# Trabalho 2 - Análise de Algoritmos
# Grafos: Algoritmos de Prim e Kruskal para MST


### Davi Augusto Neves Leite - RA: 191027383
### Luiz Fernando Sementille - RA: 191021032


---

## 1. Introdução

Uma **Árvore Geradora Mínima**, ou MST em inglês, pode ser definido formalmente da seguinte forma:

"*Dado um grafo não orientado, conexo e ponderado G = (V, A), a árvore geradora mínima é um subgrafo de G, denotado por G' = (V, A'), que conecta todos os vértices de G e com o menor peso total final.*"

Para os problemas de MST os algoritmos mais usuais são os algoritmos de **Prim** e **Kruskal**. Ambos os algoritmos usam uma abordagem gulosa para resolver problemas, em cada iteração os algoritmos "abocanham" a aresta que parece mais promissora naquele momento sem se preocupar com efeito global dessa escolha. Apesar de utilizarem essa abordagem, é sempre garantido que se alcance a melhor solução possível. Além disso, as execuções são muito rápidas.

Dessa forma, o trabalho consiste na implementação e testagem dos algoritmos de Prim e Kruskal para alcançar a MST e o seu custo total, a partir de um grafo definido pelo usuário. O intuito é avaliar as diferenças bem como a complexidade temporal de ambas abordagens, definindo qual usar dependendo da ocasião.

Para tanto, as seções foram divididas em: Funções Auxiliares, a qual contém algumas funções utilizadas para facilitar a implementação dos algoritmos de MST, bem como na exibição dos resultados obtidos; Explicação e Implementação dos Algoritmos, em que. como o próprio nome diz, traz a ideia geral de como cada algoritmo funciona e sua respectiva implementação na linguagem Python 3; Exemplos, cujo intuito está em trazer, em termos práticos, a aplicação de ambos os algoritmos para diferentes tipos de grafos e; Conclusões, na qual traz os resultados obtidos, em termos de complexidade, e as considerações finais acerca do tema.

Por fim, necessitou-se de uma biblioteca para a melhor implementação do algoritmo de Prim: a heapq, biblioteca que possui funções primárias de construção e manipulação de uma árvore do tipo Heap.

In [1]:
```
import heapq as hp  # Utilizado para importar a árvore heap para o algoritmo de Prim
```

---

## 2. Funções Auxiliares

Para facilitar a implementação e utilização dos algoritmos de Prim e Kruskal, foi necessária a criação de três funções auxiliares no código-fonte.

A primeira e a segunda estão associadas à estrutura de dados denominada **União-Busca** (ou *Disjoint Set Union*, do inglês). Essa estrutura permite o controle de um conjunto de elementos particionados em subconjuntos disjuntos. Dessa forma, as funções associadas *"juntar"* e *"encontrar"* são utilizadas no algoritmo de Kruskal para armazenar vários conjuntos disjuntos de vértices e, assim, encontrar a Árvore

Geradora Mínima a partir da chamada aresta segura (aresta que tenha o menor peso no grafo e que não tenha sido inserida na estrutura).

In [2]:
```
# Disjoint Set Union (DSU) para Kruskal
def juntar(a, b):
    if(tamanho[a] < tamanho[b]):
        a, b = b, a

    link[b] = a
    tamanho[a] += tamanho[b]

def encontrar(x):
    if(x == link[x]):
        return x

    return encontrar(link[x])
```

A terceira, denominada de *"mostraMST"*, como o próprio nome sugere, é responsável por exibir os valores da MST (em termos de vértice origem, vértice destino e peso entre eles) bem como o custo mínimo total final.

In [3]:
```
# Mostrar a MST e o custo mínimo total
def mostraMST(mst, cost):
    print('Árvore Geradora Mínima:\n')

    for aresta in mst:
        print(aresta)

    print('\nCusto:', cost)
    print("################################")
```

---

# 3. Explicação e Implementação dos Algoritmos

Nesta seção, são abordados o método genérico de implementação dos algoritmos e o funcionamento de cada algoritmo analisado, bem como a implementação em Python 3.

## Método Genérico

É feito um loop sobre as arestas do grafo dado pelo usuário e a cada passo é determinada uma aresta (u, v) que poderá ser inserida em G'. Essa aresta é chamada de **aresta segura**, pois ela pode ser adicionada com segurança em G' sem que prejudique a formação de uma árvore geradora mínima. Cada um dos algoritmos possui uma definição própria do que considera uma aresta segura.

## Prim

O algoritmo de Prim considera uma aresta segura, uma aresta (u, v) ∈ A que possua o menor custo possível e que seja incidente a um vértice que já pertença à árvore, que ainda não tenha sido inserida e não forme ciclos.

