Árvores Geradoras Mínimas

Letícia Rodrigues Bueno

UFABC

Aplicação

- **Aplicação:** projeto de redes de comunicações;
- queremos conectar n localidades;
- podemos usar $n - 1$ conexões, cada uma conectando duas localidades;
- conexões: cabos de transmissão;
- **Objetivo:** conexão que usa menor quantidade de cabos é mais desejável.

Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;

Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;

Definição do Problema

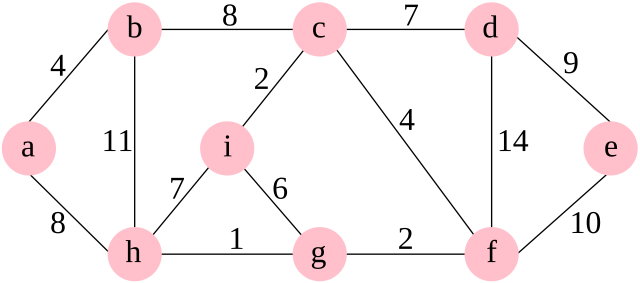
- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;
- $E(G)$: conjunto de possíveis conexões entre localidades;

Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;
- $E(G)$: conjunto de possíveis conexões entre localidades;
- para cada $uv \in E(G)$: peso $p(u, v)$ é o custo (cabo necessário) para conectar u a v;

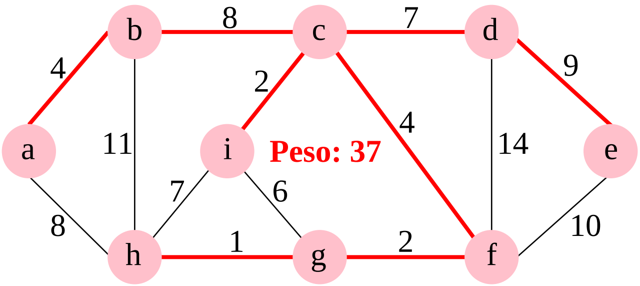
Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;
- $E(G)$: conjunto de possíveis conexões entre localidades;
- para cada $uv \in E(G)$: peso $p(u, v)$ é o custo (cabo necessário) para conectar u a v;



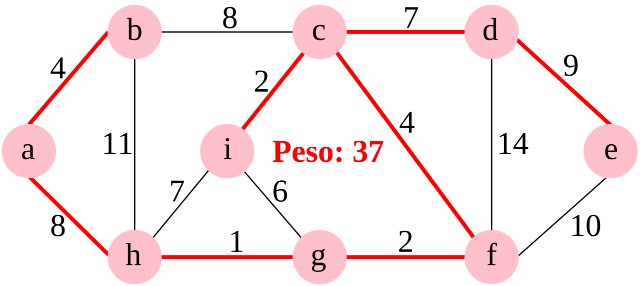
Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;
- $E(G)$: conjunto de possíveis conexões entre localidades;
- para cada $uv \in E(G)$: peso $p(u, v)$ é o custo (cabo necessário) para conectar u a v;



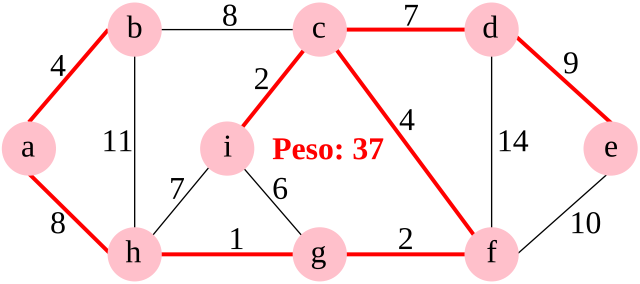
Definição do Problema

- grafo conexo não-orientado $G = (V(G), E(G))$;
- $V(G)$: conjunto de localidades;
- $E(G)$: conjunto de possíveis conexões entre localidades;
- para cada $uv \in E(G)$: peso $p(u, v)$ é o custo (cabo necessário) para conectar u a v;



Definição do Problema

- **Objetivo:** encontrar um subconjunto $T \subseteq E$ tal que:
  - T é acíclico;
  - T conecta todos os vértices de G;
  - peso total $p(T) = \sum_{uv \in T} p(u, v)$ é minimizado;
- Como T é acíclico e conecta todos vértices, T forma uma **árvore geradora** de G uma vez que T "gera" o grafo G;
- o problema de obter a árvore T é conhecido como **árvore geradora mínima**;



