Algoritmos para Árvore Geradora de Custo Mínimo

Profa. Sheila Morais de Almeida

DAINF-UTFPR-PG

junho - 2018

Este material é preparado usando como referências os textos dos seguintes livros.

Udi MANBER. *Introduction to Algorithms: a creative approach*, 1989.

Problema

Considere uma rede de computadores cujos links têm um custo para se transmitir uma mensagem.

Suponha que o custo de se enviar uma mensagem por um determinado link não depende da direção.

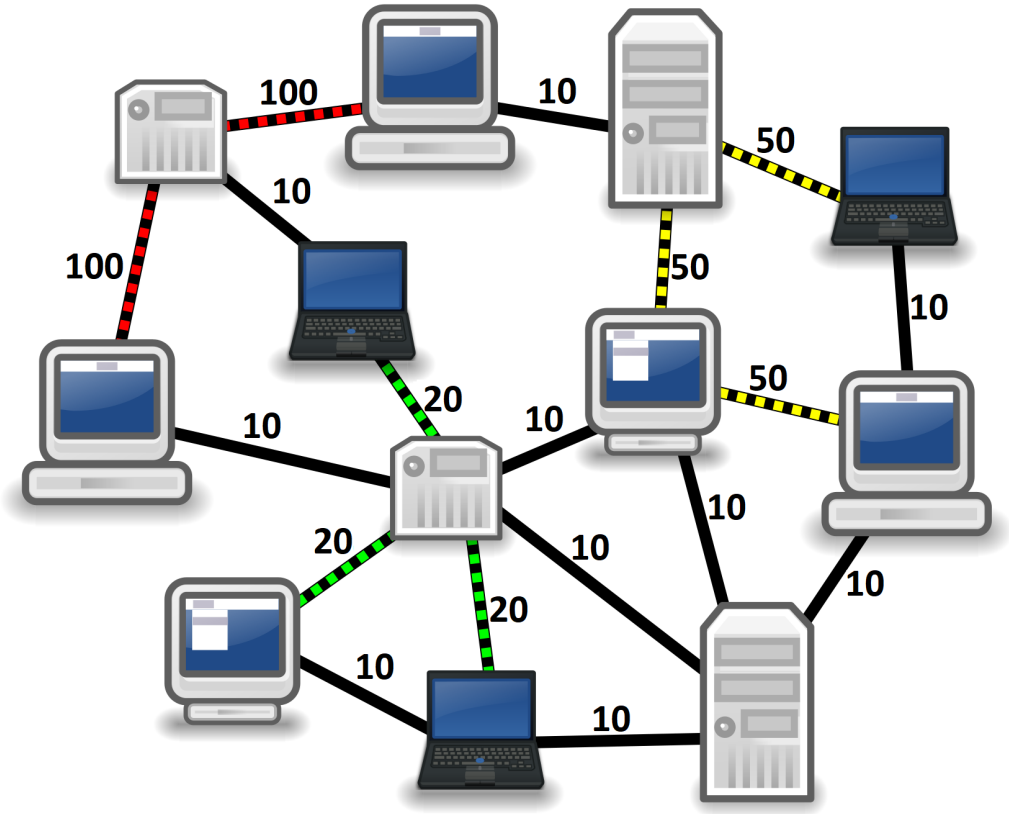
Queremos transmitir uma mensagem de um computador arbitrário para todo os computadores dessa rede.

O custo total da transmissão é a soma do custo de todos os links usados na tarefa.

Problema

Pergunta:

Quais links devem ser usados para trasmitir a mensagem a todos os computadores de forma que o custo total da transmissão seja mínimo?



Problema

Um governador decidiu contratar uma empresa para construir uma rede de gasodutos entre as cidades do estado.

Ao solicitar um orçamento, a empresa lhe entregou um mapa, com os possíveis locais para construção da rede e o preço de construção de cada trecho.

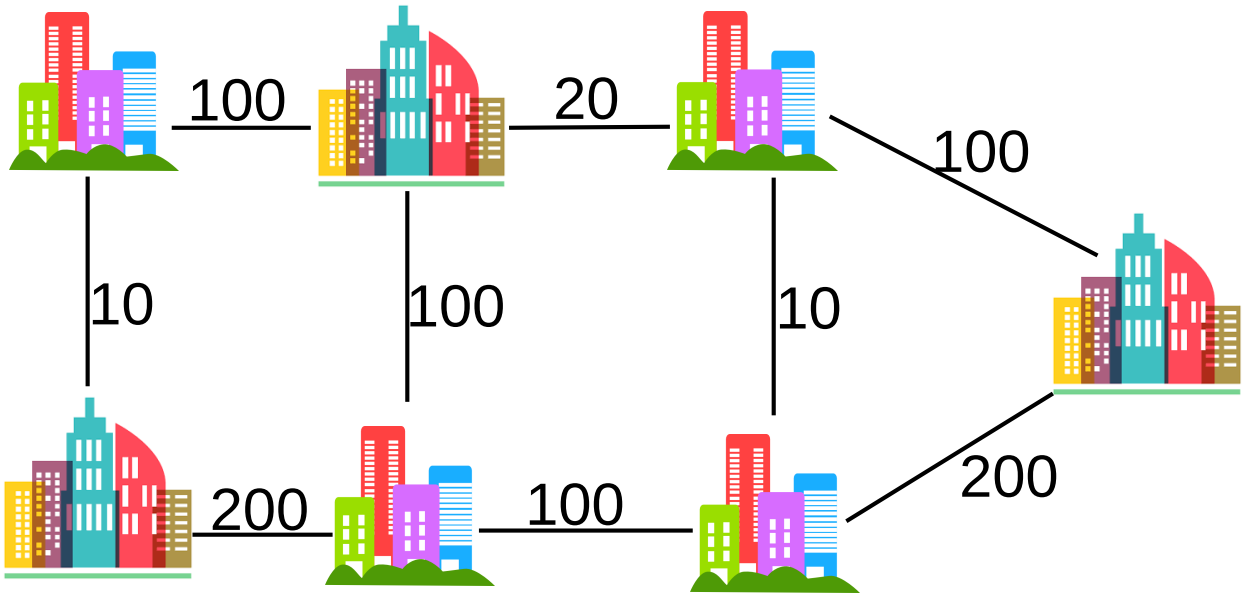
O gobernador quer que todas as cidades seja abastecidas e quer gastar o mínimo possível.

O custo total da obra é a soma dos custos dos trechos que serão efetivamente construídos.

Problema

Pergunta:

Quais trechos de gasoduto devem ser construídos para que todas as cidades sejam abastecidas e o custo total da obra seja mínimo?

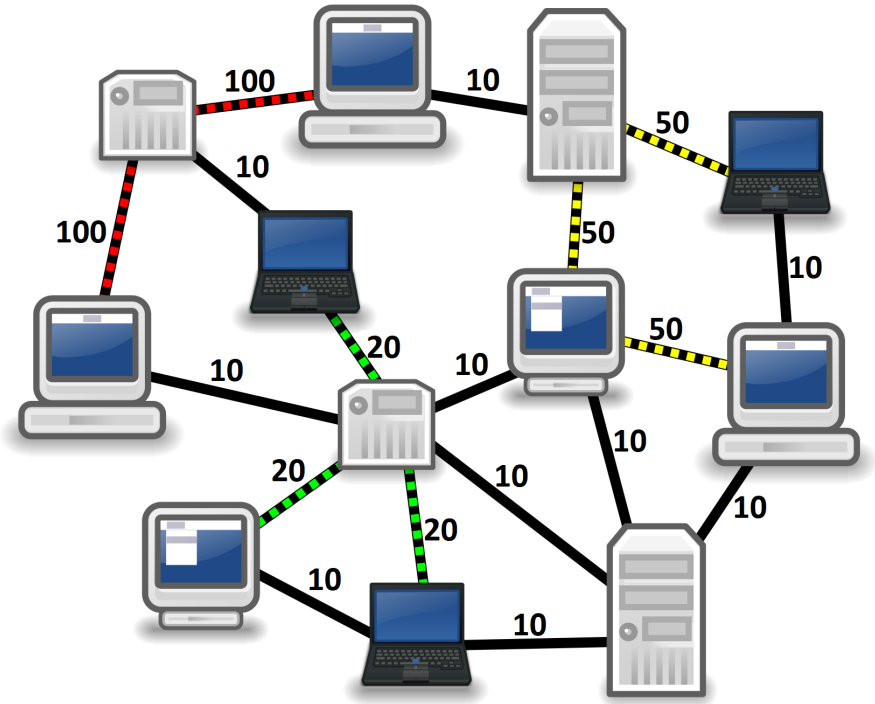


Modelando como um Grafo

Os dois problemas podem ser modelados usando grafos:

Cada computador ou cidade é um vértice.

As conexões são arestas não orientadas com pesos para representar os custos.

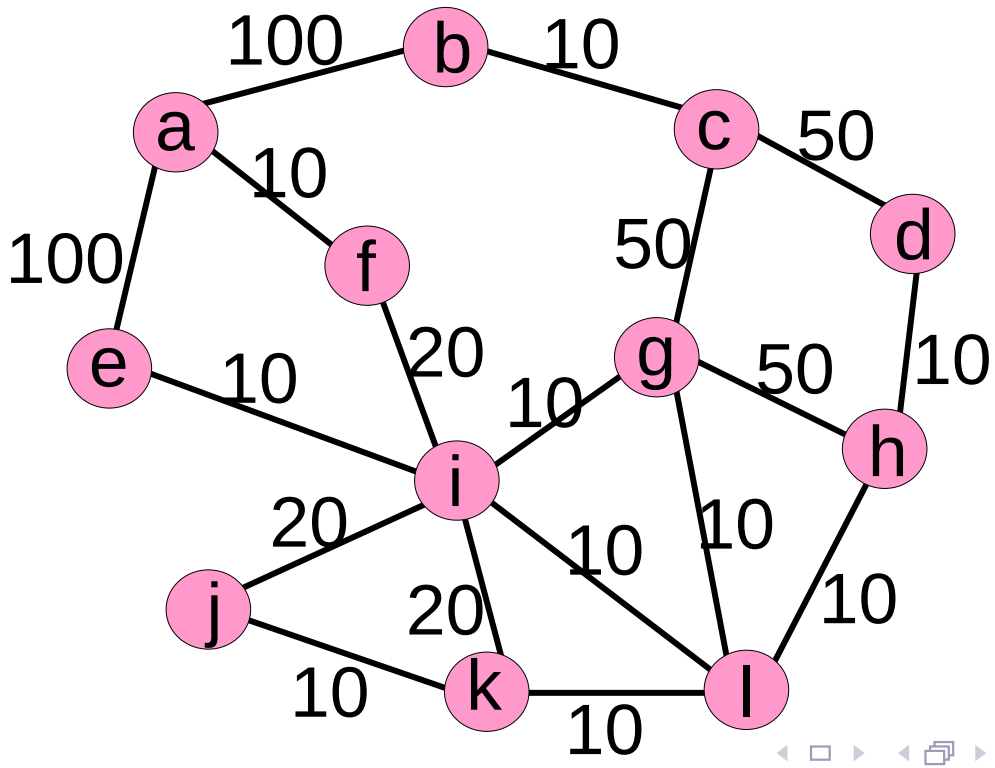


Modelando como um Grafo

Os dois problemas podem ser modelados usando grafos:

Cada computador ou cidade é um vértice.

As conexões são arestas não orientadas com pesos para representar os custos.

