

Árvores geradoras mínimas

Marco A L Barbosa

malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.

<http://github.com/malbarbo/na-grafos>

- Dado um grafo conexo não orientado $G = (V, E)$ e uma função peso $w : E \rightarrow \mathbb{R}$, queremos encontrar um subconjunto acíclico $T \subseteq E$ que conecte todos os vértices de G e cujo peso total

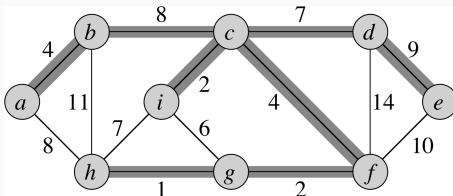
$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

seja minimizado

- Como T é acíclico e conecta todos os vértices, T forma uma árvore, que chamamos de **árvore geradora mínima** (AGM ou MST em inglês)

Introdução

Exemplo de uma AGM



Propriedades de uma AGM

- Tem $|V| - 1$ arestas
- Não tem ciclos
- Pode não ser única

- O **problema da árvore geradora mínima** consiste em encontrar uma AGM de um dado grafo com pesos nas arestas
- Veremos dois algoritmos gulosos para resolver este problema
 - Algoritmo de Kruskal
 - Algoritmo de Prim

Como construir uma árvore geradora mínima

- Como construir uma árvore geradora mínima? Uma aresta de cada vez!
- Começamos com um conjunto vazio A
- Em cada etapa, determinamos um aresta (u, v) que pode ser adicionada a A , de forma a manter a seguinte invariante
 - Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima
- A aresta (u, v) é chamada de **aresta segura** para A

Como construir uma árvore geradora mínima

GENERIC-MST(G, w)

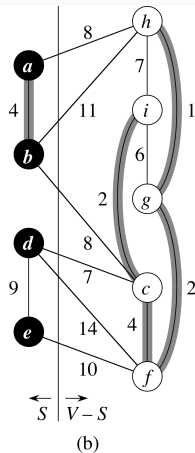
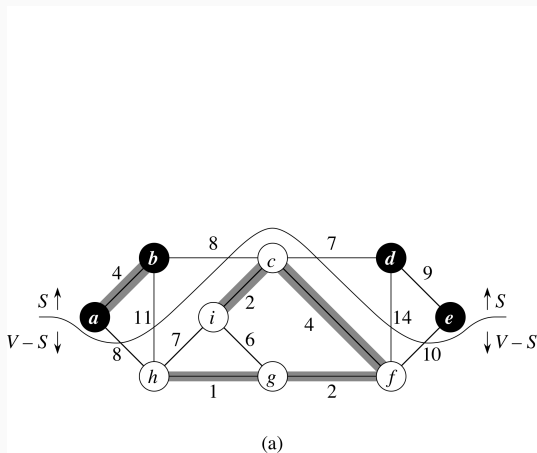
- 1 $A = \emptyset$
- 2 **while** A não forma uma árvore geradora
- 3 encontre uma aresta (u, v) que seja segura para A
- 4 $A = A \cup \{(u, v)\}$
- 5 **return** A

Como construir uma árvore geradora mínima

Vamos fornecer uma regra para reconhecer arestas seguras, mas antes precisamos de algumas definições. Seja $S \subset V$ e $A \subseteq E$

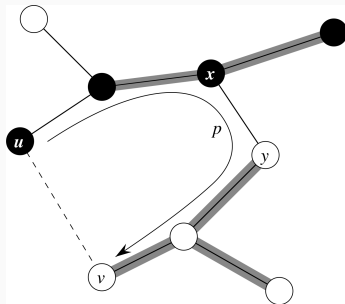
- Um **corte** $(S, V - S)$ de um grafo não orientado $G = (V, E)$ é uma partição de V
- Uma aresta $(u, v) \in E$ **cruza** o corte $(S, V - S)$ se um de seus extremos está em S e o outro em $V - S$
- Um corte **respeita** o conjunto A de arestas se nenhuma aresta em A cruza o corte
- Uma aresta é uma **aresta leve** cruzando um corte se seu peso é o mínimo de qualquer aresta que cruza o corte

Como construir uma árvore geradora mínima



Teorema 23.1

Seja $G = (V, E)$ um grafo conexo não orientado com uma função peso w de valor real definido em E . Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima de G , seja $(S, V - S)$ qualquer corte de G que respeita A e seja (u, v) uma aresta leve cruzando $(S, V - S)$. Então a aresta (u, v) é segura para A .