Estratégia Gulosa

- adiciona uma aresta por vez;
- gerencia subconjunto S de arestas tal que S é um subconjunto de uma árvore geradora mínima;
- uv é uma **aresta segura** para S se pode ser adicionada sem violar a propriedade de S;
- a cada passo uma aresta segura é determinada para ser adicionada a S;

Algoritmo Genérico para Árvore Geradora Mínima

GenericoAGM(G):
    $S = \emptyset$
    enquanto S não é árvore geradora mínima **faça**
        $uv = selecionaAresta(E)$
        se uv é segura para S **então**
            $S = S \cup \{uv\}$
    retorne S

Como escolher uma aresta segura?

- um **corte** $(V', V(G) - V')$ de G é uma partição de $V(G)$;
- uma aresta $uv \in E(G)$ **cruza** o corte $(V', V(G) - V')$ se um de seus vértices pertence a V' e o outro vértice pertence a $V(G) - V'$;
- um corte **respeita** um conjunto S de arestas se não existirem arestas em S que cruzem o corte;
- uma aresta que tenha custo mínimo sobre todas as arestas cruzando o corte é uma aresta **leve**.

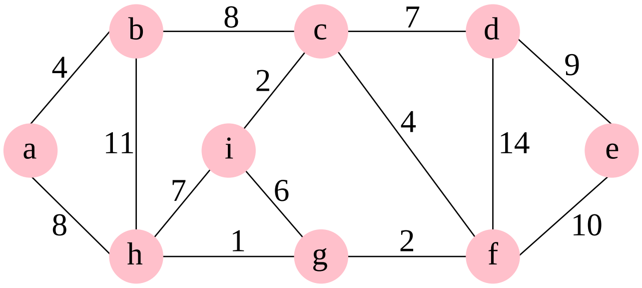
Como escolher uma aresta segura?

Teorema

Seja $G = (V, E)$ um grafo conexo, não-orientado e com pesos p sobre as arestas. Seja S um subconjunto de E que está incluído em alguma árvore geradora mínima para G, seja $(V', V - V')$ um corte qualquer que respeita S e seja uv uma aresta leve cruzando $(V', V - V')$. Logo, a aresta uv é uma aresta segura para S.

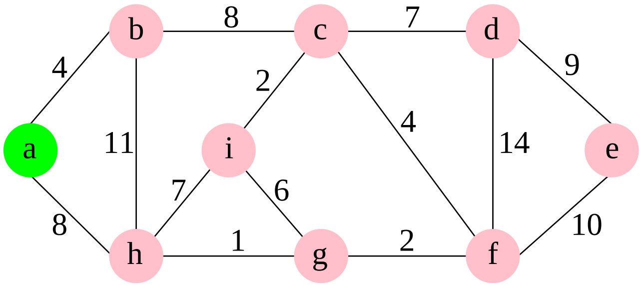
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



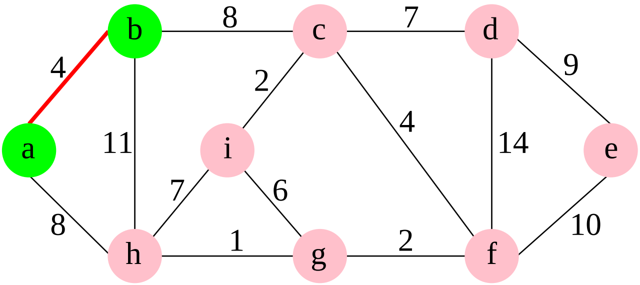
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



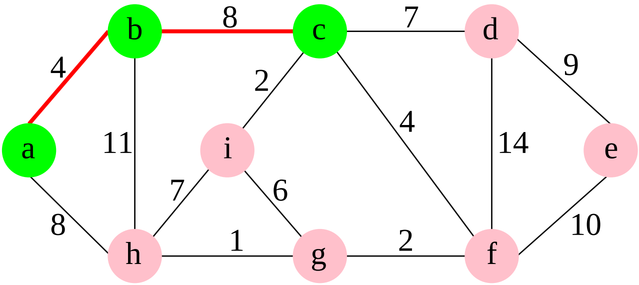
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