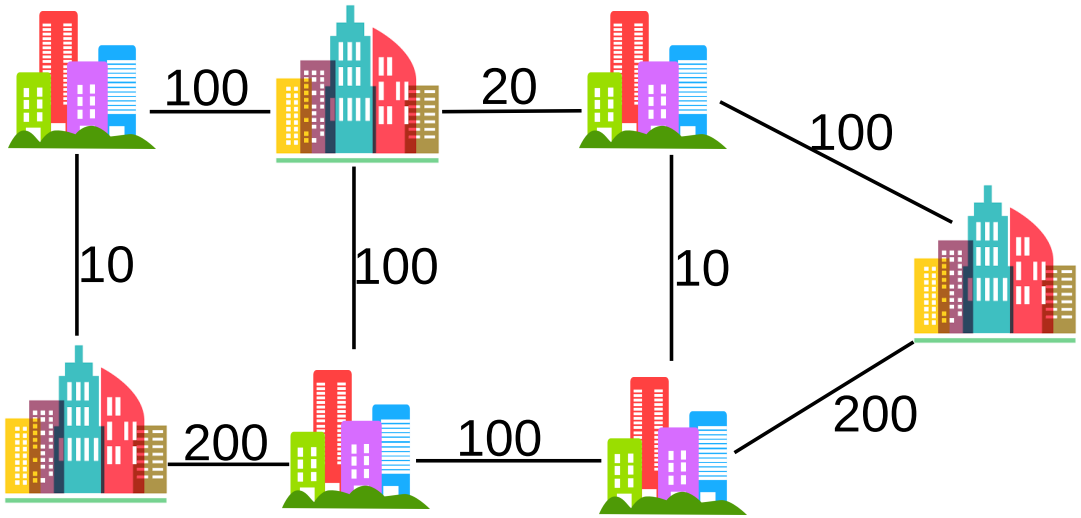


Modelando como um Grafo

Os dois problemas podem ser modelados usando grafos:

Cada computador ou cidade é um vértice.

As conexões são arestas não orientadas com pesos para representar os custos.

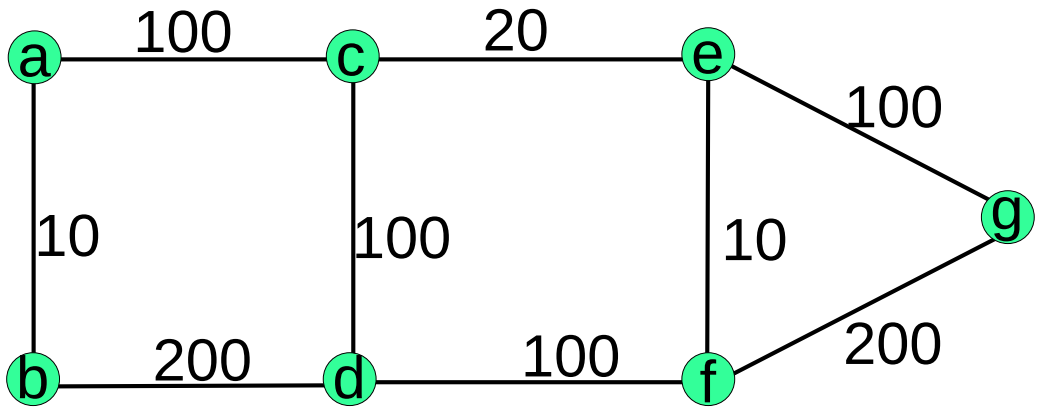


Modelando como um Grafo

Os dois problemas podem ser modelados usando grafos:

Cada computador ou cidade é um vértice.

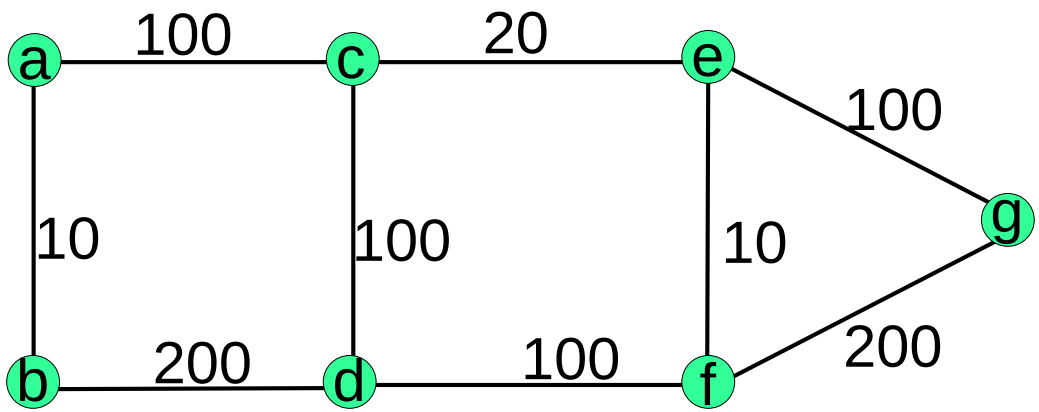
As conexões são arestas não orientadas com pesos para representar os custos.



Modelando como um grafo

Observe que ciclos têm redudância que aumenta o custo total.

Por exemplo, existem dois caminhos entre a e b, entre c e f, entre e e g, etc.



Queremos o subgrafo que seja uma árvore cuja soma das arestas é mínima.

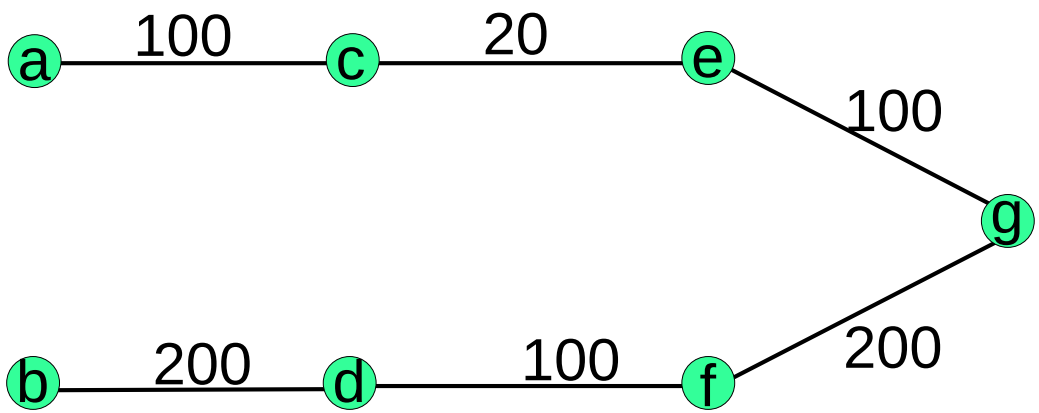
Modelando como um grafo

Definição

Seja G, um grafo ponderado (com peso nas arestas) e não orientado. Uma **árvore geradora de custo mínimo** é um subgrafo gerador de G que é uma árvore e cuja soma dos custos das arestas é a menor dentre todas as árvores geradoras de G.

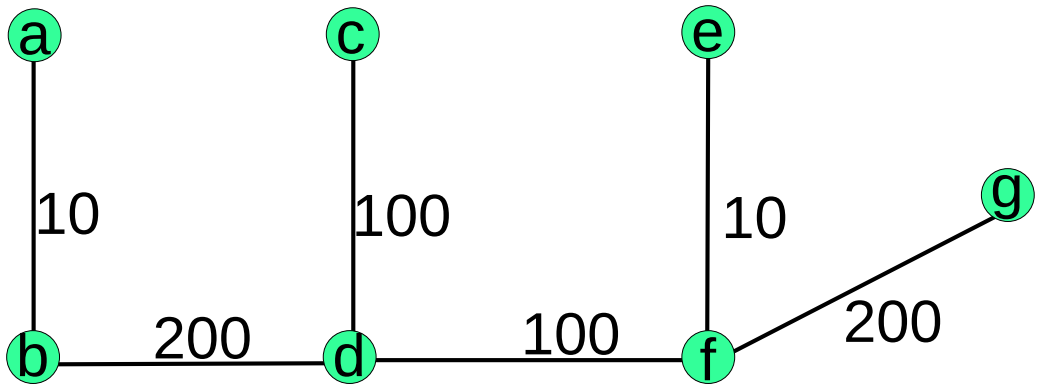
Modelando como um grafo

Árvore geradora do grafo da rede de gasodutos:



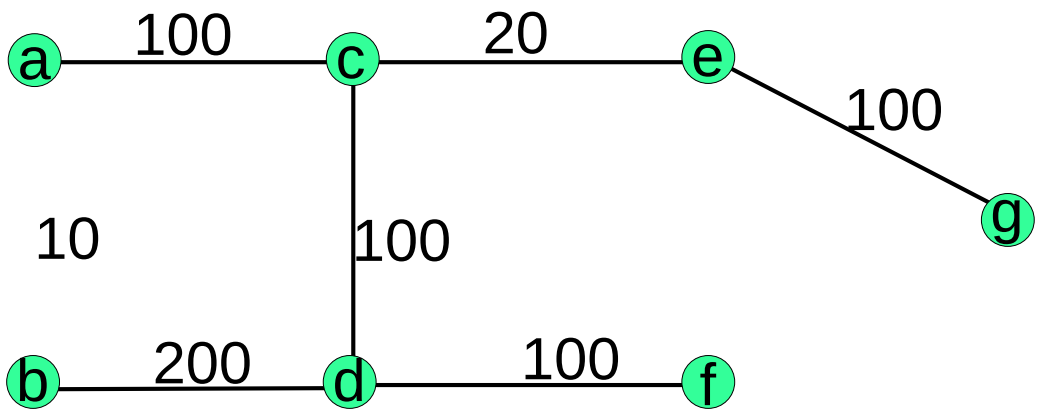
Modelando como um grafo

Árvore geradora do grafo da rede de gasodutos:



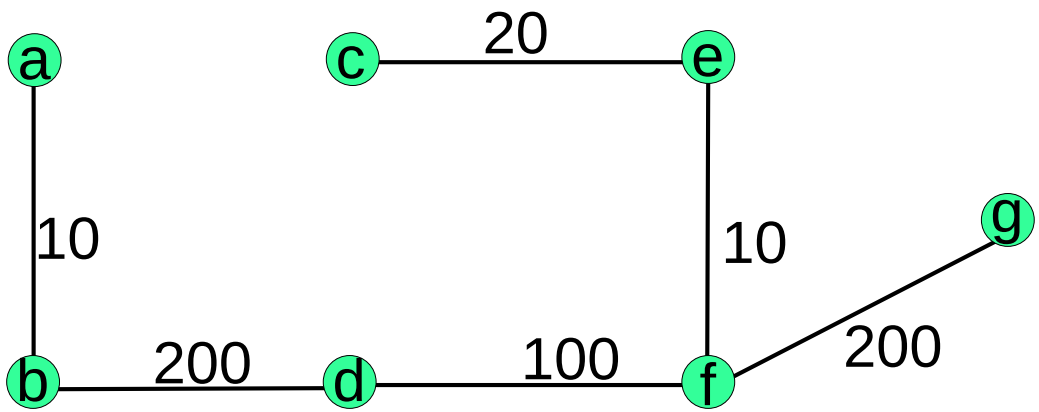
Modelando como um grafo

Árvore geradora do grafo da rede de gasodutos:



Modelando como um grafo

Árvore geradora do grafo da rede de gasodutos:



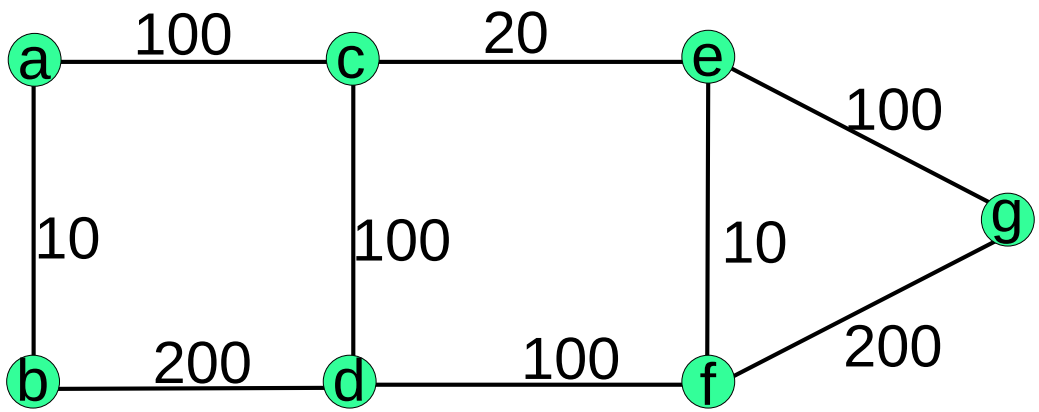
Modelando como um grafo

Problema da Árvore Geradora de Custo Mínimo

Criar um algoritmo eficiente que recebe como entrada um grafo ponderado não orientado e apresenta uma árvore geradora desse grafo que tenha custo mínimo.

