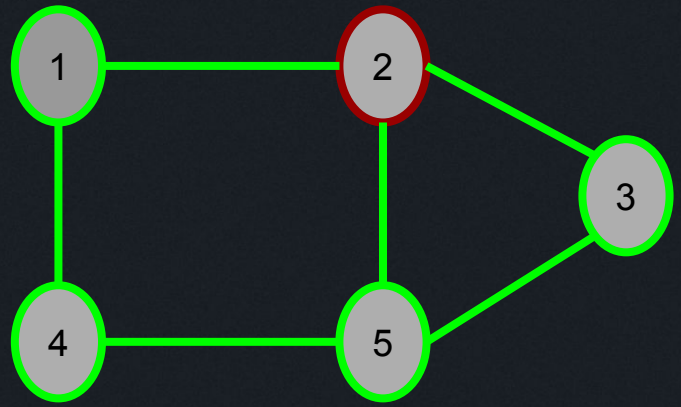


FILA: 4

BFS

Busca em Largura

Itera sobre os vizinhos do **nó de trabalho 2**
Os **vizinhos não-visitados 3 e 5** são marcados e inseridos na fila

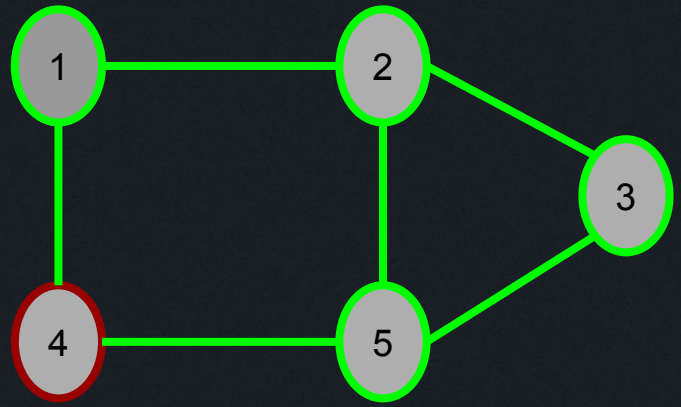


FILA: 4, 3, 5

BFS

Busca em Largura

Loop principal, retirando o **nó de trabalho 4** (em vermelho) da fila
Não há vizinhos não-visitados

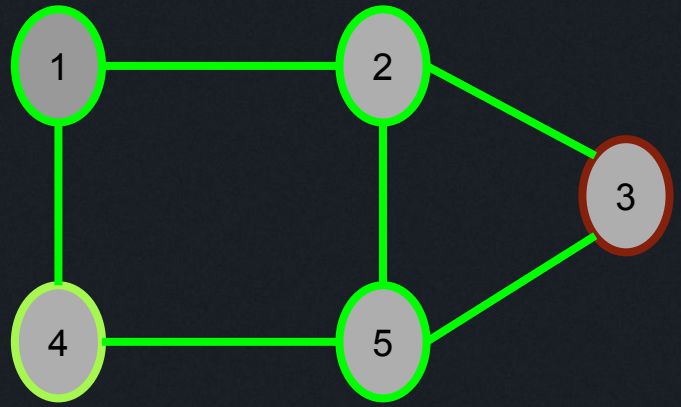


FILA: 3, 5

BFS

Busca em Largura

Loop principal, retirando o **nó de trabalho 3** (em vermelho) da fila
Não há vizinhos não-visitados

