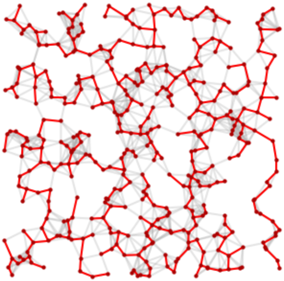
Árvores de Suporte de Custo Mínimo

Pedro Ribeiro

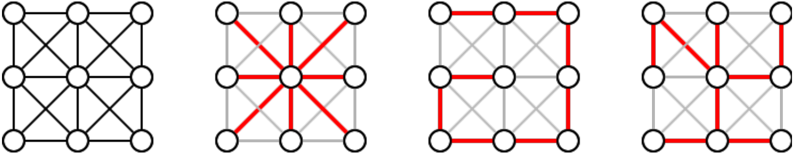
DCC/FCUP

2016/2017



Árvore de Suporte

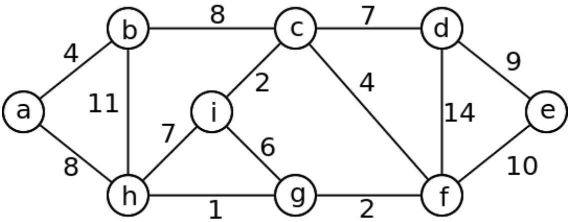
- Uma **árvore de suporte** ou **árvore de extensão** (*spanning tree*) é um subconjunto das arestas de um grafo não dirigido que forma uma árvore ligando todos os vértices.
- A figura seguinte ilustra um grafo e 3 árvores de suporte:



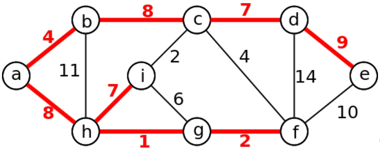
- Podem existir várias árvores de suporte para um dado grafo
- Uma árvore de suporte de um grafo terá sempre $|V| - 1$ arestas
  - Se tiver menos arestas, não liga todos os nós
  - Se tiver mais arestas, forma um ciclo

Árvore de Suporte de Custo Mínimo

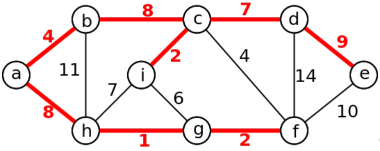
- Se o grafo for pesado (tem valores associados às arestas), existe a noção de **árvore de suporte de custo mínimo** (*minimum spanning tree* - MST), que é a árvore de suporte cuja soma dos pesos das arestas é a menor possível.
- A figura seguinte ilustra um grafo não dirigido e pesado. Qual é a sua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



