python™

Laboratório de Programação Competitiva

Profa. Silvia Brandão

2024.2

Aula de hoje

Tópicos Avançados em Algoritmos - Grafos e Estruturas de Dados.

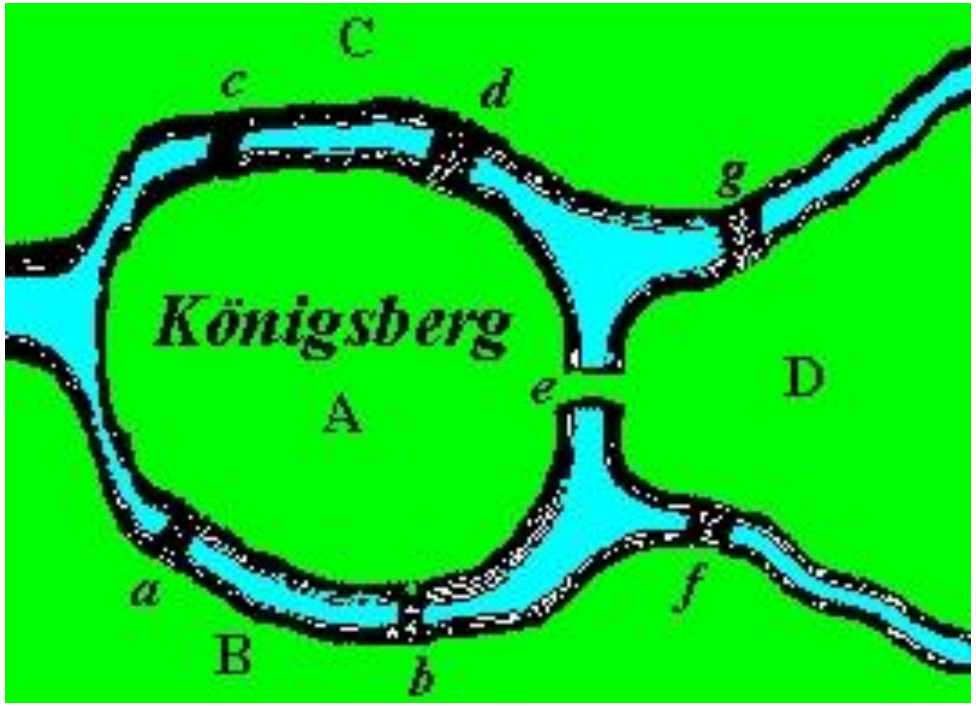
- Grafos em Python: Busca em largura (BFS), busca em profundidade (DFS), árvores, grafos ponderados.
- Estruturas de Dados em Python: Pilhas, filas, listas ligadas, árvores, heaps, segment trees.
- Abordagem de árvores e grafos ponderados.
- Estudo e implementação de estruturas de dados como pilhas, filas e árvores em Python.

MOMENTO N2:

- **Crie seu portifólio (notebook), no Google Colab, para o momento de avaliação N2**. Qualquer exercício proposto, neste momento, deverá ser implementado no portifólio para posterior correção no valor de 10 pontos. Não se esqueça de incluir NOME e RA além do enunciado de cada exercício. Compartilhar comigo: silvia.brandao@uniube.br
- Lista de exercícios no Beecrowd, valor de 5 pontos.
- Avaliação prática, valor de 10 pontos.

Grafos

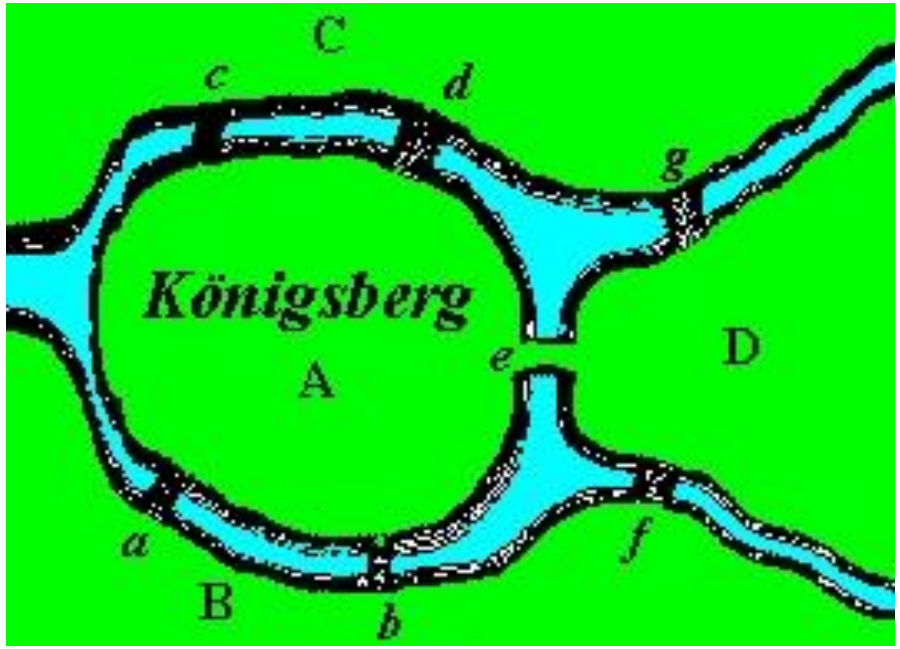
História: Euler (1736) - pontes de Königsberg