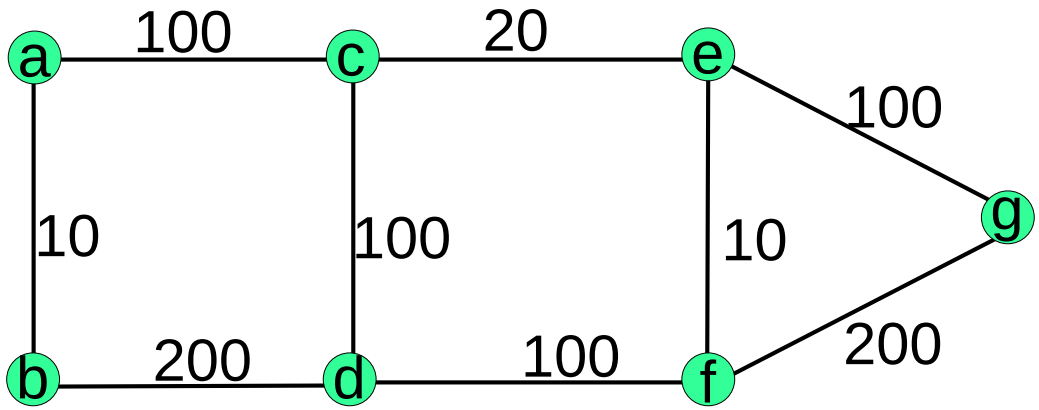
Algoritmo de Kruskal

Se você fosse o governador e tivesse uma quantia limitada de dinheiro, mas quisesse construir a maior quantidade possível de trechos do gasoduto, por quais você começaria?



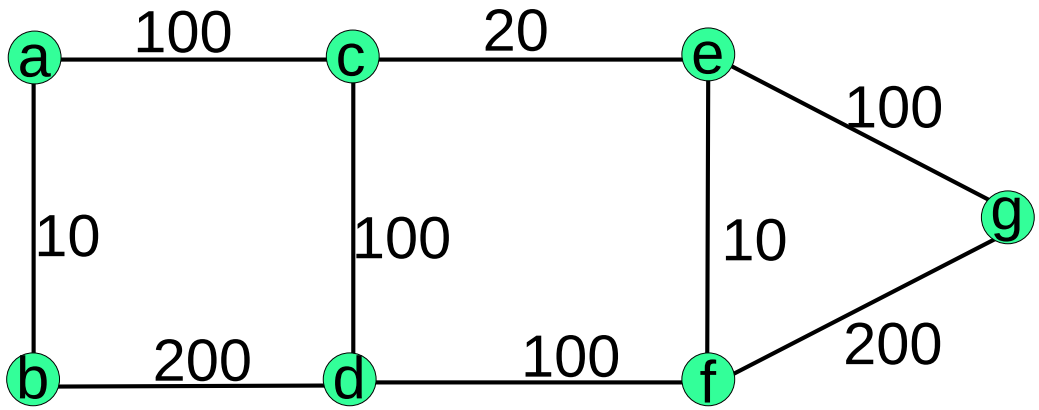
Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)



Algoritmo de Kruskal

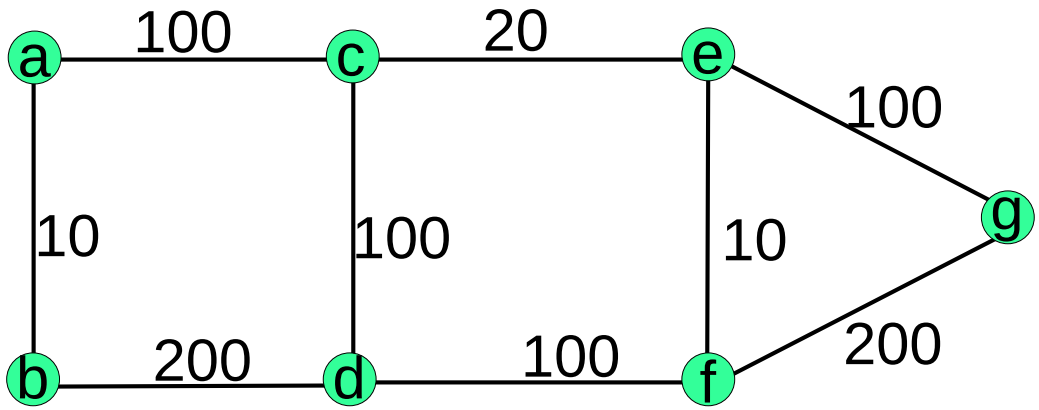
Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)



Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

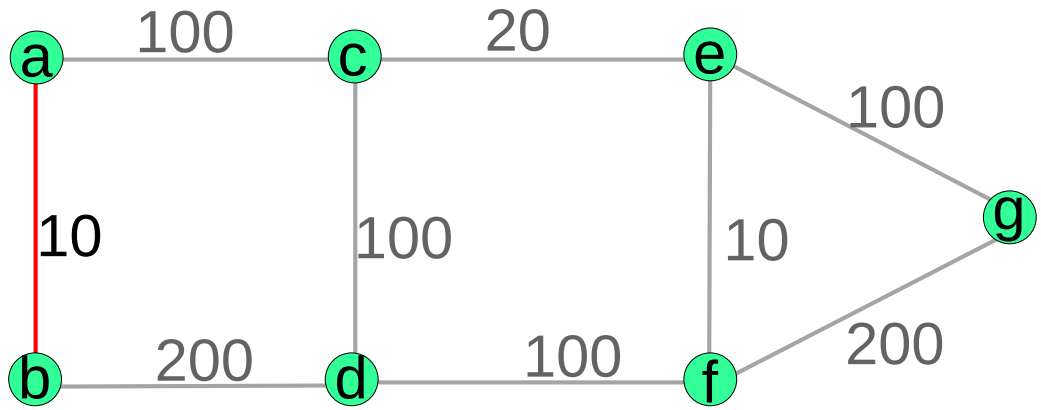


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

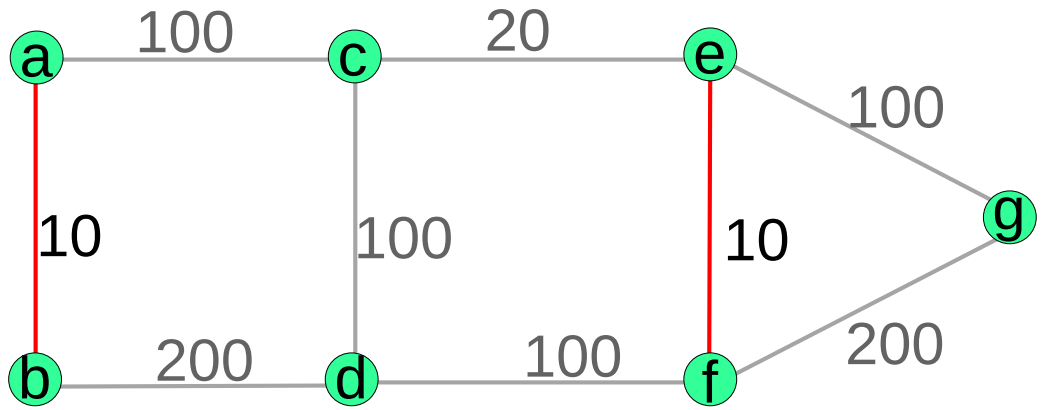


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

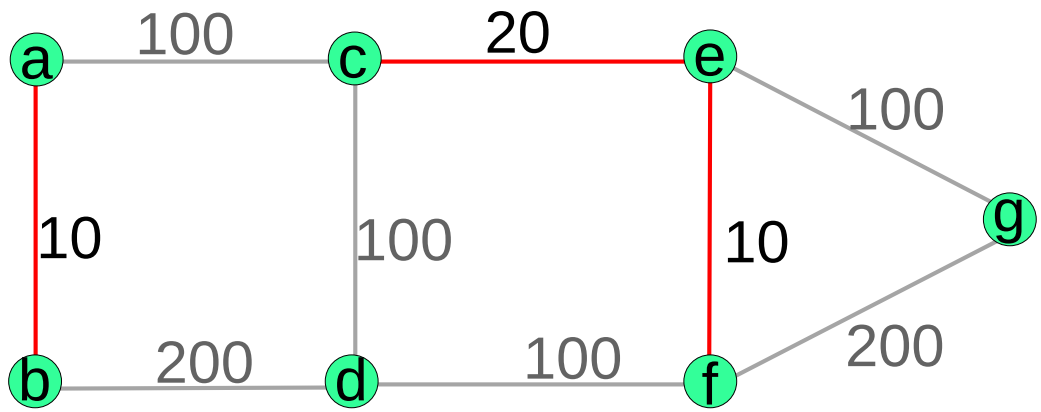


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

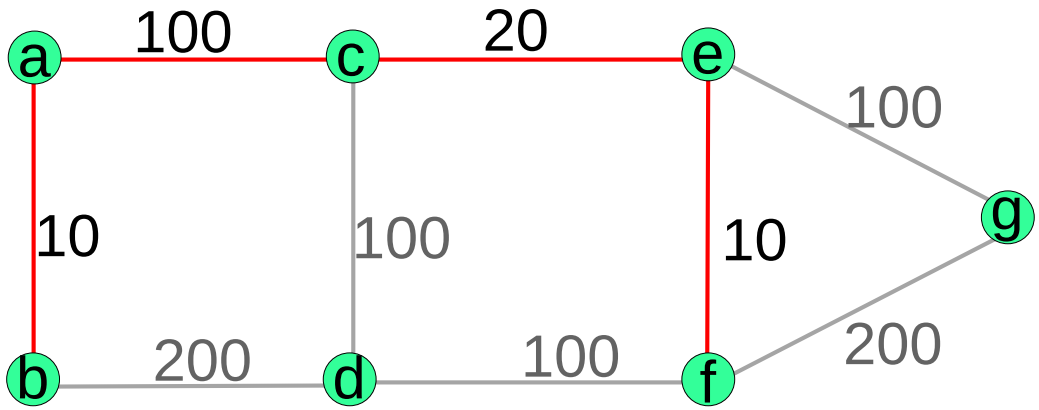


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

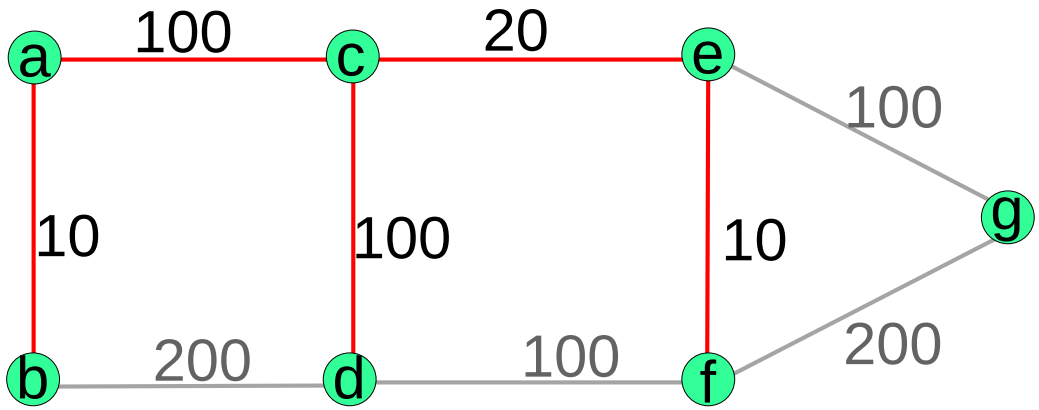


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

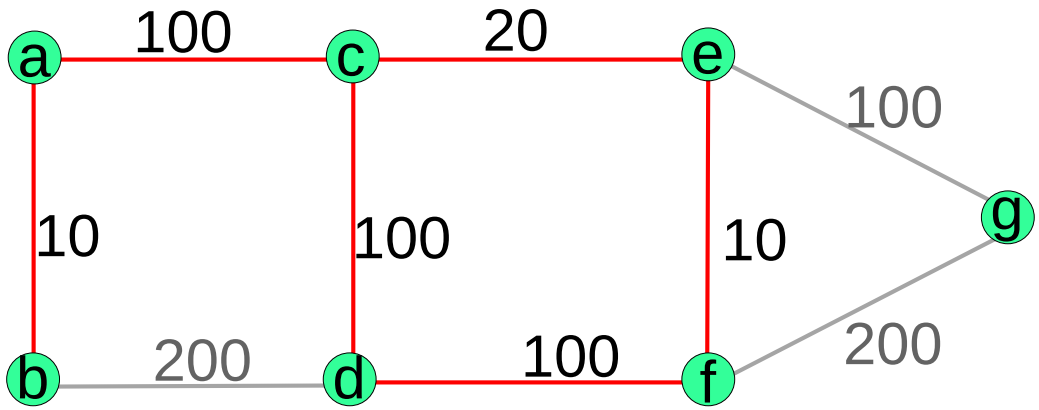


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

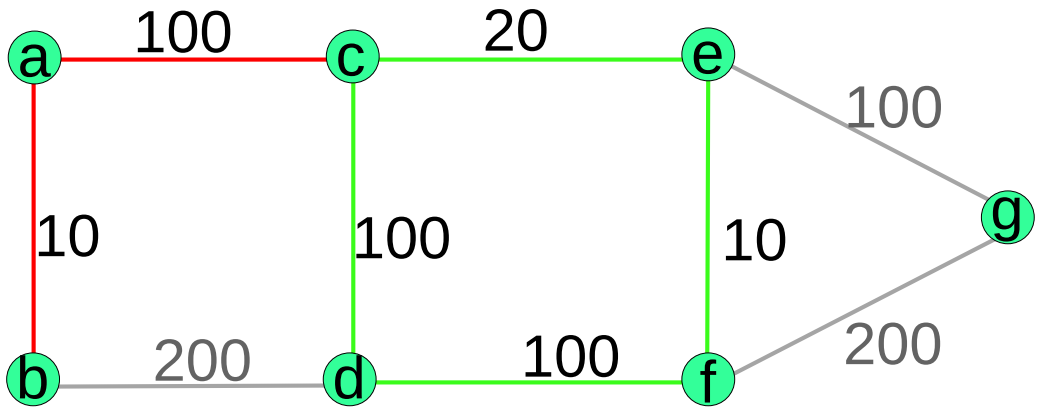


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

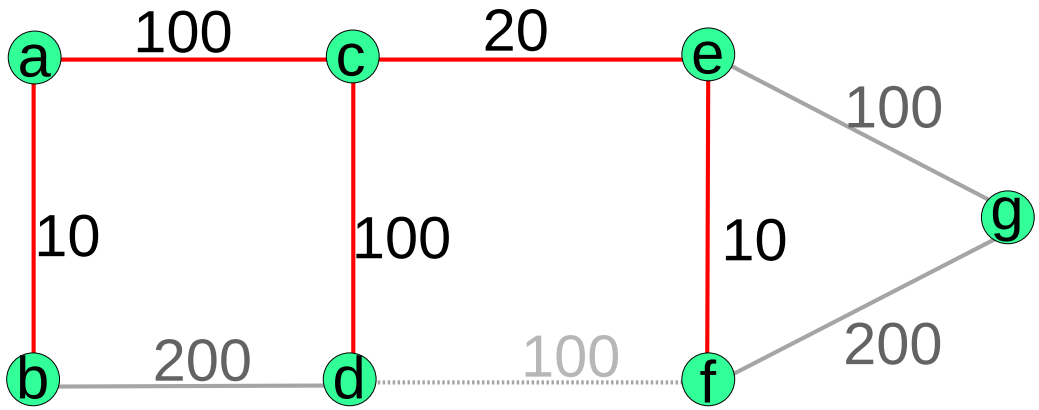


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

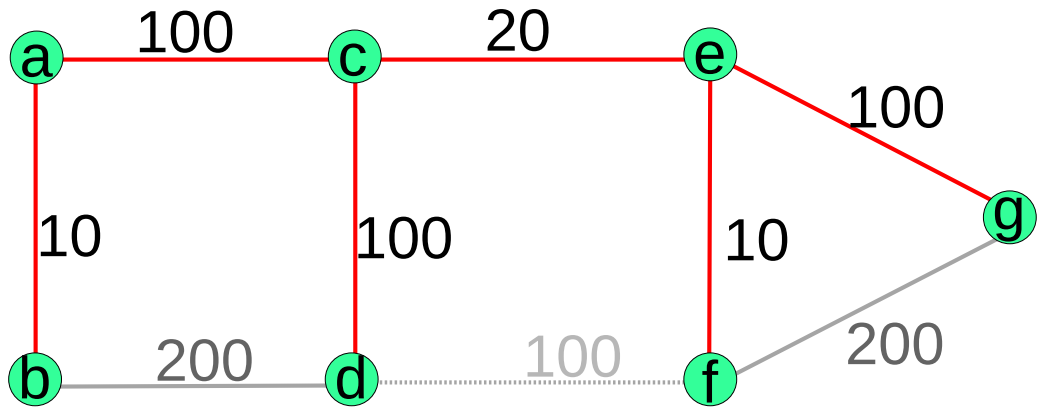


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \{d, f\}; \{e, g\}; \{b, d\}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

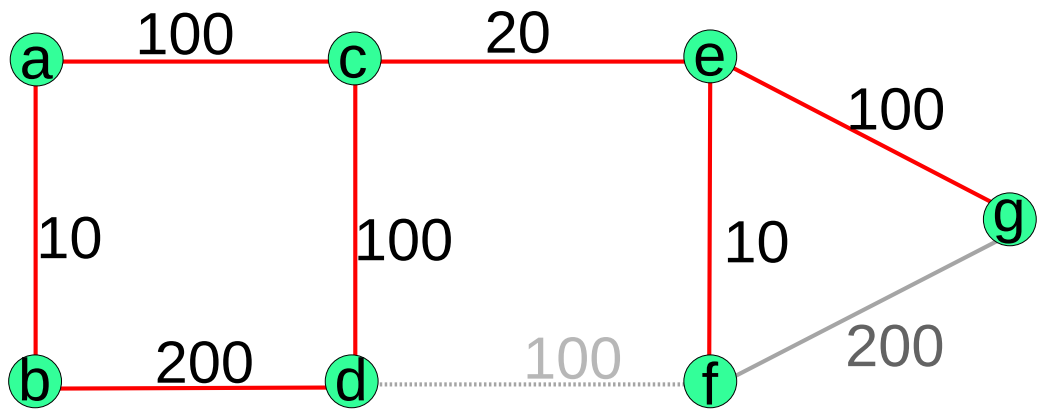


Ordenamos as arestas por custo:
$\{a,b\}; \{e,f\}; \{c,e\}; \{a,c\}; \{c,d\}; \cancel{\{d,f\}}; \{e,g\}; \{b,d\}; \{f,g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

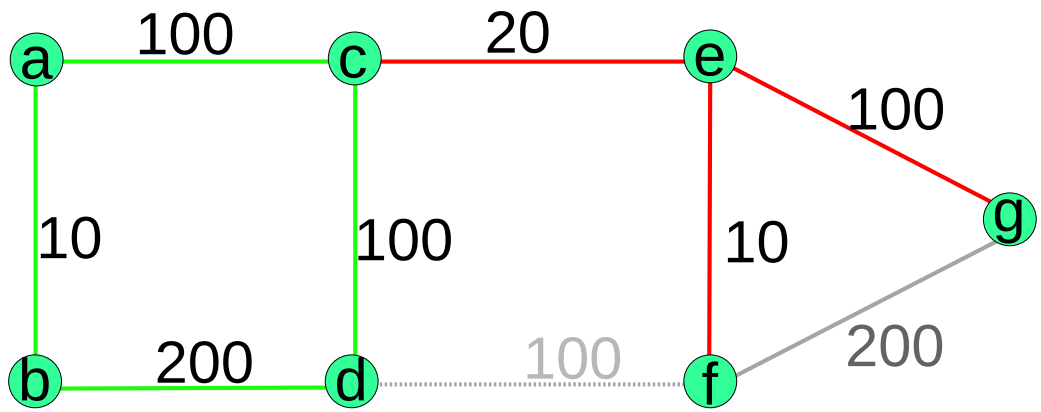


Ordenamos as arestas por custo:
$\{a,b\}; \{e,f\}; \{c,e\}; \{a,c\}; \{c,d\}; \cancel{\{d,f\}}; \{e,g\}; \{b,d\}; \{f,g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