**Complexidade de Tempo (Lista de Adjacência): O(a log v), em que *a* é o número de arestas e *v* o número de vértices.**

In [4]:
```
################################################################################
# Algoritmo de Prim
################################################################################
def prim(origem, numVertices, adjLista):
    visitado = [False for i in range(numVertices + 1)]
    parent = [-1 for i in range(numVertices + 1)]
```

```
    distancia = [float('inf') for i in range(numVertices + 1)]

    fila_prioridade = []
    distancia[origem] = 0
    hp.heappush(fila_prioridade, [distancia[origem], origem]) # Inserindo o primeiro vértice

    mst = []
    custo = 0

    while fila_prioridade:
        # Recuperando valores
        valor_fila = fila_prioridade[0]
        peso, no = valor_fila

        hp.heappop(fila_prioridade) # Eliminando da Fila

        if(not visitado[no]):
            custo += peso
            visitado[no] = True

            for par_peso_no in adjLista[no]:
                no_adj, peso_adj = par_peso_no
                if(visitado[no_adj] == False and distancia[no_adj] > peso_adj):
                    parent[no_adj] = no
                    distancia[no_adj] = peso_adj

                    hp.heappush(fila_prioridade, [peso_adj, no_adj])

    for i in range(1, numVertices + 1):
        if(parent[i] != -1):
            mst.append([parent[i], i, distancia[i]])

    return mst, custo
```

## Kruskal

O algoritmo de Kruskal considera uma aresta segura, uma aresta (u, v) que dentre todas as arestas de G tenha o menor custo e que já não tenha sido inserida. O algoritmo utiliza uma estrutura de dados auxiliar denominada ***Disjoint-Set (Union Find)*** para armazenar vários conjuntos disjuntos de vértices, isto é, inicialmente cada vértice é o líder do seu próprio conjunto.

**Complexidade de Tempo (Lista de Adjacência): O(a log v), em que *a* é o número de arestas e *v* o número de vértices.**

In [5]:

```
##############################################################################
# Algoritmo de Krukal
##############################################################################
def kruskal(numArestas, arestaLista):
    mst = []
    custo = 0

    # Ordenar por peso
    arestaLista = sorted(arestaLista)

    for i in range(numArestas):
        peso = arestaLista[i][0]
        origem = arestaLista[i][1]
        destino = arestaLista[i][2]

        origemLocalizacao = encontrar(origem)
        destinoLocalizacao = encontrar(destino)

        if(origemLocalizacao != destinoLocalizacao):
            juntar(origemLocalizacao, destinoLocalizacao)
            custo += peso
```

```
            mst.append([origem, destino, peso])

    return mst, custo
```

---

# 4. Exemplos

Os casos de teste foram gerados aleatoriamente pela ferramenta ***"Test Case Generator"*** (https://test-case-generator.herokuapp.com/), com a opção de Grafos Ponderados Aleatórios (*Random Weighted Graph*). Os grafos apresentados abaixo são formados pela tupla {origem, destino, peso da aresta}. Também, limitou-se os pesos das arestas de 1 até 10, inclusivo.

## Exemplo 1

Número de Vértices = 4
Número de Arestas = 4

Grafo 1

{2, 1, 6}
{3, 1, 2}
{3, 4, 2}
{1, 3, 1}

In [6]:

```
##################################################################################
# Main
##################################################################################
numVertices, numArestas = input().split()

numVertices = int(numVertices)  # Número de Vértices
numArestas  = int(numArestas)   # Número de Arestas

# Prim com Lista de Adjacência
adjLista = [[] for i in range(numVertices + 1)]

# Kruskal com Lista de Arestas Adjacentes
arestaLista = []
link = [i for i in range(numVertices + 1)]
tamanho = [1 for i in range(numVertices + 1)]

# Leitura do Grafo
for i in range(numArestas):
    origem, destino, peso = input().split()

    origem = int(origem)
    destino = int(destino)
    peso = int(peso)

    # Inserção no Prim
    adjLista[origem].append([destino, peso])
    adjLista[destino].append([origem, peso])

    # Inserção no Kruskal
    arestaLista.append([peso, origem, destino])

# Aplicando Prim para MST
mst, custo = prim(1, numVertices, adjLista)

# Resultado do Prim
print("##############################")
print("Algoritmo de Prim")
```

```
mostraMST(mst, custo)

# Aplicando Kruskal para MST
mst, custo = kruskal(numArestas, arestaLista)

# Resultado do Kruskal
print("Algoritmo de Kruskal")
mostraMST(mst, custo)
```

```
##############################
Algoritmo de Prim
Árvore Geradora Mínima:

[1, 2, 6]
[1, 3, 1]
[3, 4, 2]

Custo: 9
##############################
Algoritmo de Kruskal
Árvore Geradora Mínima:

[1, 3, 1]
[3, 4, 2]
[2, 1, 6]

Custo: 9
##############################
```