- Seja T uma AGM que inclui A
 - Se T contém (u, v) , é claro que (u, v) é segura para A
 - Se T não contém (u, v) , construímos outra AGM T' que inclui $A \cup \{(u, v)\}$ (feito em sala, veja o livro)

Corolário 23.2

Seja $G = (V, E)$ um grafo conexo não orientado com uma função peso w de valor real definido em E . Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima de G , e seja $C = (V_C, E_C)$ um componente conexo (árvore) na floresta $G_A = (V, A)$. Se (u, v) é uma aresta leve conectando C a algum outro componente em G_A , então (u, v) é segura para A .

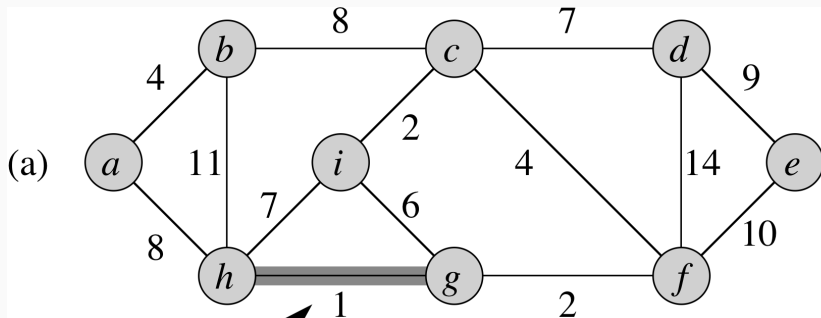
Prova

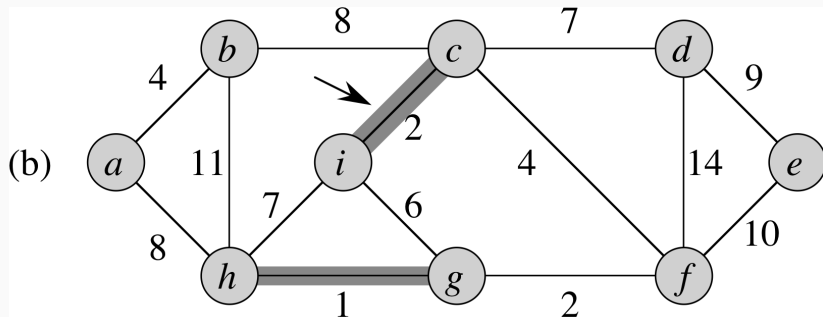
Tomamos $S = V_C$ no teorema 23.1

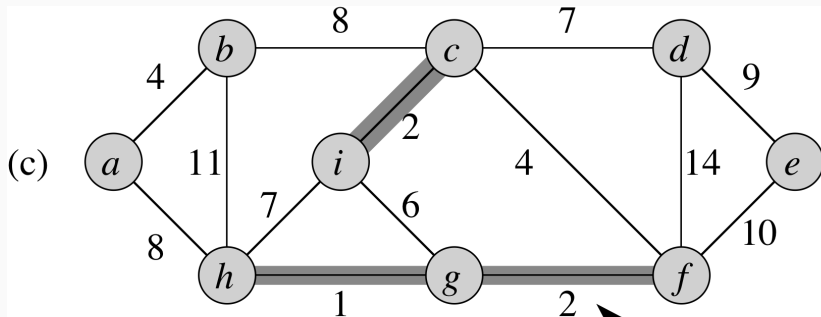
- Baseia-se diretamente no algoritmo genérico
- Inicialmente cada vértice está em sua própria componente (árvore)
- De todas as arestas que conectam duas árvores quaisquer na floresta, uma aresta (u, v) de peso mínimo é escolhida. A aresta (u, v) é segura para alguma das duas árvores