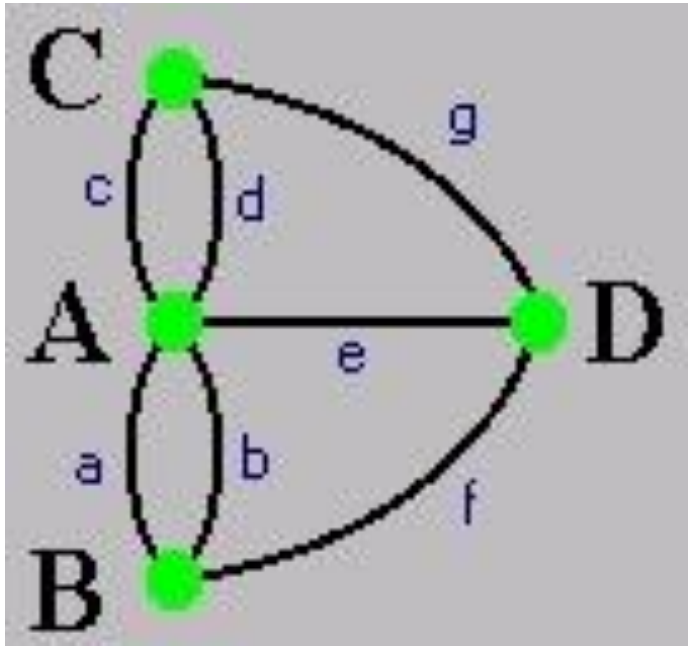


No século 18 havia na cidade de Königsberg um conjunto de sete pontes (identificadas pelas letras de **a** até **f** na figura) que cruzavam o rio Pregel. Elas conectavam duas ilhas entre si e as ilhas com as margens.

Problema: Por muito tempo os habitantes daquela cidade perguntavam-se se era possível cruzar as sete pontes numa caminhada contínua sem passar duas vezes por qualquer uma delas.

Grafos





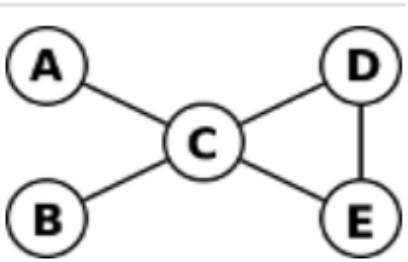
Baseado na disposição das pontes, mostrou-se que era impossível percorrer por todas as pontes passando somente uma vez

Grafos: aplicações

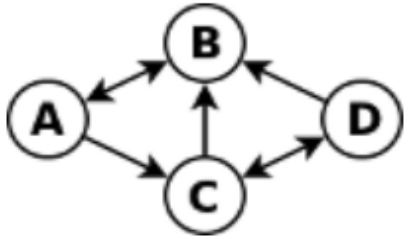
- Um vasto leque de sistemas naturais ou artificiais pode ser representado através de grafos. Por exemplo, no campo da biologia podemos ter uma cadeia alimentar, com os animais a serem os nós, e as relações de predador-presa entre eles a serem as arestas.
- Podemos ter outras estruturas biológicas, tais como uma rede neuronal (ligações entre neurónios) ou uma rede interação entre proteínas.
- Na sociologia podemos ter uma rede social de amigos com ligações representando amizades.
- No software podemos ter uma rede representando heranças entre objetos.
- Num patamar de existência mais física podemos ter grafos representando uma rede de autoestradas ligando cidades, uma rede elétrica ligando casas, uma rede computadores, ou uma rede de páginas da internet com hiperligações entre si.
- Muitos mais exemplos poderiam ser dados, mas o essencial é perceber que os grafos são uma representação abstrata muito poderosa e flexível.

Grafos: conceitos

- Um grafo é descrito por um par (V,E), onde V é o conjunto de vértices e E o conjunto das arestas que ligam pares de vértices. Um grafo pode ser dirigido ou direcionado, se as arestas tiverem uma direção, ou não dirigido, se as arestas não tiverem direção.

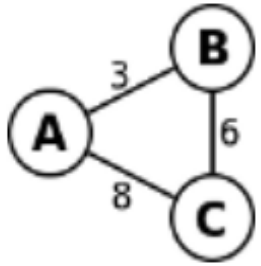


Grafo não direcionado



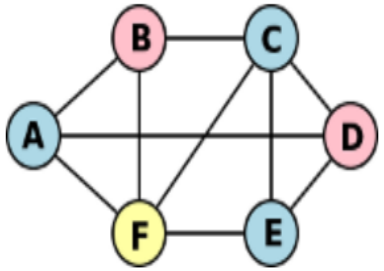
Grafo direcionado

- Se as arestas tiverem associado um peso ou custo, o grafo passa a chamar-se de grafo ponderado (ou pesado).