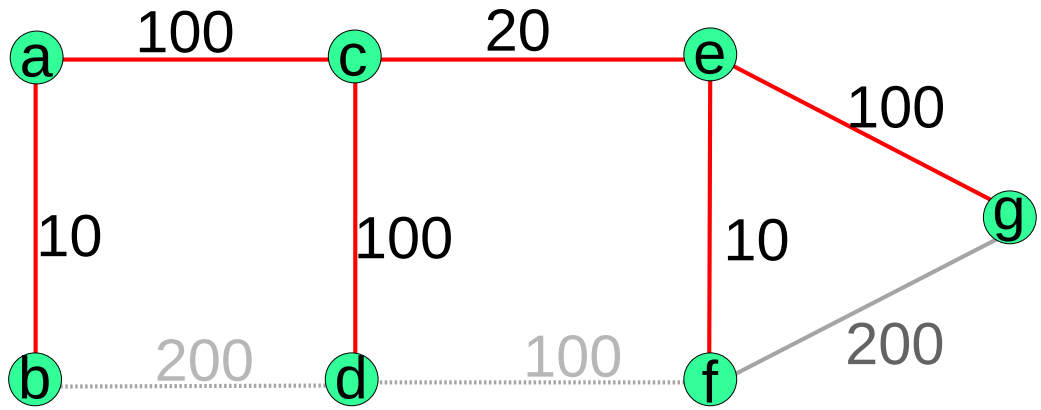


Ordenamos as arestas por custo:
$\{a,b\}; \{e,f\}; \{c,e\}; \{a,c\}; \{c,d\}; \cancel{\{d,f\}}; \{e,g\}; \{b,d\}; \{f,g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

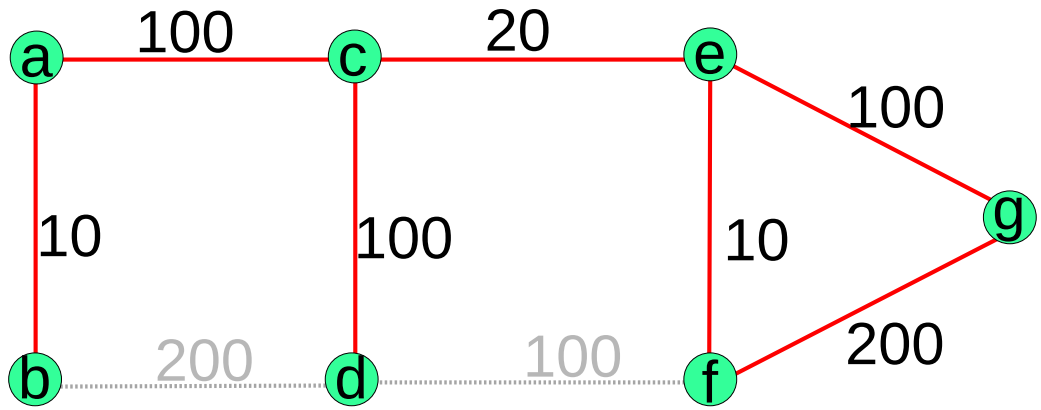


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \cancel{\{d, f\}}; \{e, g\}; \cancel{\{b, d\}}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

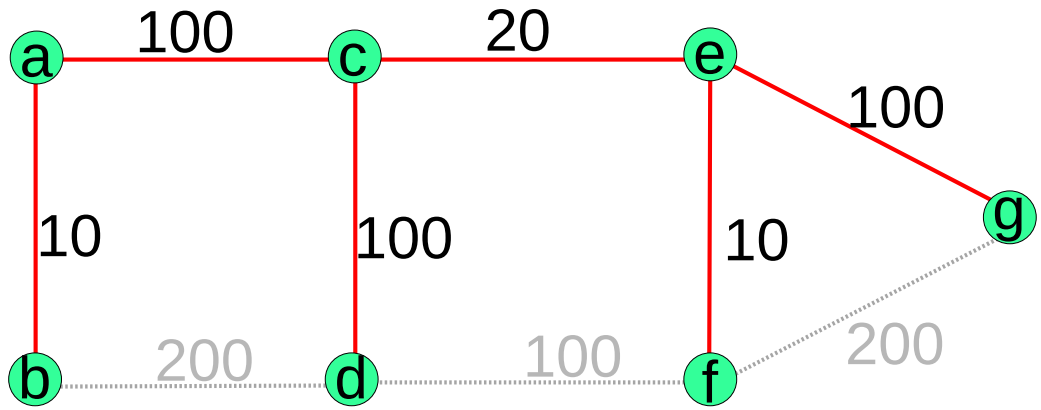


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \cancel{\{d, f\}}; \{e, g\}; \cancel{\{b, d\}}; \{f, g\}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

Algoritmo de Kruskal

Estratégia de Kruskal: Vamos começar pelas mais baratas. (Método de programação guloso!)

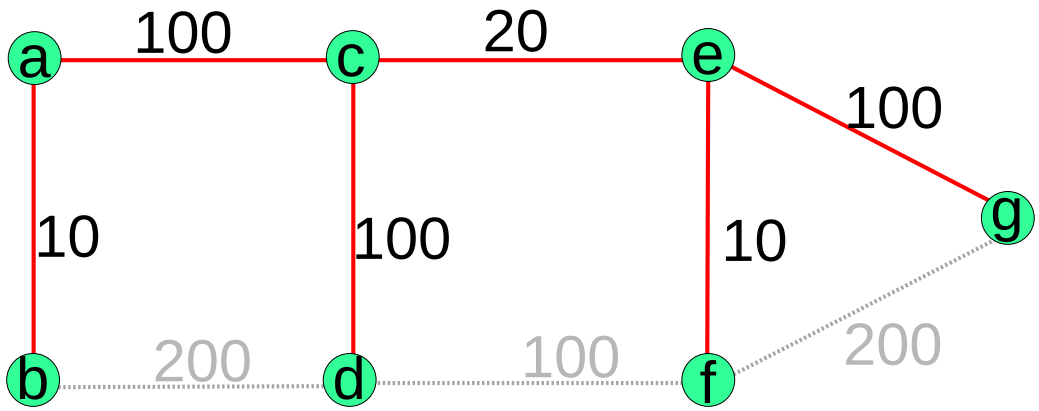


Ordenamos as arestas por custo:
$\{a, b\}; \{e, f\}; \{c, e\}; \{a, c\}; \{c, d\}; \sout{\{d, f\}}; \{e, g\}; \sout{\{b, d\}}; \sout{\{f, g\}}$

As arestas são incluídas na árvore geradora nesta ordem, desde que não formem ciclos.

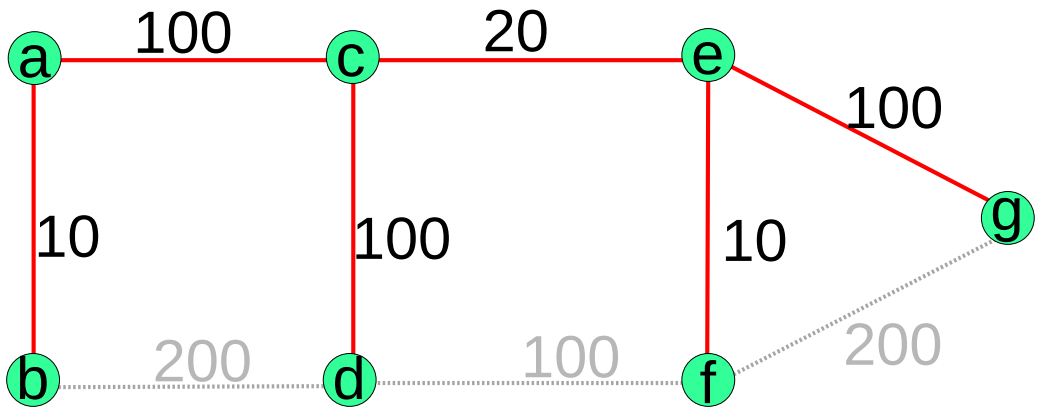
Algoritmo de Kruskal

Por que essa árvore geradora tem custo mínimo?



Algoritmo de Kruskal

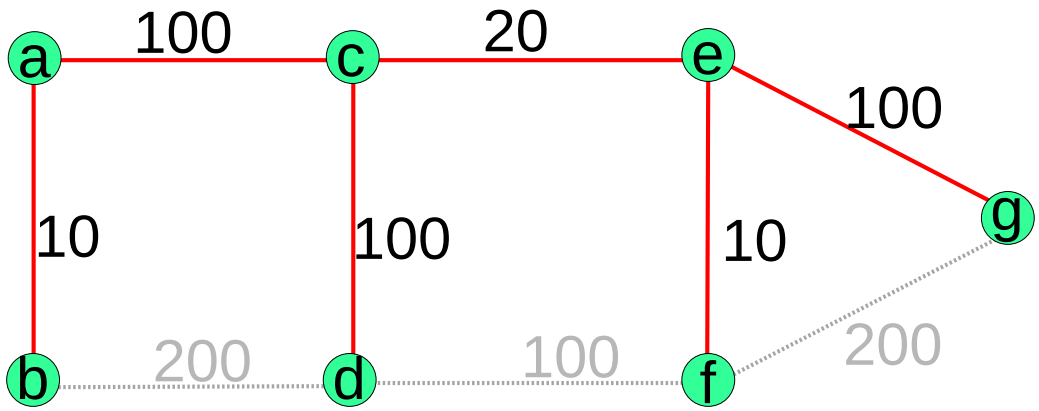
Por que essa árvore geradora tem custo mínimo?



Suponha que não. Então uma das arestas da árvore deve ser substituída por alguma que não está na árvore.

Algoritmo de Kruskal

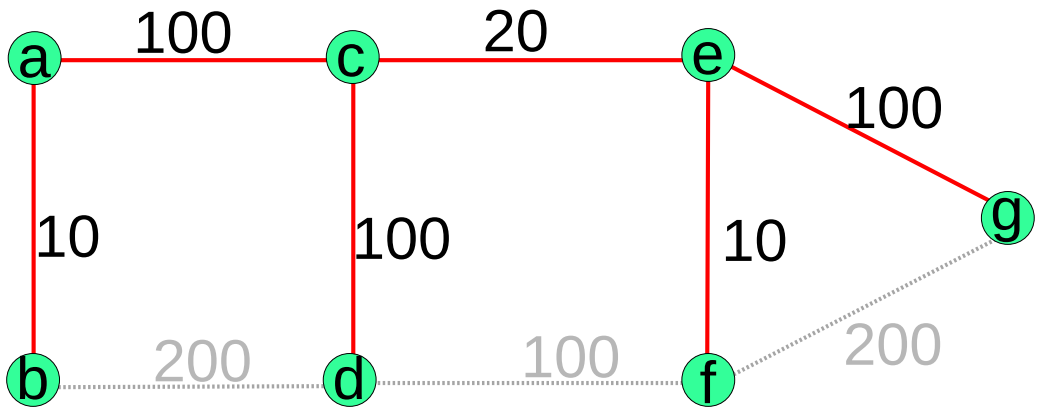
Por que essa árvore geradora tem custo mínimo?



Suponha que não. Então uma das arestas da árvore deve ser substituída por alguma que não está na árvore.
Ao se incluir qualquer uma das arestas, ela forma um ciclo no grafo. Então alguma das arestas desse ciclo deve ser removida.

Algoritmo de Kruskal

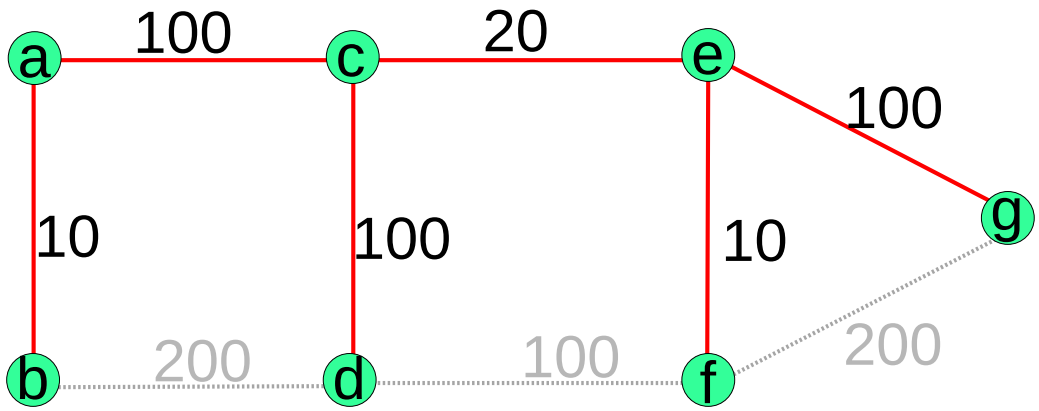
Por que essa árvore geradora tem custo mínimo?