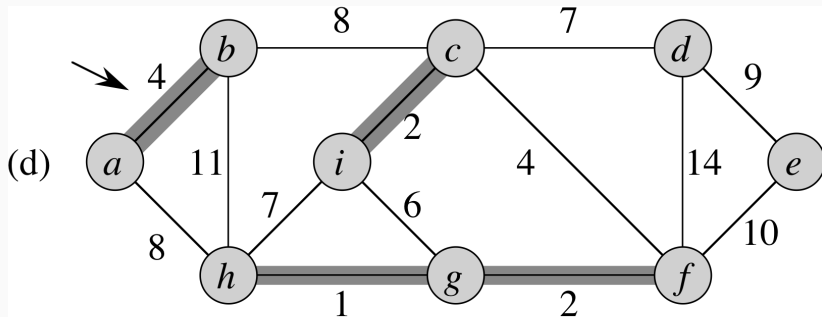
- A questão principal é como gerenciar as árvores (verificar se dois vértices estão na mesma árvore e juntar duas árvores)
 - Utilizamos uma estrutura de dados de conjuntos disjuntos
 - Cada conjunto contém os vértices de uma árvore da floresta atual

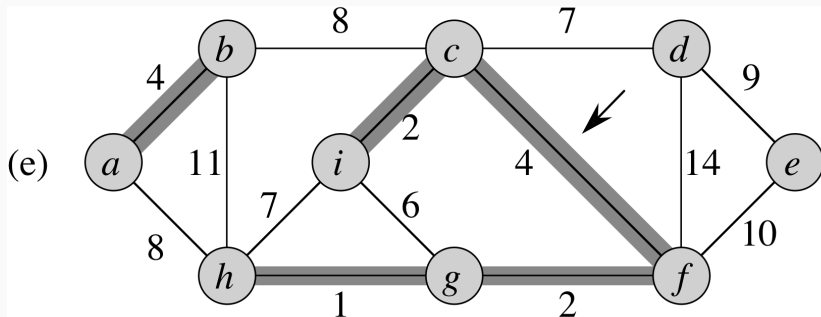
Algoritmo de Kruskal

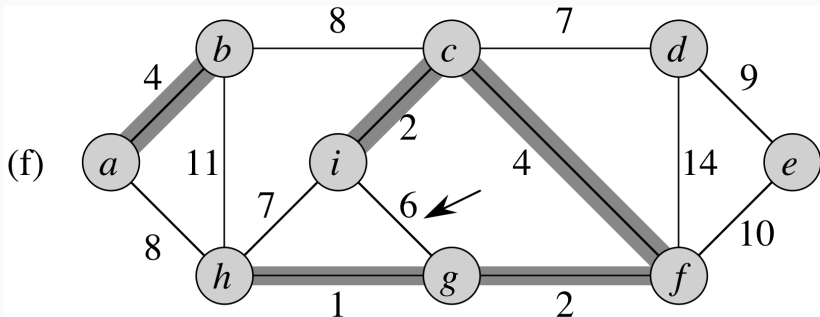