Grafo ponderado não dirigido

- Os nós podem ser de diferente tipos, representados por cores ou etiquetas. Nesse caso, o grafo é colorido.

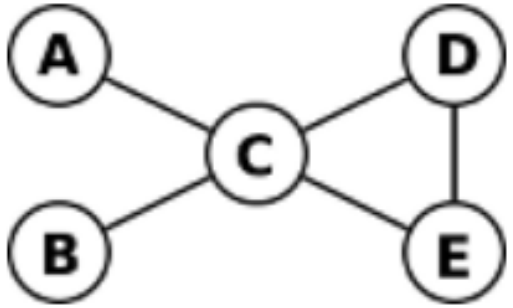


Grafo colorido não dirigido

- Quando apenas é admitida no máximo uma aresta entre dois nós, diz-se que o grafo é simples.

Grafos e a Ciência de Computadores

- Um grafo é uma que representação abstrata de um conjunto de objetos e das relações existentes entre eles. É definido por um conjunto de nós ou vértices, e pelas ligações ou arestas, que ligam pares de nós. Uma grande variedade de estruturas do mundo real podem ser representadas abstratamente através de grafos.

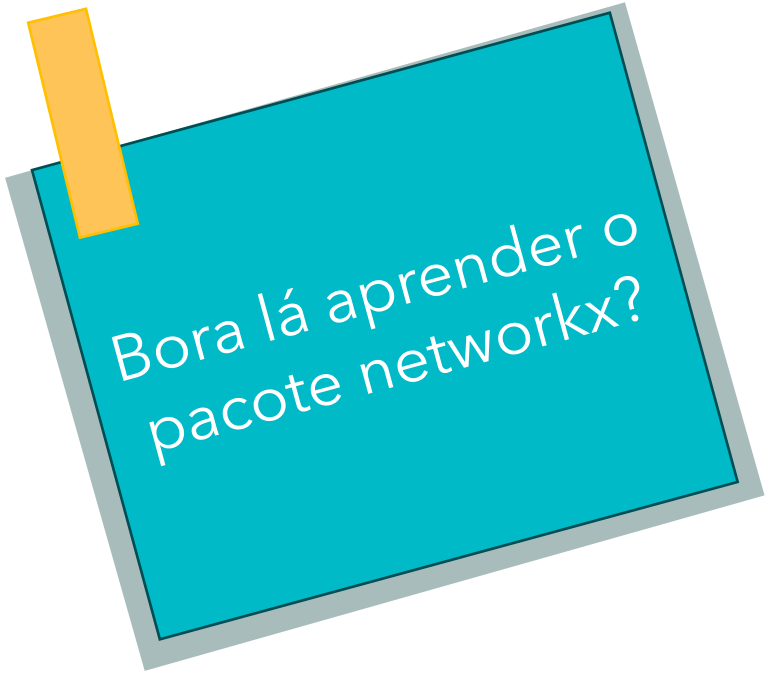


A figura ilustra um grafo não direcionado com 5 nós.

- Dada a enorme utilidade dos grafos como representações de conhecimento, existe toda uma área de Ciência de Computadores dedicada ao seu estudo e são inúmeros e muito importantes os algoritmos que manipulam grafos.
  - O **Algoritmo de Dijkstra**, que encontra o caminho mais curto entre um nó e todos os outros nós, num grafo pesado.
  - Existem muitos outros algoritmos para outras tarefas tais como **maximizar o fluxo** entre nós ou encontrar a **árvore mínima de suporte**.

Grafos x Python

- **NetworkX** é uma biblioteca da linguagem de programação Python para estudar grafos e redes; é um software livre.
- **Tutorial.** https://networkx.org/documentation/stable/tutorial.html



Introdução ao NetworkX

Instalação do NetworkX:

- `pip install networkx`

Importação do pacote.

- `import networkx as nx`

Criando um Grafo:

- `G = nx.Graph()  # Grafo não direcionado`
- `G.add_edges_from([(1, 2), (2, 3), (3, 4)])`

Visualização de Grafos através do pacote Matplotlib:

- `import matplotlib.pyplot as plt`
- `nx.draw(G, with_labels=True)`
- `plt.show()`

Operações Básicas com Grafos:

a) **Número de vértices e arestas**, grau dos vértices.