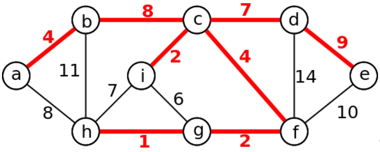
Árvore de Suporte de Custo Mínimo



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

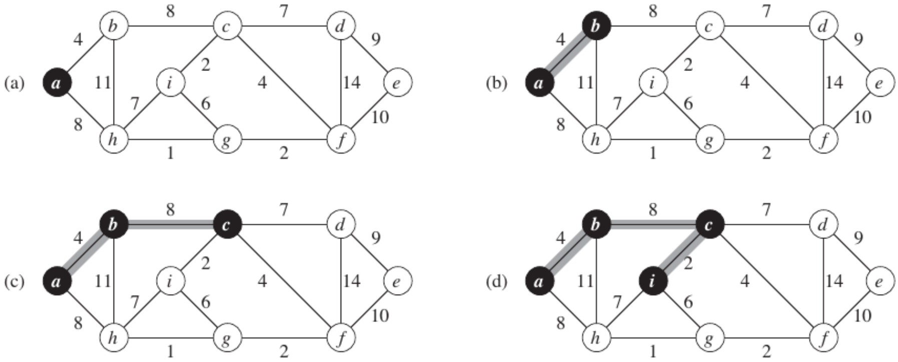
Algoritmo Genérico para MST

$A \leftarrow \emptyset$
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

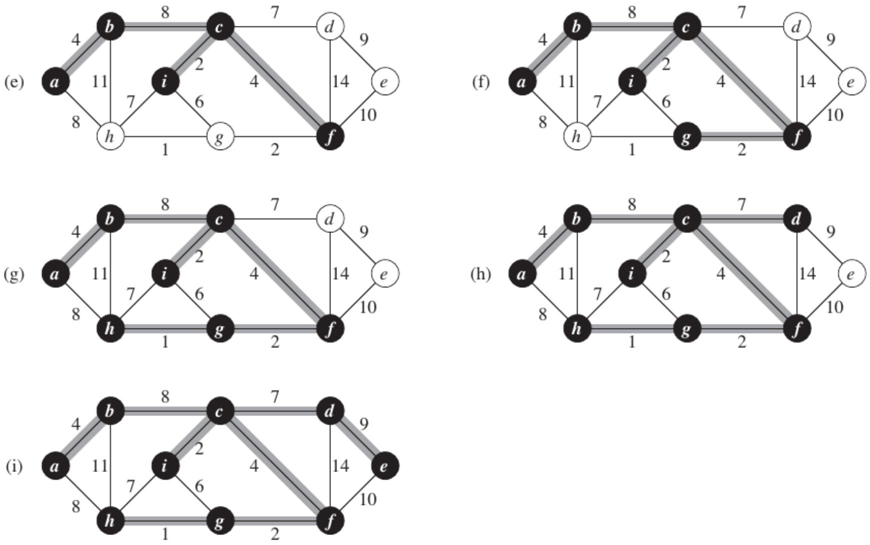
- Começar num qualquer nó
- Em cada passo **adicionar à árvore já formada o nó cujo custo seja menor** (que tenha aresta de menor peso a ligar à árvore). Em caso de empate qualquer um funciona.
- Vamos ver passo a passo para o grafo anterior...

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:

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

- A complexidade do algoritmo de Prim depende da operação EXTRAIR-MINIMO
    - Vamos chamar EXTRAIR-MINIMO $|V|$ vezes
    - Cada aresta vai ser considerada duas vezes (uma para cada um dos nós extremidade) no ciclo que actualiza os valores de *dist*
    - A complexidade final é $O(|E| + |V| \times custo(\text{EXTRAIR-MINIMO}))$

- Uma implementação "naive" em que o mínimo é descoberto de forma linear (um ciclo para ver qual o menor) daria uma complexidade de $\mathcal{O}(|E| + |V|^2)$

- É possível reduzir para um **tempo linearítmico** se usarmos uma estrutura de dados que suporte a operação de extrair o mínimo em tempo logarítmico!

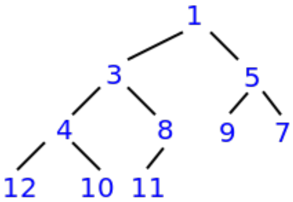
- Uma estrutura de dados para esta função (devolver o elemento mínimo ou máximo) é conhecida como **fila de prioridade**

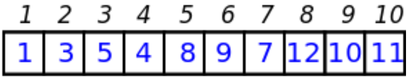
Heap: uma implementação de uma fila de prioridade

- Uma **heap** é uma estrutura de dados organizada como uma árvore binária equilibrada, implementando uma **fila de prioridade**
- Existem dois tipos básicos de heaps:
  - **max-heaps:** o elemento mais prioritário é o de máximo valor
  - **min-heaps:** o elemento mais prioritário é o de menor valor
- Para termos uma heap a seguinte condição tem de ser respeitada: **o pai de um nó tem sempre mais prioridade** do que ele. Dito de outro modo, numa max-heap os filhos de um nó têm menor valor que ele, e numa min-heap os filhos têm maior valor.
- Uma heap deve ser uma **árvore binária completa até ao seu penúltimo nível, e o último nível deve estar preenchido da esquerda para a direita**.
  - Isto garante que a altura máxima de uma árvore com n nós é proporcional a $\log_2 n$

- Uma **heap é tipicamente implementada com um array**, onde:
  - Os **filhos** do nó (i) são os nós nas posições $(i * 2)$ e $(i * 2 + 1)$
  - O **pai** de um nó (i) é o nó na posição $(i/2)$.