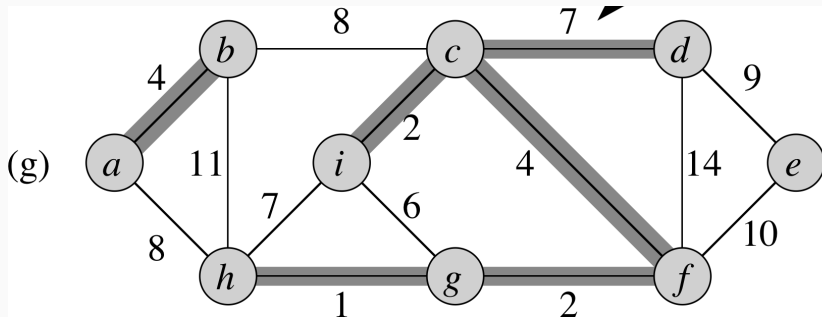


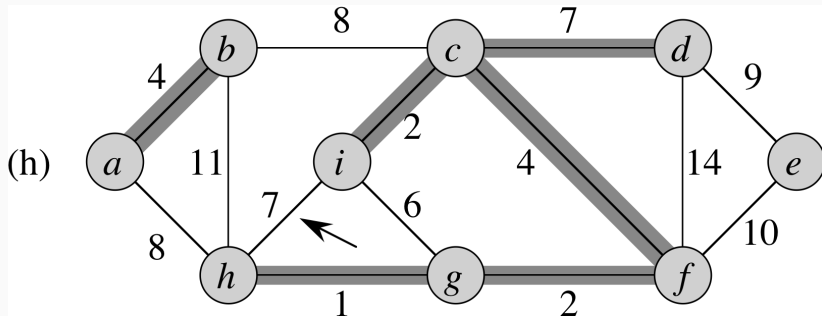




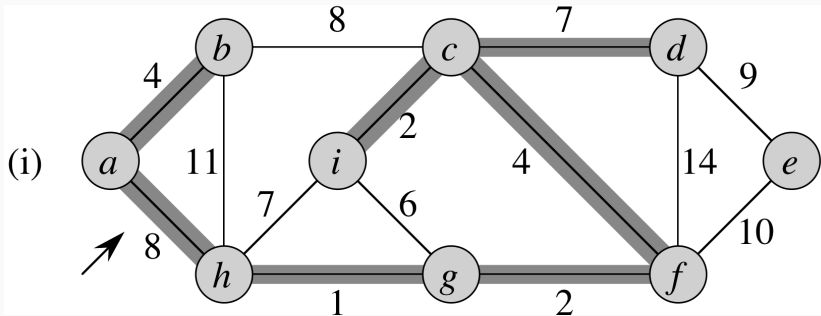


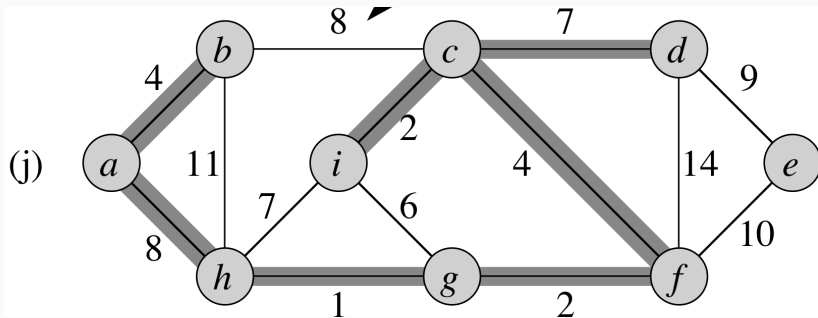
Algoritmo de Kruskal

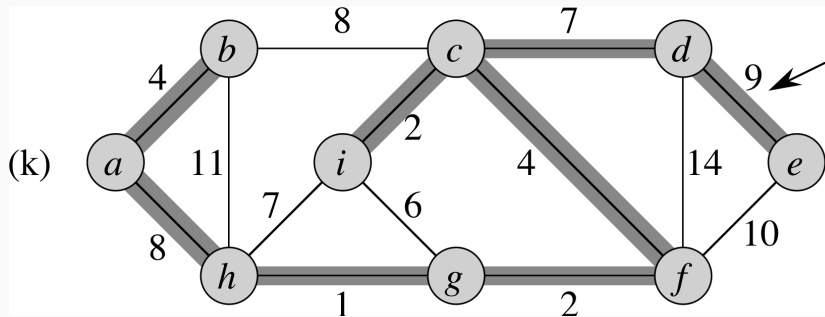




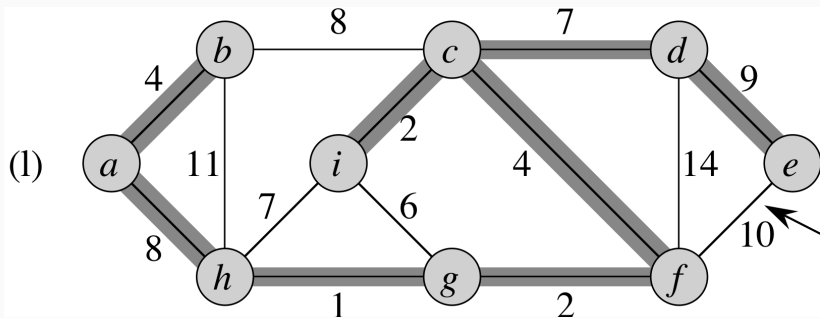
Algoritmo de Kruskal



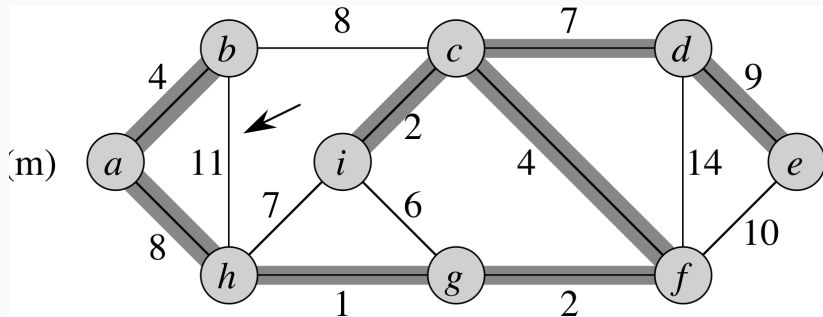




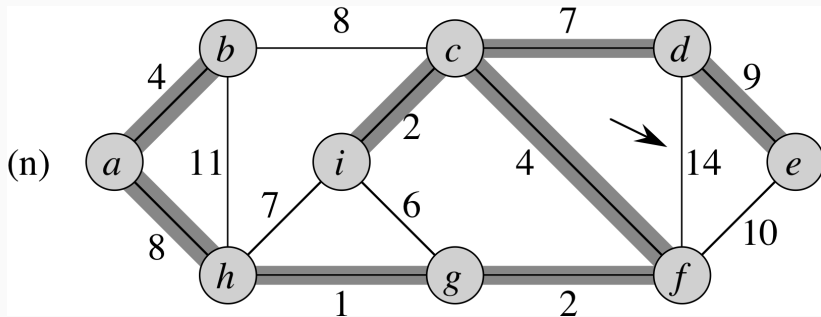
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal

MST-KRUSKAL(G, w)

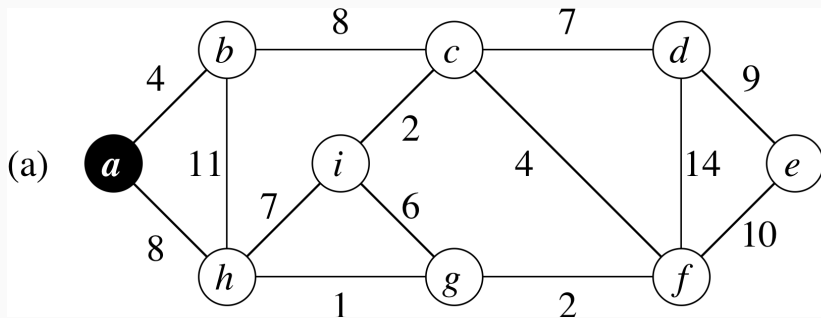
```
1   $A = \emptyset$ 
2  for  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  ordenar em ordem não decrescente as arestas de  $E$  pelo peso  $w$ 
5  for  $(u, v) \in E$ , na ordem obtida anteriormente
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{(u, v)\}$ 
8          UNION( $u, v$ )
9  return  $A$ 
```

