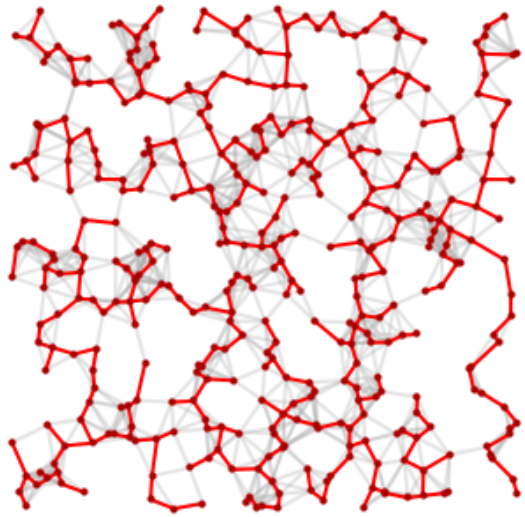
Árvores de Suporte de Custo Mínimo

Pedro Ribeiro

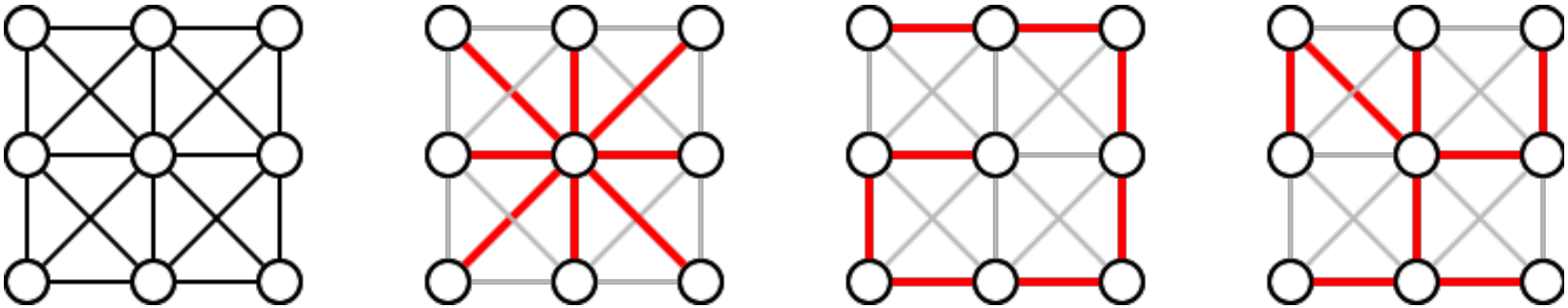
DCC/FCUP

2022/2023



Árvore de Suporte

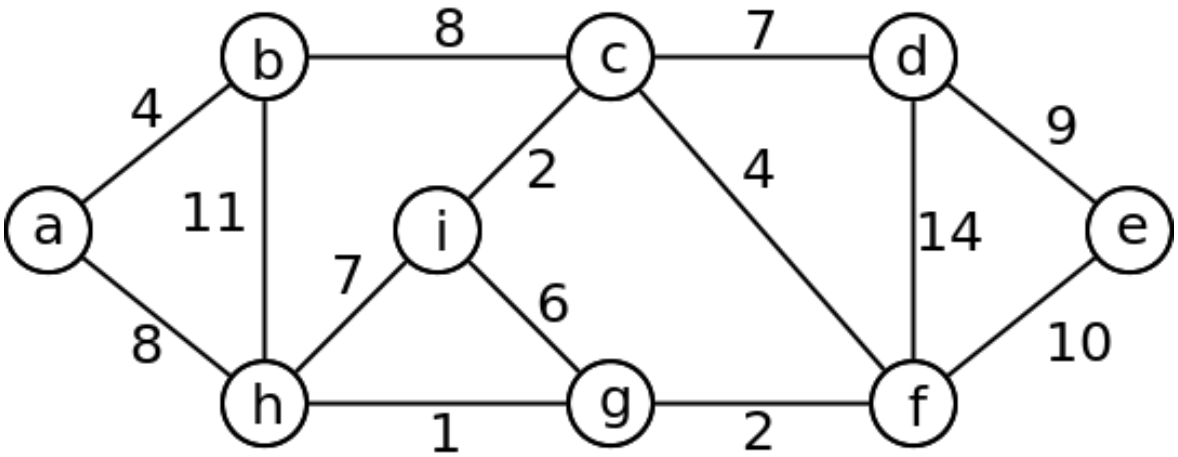
- Uma **árvore de suporte** ou **árvore de extensão** (*spanning tree*) é um subconjunto das arestas de um grafo não dirigido que forma uma árvore ligando todos os vértices.
- A figura seguinte ilustra um grafo e 3 árvores de suporte:



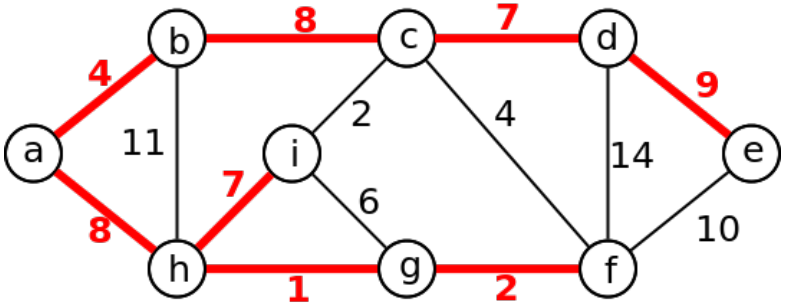
- Podem existir várias árvores de suporte para um dado grafo
- Uma árvore de suporte de um grafo terá sempre $|V| - 1$ arestas
  - Se tiver menos arestas, não liga todos os nós
  - Se tiver mais arestas, forma um ciclo

