

# Algoritmo Minimum Spanning Tree (MST)

Rafael G. Nagel

2018-06-22

# Outline

Introdução

Explicação do algoritmo (Kruskal)

Código

Conclusões

# Aplicações<sup>1</sup> e problema

- ▶ Geralmente aplicações associadas a otimização em **redes**
  - ▶ telefone
  - ▶ elétrica
  - ▶ hidráulica
  - ▶ TV a cabo
  - ▶ caminhos
- ▶ Outras aplicações indiretas
  - ▶ aprendizagem de características para verificação facial em tempo real (?)
  - ▶ caminhos de máximo gargalos

---

<sup>1</sup>[https:](https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/)

# Condições para aplicação do algoritmo

- ▶ grafo com arestas de pesos positivos ( $> 0$ )
- ▶ arestas não-direcionadas
- ▶ encontrar um conjunto de *arestas* que somadas tenha um **peso mínimo** e que conecte **todos os vértices**
- ▶ número de *arestas* que se deseja:

$$\text{arestas} = \text{número de vértices} - 1$$

# Algoritmos que implementam MST<sup>2</sup>

1. algoritmo de Kruskal
2. algoritmo de Prim
3. algoritmo de Boruvka

- 
- ▶ por que Kruskal?
    - ▶ mais fácil de aplicar e compreender

---

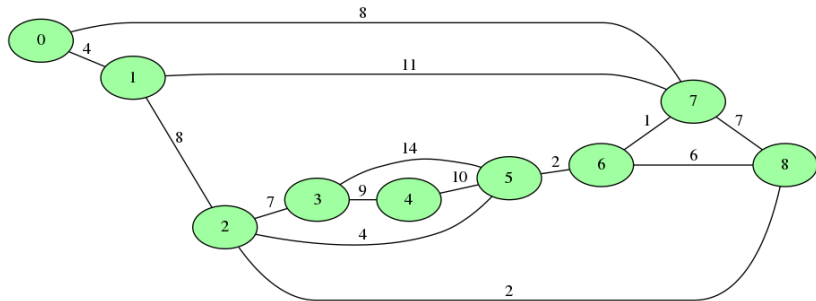
<sup>2</sup><https://www.ics.uci.edu/~eppstein/161/960206.html>

# Visão geral

1. ordena-se as arestas por peso *crescente*
2. pega-se uma aresta e verifica-se se ao adicioná-la no grafo final (MST) um **ciclo** será formado
  - ▶ se ciclo formado, então descarta-se a aresta (do MST)
  - ▶ se não forma ciclo, adicione-a ao grafo (MST)
3. repita o passo 2. até:  $\text{arestas} = \text{vértices} - 1$



## Passo a passo com um exemplo<sup>4</sup>



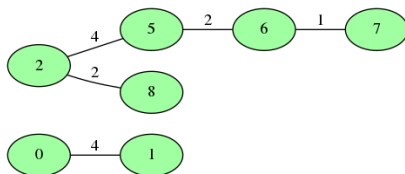
<sup>4</sup><https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/>



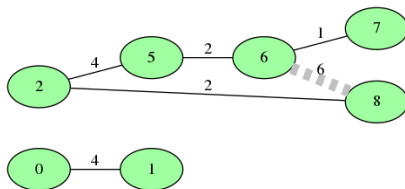
## Grafo ordenado por arestas (crescente)

weight	src	dst
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

pega-se os vértices da lista ordenada e tenta-se inserir cada um

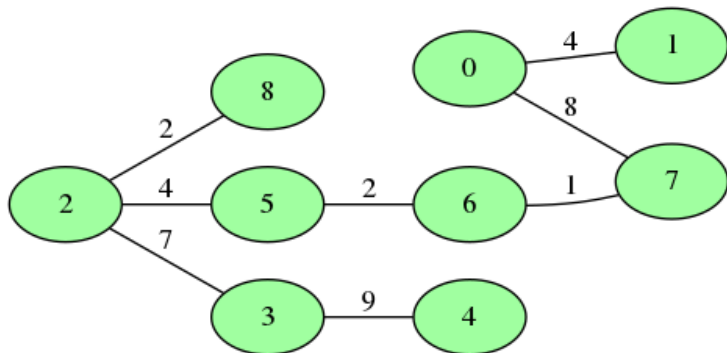


... sem ciclos até agora.



... ao incluir 8-6 forma-se um ciclo. Portanto não inclua essa aresta.

repita até: arestas = vértices - 1



# Complexidade

relembrando os passos:

1. ordenar grafo crescentemente (qsort  $\rightarrow O(n \log n)$ )
2. aplicar algoritmo MST (Kruskal)
  - 2.1 para *cada aresta* da lista ordenada:
    - 2.1.1 inclua-a no grafo MST
    - 2.1.2 checa-se se *ciclo formado*; se sim remove essa aresta (algoritmo find-union)

complexidade do algoritmo Kruskal

- ▶ O *find-union* é  $O(n)$  no *presente trabalho*
  - ▶ poderíamos melhorar isso para  $O(\log n)$  usando *union by Rank or Height*<sup>5</sup>

---

<sup>5</sup><https://www.geeksforgeeks.org/union-find/>

# Complexidade

## Considerar o pior caso

- ▶ Toda vez que um vértices é adicionado no grafo, podemos ter:

$$\text{arestas} + 1 = \text{número de vértices} - 1$$

- ▶ Assim, número de arestas no pior caso é:

$$a \approx v^2$$

quando  $a \rightarrow \infty$

Porém, essa implementação:

$$O(e \times \log e) + O(e \times O(v))$$

$$O(e \times \log e + v^2 \times v)$$
$$O(e \times \log v + v^3)$$

# Complexidade

Supondo a implementação com: union-find =  $O(\log n)$

$$O(e \times \log e) + O(e \times \log v)$$

$$e = v^2 \rightarrow \log e = \log v^2 = 2 \times \log v \approx \log v$$

$$\begin{aligned} & \quad \quad \quad \vdots \\ O(e \times (\log v + \log v)) &= O(e \times 2 \times \log v) \\ & \quad \quad \quad \vdots \\ & O(e \times \log v) \end{aligned}$$

# Código

## Estruturas

- ▶ grafo
- ▶ vertice
  - ▶ `vertice_t`
  - ▶ `*vertice_added_in_mst`
- ▶ lista
- ▶ nó

## funções (extras)

- ▶ `hasCycle()`
- ▶ `union()`
- ▶ `find()`
- ▶ `grafo_remove_ultima_aresta()`
- ▶ `compara_arestas()`
  - ▶ usada no *qsort* da *lib c*
- ▶ `get_vertice_added_in_mst`
- ▶ outras menos importante
  - ▶ e.g. `grafo_get_arestas_arr()`

# Conclusões

- ▶ Ao invés de criar novas *estruturas* ou *módulos*, adicionou-se *membros* às estruturas e novas *funções* aos módulos.

## Como checar **adição duplicada de vértices** sem ser $O(n)$ ?

- ▶ Essa questão aparece quando:
  - ▶ adiciona-se arestas ao gráfico (MST) para checar se há *ciclos*
  - ▶ ao criar novo vértice, verifica-se se há vértice duplicado (removido)
- ▶ Assim, essa complexidade teria mais um fator  $v$ , caso não houvesse um *workaround* para checagem de duplicidade:

$$O(e \times \log v + v^4)$$

Por fim: algoritmo Kruskal é  *muito mais fácil*  de implementar:

- ▶ Tradeoffs: Complexidade vs. Tempo de código