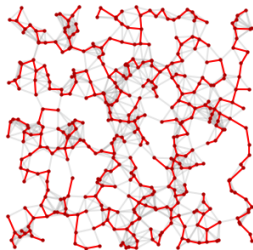


# Árvores de Suporte de Custo Mínimo

Pedro Ribeiro

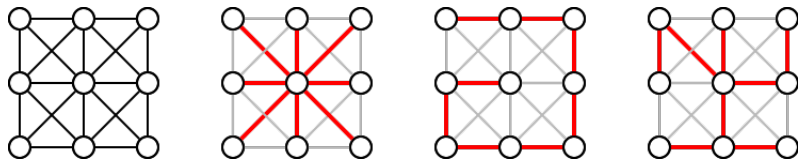
DCC/FCUP

2020/2021



# Árvore de Suporte

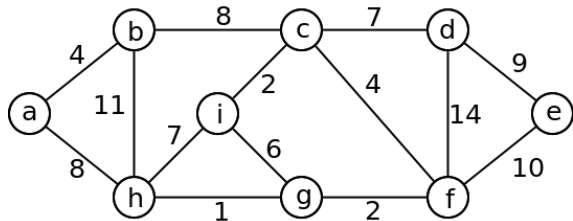
- Uma **árvore de suporte** ou **árvore de extensão** (*spanning tree*) é um subconjunto das arestas de um grafo não dirigido que forma uma árvore ligando todos os vértices.
- A figura seguinte ilustra um grafo e 3 árvores de suporte:



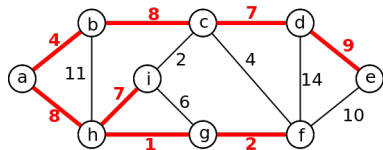
- Podem existir várias árvores de suporte para um dado grafo
- Uma árvore de suporte de um grafo terá sempre  $|V| - 1$  arestas
  - ▶ Se tiver menos arestas, não liga todos os nós
  - ▶ Se tiver mais arestas, forma um ciclo

# Árvore de Suporte de Custo Mínimo

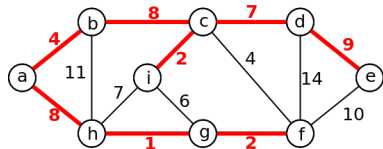
- Se o grafo for pesado (tem valores associados às arestas), existe a noção de **árvore de suporte de custo mínimo** (*minimum spanning tree* - MST), que é a árvore de suporte cuja soma dos pesos das arestas é a menor possível.
- A figura seguinte ilustra um grafo não dirigido e pesado. Qual é a sua árvore de suporte de custo mínimo?



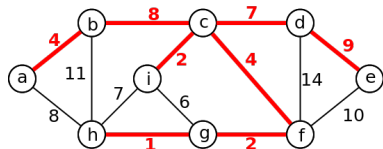
# Árvore de Suporte de Custo Mínimo



Custo total: 46 = 4+8+7+9+8+7+1+2



Custo total: 41 = 4+8+7+9+8+2+1+2



Custo total: 37 = 4+8+7+9+1+2+4+2

E de facto esta última é uma árvore de suporte de custo mínimo!

# Árvore de Suporte de Custo Mínimo

- Pode existir **mais do que uma MST**.
  - ▶ Por exemplo, no caso dos pesos serem todos iguais, qualquer árvore de suporte tem custo mínimo!
- Em termos de **aplicações**, a MST é muito útil. Por exemplo:
  - ▶ Quando queremos ligar computadores em rede gastando a mínima quantidade de cabo.
  - ▶ Quando queremos ligar casas à rede de electricidade gastando o mínimo possível de fio.
- Como **descobrir uma MST** para um dado grafo?
  - ▶ Existe um número exponencial de árvores de suporte
  - ▶ Procurar todas as árvores possíveis e escolher a melhor não é eficiente!
  - ▶ Como fazer melhor?