Árvore de Suporte de Custo Mínimo

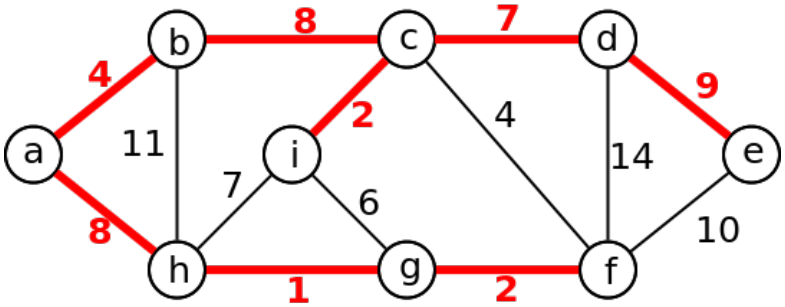
- Se o grafo for pesado (tem valores associados às arestas), existe a noção de **árvore de suporte de custo mínimo** (*minimum spanning tree* - MST), que é a árvore de suporte cuja soma dos pesos das arestas é a menor possível.
- A figura seguinte ilustra um grafo não dirigido e pesado. Qual é a sua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



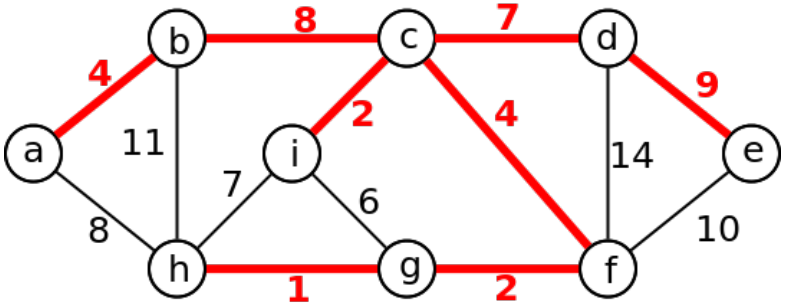
Árvore de Suporte de Custo Mínimo



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

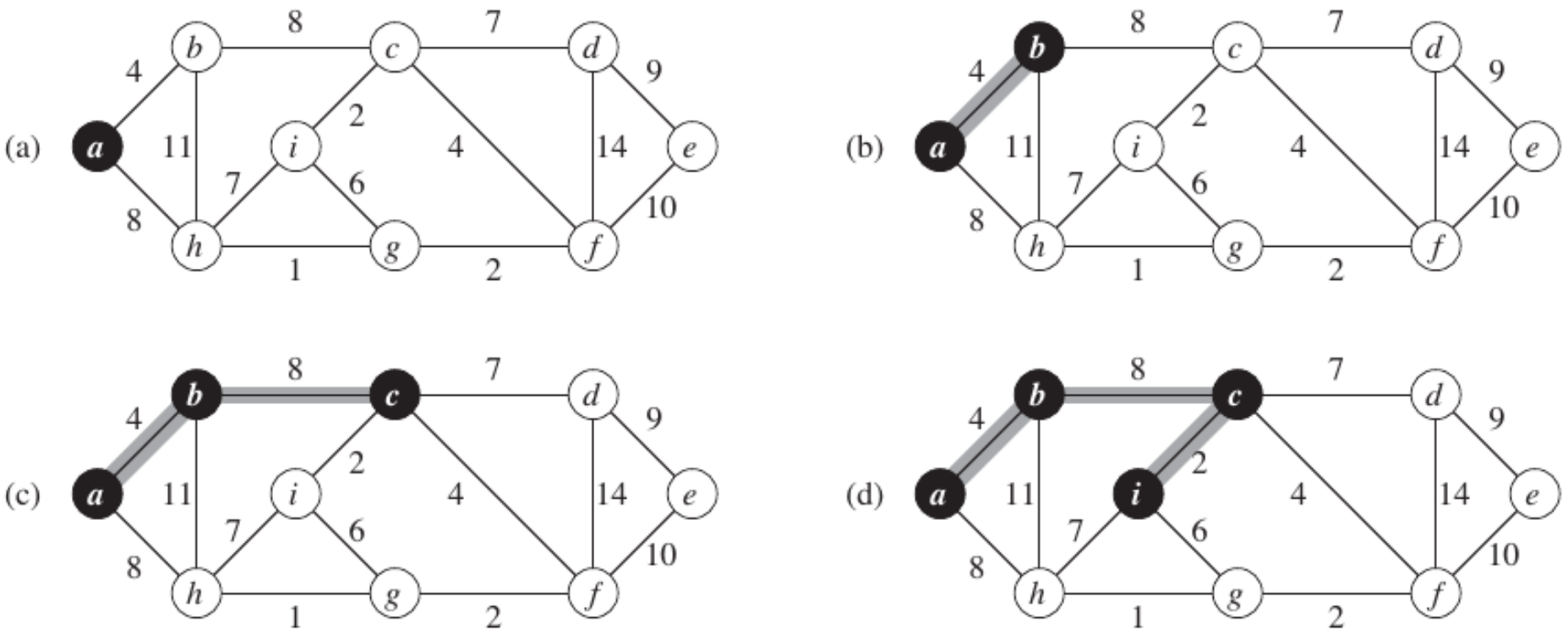
Algoritmo Genérico para MST

$A \leftarrow \emptyset$
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

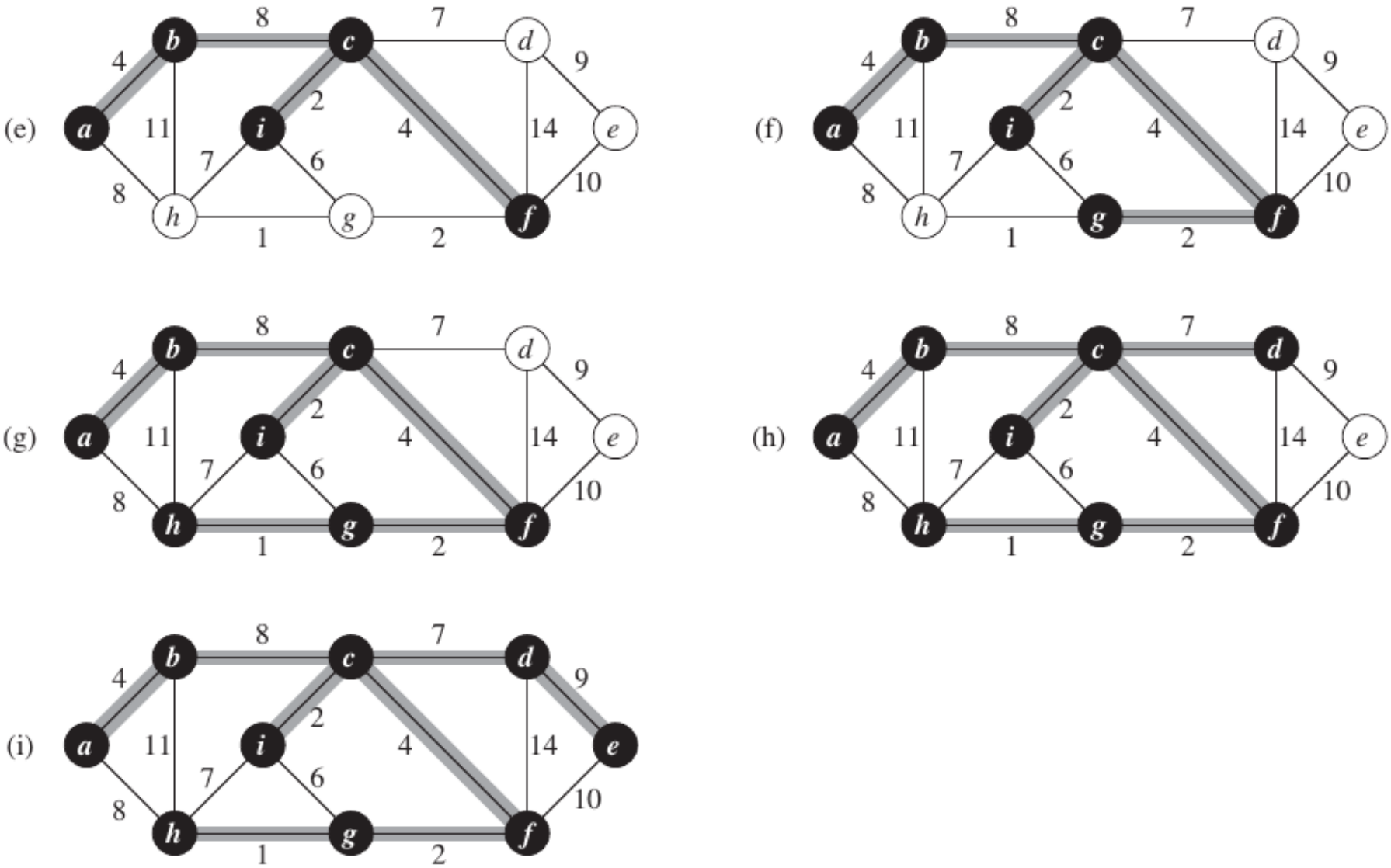
- Começar num qualquer nó
- Em cada passo **adicionar à árvore já formada o nó cujo custo seja menor** (que tenha aresta de menor peso a ligar à árvore). Em caso de empate qualquer um funciona.
