Teoria dos

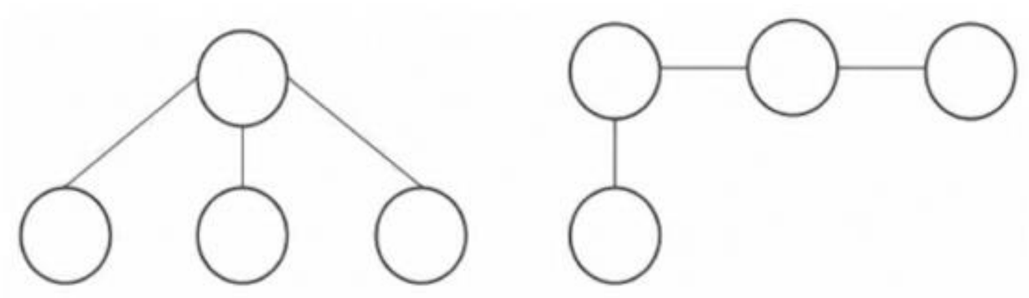
GRAFOS

Aula 06: Árvores

Prof. José Alberto S. Torres

O que são **árvores?**

- No estudo de grafos, uma árvore é um tipo especial de grafo que possui duas características principais:
  - **É um grafo conectado**: é possível ir de qualquer vértice para qualquer outro. Não existem partes isoladas.
  - **Não contém ciclos**: não há caminhos fechados. Se você começar a percorrer o grafo a partir de um vértice, nunca voltará ao ponto de partida sem ter que refazer seus passos.



O que são **árvores?**

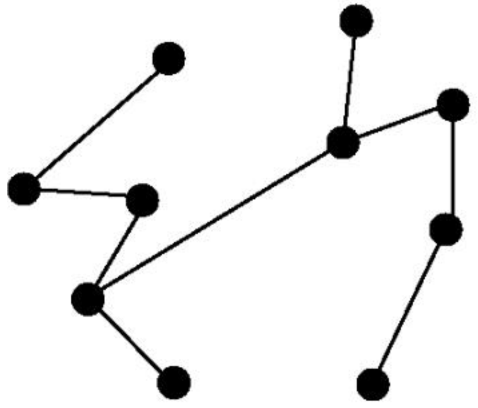
- A partir dessas duas propriedades, podemos inferir outras características importantes das árvores:
  - **Arestas e Vértices**: Uma árvore com n vértices sempre terá n−1 arestas. Por exemplo, se uma árvore tem 5 vértices, ela terá exatamente 4 arestas.
  - **Aresta de Corte (ponte):** Cada aresta em uma árvore é uma "aresta de corte". Isso significa que se você remover qualquer aresta, o grafo se tornará desconectado.
  - **Caminho Único:** Existe um e somente um caminho simples entre quaisquer dois vértices em uma árvore.

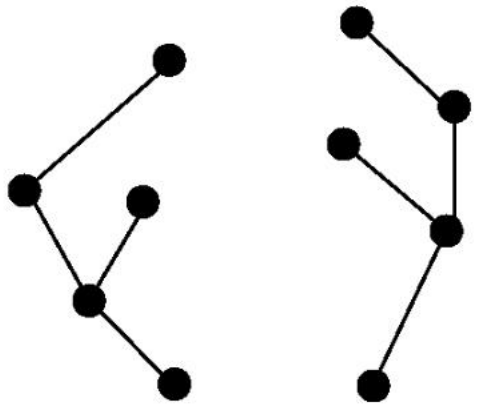
Floresta

- Em teoria dos grafos, uma **floresta** (*forest*) é uma **coleção de uma ou mais árvore**
- Formalmente, uma floresta é um **grafo não-direcionado acíclico**.
- A diferença fundamental em relação a uma árvore é que uma floresta **não precisa ser conectada**.