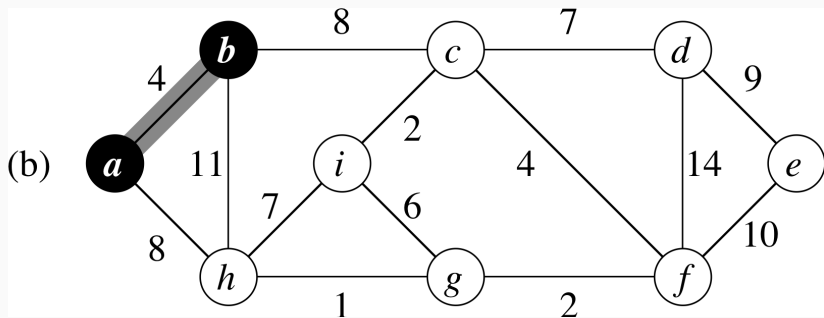
- A ordenação das arestas na linha 4 tem tempo $O(E \lg E)$
- Operações com conjuntos disjuntos (depende da implementação, supomos a implementação da seção 21.3)
 - O laço das linhas 5 a 8 executa $O(E)$ vezes FIND-SET e UNION. Juntamente com as $|V|$ operações MAKE-SET, elas têm tempo $O((V + E)\alpha(V))$, onde α é uma função de crescimento muito lento
 - Pelo fato de G ser conexo, temos que $|E| \geq |V| - 1$, portanto o tempo com operações com conjuntos disjuntos é $O(E\alpha(V))$
 - Além disso, $\alpha(|V|) = O(\lg V) = O(\lg E)$, e portanto o tempo total das operações com conjuntos disjuntos é $O(E \lg E)$

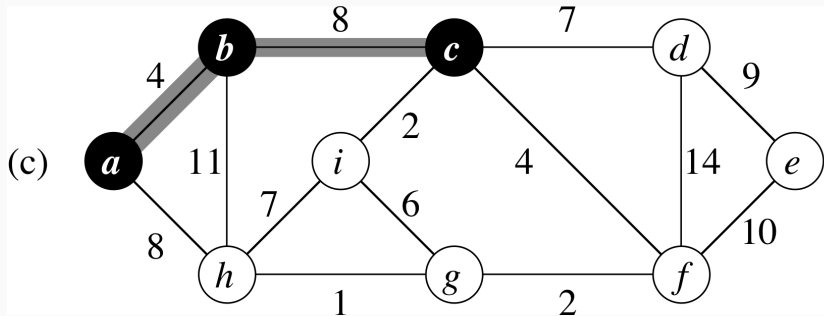
- Somando o custo de ordenação e o custo das operações com conjuntos disjuntos, temos $O(E \lg E)$. Observando que $|E| < |V|^2$, temos que $\lg |E| = O(\lg V)$, e portanto, o tempo de execução do algoritmo é $O(E \lg V)$

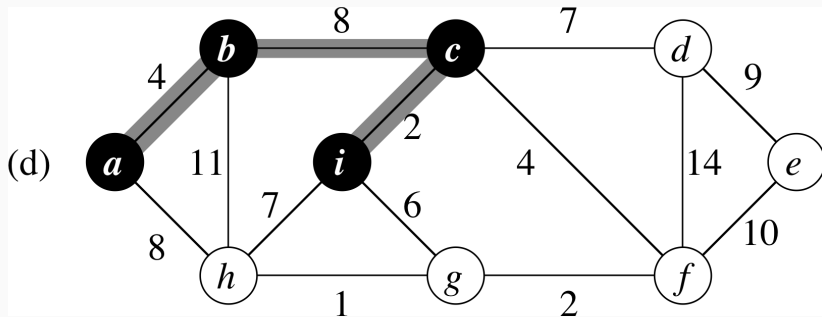
- Baseia-se diretamente no algoritmo genérico
- As arestas do conjunto A formam uma única árvore
- A árvore começa com uma raiz arbitrária r e aumenta até alcançar todos os vértices em V
- Para cada vértice v , mantemos dois atributos, $v.chave = w(v, u)$ e $v.\pi = u$, onde (u, v) é uma aresta de peso mínimo que conecta v a um vértice de V (o vértice u). Usamos $v.chave = \infty$ se não existe tal aresta
- Em cada passo, um vértice v com a menor chave é adicionado a árvore junto com a aresta $(v.\pi, v)$