  ("árvore já formada" são todos os nós já adicionados anteriormente)
- Vamos ver passo a passo para o grafo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

Algoritmo de Prim



(imagem de Introduction to Algorithms, 3rd Edition)

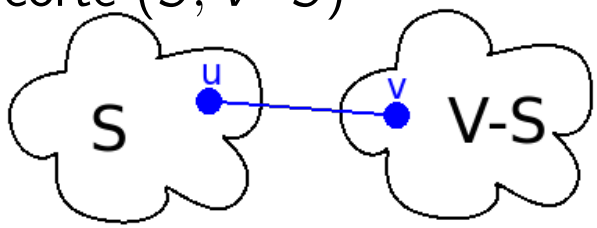
Algoritmo de Prim



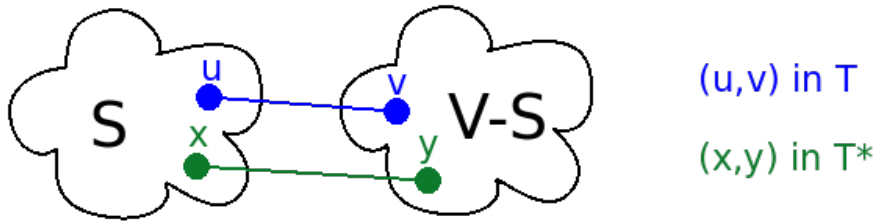
(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Prim - Esboço de uma prova de correção

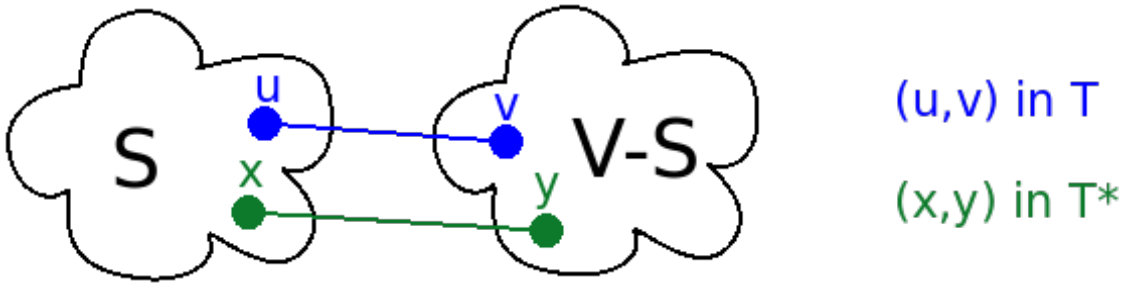
- Seja T uma árvore de suporte descoberta pelo alg. de Prim e T^* uma qualquer MST de G. Queremos mostrar que $custo(T) = custo(T^*)$.
- Se $T = T^*$, então $custo(T) = custo(T^*)$ e temos o desejado
- Caso contrário, $T \neq T^*$, e então $T - T^* \neq \emptyset$.
  - Seja (u, v) uma qualquer aresta de $T - T^*$
  - Quando (u, v) é adicionado a T, era a aresta de menor peso atravessando um corte $(S, V–S)$