# Algoritmos para Calcular MST

- Vamos falar essencialmente de dois algoritmos diferentes: **Prim** e **Kruskal**
- Ambos os algoritmos são **greedy**: em cada passo adicionam uma nova aresta tendo o cuidado de garantir que as arestas já selecionadas são parte de uma MST

## Algoritmo Genérico para MST

$A \leftarrow \emptyset$

**Enquanto**  $A$  não forma uma MST **fazer**

    Descobrir uma aresta  $(u, v)$  que é "segura" para adicionar

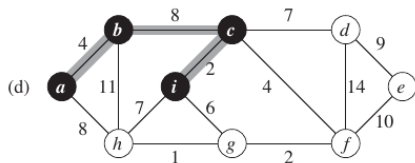
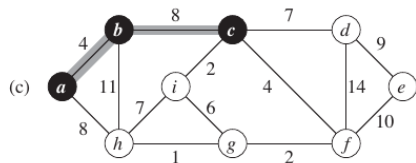
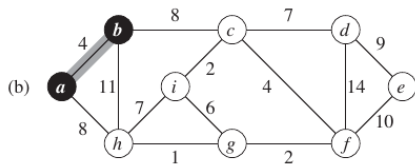
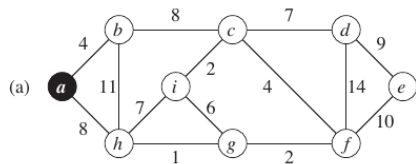
$A \leftarrow A \cup (u, v)$

**retorna**( $A$ )

# Algoritmo de Prim

- Começar num qualquer nó
- Em cada passo **adicionar à árvore já formada o nó cujo custo seja menor** (que tenha aresta de menor peso a ligar à árvore). Em caso de empate qualquer um funciona.  
(“árvore já formada” são os nós já adicionados anteriormente)
- Vamos ver passo a passo para o grafo anterior...

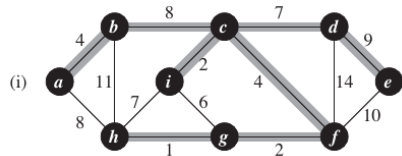
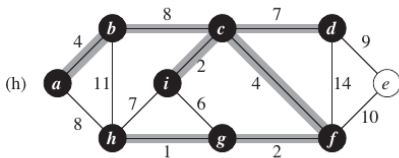
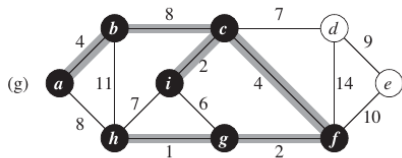
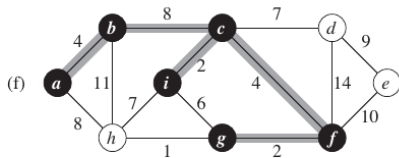
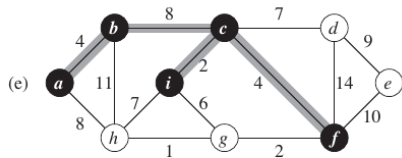
# Algoritmo de Prim



(imagem de *Introduction to Algorithms, 3rd Edition*)



# Algoritmo de Prim



(imagem de Introduction to Algorithms, 3rd Edition)

# Algoritmo de Prim

Vamos operacionalizar isto em código:

(notem as semelhanças, mas também as diferenças, em relação ao algoritmo de Dijkstra)

## Algoritmo de Prim para descobrir MST de $G$ (começar no nó $r$ )

Prim( $G, r$ ):

**Para** todos os nós  $v$  de  $G$  **fazer**:

$v.dist \leftarrow \infty$

$v.pai \leftarrow NULL$

$r.dist \leftarrow 0$

$Q \leftarrow G.V$  /\* Todos os vértices de  $G$  \*/

**Enquanto**  $Q \neq \emptyset$  **fazer**

$u \leftarrow \text{EXTRAIR-MINIMO}(Q)$  /\* Nó com menor  $dist$  \*/

**Para** todos os nós  $v$  adjacentes a  $u$  **fazer**

**Se**  $v \in Q$  e  $\text{peso}(u, v) < v.dist$  **então** /\* Actualizar distâncias \*/

$v.pai \leftarrow u$

$v.dist \leftarrow \text{peso}(u, v)$

# Algoritmo de Prim - Complexidade

- Qual a **complexidade** do algoritmo de Prim?
  - ▶ No início fazemos  $\mathcal{O}(|V|)$  inicializações
  - ▶ Depois fazemos:
    - ★  $\mathcal{O}(|V|)$  escolhas de nós mínimos (*EXTRAIR-MINIMO*)
    - ★  $\mathcal{O}(|E|)$  atualizações de distâncias ("*relaxamentos*")
- Gastamos  $\mathcal{O}(|V| + |V| \times \text{EXTRAIR-MINIMO} + |E| \times \text{relaxamento})$  [assumindo o uso de uma lista de adjacências]
- Consideremos uma implementação *naive* com **um ciclo para descobrir a distância mínima**
  - ▶ um EXTRAIR-MINIMO custaria  $\mathcal{O}(|V|)$  [ciclo pelos nós]
  - ▶ um relaxamento custaria  $\mathcal{O}(1)$  [atualizar distância]

Ficaríamos com complexidade total  $\mathcal{O}(|V| + |V|^2 + |E|)$ . Como o número de arestas é no máximo  $|V|^2$ , a complexidade pode ser simplificada para  $\mathcal{O}(|V|^2)$ .