## Exemplo 2

Número de Vértices = 5

Número de Arestas = 4

Grafo 2

{3, 4, 1}

{1, 4, 6}

{5, 2, 2}

{4, 5, 5}

In [7]:

```
##############################################################################
# Main
##############################################################################
numVertices, numArestas = input().split()

numVertices = int(numVertices)  # Número de Vértices
numArestas  = int(numArestas)   # Número de Arestas

# Prim com Lista de Adjacência
adjLista = [[] for i in range(numVertices + 1)]

# Kruskal com Lista de Arestas Adjacentes
arestaLista = []
link = [i for i in range(numVertices + 1)]
tamanho = [1 for i in range(numVertices + 1)]

# Leitura do Grafo
for i in range(numArestas):
    origem, destino, peso = input().split()

    origem = int(origem)
    destino = int(destino)
    peso = int(peso)

    # Inserção no Prim
```

```
    adjLista[origem].append([destino, peso])
    adjLista[destino].append([origem, peso])

    # Inserção no Kruskal
    arestaLista.append([peso, origem, destino])

# Aplicando Prim para MST
mst, custo = prim(1, numVertices, adjLista)

# Resultado do Prim
print("##############################")
print("Algoritmo de Prim")
mostraMST(mst, custo)

# Aplicando Kruskal para MST
mst, custo = kruskal(numArestas, arestaLista)

# Resultado do Kruskal
print("Algoritmo de Kruskal")
mostraMST(mst, custo)
```

```
##############################
Algoritmo de Prim
Árvore Geradora Mínima:

[5, 2, 2]
[4, 3, 1]
[1, 4, 6]
[4, 5, 5]

Custo: 14
##############################
Algoritmo de Kruskal
Árvore Geradora Mínima:

[3, 4, 1]
[5, 2, 2]
[4, 5, 5]
[1, 4, 6]

Custo: 14
##############################
```

## Exemplo 3

Número de Vértices = 4

Número de Arestas = 6

Grafo 3

{2, 1, 6}

{3, 1, 5}

{3, 4, 8}

{4, 1, 4}

{3, 2, 1}

{1, 4, 4}

In [8]:

```
##############################################################################
# Main
##############################################################################
numVertices, numArestas = input().split()

numVertices = int(numVertices)  # Número de Vértices
numArestas  = int(numArestas)   # Número de Arestas
```

```
# Prim com Lista de Adjacência
adjLista = [[] for i in range(numVertices + 1)]

# Kruskal com Lista de Arestas Adjacentes
arestaLista = []
link = [i for i in range(numVertices + 1)]
tamanho = [1 for i in range(numVertices + 1)]

# Leitura do Grafo
for i in range(numArestas):
    origem, destino, peso = input().split()

    origem = int(origem)
    destino = int(destino)
    peso = int(peso)

    # Inserção no Prim
    adjLista[origem].append([destino, peso])
    adjLista[destino].append([origem, peso])

    # Inserção no Kruskal
    arestaLista.append([peso, origem, destino])

# Aplicando Prim para MST
mst, custo = prim(1, numVertices, adjLista)

# Resultado do Prim
print("##############################")
print("Algoritmo de Prim")
mostraMST(mst, custo)

# Aplicando Kruskal para MST
mst, custo = kruskal(numArestas, arestaLista)

# Resultado do Kruskal
print("Algoritmo de Kruskal")
mostraMST(mst, custo)
```

```
##############################
Algoritmo de Prim
Árvore Geradora Mínima:

[3, 2, 1]
[1, 3, 5]
[1, 4, 4]

Custo: 10
##############################
Algoritmo de Kruskal
Árvore Geradora Mínima:

[3, 2, 1]
[1, 4, 4]
[3, 1, 5]

Custo: 10
##############################
```