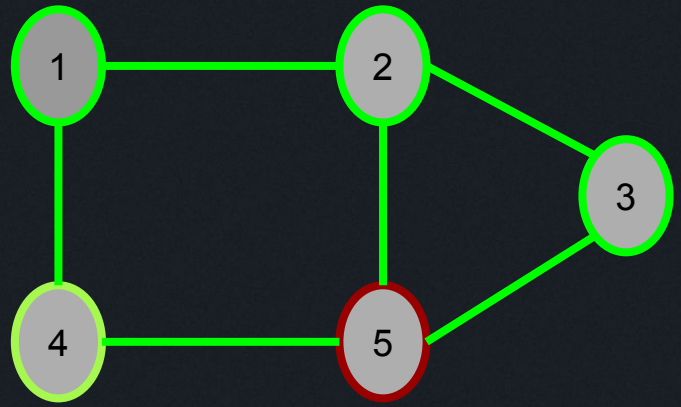


FILA: 5

BFS

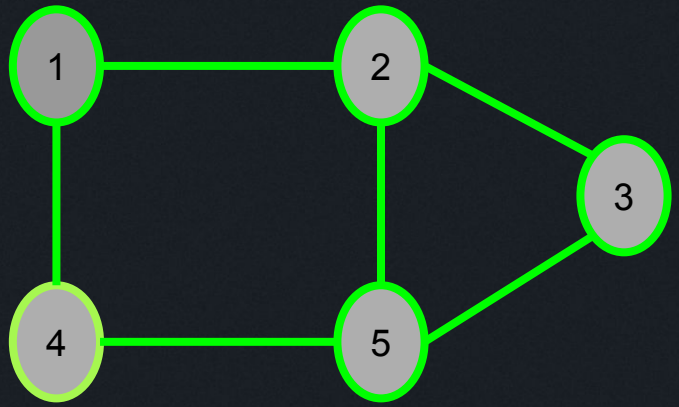
Busca em Largura

Loop principal, retirando o **nó de trabalho 5** (em vermelho) da fila
Não há vizinhos não-visitados



FILA:

O algoritmo pára porque a fila ficou vazia



FILA:
ORDEM DE VISITA DOS NÓS: 1, 2, 4, 3, 5

BFS

Busca em Largura

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
```

BFS

Busca em Largura

BFS que apenas percorre o grafo

```
class Graph:

    def bfs(self, node):
        visited = set()
        visited.add(node) #starting node is visited
        q = deque()
        q.append(node) #starting node in queue
        while q:
            workingNode = q.popleft() #working node
            for neigh in self.adj[workingNode]:
                if neigh not in visited:
                    visited.add(neigh)
                    q.append(neigh)
```

BFS

Busca em Largura

BFS que retorna a lista de nós na ordem que foram visitados
As linhas acrescentadas estão destacadas em amarelo

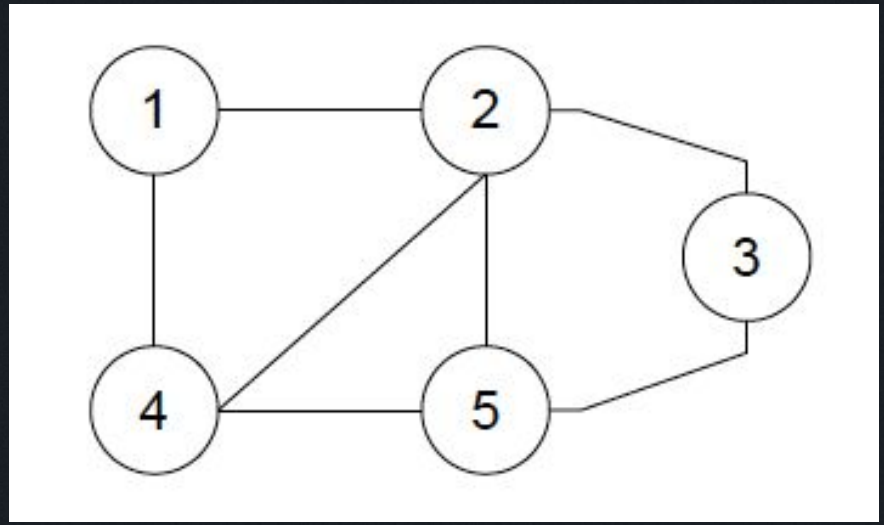
```
class Graph:
    def bfsVisitedList(self, node):
        visited = set()
        visited.add(node)
        q = deque()
        q.append(node)
        listOfVisitedNodes = []
        while q:
            workingNode = q.popleft()
            listOfVisitedNodes.append(workingNode)
            for neigh in self.adj[workingNode]:
                if neigh not in visited:
                    visited.add(neigh)
                    q.append(neigh)
        return listOfVisitedNodes
```

BFS

Busca em Largura

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.bfsVisitedList(1) )
```



Saída [1, 2, 4, 3, 5]

ada

Prática no LeetCode

https://leetcode.com/problems/keys-and-rooms/

Medium | Topics | Companies

There are `n` rooms labeled from `0` to `n - 1` and all the rooms are locked except for room `0`. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room `i`, return `true` *if you can visit* ***all*** *the rooms, or* `false` *otherwise.*

Example 1:

```
Input: rooms = [[1],[2],[3],[]]
Output: true
Explanation:
We visit room 0 and pick up key 1.
We then visit room 1 and pick up key 2.
We then visit room 2 and pick up key 3.
We then visit room 3.
Since we were able to visit every room, we return true.
```

Example 2:

```
Input: rooms = [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can not enter room number 2 since the only key that unlocks it is in that room.
```

Constraints:

- `n == rooms.length`
- `2 <= n <= 1000`
- `0 <= rooms[i].length <= 1000`
- `1 <= sum(rooms[i].length) <= 3000`
- `0 <= rooms[i][j] < n`
- All the values of `rooms[i]` are **unique**.