Como melhorar?

# Algoritmo de Prim - Complexidade

- Prim:  $\mathcal{O}(|V| + |V| \times \text{EXTRAIR-MINIMO} + |E| \times \text{relaxamento})$   
[assumindo o uso de uma lista de adjacências]
- Consideremos uma implementação com uma **fila de prioridade** (ex: uma **min-heap**)
  - ▶ um EXTRAIR-MINIMO custaria  $\mathcal{O}(\log |V|)$   
[retirar elemento da fila de prioridade]
  - ▶ um relaxamento custaria  $\mathcal{O}(\log |V|)$   
[atualizar prioridade fazendo elemento "subir" na heap]

Ficaríamos no total com  $\mathcal{O}(|V| + |V| \times \log |V| + |E| \times \log |V|)$ .  
Assumindo que  $|E| \geq |V|$ , a parte dos relaxamentos vai dominar o tempo e a complexidade pode ser simplificada para  $\mathcal{O}(|E| \log |V|)$ .

(isto é em tudo semelhante ao Dijkstra: no fundo só muda o que é feito no "relaxamento" de uma aresta)

# Algoritmo de Prim e Filas de Prioridade

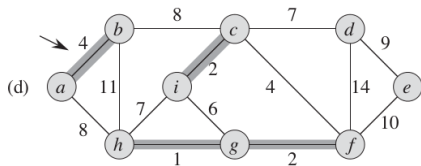
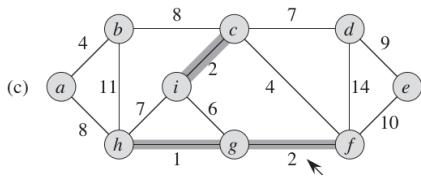
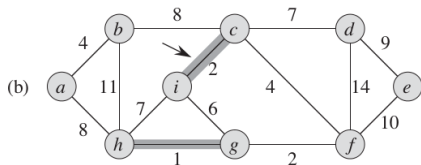
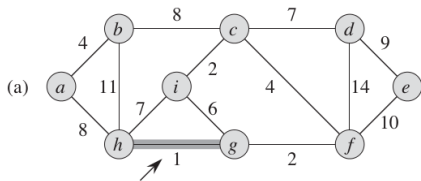
(isto já foi explicado para o algoritmo de Dijkstra, mas fica aqui novamente para o caso de estarem a ver só este capítulo)

- As linguagens de programação tipicamente trazem já disponível uma fila de prioridade que garante complexidade logarítmica para **inserção** de um novo valor e **remoção do mínimo**:
  - ▶ **C++**: `priority_queue` / **Java**: `PriorityQueue`
- Estas implementações não trazem tipicamente a parte de actualizar um valor (nem a hipótese de retirar um valor no meio da fila).
- Três possíveis hipóteses para lidar com actualização de valor:
  - 1 Usar heap "manualmente" (complexidade final:  $\mathcal{O}(|E| \log |V|)$ ) (podemos chamar *up\_heap* em qualquer nó no meio da fila())
  - 2 Usar uma *PriorityQueue* e actualizar ser feito via inserção de novo elemento na heap com a nova distância (ignorar depois nó "repetido") (cada nó será inserido no máximo tantas vezes quanto o seu grau)  
Complexidade final:  $\mathcal{O}(|E| \log |E|)$  ou
  - 3 Usamos uma BST (ex: um *set*) e actualizar ser feito via remoção + inserção (ambas as operações em tempo logarítmico)  
Complexidade final:  $\mathcal{O}(|E| \log |V|)$