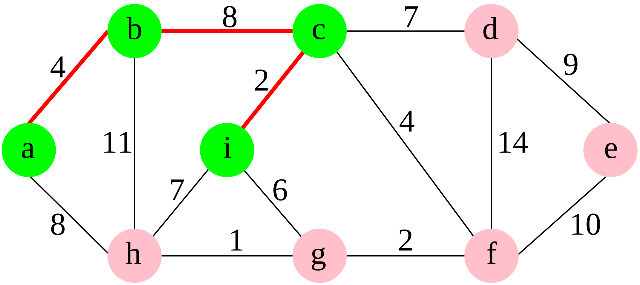
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



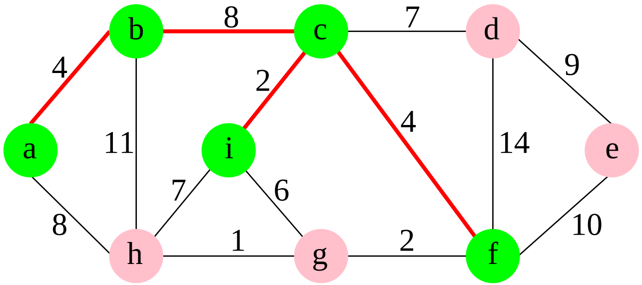
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



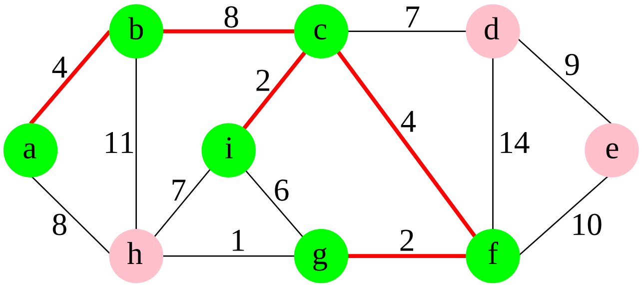
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



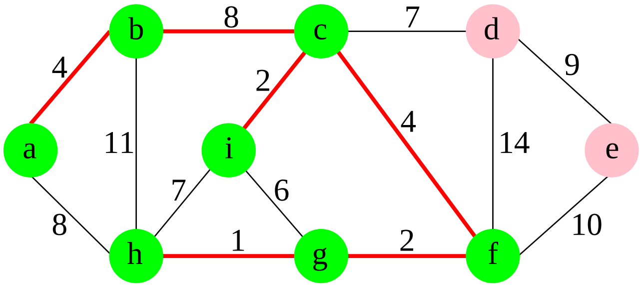
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



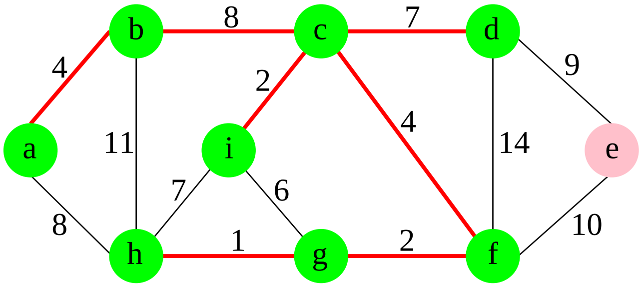
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



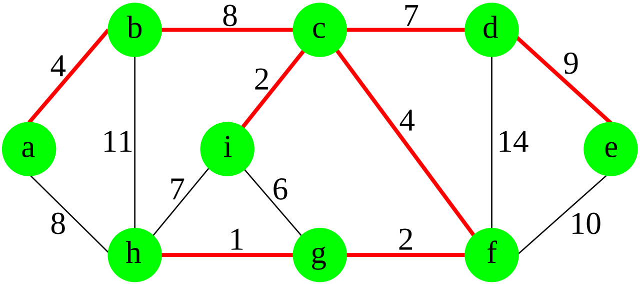
Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



Algoritmo de Prim

- **escolha gulosa:** árvore aumenta acrescentando uma aresta leve por vez.
- *S* forma uma árvore única;
- aresta segura adicionada é aresta leve que conecta árvore a vértice não presente na árvore.