  - Como T^* é uma MST, tem de existir um caminho de u para v em T^*
  - Este caminho começa em S e termina em $V - S$; deve existir por isso uma aresta (x, y) ao longo desse caminho onde $x \in S$ e $y \in V - S$



Algoritmo de Prim - Esboço de uma prova de correção



- Como (u, v) é uma aresta de menor peso atravessando o corte $(S, V - S)$, temos que $custo(u, v) \leq custo(x, y)$
- Seja $T^{**} = T^* \cup \{(u, v)\} - \{(x, y)\}$
  - Como (x, y) está num ciclo formado ao adicionar (u, v), isto significa que T^{**} é uma árvore de suporte
  - $custo(T^{**}) = custo(T*) + custo(u, v) - custo(x, y) \leq custo(T*)$
  - Como T^* é uma MST, então $custo(T^{**}) \geq custo(T^*)$
  - Então, $custo(T^*) = custo(T^{**})$.
- Se repetirmos este processo para todas as arestas de $T - T^*$, teremos convertido T^* em T, preservando $custo(T^*)$.
  Então $custo(T) = custo(T^*)$ □

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**
    $u \leftarrow$ EXTRAIR-MINIMO(Q) /* Nó com menor *dist* */
    Para todos os nós v adjacentes a u **fazer**
      Se $v \in Q$ e $peso(u, v) < v.dist$ **então** /* Actualizar distâncias */
        $v.pai \leftarrow u$
        $v.dist \leftarrow peso(u, v)$

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 $\mathcal{O}(|V| + |V| \times$ EXTRAIR-MINIMO $+|E| \times$ *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: $\mathcal{O}(|V| + |V| \times$ EXTRAIR-MINIMO $+|E| \times$ *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

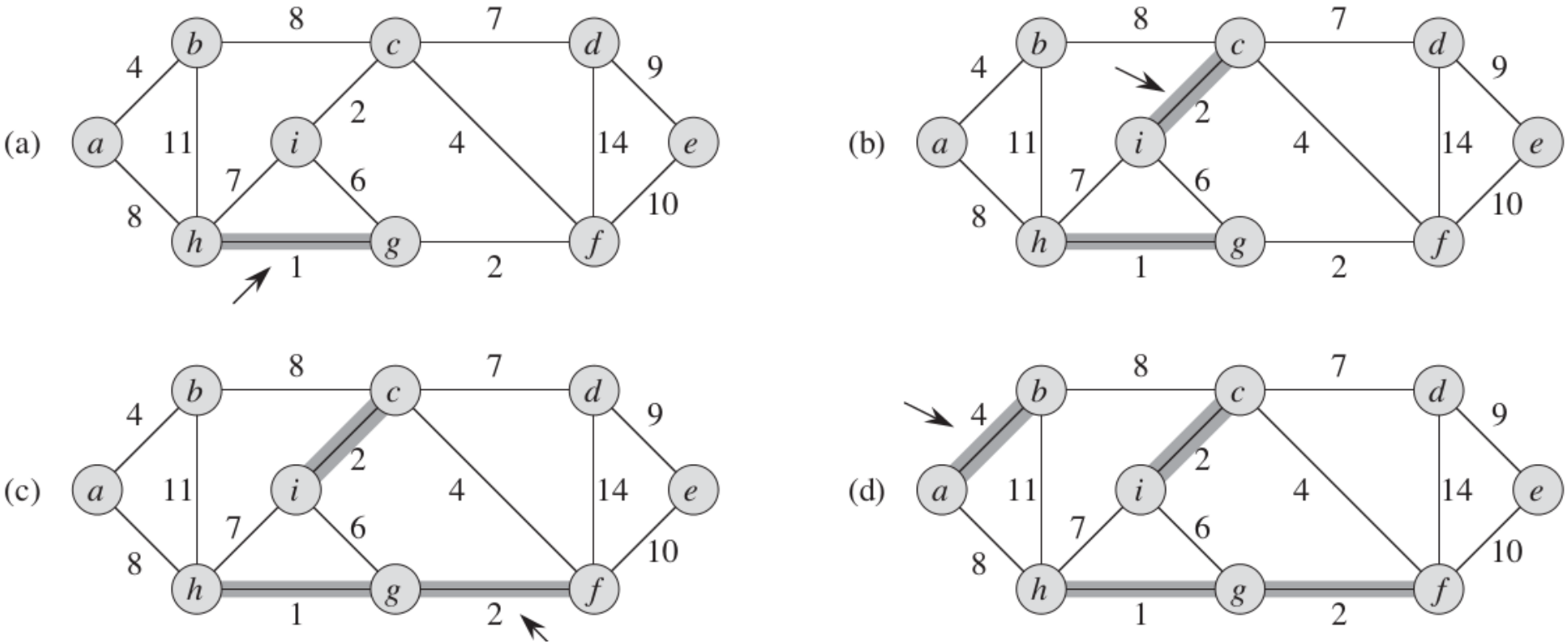
(isto já foi explicado para o algoritmo de Dijkstra, mas fica aqui novamente para o caso de estarem a ver só este capítulo)

- As linguagens de programação tipicamente trazem já disponível uma fila de prioridade que garante complexidade logarítimica 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() ou
  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