Árvore X Floresta

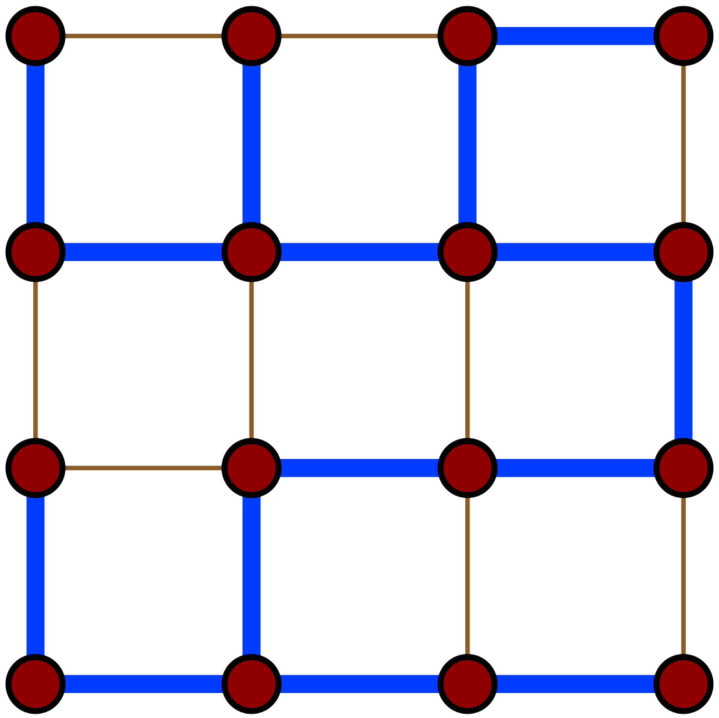
- Uma **Árvore** é um grafo **acíclico E conectado**.
- Uma **Floresta** é um grafo **acíclico** (que pode ser conectado ou desconectado).
- Toda árvore é, tecnicamente, uma floresta (uma floresta com apenas um componente conectado).
- Uma floresta é um grafo cujos componentes conectados são, individualmente, árvores.





Árvore Geradora

- Uma Árvore Geradora de um grafo G conectado e não-direcionado é um subgrafo de G que satisfaz três condições:
    - Inclui todos os vértices do grafo original G. (Esta é a parte "geradora" ou "spanning").
    - É acíclica (não contém ciclos) e é conectada.
    - É formada por um subconjunto das arestas do grafo original G.



Algoritmo de **Kruskal**

- Mais interessante do que simplesmente notar que um grafo conectado G possui uma árvore geradora, é **encontrar uma árvore geradora mínima em um grafo conectado ponderado G**.
- Em outras palavras, nosso objetivo é encontrar uma árvore geradora T com peso mínimo entre todas as árvores geradoras de G.
- Um algoritmo que constrói eficientemente tal árvore foi projetado por Kruskal.

Algoritmo de **Kruskal**

- O Algoritmo de Kruskal é um exemplo clássico de algoritmo guloso (greedy algorithm).
- A estratégia gulosa aqui é: **Sempre escolha a opção que parece ser a melhor no momento**.
- Para Kruskal, a "**melhor opção**" é sempre **a aresta de menor peso disponível que ainda não foi escolhida**.
- Ele constrói a árvore aresta por aresta, **começando pelas mais "baratas".**
- A única regra é: ao adicionar uma nova aresta, você **não pode criar um ciclo** com as arestas que já foram adicionadas.

Kruskal passo a passo

1. **Ordenar as Arestas:** Crie uma lista de todas as arestas do grafo e ordene-a em ordem crescente de peso (da mais leve para a mais pesada).
2. **Inicializar a Floresta:** Crie uma "floresta" onde cada vértice do grafo é sua própria árvore individual. Em outras palavras, no início, nenhum vértice está conectado a outro.