Algoritmo de Prim

Prim(G,r):
    para cada $v \in V(G)$ **faça**
        $p[v] = \infty$
        $pai[v] = -1$
    $p[r] = 0$
    Constrói heap mínimo A com $V(G)$ (com base em p)
    $S = \emptyset$
    enquanto $|A| > 1$ **faça**
        $u = RetiraMin(A)$; Refaz heap
        $S = S \cup \{u\}$
        para $v \in adj(u)$ **faça**
            se $(v \in A)$ **e** $(p[v] > p(u,v))$ **então**
                $p[v] = p(u,v)$
                $pai[v] = u$; Refaz heap

Algoritmo de Prim

Análise da complexidade:

- Constrói *heap* (linha 6): custa $O(n)$;

Algoritmo de Prim

Análise da complexidade:

- Constrói *heap* (linha 6): custa $O(n)$;
- Refazer heap (linha 9): custa $(\log n)$ e é chamada n vezes. Total: $(n \log n)$;

Algoritmo de Prim

Análise da complexidade:

- Constrói *heap* (linha 6): custa $O(n)$;
- Refazer heap (linha 9): custa $(\log n)$ e é chamada n vezes. Total: $(n \log n)$;
- Laço “para” (linha 11) executa $O(m)$ vezes;

Algoritmo de Prim

Análise da complexidade:

- Constrói *heap* (linha 6): custa $O(n)$;
- Refazer heap (linha 9): custa $(\log n)$ e é chamada n vezes. Total: $(n \log n)$;
- Laço “para” (linha 11) executa $O(m)$ vezes;
- Refazer heap (linha 14): custa $(\log n)$ e é chamada $2m$ vezes. Total: $(m \log n)$;

Algoritmo de Prim

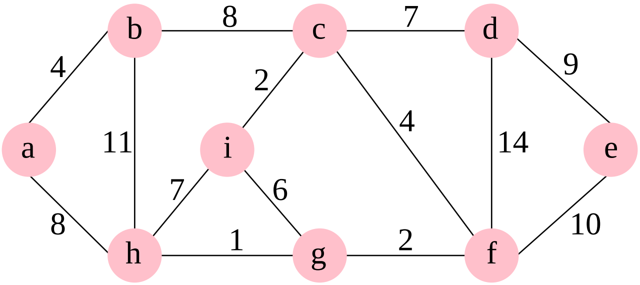
Análise da complexidade:

- Constrói *heap* (linha 6): custa $O(n)$;
- Refazer heap (linha 9): custa $(\log n)$ e é chamada n vezes. Total: $(n \log n)$;
- Laço "para" (linha 11) executa $O(m)$ vezes;
- Refazer heap (linha 14): custa $(\log n)$ e é chamada $2m$ vezes. Total: $(m \log n)$;
- **Complexidade total:** $(n \log n) + (m \log n)$;

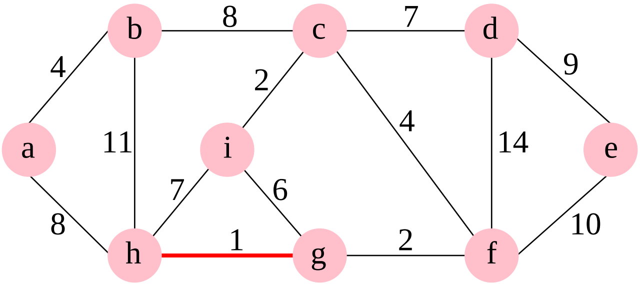
Algoritmo de Kruskal

- escolha gulosa sempre faz escolha que parece melhor no momento;
- nem sempre garante encontrar solução ótima global;
- para árvore geradora mínima, estratégias gulosas obtêm árvore geradora de peso total mínimo;
- **algoritmo de Kruskal:** S é floresta e aresta segura adicionada é sempre aresta leve que conecta dois componentes distintos;
- **escolha gulosa:** árvore aumenta acrescentando-se uma aresta leve por vez.