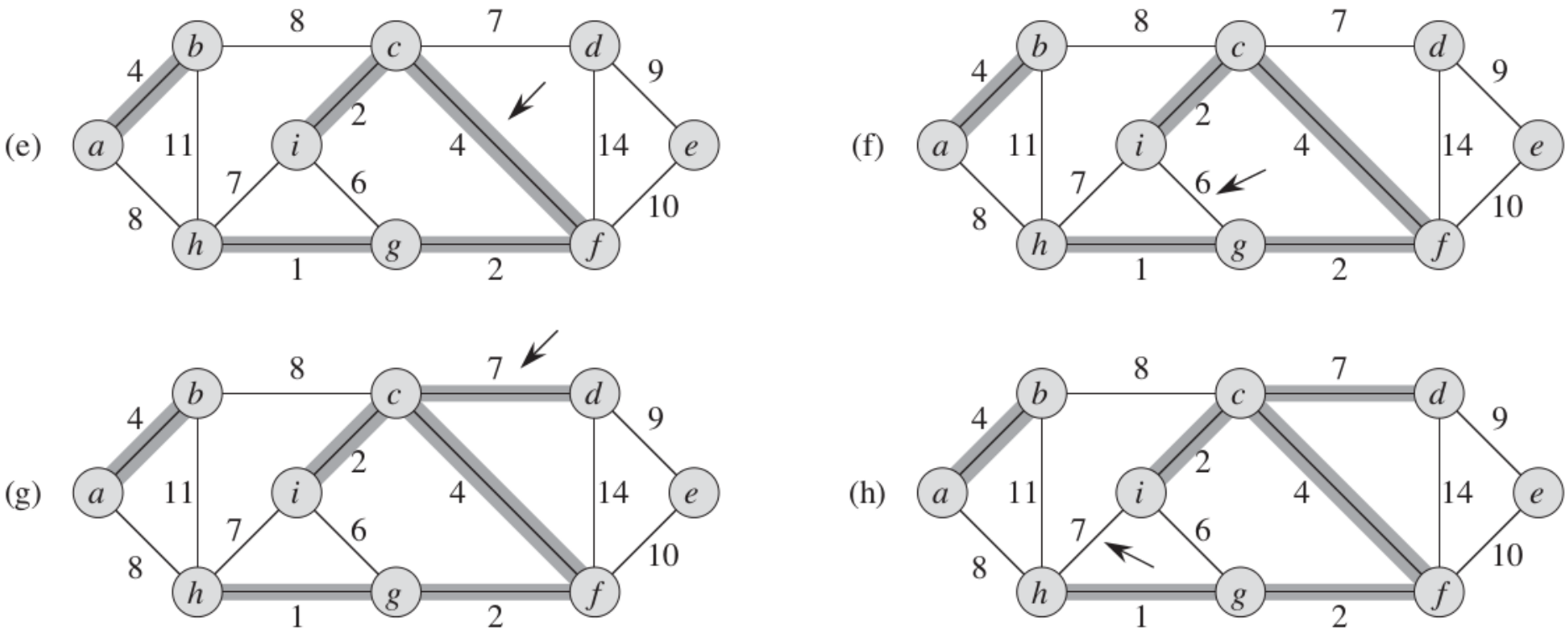
- Manter uma floresta (conjunto de árvores), onde no início cada nó é uma árvore isolada e no final todos os nós fazem parte da mesma árvore
- Ordenar as arestas por ordem crescente de peso
- Em cada passo **selecionar a aresta de menor valor que ainda não foi testada** e, caso esta aresta junte duas 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



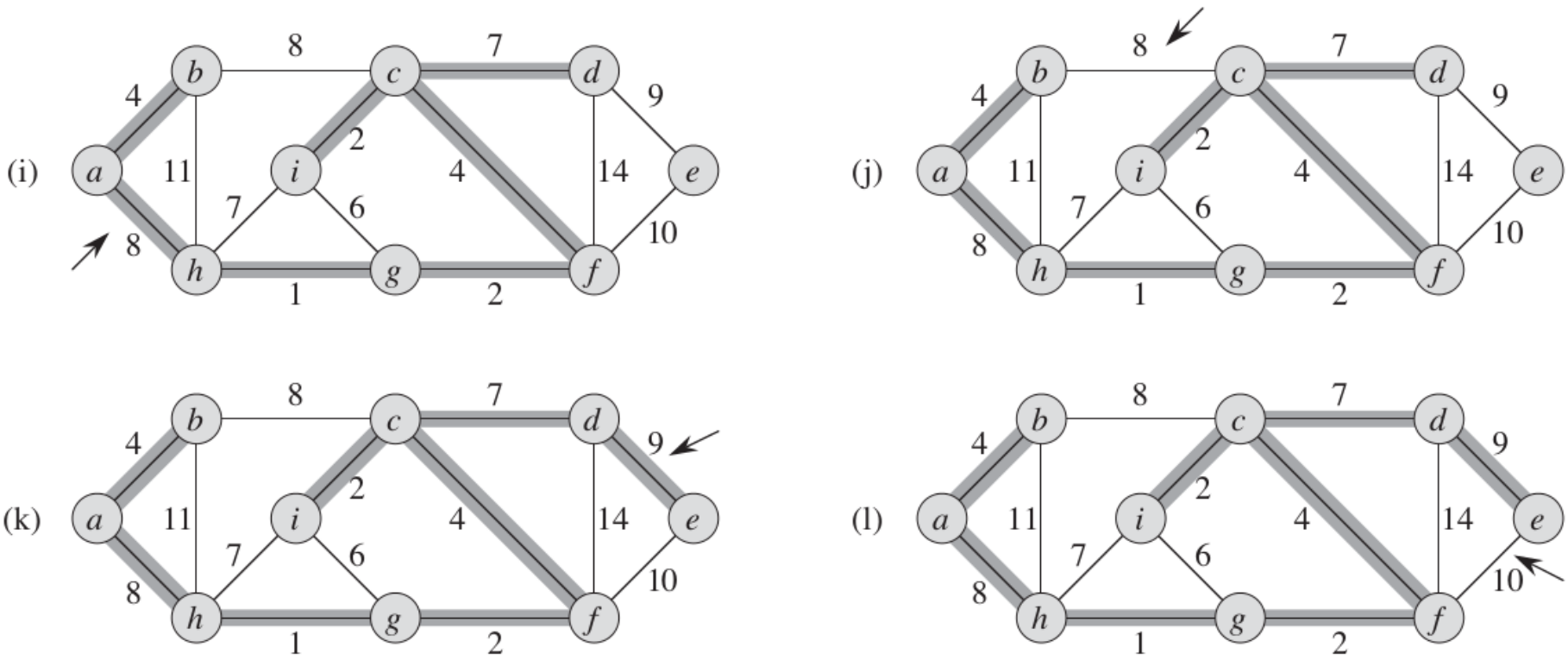
(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Kruskal