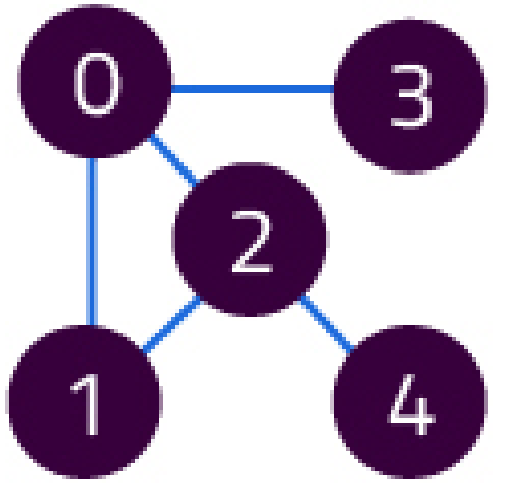
- `num_nodes = G.number_of_nodes()`
- `num_edges = G.number_of_edges()`

b) **Busca em Grafos**: exemplificar busca em profundidade (DFS) e busca em largura (BFS).

- `dfs_edges = list(nx.dfs_edges(G, source=1))`
- `bfs_edges = list(nx.bfs_edges(G, source=1))`

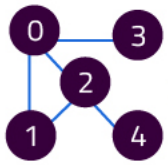
Buscas iterativas em Grafos

- Consiste em explorar um grafo, ou seja, obter um processo sistemático de como caminhar por seus vértices e arestas.
- Raiz da busca:
  - o vértice inicial da busca.
- Algoritmos básicos:
  - Busca em profundidade
  - Busca em largura



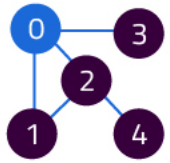
Funcionamento do algoritmo de busca em profundidade (DFS)

Busca em profundidade



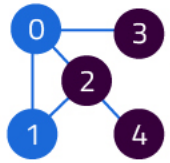
Visitados

pilha



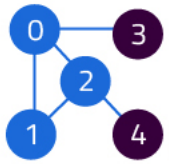
Visitados	0				
pilha	1	2	3		

Começando do vértice 0, o algoritmo o adiciona à lista de visitados e coloca todos os seus vértices adjacentes na **pilha**.

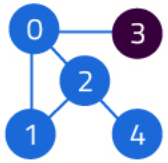


Visitados	0	1			
pilha	2	3			

Na sequência, visitamos o elemento no topo da pilha (ou seja, 1) e vamos para seus nós adjacentes. Como 0 já foi visitado, visitamos 2.

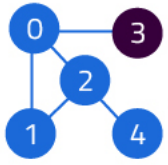


Visitados	0	1	2		
pilha	3				



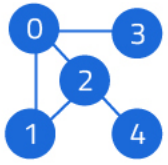
Visitados	0	1	2	4	
pilha	4	3			

O vértice 2 tem um vértice adjacente não visitado (o vértice 4), então o adicionamos ao topo da pilha e, em seguida, ele é visitado.



Visitados	0	1	2	4	
pilha	3				

O vértice 2 tem um vértice adjacente não visitado (o vértice 4), então o adicionamos ao topo da pilha e, em seguida, ele é visitado.

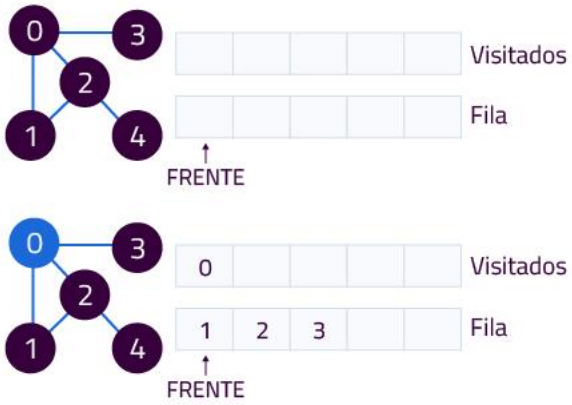


Visitados	0	1	2	4	3
pilha					

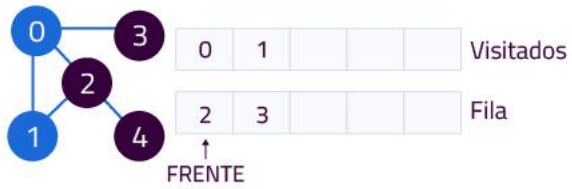
Depois de visitarmos o último elemento (3), ele não tem nenhum nó adjacente que ainda não tenha sido visitado. Portanto, a busca em profundidade é concluída.

Funcionamento do algoritmo de busca em largura (BFS)

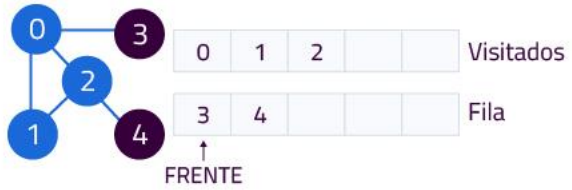
Busca em largura



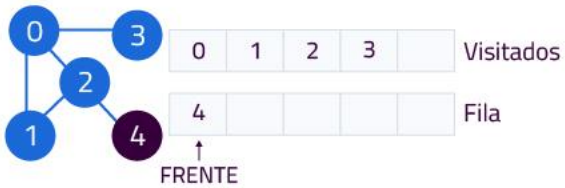
Começando do vértice 0, o algoritmo o adiciona à lista de visitados e coloca todos os seus vértices adjacentes na **fila**.