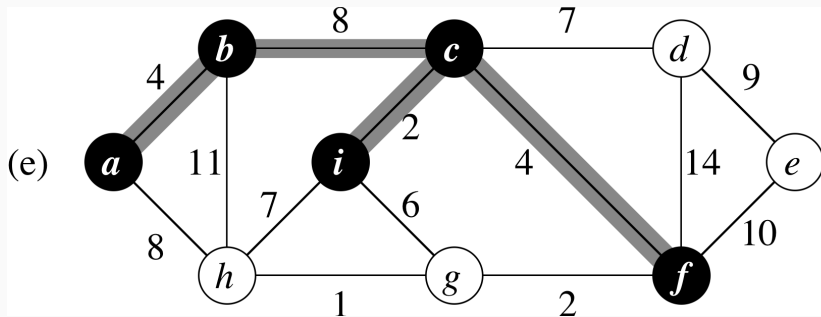
- A questão principal para implementar o algoritmo de Prim de forma eficiente é tornar fácil a seleção de uma nova aresta a ser adicionada à árvore
 - Usamos uma fila de prioridades

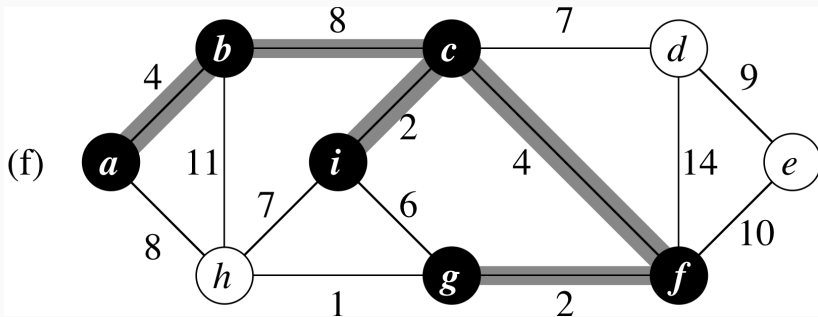


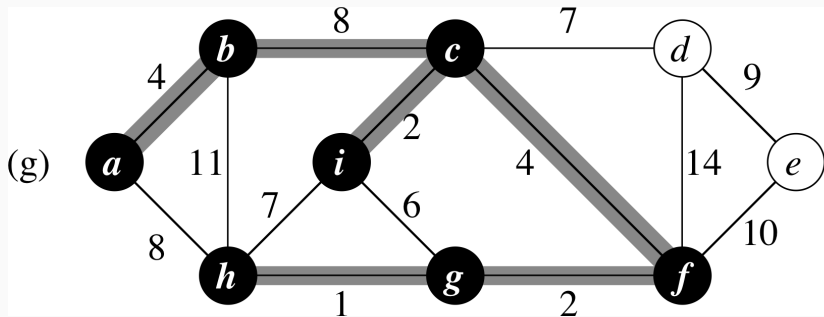


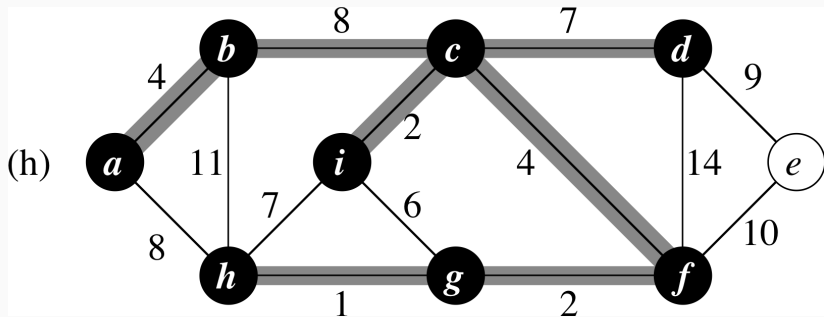


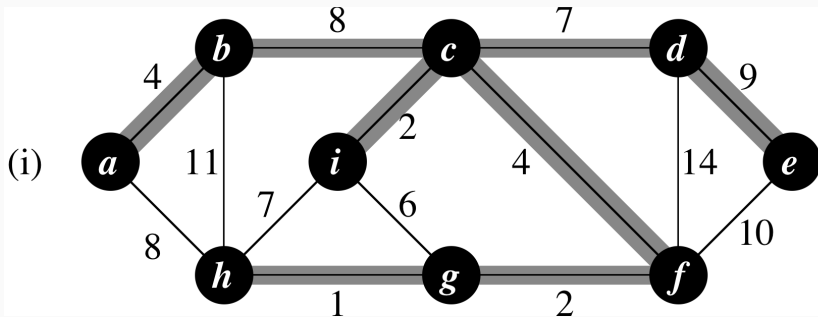












Algoritmo de Prim

MST-PRIM(G, w, r)

```
1  for  $u \in G.V$ 
2       $u.chave = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.chave = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for  $v \in G.Adj[u]$ 
9          if  $v \in Q$  e  $w(u, v) < v.chave$ 
10               $v.\pi = u$ 
11               $v.chave = w(u, v)$ 
```

Análise do tempo de execução

- A inicialização nas linhas de 1 a 3 tem tempo $O(V)$
- A inicialização da fila na linha 4 requer $|V|$ operações INSERT na fila (ou pode ser feito com uma única operação CREATE)
- A operação EXTRACT-MIN é executada uma vez para cada vértice
- O laço for das linhas 8 a 11 é executado no total $O(E)$ vezes