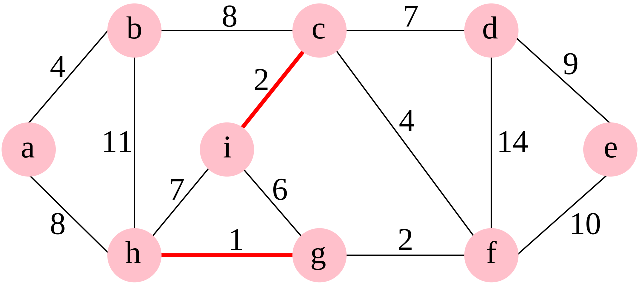
Algoritmo de Kruskal



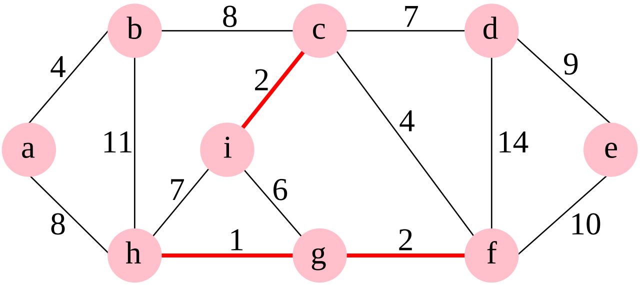
Algoritmo de Kruskal



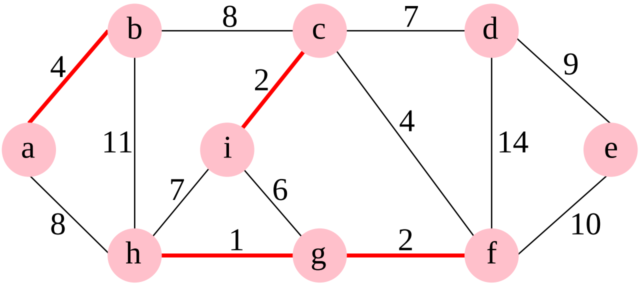
Algoritmo de Kruskal



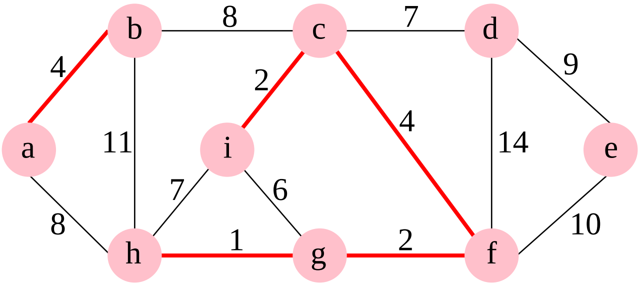
Algoritmo de Kruskal



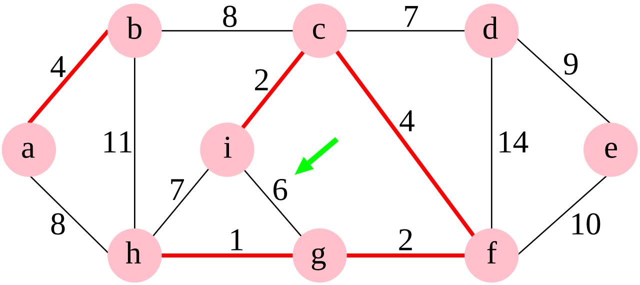
Algoritmo de Kruskal



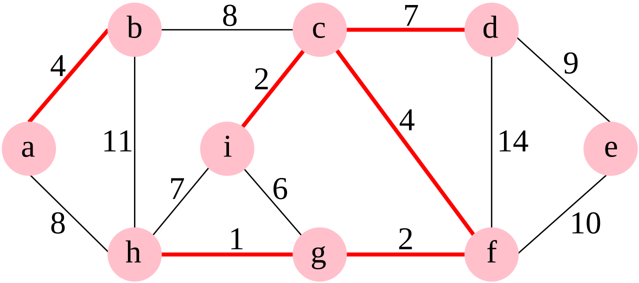
Algoritmo de Kruskal



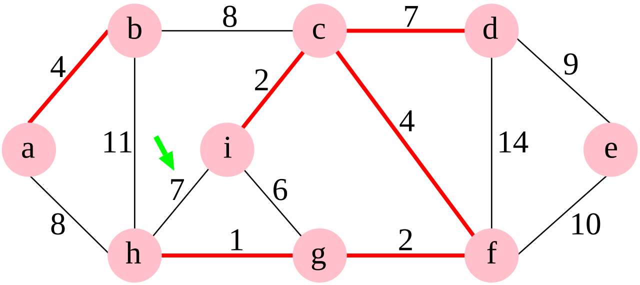
Algoritmo de Kruskal



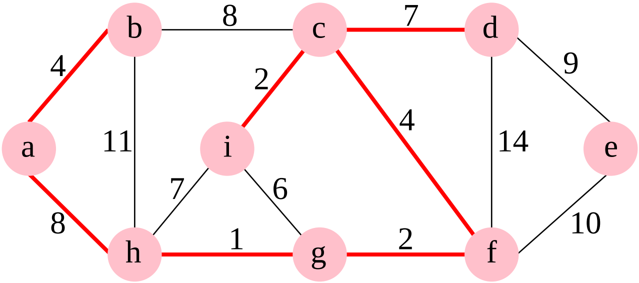
Algoritmo de Kruskal



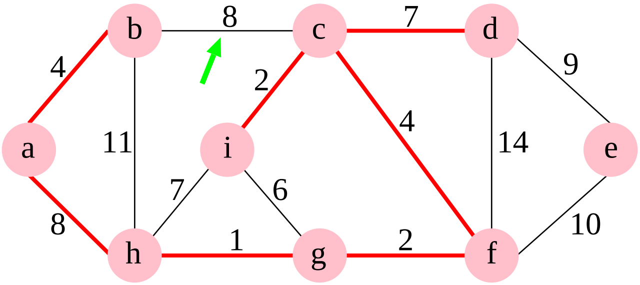
Algoritmo de Kruskal



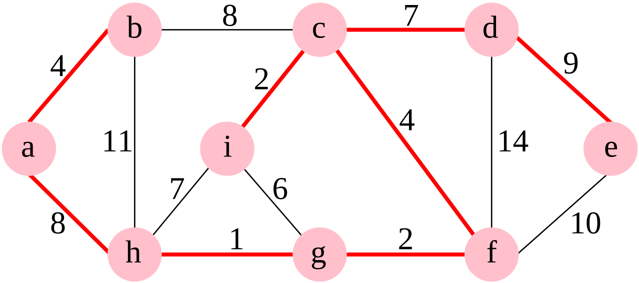
Algoritmo de Kruskal



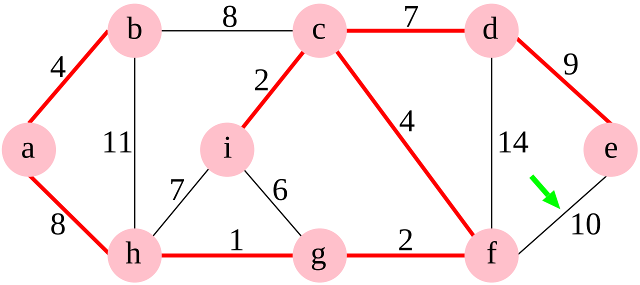
Algoritmo de Kruskal



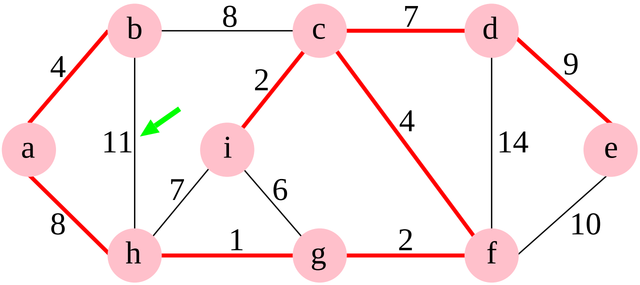