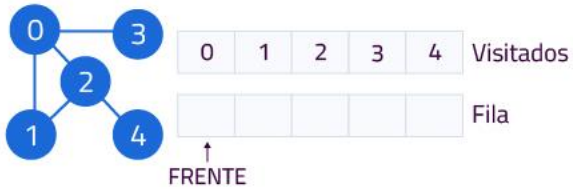
Em seguida, o vértice 1, que está na frente da fila, é visitado e seus nós adjacentes são examinados. Como 0 já foi visitado, visitamos 2.



O vértice 2 tem o vértice adjacente 4 não visitado, então ele é adicionado ao final da fila, e o vértice 3 é visitado, já que está na frente da fila.



Apenas o vértice 4 permanece na fila, pois o único nó adjacente a 3 (ou seja, 0) já foi visitado. Logo, 4 é visitado.



Como a fila está vazia, a busca em largura do grafo é concluída.

Referências

- Algoritmos para grafos. https://www.ime.usp.br/~pf/algoritmos_para_grafos/index.html#contents
- Networkx. https://networkx.org/documentation/stable/index.html

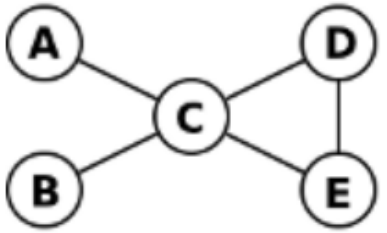
Não se esqueçam do Uniube+

Grafos e Estruturas de Dados

➢ Do ponto de vista da Programação, um grafo é uma estrutura de dados e existem duas maneiras base de o implementar na prática:

Matriz de Adjacências: uma matriz de V por V células, onde a célula na linha i e coluna j nos dá informação sobre a ligação entre os nós i e j.

- Por exemplo, se o grafo não for pesado, a célula podia conter o valor verdadeiro (ou um) quando existisse uma ligação entre i e j e o valor falso (ou zero) quando tal não acontecesse.
- Se o grafo for pesado, cada célula pode conter um número representando o peso da aresta.

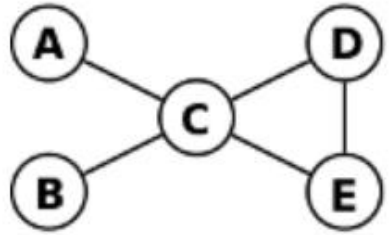


	A	B	C	D	E
A	0	0	1	0	0
B	0	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	1
E	0	0	1	1	0

Grafo não dirigido e sua correspondente matriz de adjacências

Lista de Adjacências: um vetor de listas onde a posição i do vetor contém uma lista dos nós aos quais o nó i se encontra ligado.

- Se o grafo for pesado, cada elemento da lista contém também o peso da ligação.



A: C
B: C
C: A, B, D, E
D: C, E
E: C, D

Grafo não dirigido e sua correspondente lista de adjacências.

Introdução ao NetworkX

```
import numpy as np
import networkx as nx

# Criar um grafo
G = nx.Graph()
G.add_edges_from([(0, 1, {'weight': 4}), (0, 2, {'weight': 1}),
                  (1, 2, {'weight': 2}), (1, 3, {'weight': 5}),
                  (2, 3, {'weight': 8})])

# Número de vértices
num_nodes = G.number_of_nodes()

# Criar matriz de adjacência
adj_matrix = np.zeros((num_nodes, num_nodes))

for (u, v, wt) in G.edges(data=True):
    adj_matrix[u][v] = wt['weight']
    adj_matrix[v][u] = wt['weight']  # Para grafos não direcionados

print("Matriz de Adjacência:")
print(adj_matrix)
```

Introdução ao NetworkX

```
# Criar lista de adjacências
adj_list = {i: [] for i in range(num_nodes)}

for (u, v, wt) in G.edges(data=True):
    adj_list[u].append((v, wt['weight']))
    adj_list[v].append((u, wt['weight']))  # Para grafos não direcionados

print("Lista de Adjacências:")
for key, value in adj_list.items():
    print(f"{key}: {value}")
```