 - O teste de pertinência da linha 9 pode ser implementa em tempo constante usando um atributo no vértice
 - A atribuição na linha 11 envolve uma operação implícita de DECREASE-KEY
- Portanto, o tempo de execução total é $O(V) \times O(\text{INSERT}) + O(V) \times O(\text{EXTRACT-MIN}) + O(E) \times O(\text{DECREASE-KEY})$

Podemos utilizar diversas implementação para a fila de prioridades

- Arranjo
 - CREATE inicializa um arranjo com todos os vértices, $O(V)$
 - DECREASE-KEY não faz nada, portanto $O(1)$
 - EXTRACT-MIN, faz uma busca linear no arranjo e remove o vértice com menor chave, tempo $O(V)$
- Heap (Seção 6.5)
- Heap de Fibonacci (Capítulo 19)
 - Interesse teórico, na prática é superado pelo Heap

Análise do algoritmo de Prim

Tempos das operações

Operação	Arranjo	Heap	Heap de Fibonacci
CREATE	$O(V)$	$O(V)$	$O(V)$
EXTRACT-MIN	$O(V)$	$O(\lg V)$	$O(\lg V)$ (amortizado)
DECREASE-KEY	$O(1)$	$O(\lg V)$	$O(1)$ (amortizado)

Tempo de MST-PRIM

Tempo	Arranjo	Heap	Heap de Fibonacci
Grafo qualquer	$O(V^2)$	$O(E \lg V)$	$O(E + V \lg V)$
Grafo denso	$O(V^2)$	$O(V^2 \lg V)$	$O(V^2)$

- Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition.
Capítulo 23.