Prática no LeetCode

https://leetcode.com/problems/keys-and-rooms/

```python
from collections import deque



class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        q = deque([0])
        visited = set()
        visited.add(0)
        while q:
            room = q.popleft()
            for neigh in rooms[room]:
                if neigh not in visited:
                    visited.add(neigh)
                    q.append(neigh)
        return len(visited) == len(rooms) #did I visited all rooms?
```

Prática no LeetCode

https://leetcode.com/problems/coin-change

Medium | Topics | Companies

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Example 3:

```
Input: coins = [1], amount = 0
Output: 0
```

Constraints:

- `1 <= coins.length <= 12`
- `1 <= coins[i] <= 2^31 - 1`
- `0 <= amount <= 10^4`

Prática no LeetCode

https://leetcode.com/problems/coin-change

```python
from collections import deque

    def coinChange(self, coins: List[int], amount: int) -> int:
        visited = set()
        visited.add(0)
        q = deque()
        q.append( (0, 0) ) #0 amount, 0 totalCoins
        while q:
            atual, totalCoins = q.popleft()
            if atual == amount:
                return totalCoins
            for coin in coins:
                if atual+coin <= amount and not atual+coin in visited:
                    q.append( (atual+coin, totalCoins+1) )
                    visited.add(atual+coin)
        return -1
```

Prática no LeetCode

https://leetcode.com/problems/rotting-oranges

Medium | Topics | Companies

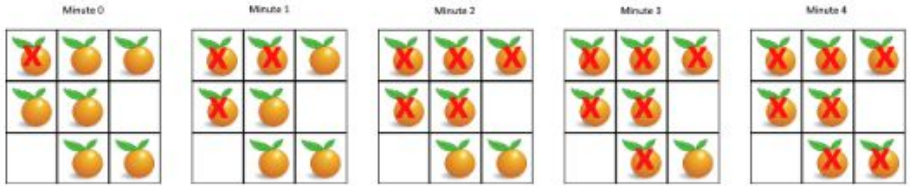
You are given an `m x n` `grid` where each cell can have one of three values:

- `0` representing an empty cell,
- `1` representing a fresh orange, or
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return* `-1`.

Example 1:




```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

Example 2:

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is
never rotten, because rotting only happens 4-directionally.
```

Example 3:

```
Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the
answer is just 0.
```

ada

Prática no LeetCode

https://leetcode.com/problems/rotting-oranges

```python
from collections import deque

class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        q = deque()
        freshOranges = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j]==2:
                    q.append( (i,j,0) )
                elif grid[i][j]==1:
                    freshOranges += 1
        if freshOranges==0:
            return 0

( ...  continua ao lado ... )
```

```python
        while q:
            row, col, minutes = q.popleft()

            moves = [ (row-1, col), (row+1, col),
                      (row, col-1), (row, col+1) ]
            for newRow, newCol in moves:
                if newRow<0 or newRow>=len(grid):
                    continue
                if newCol<0 or newCol>=len(grid[newRow]):
                    continue
                if grid[newRow][newCol]==1:
                    grid[newRow][newCol]=2
                    freshOranges -= 1
                    if freshOranges==0:
                        return minutes+1
                    q.append( (newRow, newCol, minutes+1) )

        return -1
```

DFS - evitando a recursão com uma pilha

BFS

Busca em Largura

Olhando para o código do BFS, ele pode ser adaptado para virar um DFS
Basta trocar a **fila** por uma **pilha**!

```
class Graph:

    def bfs(self, node):
        visited = set()
        visited.add(node) #starting node is visited
        q = deque()
        q.append(node) #starting node in queue
        while q:
            workingNode = q.popleft() #working node
            for neigh in self.adj[workingNode]:
                if neigh not in visited:
                    visited.add(neigh)
                    q.append(neigh)
```

DFS

Busca em Profundidade

Olhando para o código do BFS, ele pode ser adaptado para virar um DFS
Basta trocar a **fila** por uma **pilha**!

```
class Graph:

    def dfs(self, node):
        visited = set()
        visited.add(node) #starting node is visited
        stack = []
        stack.append(node) #starting node in stack
        while stack:
            workingNode = stack.pop() #working node
            for neigh in self.adj[workingNode]:
                if neigh not in visited:
                    visited.add(neigh)
                    stack.append(neigh)
```

DFS & BFS

- Isso significa que DFS e BFS são percursos implementados de forma bem parecida, mas com resultados bem diferentes
- A maior parte dos problemas que você pode resolver com um DFS, também pode resolver com um BFS
- Geralmente você resolve apenas com BFS quando quer saber o mínimo de arestas entre dois nós do grafo

Obrigado