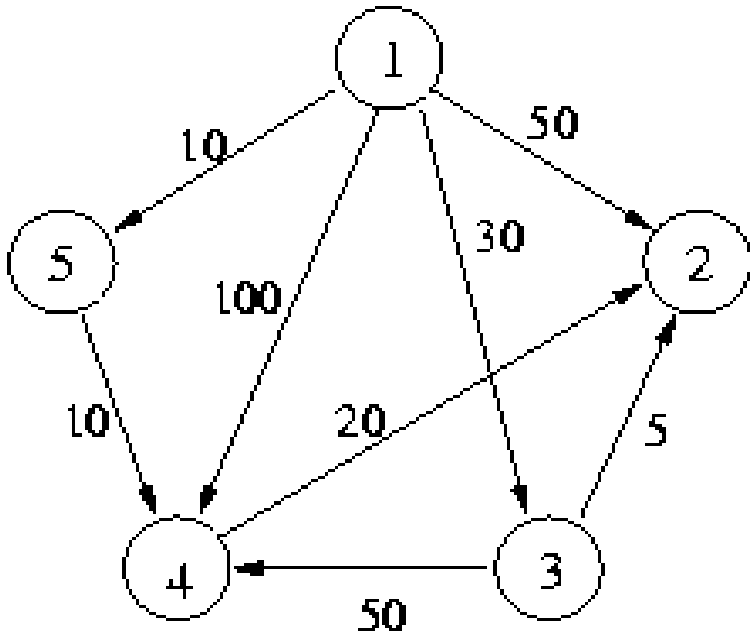
O problema do Caminho mais curto

- Um motorista deseja encontrar o caminho, mais curto possível, entre duas cidades do Brasil. Caso ele receba um mapa das estradas de rodagem do Brasil, no qual a distância entre cada par adjacente de cidades está exposta, como poderíamos determinar uma rota mais curta entre as cidades desejadas?
  - Uma maneira possível é enumerar todas as rotas possíveis que levam de uma cidade à outra, e então selecionar a menor.
  - O problema do menor caminho consiste em determinar um menor caminho entre um vértice de origem $s \in V$ e todos os vértices v de V.
  - Variantes do problema:
    - Menor caminho com destino único: encontrar um caminho mais curto para um vértice destino v
    - Menor caminho para um par: encontrar um caminho mais curto para um determinado par de vértices u e v
    - Menor caminho para todos os pares: encontrar um caminho mais curto de u para v, para todos e quaisquer pares u e v

O problema do Caminho mais curto

Algoritmo de Dijkstra para cálculo do Caminho de Custo Mínimo

Como obter um caminho mínimo partindo de 1 ?



Passo	v	C	D
Início	-	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,20,10]
2	4	{2,3}	[40,30,20,10]
3	3	{2}	[35,30,20,10]

Dessa maneira, obtivemos o **custo do caminho mínimo** para quaisquer vértices, mas não obtivemos o caminho mínimo. Para identificar esse caminho mínimo, é só acrescentar mais um vetor P[2..n], onde P[v] indica o vértice que precede v no caminho mais curto. Para isso, primeiro devemos inicializar esse vetor. Segundo, no mesmo momento que o vetor D é atualizado, atualizamos também o vetor P.

Introdução ao NetworkX

Operações Básicas com Grafos - continuando

Encontrar o caminho mais curto entre dois vértices, usando o algoritmo de Dijkstra:

- `path = nx.shortest_path(G, source=1, target=4, weight='weight')`

Árvore Geradora Mínima
(Minimum Spanning Tree)

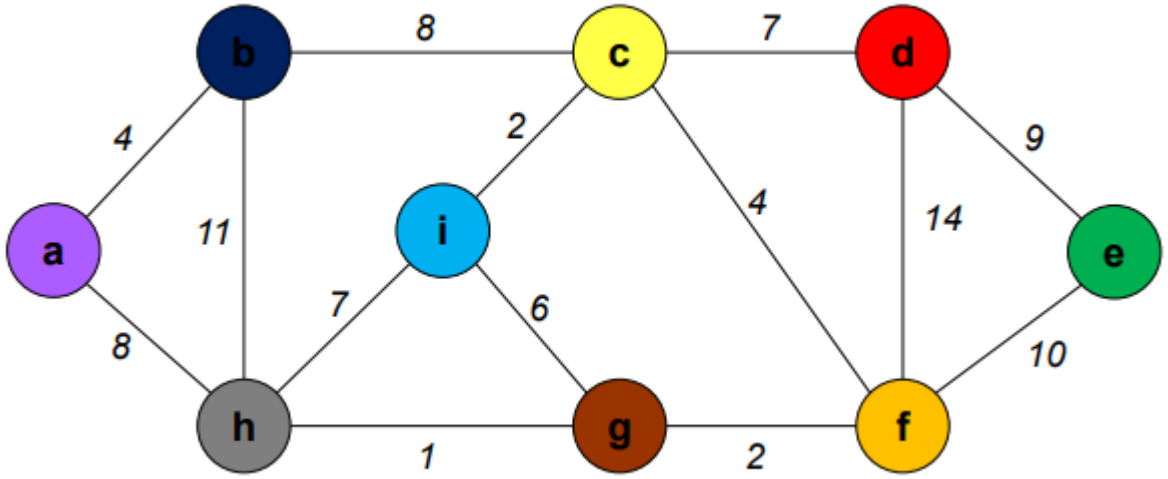
- Seja G= (V,E) um grafo conexo com uma função de peso w: E → R
- O peso de uma árvore geradora T de G é definido como a soma dos pesos de todas as arestas de T:

$$w(T) = \sum_{(u,v)\in T} w(\in u, v)$$

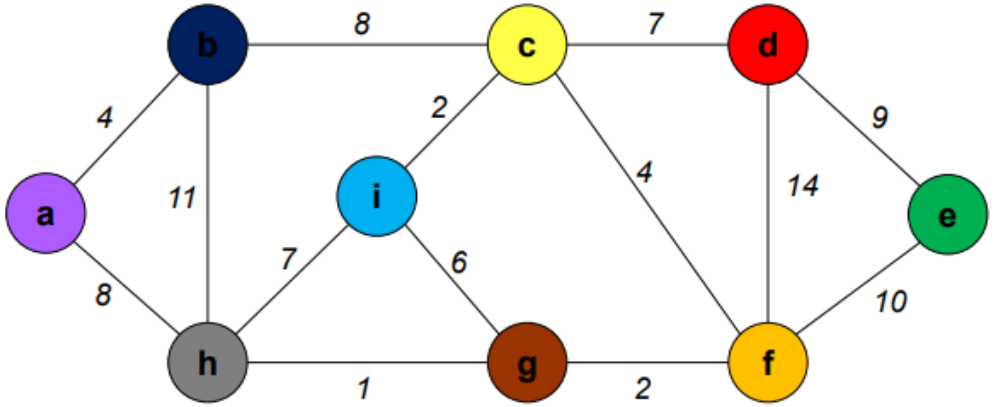
- A meta é achar uma **árvore geradora de peso mínimo** para G, ou seja, a Árvore Geradora Mínima.
- Dois algoritmos clássicos:
  - Kruskal
  - Prim

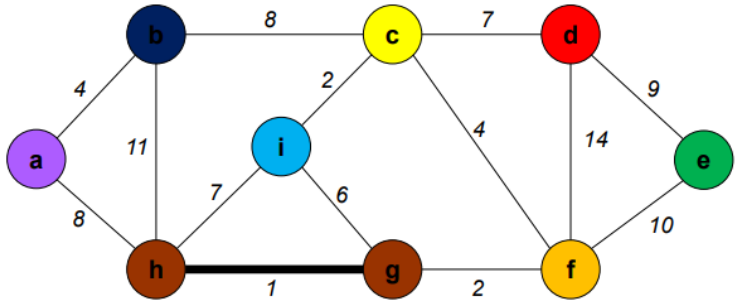
Árvore Geradora Mínima
Algoritmo de Kruskal

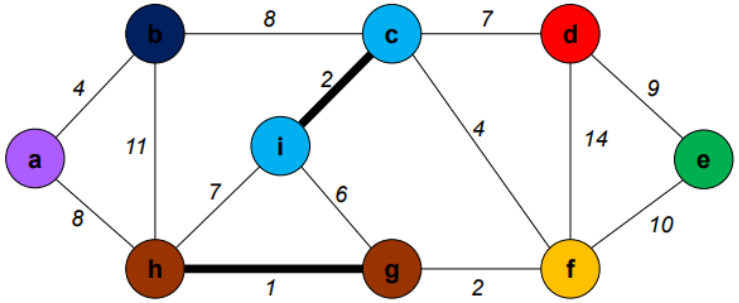
- **Princípio:** a aresta de menor peso sempre pertence à AGM
- Escolha a aresta de menor peso entre todas as arestas que conectam quaisquer dois vértices em A
- A nova aresta não pode ligar vértices na mesma árvore (não pode formar ciclo)

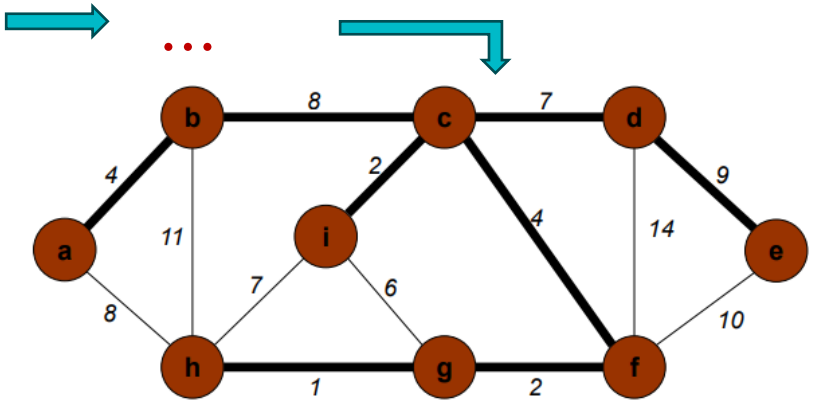


Árvore Geradora Mínima – Algoritmo de Kruskal









Introdução ao NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

# Criar um grafo
G = nx.Graph()
G.add_edges_from([(0, 1, {'weight': 4}), (0, 2, {'weight': 1}),
                  (1, 2, {'weight': 2}), (1, 3, {'weight': 5}),
                  (2, 3, {'weight': 8}), (2, 4, {'weight': 10}),
                  (3, 4, {'weight': 3})])