Kruskal passo a passo

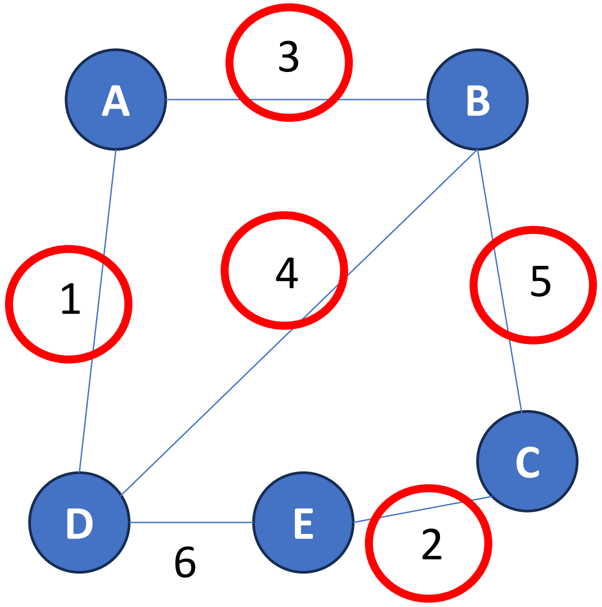
3. **Iterar e Construir a Árvore:** Percorra a lista de arestas ordenadas (da mais barata para a mais cara):
   - Para cada aresta (u, v):
   - Verifique se os vértices u e v já pertencem à mesma árvore (ou ao mesmo componente conectado).
     - **Se NÃO pertencem à mesma árvore:** A aresta (u, v) é segura. Adicioná-la não criará um ciclo. Então:
       - Adicione a aresta (u, v) à sua Árvore Geradora Mínima.
       - Junte (una) as duas árvores que continham u e v em uma única árvore.
     - **Se JÁ pertencem à mesma árvore:** Descarte a aresta (u, v). Adicioná-la criaria um ciclo.

IESB
Centro Universitário

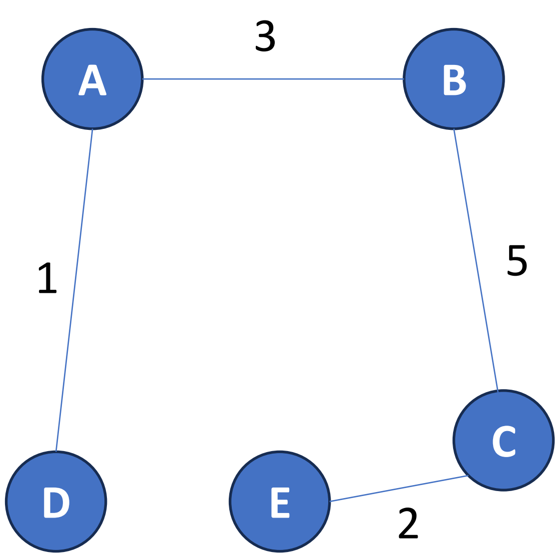
Kruskal passo a passo

4. **Condição de Parada:** O algoritmo termina quando a Árvore Geradora Mínima tiver **n - 1** arestas, onde n é o número de vértices. Nesse ponto, todos os vértices estarão conectados em uma única árvore.

Exemplo Prático



AGM



Componentes = {A}, {B}, {C}, {D}, {E}

A e D estão em componentes diferentes.

Componentes = {A, D}, {B}, {C}, {E}

C e E estão em componentes diferentes.

Componentes = {A, D}, {B}, {C,E}

A e B estão em componentes diferentes.

Componentes = {A, D, B}, {C,E}

B e D estão no mesmo componente

B e C estão em componentes diferentes. Unir {A,B,D} e {C, E}

Componentes = {A, D, B, C, E}

Todos os vértices conectados! Fim!