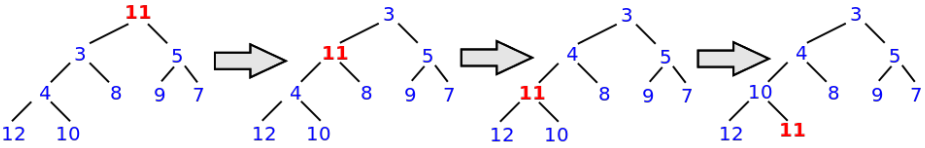
A figura seguinte ilustra uma **min-heap** e o array correspondente:





Heap: uma implementação de uma fila de prioridade

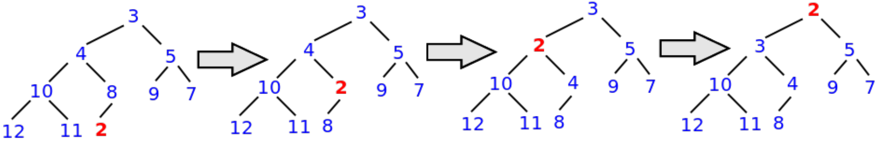
- Existem duas operações importantes numa heap: **remover** e **inserir**
- **Remover** um elemento passa por remover a raiz
  - Numa min-heap a raíz é o menor elemento de todos
  - Numa max-heap a raíz é o maior elemento de todos
- Depois de remover a raíz é necessário repor as condições de heap. Para isso, faz-se o seguinte:
  - Pega-se no último elemento e coloca-se na posição da raíz
  - O elemento "baixa" (**down-heap**), trocando com o mais prioritário dos filhos, até que a condição de heap esta reposta
  - No máximo faz-se $\mathcal{O}(\log n)$ operações, porque a árvore é equilibrada!



Heap: uma implementação de uma fila de prioridade

- **Inserir** um elemento passa por:
  - Colocá-lo na última posição
  - O elemento "sobe" (**up-heap**), trocando com o pai, até que a condição de heap esteja reposta
  - No máximo faz-se $\mathcal{O}(\log n)$ operações, porque a árvore é equilibrada!

Exemplo para inserção do elemento 2



Algoritmo de Prim e Filas de Prioridade

- Recorda que a complexidade do algoritmo de Prim é $\mathcal{O}(|E| + |V| \times \mathit{custo}(\text{EXTRAIR-MINIMO}))$
- Supondo que usamos uma estrutura de dados especializada para EXTRAIR-MINIMO necessitamos de ter em conta o tempo para actualizar (baixar) o valor da distância de um nó: $\mathcal{O}(|E| \times \mathit{custo}(\text{ACTUALIZAR}) + |V| \times \mathit{custo}(\text{EXTRAIR-MINIMO}))$
- Com uma **min-heap**:
  - Cada operação de retirar o nó mais perto vai custar $\mathcal{O}(\log |V|)$ (é só chamar a remoção da heap)
  - Cada operação de actualização vai também custar $\mathcal{O}(\log |V|)$ (como uma actualização só pode reduzir o valor, é chamar um up-heap)
- A complexidade final é $\mathcal{O}(|E| \log |V| + |V| \log |V|$, que é o mesmo que $\mathcal{O}(|E| \log |V|)$ (existem assintoticamente pelo menos tantas arestas como nós, caso contrário nem uma árvore de suporte conseguiriamos fazer)

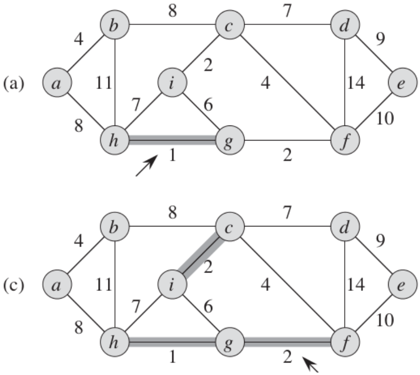
Algoritmo de Prim e Filas de Prioridade

- 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. Implementar heap "manualmente"
     (podemos chamar up-heap em qualquer nó no meio da fila()
     Complexidade do Prim: $\mathcal{O}(|E| \log |V|)$ ou
  2. Usar uma *PriorityQueue* e actualizar ser feito via inserção de novo elemento na heap com a nova distância
     (cada nó será inserido no máximo tantas vezes quanto o seu grau)
     Complexidade do Prim: $\mathcal{O}(|E| \log |E|)$ ou
  3. Usamos uma BST (ex: um *set*) e actualizar ser feito via remoção + inserção (ambas as operações em tempo logarítmico)
     Complexidade do Prim: $\mathcal{O}(|E| \log |V|)$