## Exemplo 4

Número de Vértices = 6

Número de Arestas = 4

Grafo 4

{4, 1, 2}

{4, 3, 1}

{4, 5, 3}

{3, 1, 8}

In [9]:

```
##########################################################################################
# Main
##########################################################################################
numVertices, numArestas = input().split()

numVertices = int(numVertices)  # Número de Vértices
numArestas  = int(numArestas)   # Número de Arestas

# Prim com Lista de Adjacência
adjLista = [[] for i in range(numVertices + 1)]

# Kruskal com Lista de Arestas Adjacentes
arestaLista = []
link = [i for i in range(numVertices + 1)]
tamanho = [1 for i in range(numVertices + 1)]

# Leitura do Grafo
for i in range(numArestas):
    origem, destino, peso = input().split()

    origem = int(origem)
    destino = int(destino)
    peso = int(peso)

    # Inserção no Prim
    adjLista[origem].append([destino, peso])
    adjLista[destino].append([origem, peso])

    # Inserção no Kruskal
    arestaLista.append([peso, origem, destino])

# Aplicando Prim para MST
mst, custo = prim(1, numVertices, adjLista)

# Resultado do Prim
print("##############################")
print("Algoritmo de Prim")
mostraMST(mst, custo)

# Aplicando Kruskal para MST
mst, custo = kruskal(numArestas, arestaLista)

# Resultado do Kruskal
print("Algoritmo de Kruskal")
mostraMST(mst, custo)
```

```
##############################
Algoritmo de Prim
Árvore Geradora Mínima:

[4, 3, 1]
[1, 4, 2]
[4, 5, 3]

Custo: 6
##############################
Algoritmo de Kruskal
Árvore Geradora Mínima:

[4, 3, 1]
[4, 1, 2]
[4, 5, 3]
```

```
Custo: 6
##############################
```

## Exemplo 5

Número de Vértices = 6

Número de Arestas = 6

Grafo 5

{5, 2, 6}

{4, 1, 5}

{6, 4, 5}

{6, 5, 2}

{1, 2, 9}

{3, 5, 7}

In [10]:

```
################################################################################
# Main
################################################################################
numVertices, numArestas = input().split()

numVertices = int(numVertices)  # Número de Vértices
numArestas  = int(numArestas)   # Número de Arestas

# Prim com Lista de Adjacência
adjLista = [[] for i in range(numVertices + 1)]

# Kruskal com Lista de Arestas Adjacentes
arestaLista = []
link = [i for i in range(numVertices + 1)]
tamanho = [1 for i in range(numVertices + 1)]

# Leitura do Grafo
for i in range(numArestas):
    origem, destino, peso = input().split()

    origem = int(origem)
    destino = int(destino)
    peso = int(peso)

    # Inserção no Prim
    adjLista[origem].append([destino, peso])
    adjLista[destino].append([origem, peso])

    # Inserção no Kruskal
    arestaLista.append([peso, origem, destino])

# Aplicando Prim para MST
mst, custo = prim(1, numVertices, adjLista)

# Resultado do Prim
print("##############################")
print("Algoritmo de Prim")
mostraMST(mst, custo)

# Aplicando Kruskal para MST
mst, custo = kruskal(numArestas, arestaLista)

# Resultado do Kruskal
print("Algoritmo de Kruskal")
mostraMST(mst, custo)
```

```
##############################
```

```
Algoritmo de Prim
Árvore Geradora Mínima:

[5, 2, 6]
[5, 3, 7]
[1, 4, 5]
[6, 5, 2]
[4, 6, 5]

Custo: 25
##############################
Algoritmo de Kruskal
Árvore Geradora Mínima:

[6, 5, 2]
[4, 1, 5]
[6, 4, 5]
[5, 2, 6]
[3, 5, 7]

Custo: 25
##############################
```

---

# 5. Considerações Finais

O objetivo deste trabalho foi realizar uma comparação acerca dos dois principais algoritmos de cálculo da chamada Árvore Geradora Mínima, por meio de cinco testes com grafos diferentes gerados automaticamente. Além disso, um ponto importante abordado foi a complexidade de ambos os algoritmos para grafos do tipo de Lista de Adjacência e que serviu como base na observação de cada teste.

Diante disso, cada algoritmo foi implementado na linguagem de programação Python 3, com intuito de facilitar a manipulação dos dados e das funções necessárias para a realização dos testes. Também utilizou-se uma ferramenta para a geração dos grafos de testes, como consta no início do capítulo 4 deste trabalho.

Tendo em vista os algoritmos e seus objetivos, isto é, calcular a MST, pode-se elencar as seguintes principais diferenças entre ambos:

- A complexidade temporal de Prim é melhor do que a do Kruskal, desde que utilizada uma ***Heap de Fibonacci***, transformando de **O(A log V)** para **O(A + V log V)**, tendo *A* como número de arestas e *V* como número de vértices do grafo na forma de Lista de Adjacência.
- O algoritmo de Prim é iniciado com a escolha de um vértice, enquanto que o de Kruskal é iniciado na escolha de uma aresta ordenada. Dessa forma, Prim necessita da existência de um grafo conexo, enquanto Kruskal pode funcionar em grafos desconexos.
- Prim gera uma árvore única ao decorrer de sua execução, ao passo que Kruskal gera uma floresta e somente tem-se a garantia de uma árvore após a última iteração.

Portanto, infere-se que o algoritmo de Prim é mais eficiente quando há um **conjunto de dados muito grande (grafos densos)**; enquanto que o algoritmo de Kruskal é mais recomendado para um **conjunto de dados menores (grafos esparsos)**, pois utiliza estrutura de dados mais simples (*Disjoint-Set*).