Atividade 1

- Crie o algoritmo para implementar o código de Kruskal.
- Adaptação do Grafo Esparso para conter pesos

Kruskal.py

```python
def kruskal(grafo:GrafoEsparsoPonderado):

    # 1. Ordena as arestas pelo peso
    arestas = grafo.get_arestas()
    #Em vez de ordenar os elementos com base em seu valor completo (o tuplo inteiro), ela usará o valor
    #retornado pela função key para fazer a ordenação.
    sorted_edges = sorted(arestas, key=lambda item: item[2])

    # 2. Inicializa a estrutura BuscaUniaoSimples
    uf = BuscaUniaoSimples(grafo.get_vertices())

    minimum_spanning_tree = []
    total_weight = 0

    # 3. Itera sobre as arestas ordenadas
    for u, v, weight in sorted_edges:
        # 4. Se adicionar a aresta não formar um ciclo...
        #    (ou seja, se u e v estiverem em conjuntos diferentes)
        if uf.find(u) != uf.find(v):
            # ...une os conjuntos e adiciona a aresta à árvore.
            uf.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
            total_weight += weight

    return minimum_spanning_tree, total_weight
```

Kruskal.py

```python
class BuscaUniaoSimples():

    def __init__(self, vertices):
        """
        Cada vértice começa como seu próprio "pai" (em seu próprio conjunto).
        """
        self.parent = {v: v for v in vertices}
        print(self.parent)

    def find(self, v):
        """
        Encontra a raiz do conjunto ao qual 'v' pertence.
        Faz isso simplesmente seguindo os ponteiros de "pai" até o topo
        """
        root = v
        while self.parent[root] != root:
            root = self.parent[root]
        return root

    def union(self, u, v):
        """
        Une os conjuntos de 'u' e 'v' de forma arbitrária.
        """
        root_u = self.find(u)
        root_v = self.find(v)

        # Se não estiverem no mesmo conjunto, une os dois
        if root_u != root_v:
            # Simplesmente faz a raiz de um apontar para a raiz do outro
            self.parent[root_v] = root_u
            return True
        return False
```

Kruskal.py

```python
if __name__ == "__main__":
    vertices_exemplo = ['A', 'B', 'C', 'D', 'E']
    grafo = GrafoEsparsoPonderado(labels=vertices_exemplo)
    grafo.adicionar_aresta('A', 'B', 3)
    grafo.adicionar_aresta('A', 'D', 1)
    grafo.adicionar_aresta('B', 'C', 5)
    grafo.adicionar_aresta('B', 'D', 4)
    grafo.adicionar_aresta('C', 'E', 2)
    grafo.adicionar_aresta('D', 'E', 6)
    grafo.imprimir()


    agm, custo = kruskal(grafo)

    print("\nArestas da Árvore Geradora Mínima:")
    for u, v, weight in agm:
        print(f"  - De {u} para {v} com custo {weight}")

    print(f"\nCusto Total da Árvore Geradora Mínima: {custo}")
```

```
res/Library/CloudStorage/OneDrive-Pessoal/IESB/Grafos
Aresta adicionada entre A e B
Aresta adicionada entre A e D
Aresta adicionada entre B e C
Aresta adicionada entre B e D
Aresta adicionada entre C e E
Aresta adicionada entre D e E

Lista de Adjacências:
  A -> [ [('B', 3), ('D', 1)] ]
  B -> [ [('A', 3), ('C', 5), ('D', 4)] ]
  C -> [ [('B', 5), ('E', 2)] ]
  D -> [ [('A', 1), ('B', 4), ('E', 6)] ]
  E -> [ [('C', 2), ('D', 6)] ]

{'A': 'A', 'B': 'B', 'C': 'C', 'D': 'D', 'E': 'E'}

Arestas da Árvore Geradora Mínima:
  - De A para D com custo 1
  - De C para E com custo 2
  - De A para B com custo 3
  - De B para C com custo 5

Custo Total da Árvore Geradora Mínima: 11
```