Suponha que não. Então uma das arestas da árvore deve ser substituída por alguma que não está na árvore.
Ao se incluir qualquer uma das arestas, ela forma um ciclo no grafo. Então alguma das arestas desse ciclo deve ser removida.
Mas se essa foi a aresta que fechou o ciclo, então foi a última do ciclo que tentamos inserir.

Algoritmo de Kruskal

Por que essa árvore geradora tem custo mínimo?

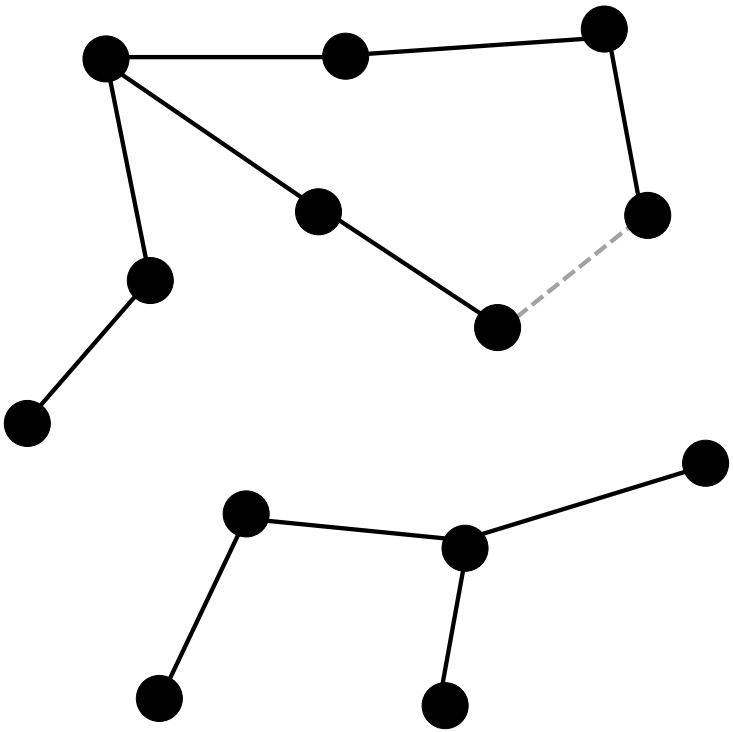


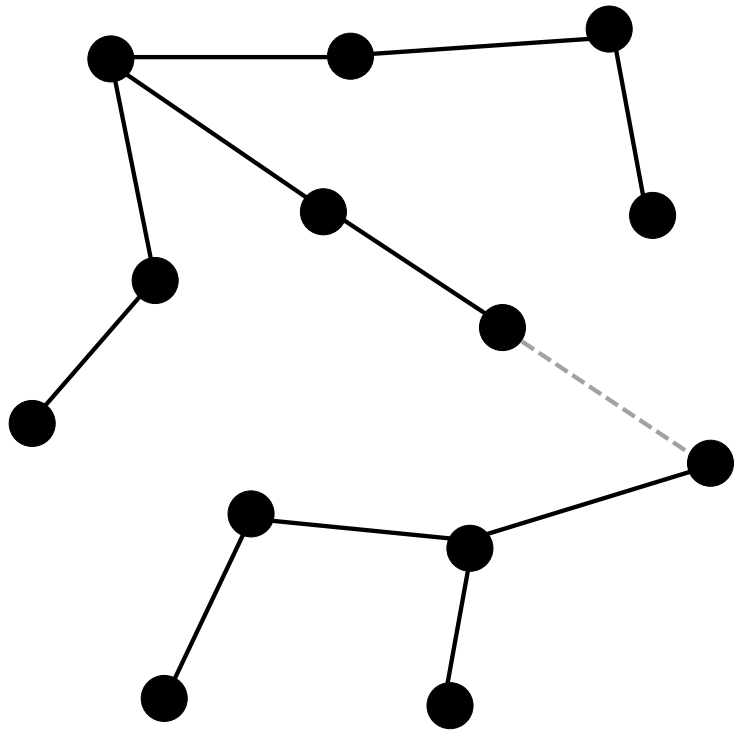
Suponha que não. Então uma das arestas da árvore deve ser substituída por alguma que não está na árvore.
Ao se incluir qualquer uma das arestas, ela forma um ciclo no grafo. Então alguma das arestas desse ciclo deve ser removida.
Mas se essa foi a aresta que fechou o ciclo, então foi a última do ciclo que tentamos inserir.
Então seu custo é maior ou igual que o das outras arestas do ciclo e a troca não diminui o custo total (permanece igual ou aumenta).

Algoritmo de Kruskal

Como verificar ciclos?

Ciclos se formam quando tentamos inserir uma aresta entre dois vértices da mesma árvore.

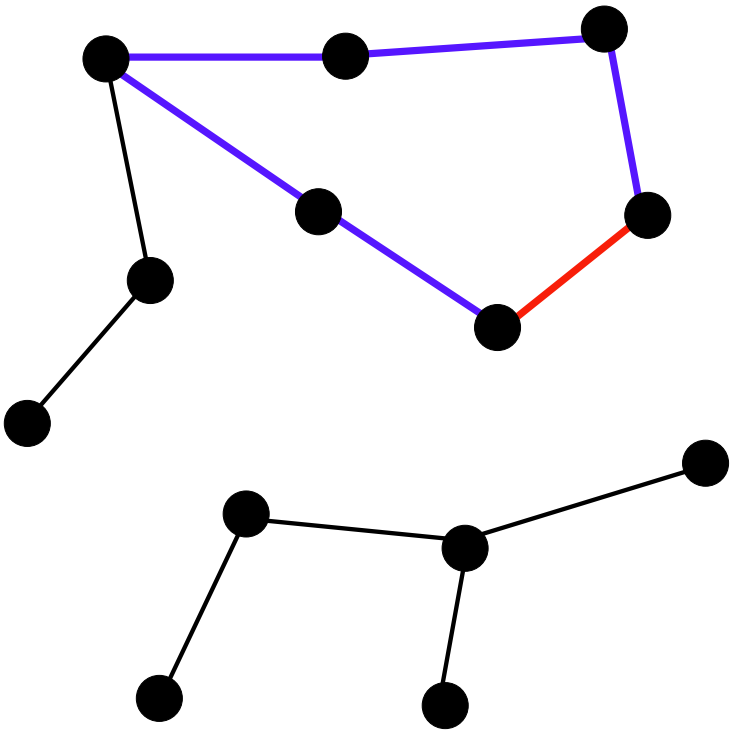


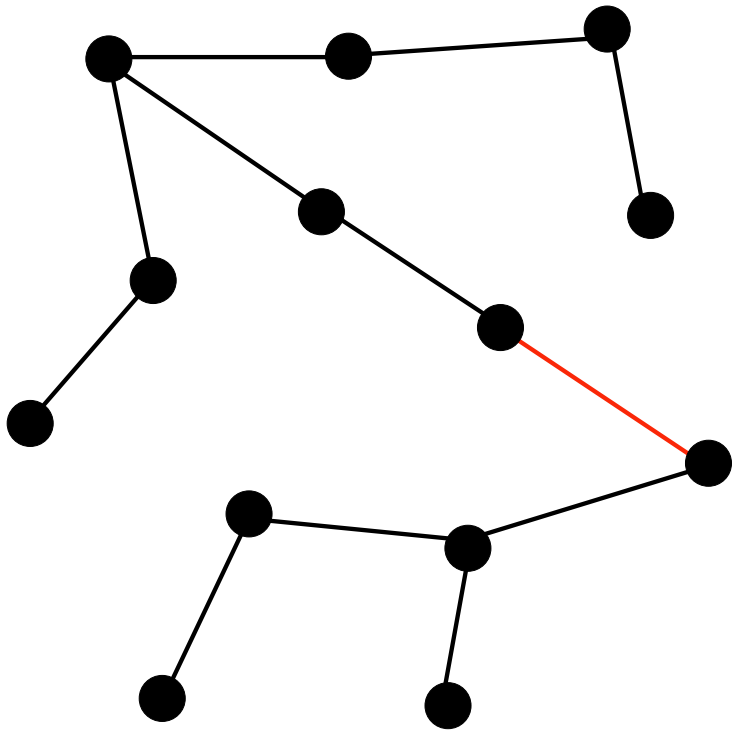


Algoritmo de Kruskal

Como verificar ciclos?

Ciclos se formam quando tentamos inserir uma aresta entre dois vértices da mesma árvore.





Algoritmo de Kruskal

Como verificar ciclos?

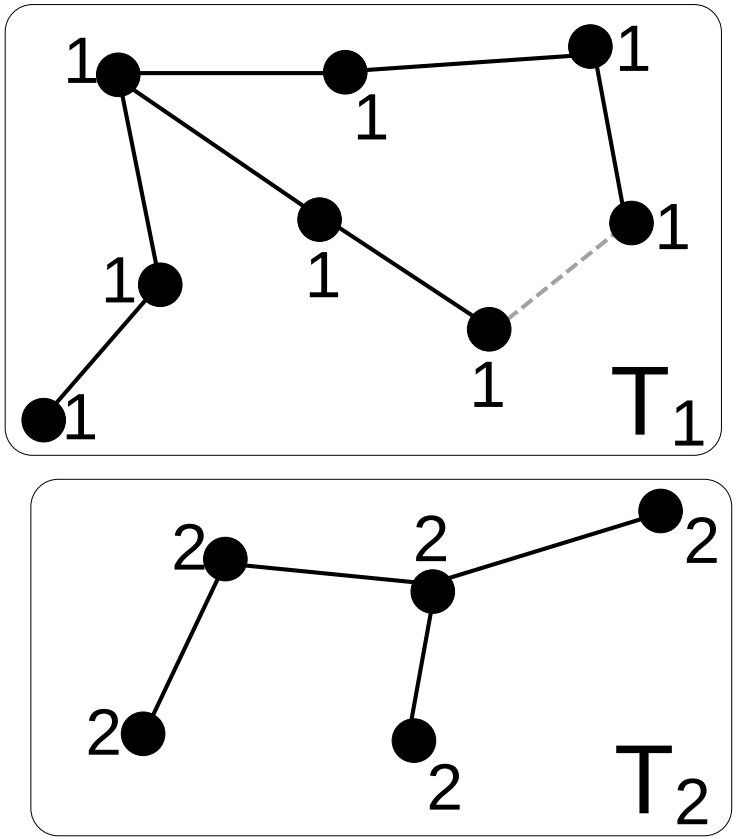
É suficiente que:

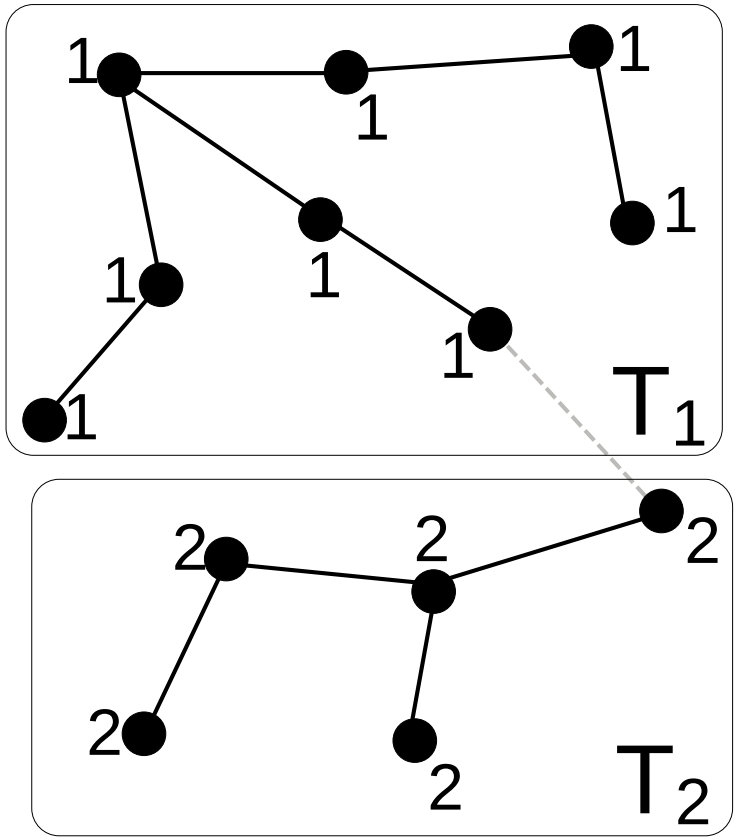
- exista uma identificação das árvores;
- cada vértice saiba a qual árvore pertence.

Algoritmo de Kruskal

Como verificar ciclos?

Ciclos se formam quando tentamos inserir uma aresta entre dois vértices da mesma árvore.

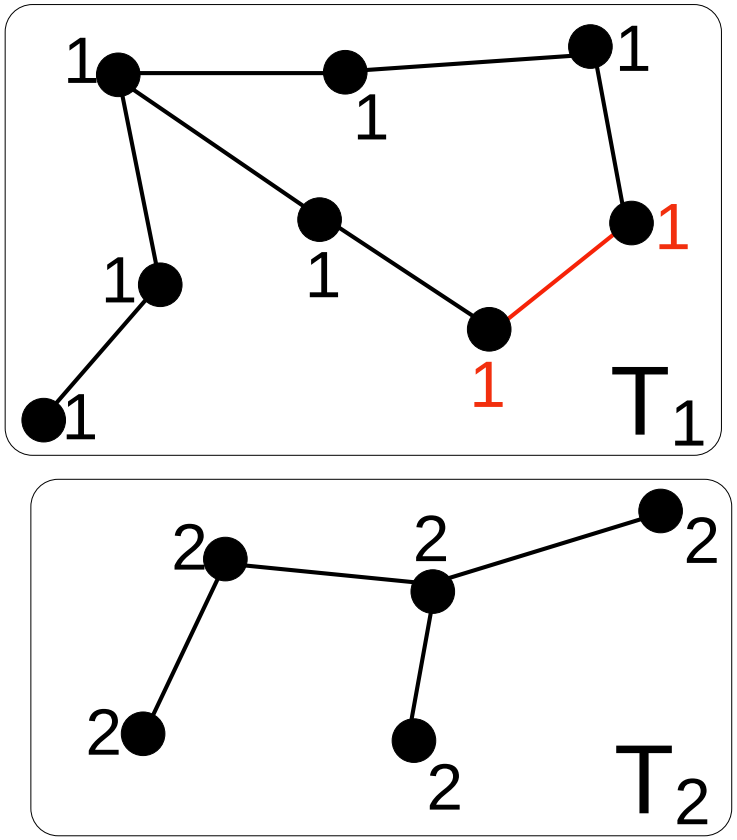


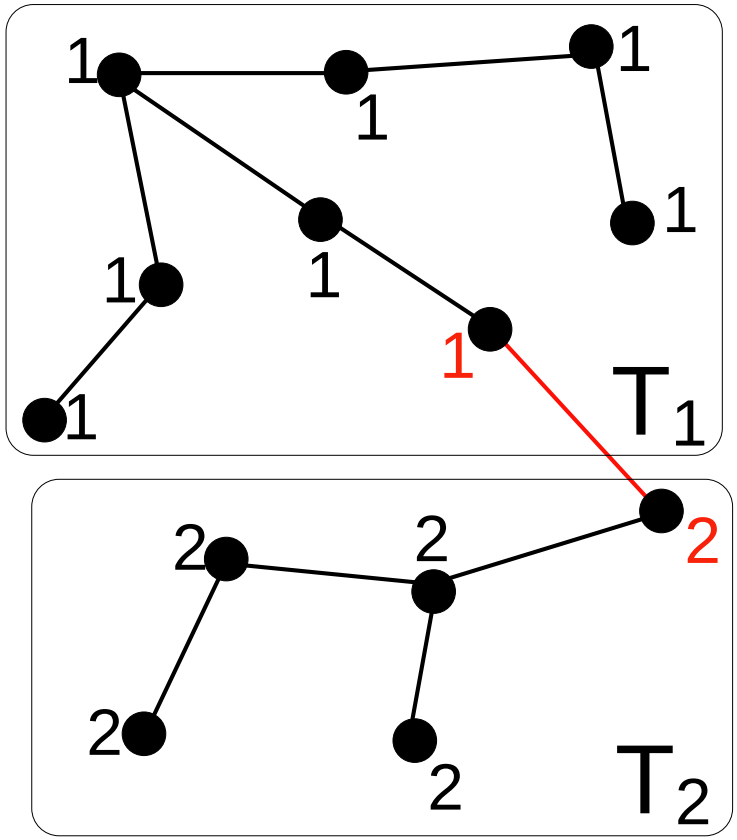


Algoritmo de Kruskal

Como verificar ciclos?

Ciclos se formam quando tentamos inserir uma aresta entre dois vértices da mesma árvore.

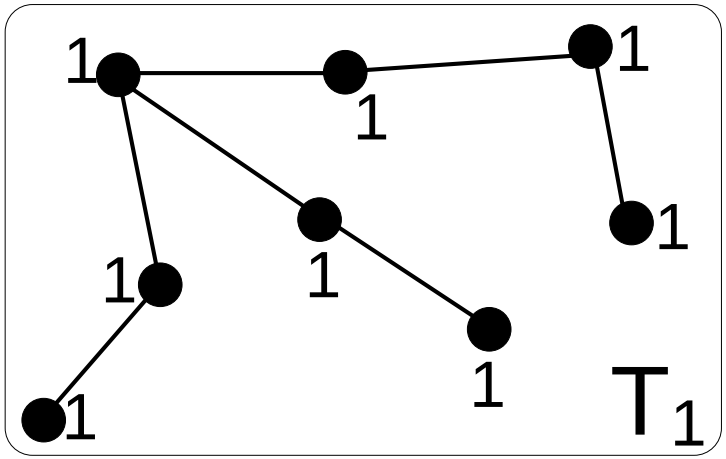


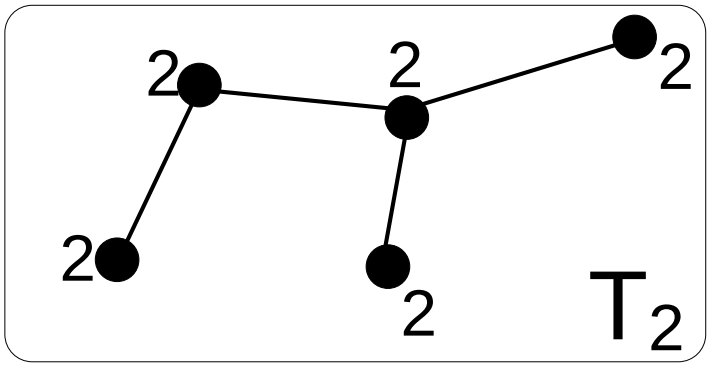


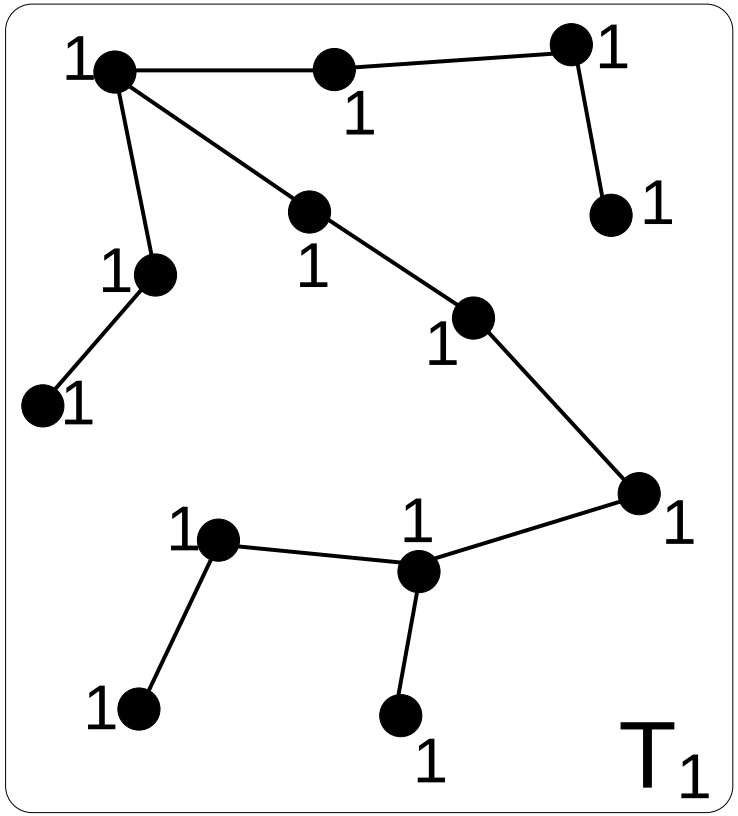
Algoritmo de Kruskal

Como verificar ciclos?

Ciclos se formam quando tentamos inserir uma aresta entre dois vértices da mesma árvore.







Algoritmo de Kruskal

Kruskal(G)

Ordene as arestas de G, em ordem crescente de custo
$c \leftarrow 1$; $T \leftarrow \{\ \}$
Para cada vértice $v \in V(G)$ faça:
    $comp(v) \leftarrow c$;
    $c \leftarrow c + 1$;
Para cada aresta $\{u, v\}$, na ordem estabelecida faça:
    Se $comp(u) \neq comp(v)$ então:
        $T \leftarrow T + \{u, v\}$;
        Para cada vértice $w \in V(G)$ faça:
            Se $comp(w) = comp(v)$, então:
                $comp(w) \leftarrow comp(u)$;

Kruskal(G)

Ordene as arestas de G, em ordem crescente de custo $O(m \log m)$

$c \leftarrow 1$; $T \leftarrow \{\ \}$ $O(1)$

Para cada vértice $v \in V(G)$ faça: $O(n)$

    $comp(v) \leftarrow c$; $O(n)$

    $c \leftarrow c + 1$; $O(n)$

Para cada aresta $\{u, v\}$, na ordem estabelecida faça: $O(m)$

    Se $comp(u) \neq comp(v)$ então: $O(m)$

        $T \leftarrow T + \{u, v\}$; $O(n)$ (linha anterior verdadeira $n - 1$ vezes)

        Para cada vértice $w \in V(G)$ faça: $O(n^2)$

            Se $comp(w) = comp(v)$, então: $O(n^2)$

                $comp(w) \leftarrow comp(u)$; $O(n^2)$

Complexidade do algoritmo: $O(n^2 + m \log m)$.