Algoritmo de Kruskal

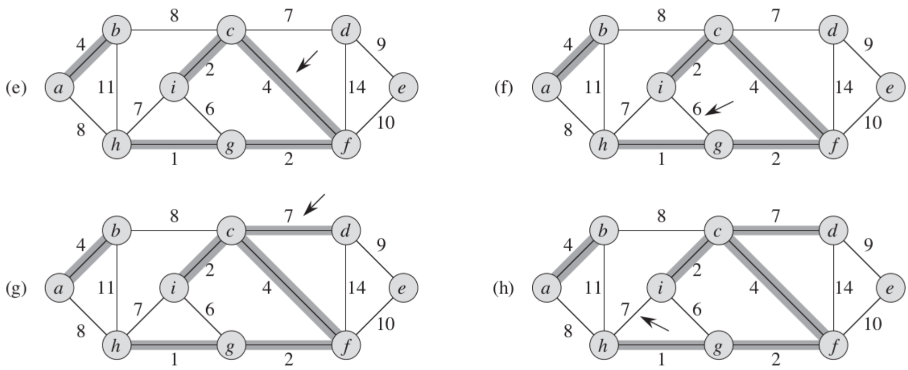
- Manter uma floresta (conjunto de árvores), onde no início cada nó é uma árvore isolada e no final todos os nós fazem parte da mesma árvore
- Ordenar as arestas por ordem crescente de peso
- Em cada passo **selecionar a aresta de menor valor que ainda não foi testada** e, caso esta aresta junte duas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



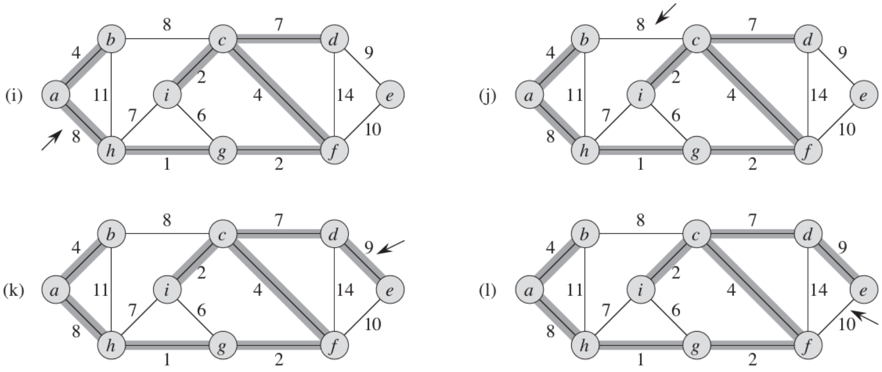
(imagem de Introduction to Algorithms, 3rd Edition)

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 Kruskall para descobrir MST de G

Kruskal(G, r):
  $A \leftarrow \emptyset$
  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 difer. */
    $A \leftarrow A \cup \{(u, v)\}$
    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

- Para além da ordenação, a complexidade do algoritmo de Kruskall 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 $\mathcal{O}(|E| \cdot |V|)$
- 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 linearítmico** se usarmos uma estrutura de dados que suporte estas operações em tempo logarítmico ou constante (supondo que a ordenação demora $\mathcal{O}(n \log n)$)
- Uma estrutura de dados para esta função (manter conjuntos, suportando as operaçãos 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 raíz da árvore do conjunto

Union-Find

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

Naive UNION-FIND

MAKE-SET(x):
  $x.pai \leftarrow x$ /* Raíz 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 desiquilibradas (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):
  $x.pai \leftarrow x \quad x.rank \leftarrow 0$

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 raíz, 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 Kruskall passa a ser dominado... pela ordenação das arestas!