(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Kruskal



(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Kruskal - Esboço de uma prova de correção

- Seja T uma árvore de suporte descoberta pelo alg. de Kruskall e T^* uma qualquer MST de G. Queremos mostrar que $custo(T) = custo(T^*)$.
- Se $T = T^*$, então $custo(T) = custo(T^*)$ e temos o desejado
- Caso contrário, $T \neq T^*$:
  - Seja e a aresta de menor peso de T^* que não está em T
  - $T \cup \{e\}$ contém um ciclo C tal que:
    - ★ Qualquer aresta de C tem peso $\leq w(e)$ (pela maneira como o algoritmo construiu T)
    - ★ Existe uma aresta f em C que não está em T^* (porque T^* não contém o ciclo C)
  - Consideremos a árvore $T_2 = T \cup \{e\} - \{f\}$:
    - ★ T_2 é uma árvore de suporte
    - ★ T_2 tem mais arestas em comum com T^* que T
    - ★ $custo(T) \leq custo(T_2)$, dado que $w(f) \leq w(e)$
- Podemos repetir este processo em T_2 para obter T_3 e por aí adiante que obtemos T^* e: $custo(T) \leq custo(T_2) \leq custo(T_3) \leq \ldots \leq custo(T^*)$
- Como T^* é uma MST, então $custo(T) = custo(T^*)$ □

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

- Vamos assumir que usamos um algoritmo de ordenação comparativo com tempo $\mathcal{O}(n \log n)$
  (como as funções de ordenação disponíveis no C, C++ ou Java)

- 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 dar origem a duas chamadas a FIND-SET
  - UNION vai ser chamada $|V| - 1$ vezes
    (uma por cada aresta adicionada)

- O custo total será:
  $\mathcal{O}(|E| \log |E| + |V| \times \text{MAKE-SET} + |E| \times \text{FIND-SET} + |V| \times \text{UNION})$

Algoritmo de Kruskal - Complexidade

- O custo total será:
  $\mathcal{O}(|E| \log |E| + |V| \times \text{MAKE-SET} + |E| \times \text{FIND-SET} + |V| \times \text{UNION})$

- 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 um custo de $\mathcal{O}(|E| \log |E| + |V| + |E| \times |V| + |V|)$ que é dominado pelo termo $|E| \times |V|$

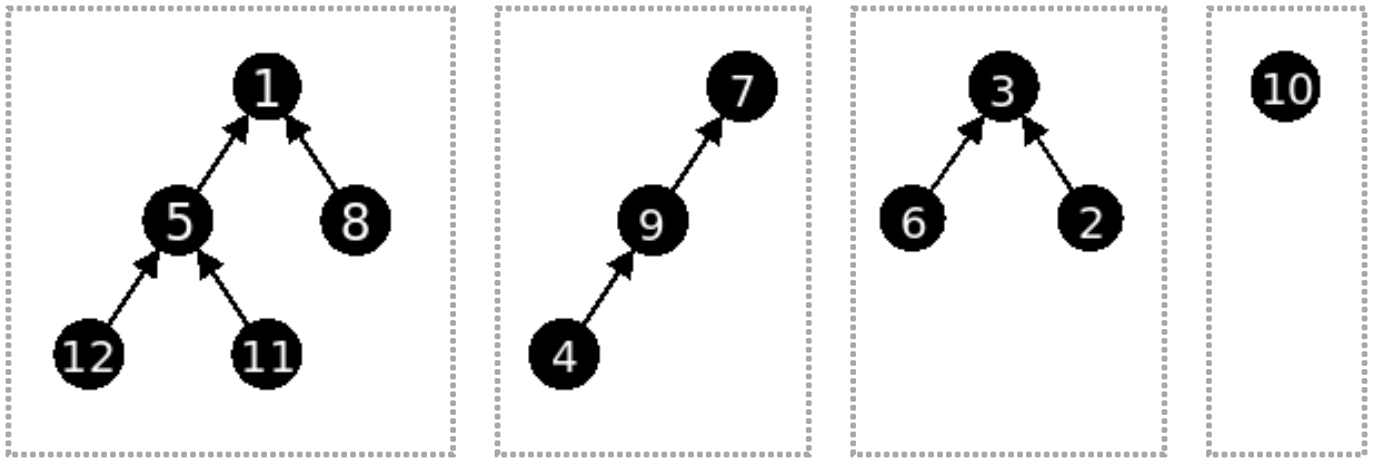
- Se mantivermos um atributo auxiliar para cada nó dizendo qual o conjunto onde está, podemos fazer o FIND-SET em $\mathcal{O}(1)$, mas o UNION passa a custar $\mathcal{O}(|V|)$ (mudar esse atributo para os nós de uma das listas a ser unida), pelo que a complexidade final seria quadrática no número de vértices.

Conjuntos-Disjuntos (*Union-Find*) - Ideia Principal

- Podemos fazer muito melhor se melhorarmos as nossas operações

- Uma estrutura de dados que permite manter conjuntos e suporta as operações UNION e FIND-SET é conhecida como **union-find**, e uma maneira eficiente 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 raíz da árvore do conjunto

  Aqui fica um exemplo de representação de 4 conjuntos:
  $\{1, 5, 8, 11, 12\}, \{4, 7, 9\}, \{2, 3, 6\}$ e$\{10\}$



Conjuntos-Disjuntos - Uma primeira implementação

Uma maneira "naive" de implementar conjuntos disjuntos:

Conjuntos-Disjuntos - "naive"

MAKE-SET(x):
  $x.pai \leftarrow x$ /* raíz aponta para si própria */

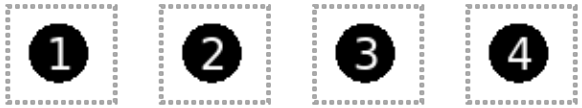
FIND(x):
  Se $x.pai = x$ **então retorna** x
  Senão retorna FIND($x.pai$)

UNION(x, y):
  $xRaiz \leftarrow FIND(x)$
  $yRaiz \leftarrow FIND(y)$
  $yRaiz.pai \leftarrow xRaiz$