# Algoritmo de Kruskal

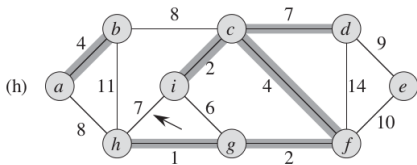
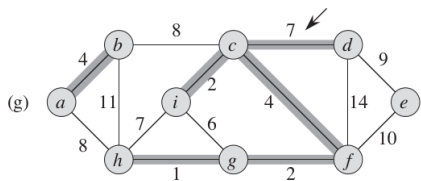
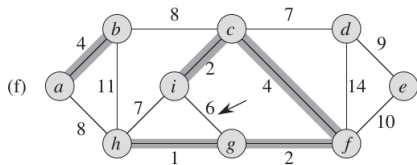
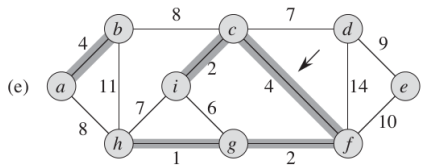
- Manter uma floresta (conjunto de árvores), onde no início cada nó é uma árvore isolada e no final todos os nós fazem parte da mesma árvore
- Ordenar as arestas por ordem crescente de peso
- Em cada passo **selecionar a aresta de menor valor que ainda não foi testada** e, caso esta aresta junte duas árvores ainda não "ligadas", então juntar a aresta, combinando as duas árvores numa única árvore.
- Vamos ver passo a passo para o grafo anterior...

# Algoritmo de Kruskal



(imagem de Introduction to Algorithms, 3rd Edition)

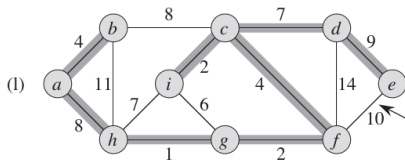
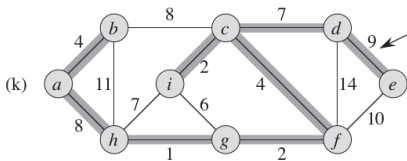
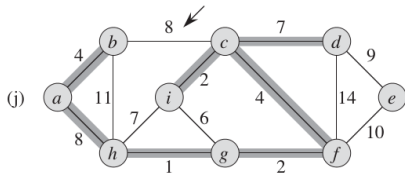
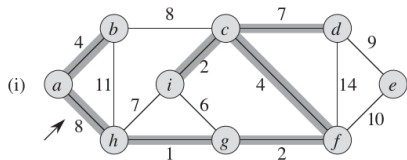
# Algoritmo de Kruskal



(imagem de *Introduction to Algorithms, 3rd Edition*)



# Algoritmo de Kruskal



(imagem de Introduction to Algorithms, 3rd Edition)

# Algoritmo de Kruskal

Vamos operacionalizar isto em código:

## Algoritmo de Kruskal para descobrir MST de $G$

Kruskal( $G$ ):

$A \leftarrow \emptyset$

**Para** todos os nós  $v$  de  $G$  **fazer**:

MAKE-SET( $v$ ) /\* criar árvore para cada nó \*/

Ordenar arestas de  $G$  por ordem crescente de peso

**Para** cada aresta  $(u, v)$  de  $G$  **fazer**: /\* segue ordem anterior \*/

**Se** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) **então** /\* estão em árvores dif. \*/

$A \leftarrow A \cup \{(u, v)\}$

UNION( $u, v$ ) /\* juntar duas árvores \*/

**retorna**( $A$ )

- MAKE-SET( $v$ ): criar conjunto apenas com  $v$
- FIND-SET( $v$ ): descobrir qual o conjunto de  $v$
- UNION( $u, v$ ): unir os conjuntos de  $u$  e  $v$

# Algoritmo de Kruskal - Complexidade

- Para além da ordenação, a complexidade do algoritmo de Kruskal depende das operações MAKE-SET, FIND-SET e UNION
  - ▶ Vamos chamar MAKE-SET no início  $|V|$  vezes
  - ▶ Cada aresta vai levar a duas chamadas a FIND-SET e potencialmente a uma chamada a UNION
- Uma implementação "naive" em que um conjunto é mantido numa lista, daria um MAKE-SET com  $\mathcal{O}(1)$  (criar lista com o nó), um FIND-SET com  $\mathcal{O}(|V|)$  (procurar lista com elemento) e um UNION com  $\mathcal{O}(1)$  (juntar duas filas é só fazer o apontador do último nó de uma lista apontar para o início do primeiro nó da outra lista.) Isto daria uma complexidade de  $\mathcal{O}(|E| \times |V|)$
- Se mantivermos um atributo auxiliar para cada nó dizendo qual o conjunto onde está, podemos fazer o FIND-SET em  $\mathcal{O}(1)$ , mas o UNION passa a custar  $\mathcal{O}(|V|)$  (mudar esse atributo para os nós de uma das listas a ser unida), pelo que a complexidade final não melhora.