# Encontrar a árvore geradora mínima usando o algoritmo de Kruskal
mst_kruskal = nx.minimum_spanning_tree(G, algorithm='kruskal')

# Exibir as arestas da árvore geradora mínima
print("Arestas da Árvore Geradora Mínima (Kruskal):")
print(mst_kruskal.edges(data=True))

# Visualizar a árvore geradora mínima
pos = nx.spring_layout(mst_kruskal)
nx.draw(mst_kruskal, pos, with_labels=True, node_color='lightblue', edge_color='black')
edge_labels = nx.get_edge_attributes(mst_kruskal, 'weight')
nx.draw_networkx_edge_labels(mst_kruskal, pos, edge_labels=edge_labels)
plt.title("Árvore Geradora Mínima (Kruskal)")
plt.show()
```

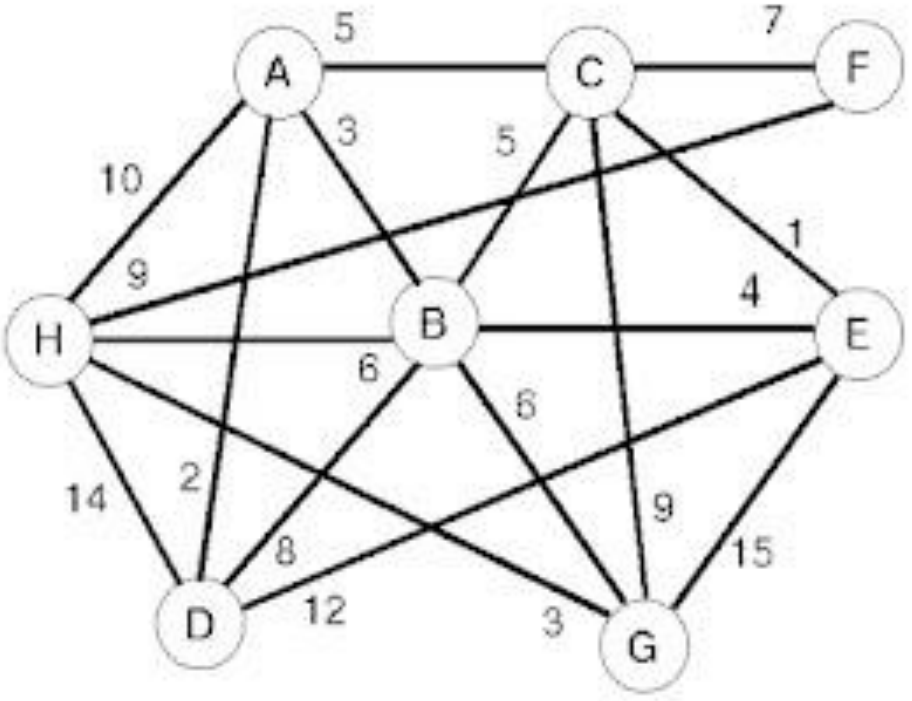
O algoritmo de Kruskal funciona bem com grafos ponderados, adicionando as arestas mais leves primeiro, desde que não formem ciclos.

Introdução ao NetworkX

```
# Encontrar a árvore geradora mínima usando o algoritmo de Prim
mst = nx.minimum_spanning_tree(G)

print("Arestas da Árvore Geradora Mínima:")
print(mst.edges(data=True))

# Para visualizar a árvore geradora mínima
import matplotlib.pyplot as plt

pos = nx.spring_layout(mst)
nx.draw(mst, pos, with_labels=True)
edge_labels = nx.get_edge_attributes(mst, 'weight')
nx.draw_networkx_edge_labels(mst, pos, edge_labels=edge_labels)
plt.title("Árvore Geradora Mínima")
plt.show()
```

Exercício: Estudando grafos com o Networkx

8. Para o grafo ponderado ao lado, defina e dê exemplos visuais do que são:
   - Determine sua matriz de adjacência.
   - Represente-o utilizando Lista de adjacências.
   - Liste vértices e arestas.
   - Mostre o grau do nó C.
   - Faça uma busca em largura (BFS) a partir do nó A.
   - Faça uma busca em profundidade (DFS) a partir do nó A.
   - Indique um caminho válido de A até E que possua custo igual a 24.
   - Encontre o caminho mais curto entre dois vértices, usando o algoritmo de Dijkstra.
   - Encontre a árvore geradora mínima.



Entrega pelo portifólio (notebook), no Google Colab.

CORREÇÃO NA PRÓXIMA AULA

Vale nota

Referências

- Algoritmos para grafos. https://www.ime.usp.br/~pf/algoritmos_para_grafos/index.html#contents
- Networkx. https://networkx.org/documentation/stable/index.html

Não se esqueçam do Uniube+