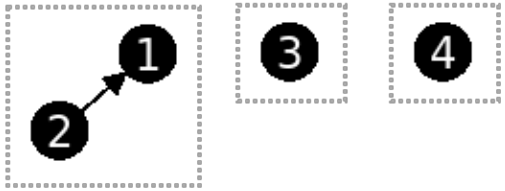
Conjuntos-Disjuntos - Uma primeira implementação

- Exemplo de execução:

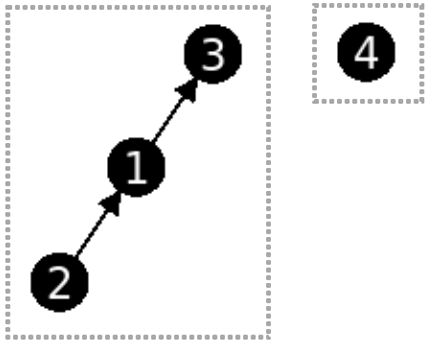
MAKE-SET(1), MAKE-SET(2), MAKE-SET(3), MAKE-SET(4)



UNION(1, 2) [2's parent becomes 1]



UNION(3, 2) [2's root is 1, its new parent becomes 3]



- Com esta implementação podemos continuar a ter **tempo linear por operação** porque as árvores podem ficar muito desiquilibradas (e com altura igual ao número de nós)

Conjuntos-Disjuntos - Melhoria: "Union by Rank"

- Como podemos melhorar o nosso algoritmo?
- Vamos discutir duas estratégias que levam a **árvores mais equilibradas**
- A primeira estratégia é a **"Union by Rank"**: ao unir, juntar sempre a árvore de menor altura à outra árvore:
  - Queremos evitar que a pior altura de uma árvore cresça
  - Isto garante que a pior altura apenas irá crescer se as duas árvores a juntar tiverem igual altura
- A ideia é guardar num atributo extra o **rank**, que essencialmente dá-nos a profundidade máxima abaixo do nó, e com isso decidir qual árvore se junta a qual

Conjuntos-Disjuntos - Melhoria: "Union by Rank"

Conjuntos-Disjuntos com "Union by Rank"

MAKE-SET(x):
  $x.pai \leftarrow x$ $x.rank \leftarrow 0$

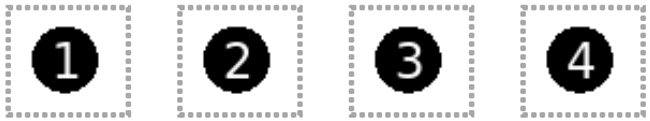
UNION(x, y):
  $xRaiz \leftarrow FIND(x)$
  $yRaiz \leftarrow FIND(y)$
  Se $xRaiz = yRaiz$ **então retorna**

  /* x e y não estão no mesmo conjunto - temos de os unir */
  If $xRaiz.rank < yRaiz.rank$ **então**
    $xRaiz.pai \leftarrow yRaiz$
  Else If $xRaiz.rank > yRaiz.rank$ **então**
    $yRaiz.pai \leftarrow xRaiz$
  Else
    $yRaiz.pai \leftarrow xRaiz$
    $xRaiz.rank \leftarrow xRaiz.rank + 1$

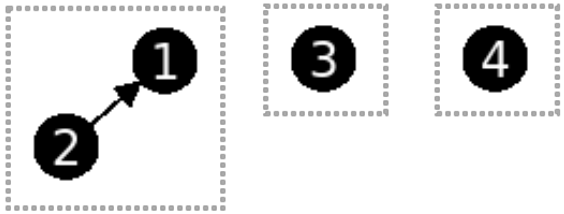
Conjuntos-Disjuntos - Melhoria: "Union by Rank"

- Exemplo de execução com "Union by Rank":

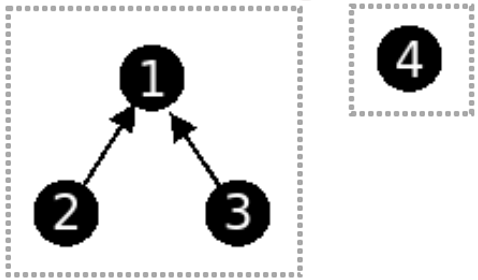
MAKE-SET(1), MAKE-SET(2), MAKE-SET(3), MAKE-SET(4)



UNION(1, 2) [1's rank increases from zero to one]



UNION(2, 3) [since 3's rank is smaller, join it to 1, which is 2's root]



- Esta melhoria, por si só, já garante uma **complexidade logarítmica** para os FIND e UNION!

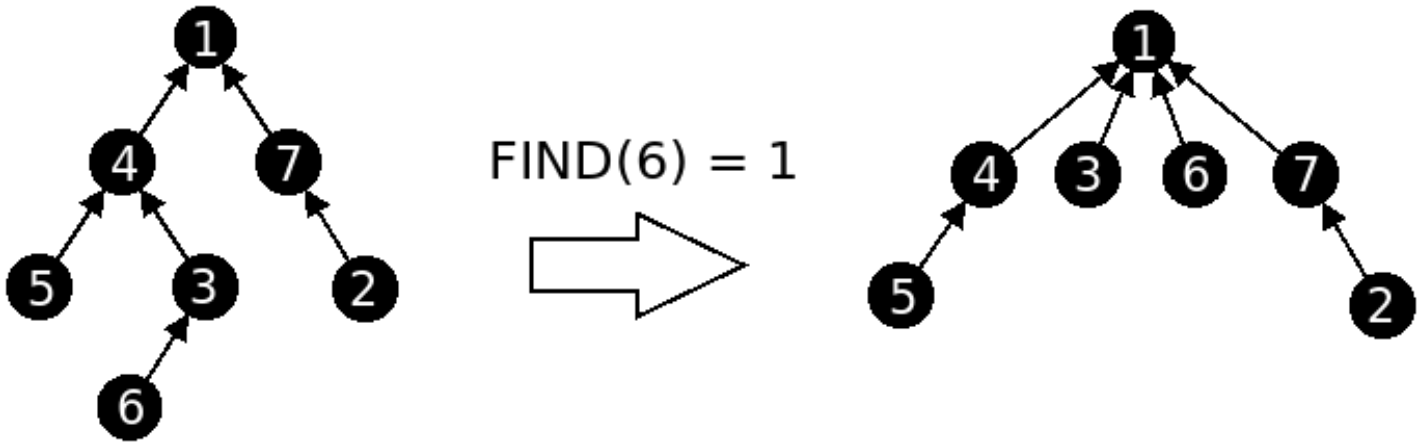
Conjuntos-Disjuntos - Melhoria: "Path Compression"