# Union-Find

- É possível reduzir para um **tempo linearítico** se usarmos uma estrutura de dados que suporte estas operações em tempo logarítimico ou constante (supondo que a ordenação demora  $\mathcal{O}(n \log n)$ )
- Uma estrutura de dados para esta função (manter conjuntos, suportando as operações FIND-SET e UNION) é conhecida como **union-find**, e uma boa maneira de a implementar é usando **florestas de conjuntos disjuntos**.
  - ▶ Cada conjunto é representando por uma árvore
  - ▶ Cada nó guarda uma referência para o seu pai
  - ▶ O representante de um conjunto é o nó raiz da árvore do conjunto

# Union-Find

Uma maneira "naive" de implementar florestas de conjuntos disjuntos:

## Naive UNION-FIND

MAKE-SET( $x$ ):

$x.pai \leftarrow x$  /\* Raiz aponta para ela própria \*/

FIND( $x$ ):

**Se**  $x.pai = x$  **então retorna**  $x$

**Senão retorna** FIND( $x.pai$ )

UNION( $x, y$ ):

$xRaiz \leftarrow FIND(x)$

$yRaiz \leftarrow FIND(y)$

$xRaiz.pai \leftarrow yRaiz$

- Com esta implementação podemos continuar a ter tempo linear por operação porque as árvores podem ficar desequilibradas (e com altura igual ao número de nós). Para melhorar vamos usar duas coisas...

# Union-Find - Melhoria "Union by Rank"

- **Union by Rank** - Juntar sempre a árvore mais pequena à árvore maior quando se faz uma união.
  - ▶ O que queremos é não fazer subir tanto a altura das árvores
  - ▶ Isto garante que a altura das árvores só aumenta se as duas árvores já tiveram altura igual.
- A ideia é manter um atributo **rank** que nos diz essencialmente a altura da árvore.
- Esta melhoria, por si só, já garante uma complexidade logarítmica para os FIND e UNION!

# Union-Find - Melhoria "Union by Rank"

## UNION-FIND com "Union by Rank"

MAKE-SET( $x$ ):

$x.pai \leftarrow x$     $x.rank \leftarrow 0$

UNION( $x, y$ ):

$xRaiz \leftarrow FIND(x)$

$yRaiz \leftarrow FIND(y)$

**Se**  $xRaiz = yRaiz$  **então retorna**

*/\* x e y não estão no mesmo conjunto - temos de os unir \*/*

**Se**  $xRaiz.rank < yRaiz.rank$  **então**

$xRaiz.pai \leftarrow yRaiz$

**Senão, Se**  $xRaiz.rank > yRaiz.rank$  **então**

$yRaiz.pai \leftarrow xRaiz$

**Senão**

$yRaiz.pai \leftarrow xRaiz$

$xRaiz.rank \leftarrow xRaiz.rank + 1$

## Union-Find - Melhoria "Path Compression"

- A segunda melhoria é **comprimir as árvores** ("path compression"), fazendo que todos os nós que um FIND percorre passem a apontar directamente para a raiz, potencialmente diminuindo assim a altura da árvore

### UNION-FIND com "Path Compression"

FIND( $x$ ):

**Se**  $x.pai \neq x$  **então**

$x.pai \leftarrow FIND(x.pai)$

**retorna**  $x.pai$

- Com "union by rank" e "path compression" **o custo amortizado por operação é, na prática, constante** (para mais pormenores espreitar por exemplo o livro desta unidade curricular).
- O tempo para o algoritmo de Kruskal passa a ser dominado... pela ordenação das arestas!



# Algoritmo de Kruskal - Complexidade

- No final de contas, supondo que usamos uma típica ordenação comparativa (com complexidade linearítmica:  $\mathcal{O}(n \log n)$ ) o algoritmo de Kruskal fica então com complexidade  $\mathcal{O}(|E| \log |E|)$
- Notem que  $\mathcal{O}(|E| \log |E|)$  é efetivamente  $\mathcal{O}(|E| \log |V|)$ , pois num grafo simples  $|E| \leq |V|^2$  e  $\log |V|^2 = 2 \times \log |V| = \mathcal{O}(\log |V|)$   
Kruskal e Prim são por isso muito "equivalentes" em termos de complexidade
- O Kruskal poderia ser ainda mais rápido se usamos ordenação não comparativa (ex: se os pesos forem todos pequenos e inteiros podemos usar algo como um *counting sort*)