Algoritmo de Kruskal



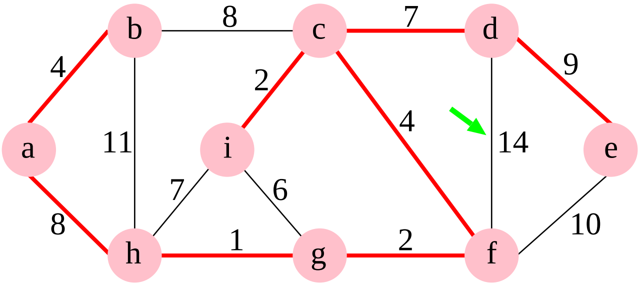
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal

Kruskal(G):
    para $v \in V(G)$ **faça**
        MAKE-SET(v)
    Ordena arestas de $E(G)$ por p não decrescente
    $S = \emptyset$
    para cada $(u, v) \in E(G)$ em ordem não-decrescente **faça**
        se FIND-SET(u) $\neq$ FIND-SET(v)) **então**
            $S = S \cup \{(u, v)\}$
            UNION(u, v)
    retorne S

Algoritmo de Kruskal

Complexidade:

Algoritmo de Kruskal

Complexidade:

- Complexidade total: $O(m \log n)$;

Bibliografia Utilizada

- CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.
- ZIVIANI, N. Projeto de Algoritmos com Implementações em Java e C++. Thomson, 2007.