- A segunda estratégia é **comprimir as árvores** ("path compression"), fazendo com que todos os nós que um FIND percorre passem a apontar directamente para a raíz, potencialmente diminuindo a altura

Conjuntos-Disjuntos com "Path Compression"

FIND(x):
  Se $x.pai \neq x$ **então**
    $x.pai \leftarrow FIND(x.pai)$
  retorna $x.pai$

- Um exemplo de uma operação FIND (com "Path Compression"):



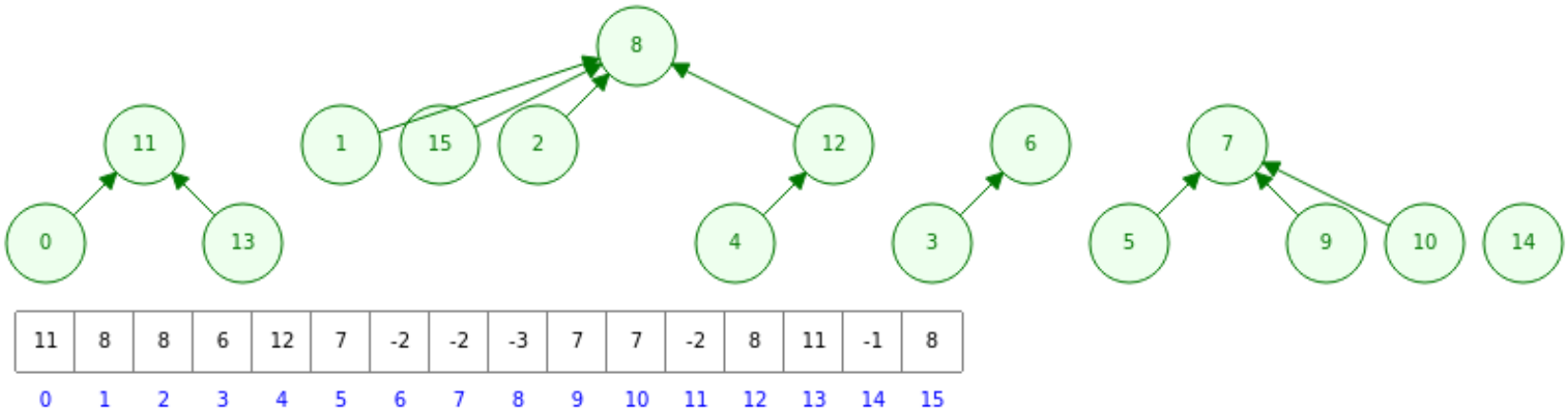
Conjuntos-Disjuntos - Complexidade e Visualização

- Com "union by rank" e "path compression" **o custo amortizado por operação, na prática, é constante**

  A complexidade real é $\mathcal{O}(m\alpha(m, n))$ para uma sequência de m operações, onde *alpha* é **a inversa da função de Ackermann**: esta função cresce tão devagar que é menor que 5 para qualquer input que possa ser escrito no nosso universo físico. Uma análise detalhada disto está fora do escopo desta UC, mas se tiverem curiosidade podem espreitar por exemplo o seguinte artigo científico: *"Worst-case analysis of set union algorithms"* (JACM, 1984)

- Podem visualizar conjuntos-disjuntos e as suas operações:

  https://www.cs.usfca.edu/~galles/visualization/DisjointSets.html



Algoritmo de Kruskal - Complexidade Temporal

- Relembremos que o custo do **Algoritmo de Kruskal** era: $\mathcal{O}(|E| \log |E| + |V| \times \text{MAKE-SET} + |E| \times \text{FIND-SET} + |V| \times \text{UNION})$
- Se o tempo para cada operação é constante, então a complexidade temporal é agora **dominada pela... ordenação das arestas**!
- A complexidade temporal do algoritmo de Kruskal usando conjuntos-disjuntos é então $\mathcal{O}(|E| \log |E|)$
- Notem que $\mathcal{O}(|E| \log |E|)$ é efetivamente $\mathcal{O}(|E| \log |V|)$, uma vez que num grafo simples $|E| \leq |V|^2$ e $\log |V|^2 = 2 \times \log |V| = \mathcal{O}(\log |V|)$
- O Kruskal e o Prim são por isso "equivalentes" em termos de complexidade temporal e ambos calculam uma MST em tempo $\mathcal{O}(|E| \log |V|)$
- O Kruskal poderia ser ainda mais rápido se usarmos uma ordenação não comparativa (ex: se os pesos forem todos pequenos e inteiros podemos usar algo como um *counting sort*)

Árvores de Suporte de Custo Mínimo - Resumo

- Uma MST é o "melhor" conjunto de arestas que mantém o grafo conexo
- Existem dois principais algoritmos greedy "clássicos" para esta tarefa:
  - **Prim:** adicionar nó a nó
    (selecionar o nó que liga à árvore atual com menor custo)
  - **Kruskal:** adicionar aresta a aresta
    (por ordem crescente de peso, desde que não se introduzam ciclos)
- Ambos os algoritmos ficam com complexidade temporal $\mathcal{O}(|E| \log |V|)$ se usarmos as estruturas de dados corretas

- Podem ver visualizações dos algoritmos:
  - David Galles: Prim, Kruskal
  - VisuAlgo: Min Spanning Tree
- E para as estruturas de dados associadas:
  - David Galles: Heaps, Disjoint-Sets
  - VisuAlgo: Heaps, Disjoint-Sets