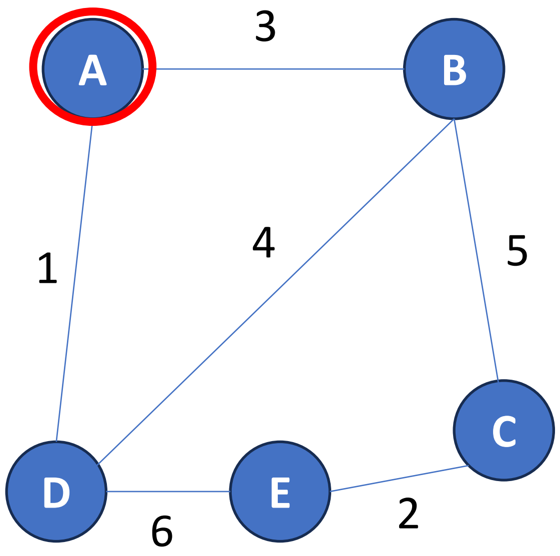
Algoritmo de **Prim**

- Assim, como Kruskal, é um algoritmo de busca de caminhos em grafos, utilizado para encontrar a Árvore Geradora Mínima (AGM) de um grafo conectado e ponderado.
- O algoritmo de Prim também opera de forma "gulosa" (greedy), ou seja, ele toma a melhor decisão a cada passo, sem se preocupar com o resultado final.
- A principal diferença para o de kruskal é a abordagem:
  - **Prim** constrói a AGM a partir de um único vértice, adicionando arestas a uma única componente em crescimento.
  - **Kruskal** constrói a AGM adicionando arestas ao grafo em ordem crescente de peso, formando várias componentes que são gradualmente unidas.

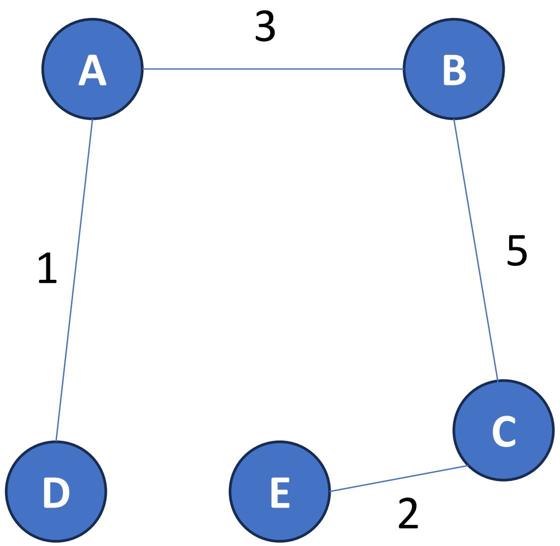
Algoritmo de **Prim**

- O processo é o seguinte:
  - **Escolha um vértice inicial:** Comece com qualquer vértice do grafo.
  - **Adicione a aresta de menor peso:** Encontre a aresta de menor peso que conecta um vértice na subárvore em crescimento a um vértice fora dela.
  - **Adicione o vértice:** Inclua essa aresta e o vértice que ela conecta na subárvore.
  - **Repita:** Repita os passos 2 e 3 até que todos os vértices do grafo original estejam na sua AGM.

Exemplo Prático



AGM



AGM = {A}

Arestas candidatas (A,D,1) e (A,B,3)

Escolhida a de menor peso (A,D)

AGM = {A,D}

Arestas candidatas (A,B,3) (D,E,6) e (D,B,4)

Escolhida a de menor peso (A,B)

AGM = {A,D, B}

Arestas candidatas (D,E,6) e (B,C,5)
(D,B,4) ignorada pois B está em AGM

Escolhida a de menor peso (B,C)

AGM = {A,D, B, C}

Arestas candidatas (D,E,6) e (C,E, 2)

Escolhida a de menor peso (C,E)

AGM = {A,D, B, C, E}

Atividade 2

- Crie o algoritmo para implementar o código de Prim.

DÚVIDAS

IESB
Centro Universitário