Como $m < n^2$: $\log m \in O(\log n)$.

Complexidade do algoritmo: $O(n^2 + m \log n)$.

Algoritmo de Kruskal

É possível melhorar a etapa de identificação de ciclos.

Para melhorar: usamos union-find!

Union-Find

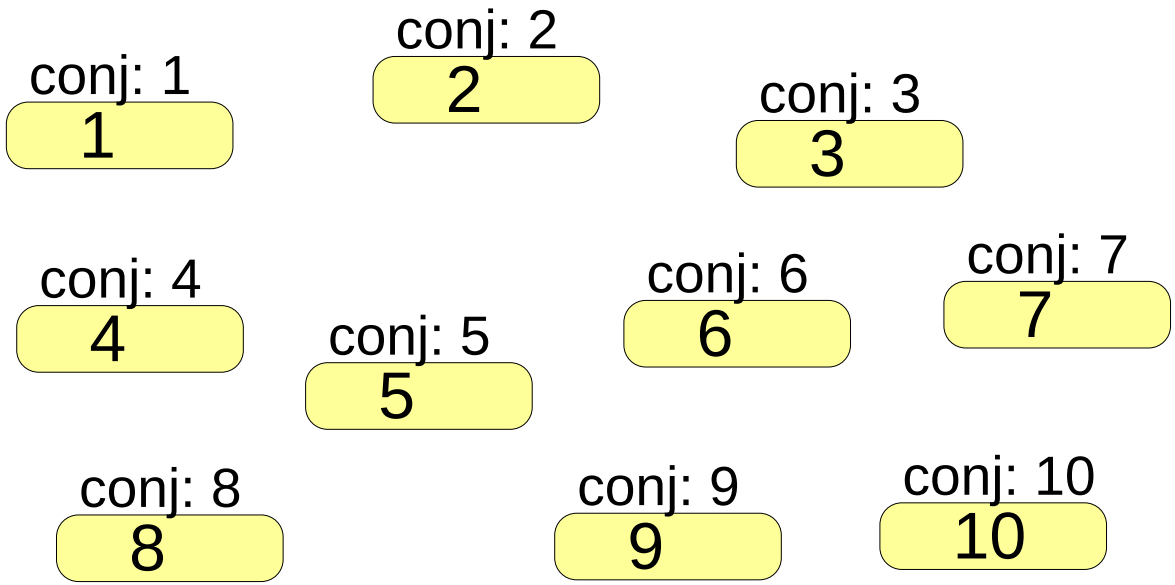
Problema do Union-find

Dado um conjunto de conjuntos de elementos, pode-se realizar dois tipos de operações:

Union: unir dois conjuntos em um único conjunto.

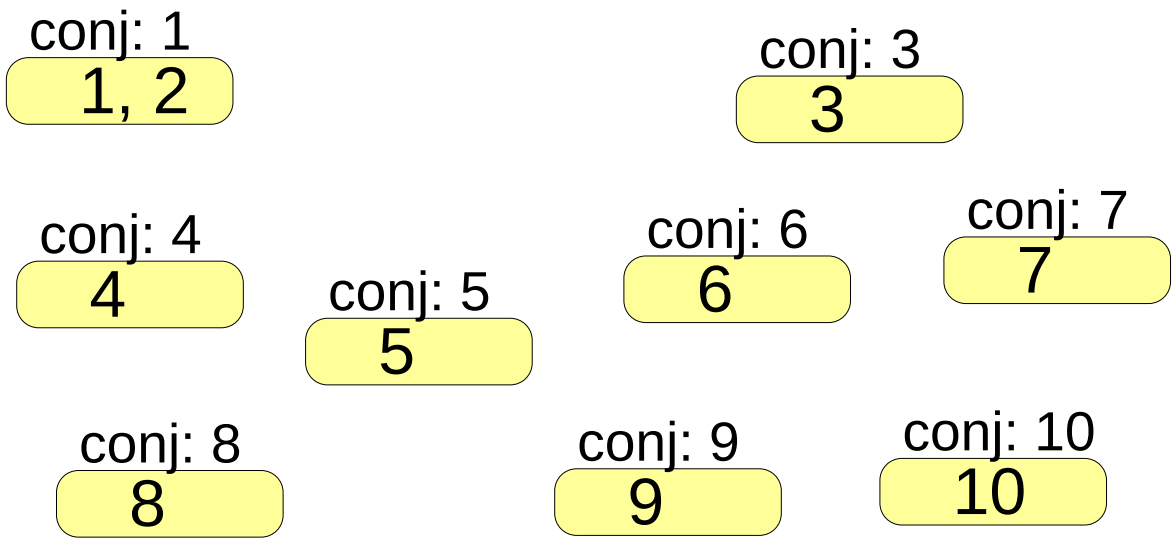
Find: descobrir qual é o conjunto ao qual um determinado elemento pertence.

Union-Find


conj: 1
1
conj: 2
2
conj: 3
3
conj: 4
4
conj: 5
5
conj: 6
6
conj: 7
7
conj: 8
8
conj: 9
9
conj: 10
10

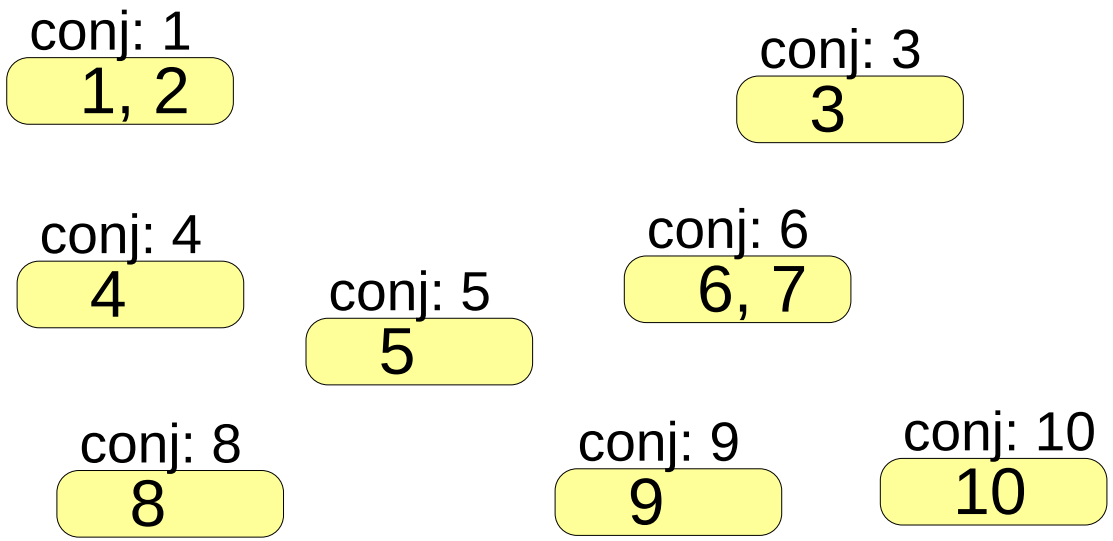
Union-Find

Union(1, 2):



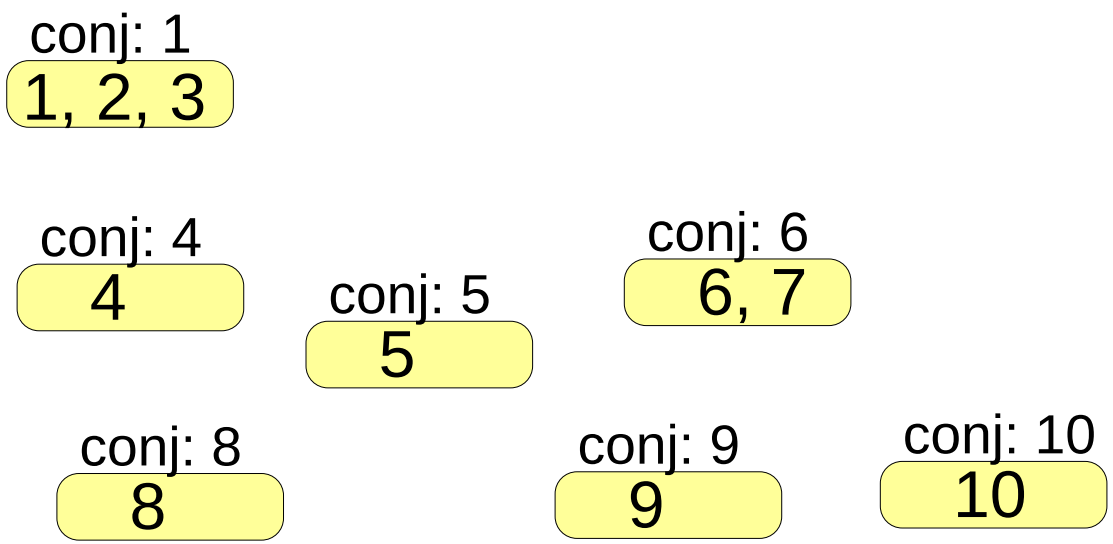
Union-Find

Union(6, 7):



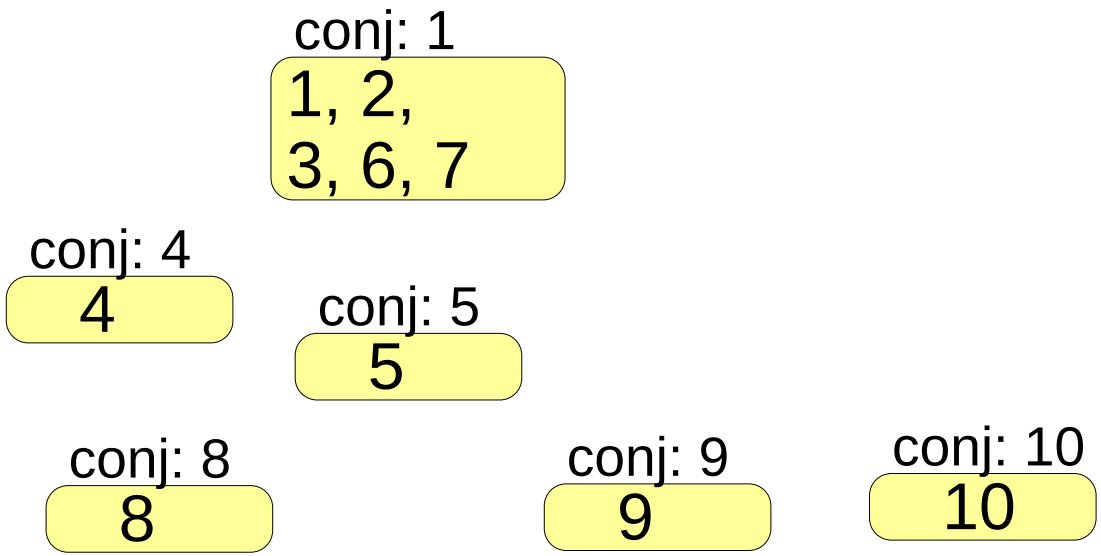
Union-Find

Union(1, 3):



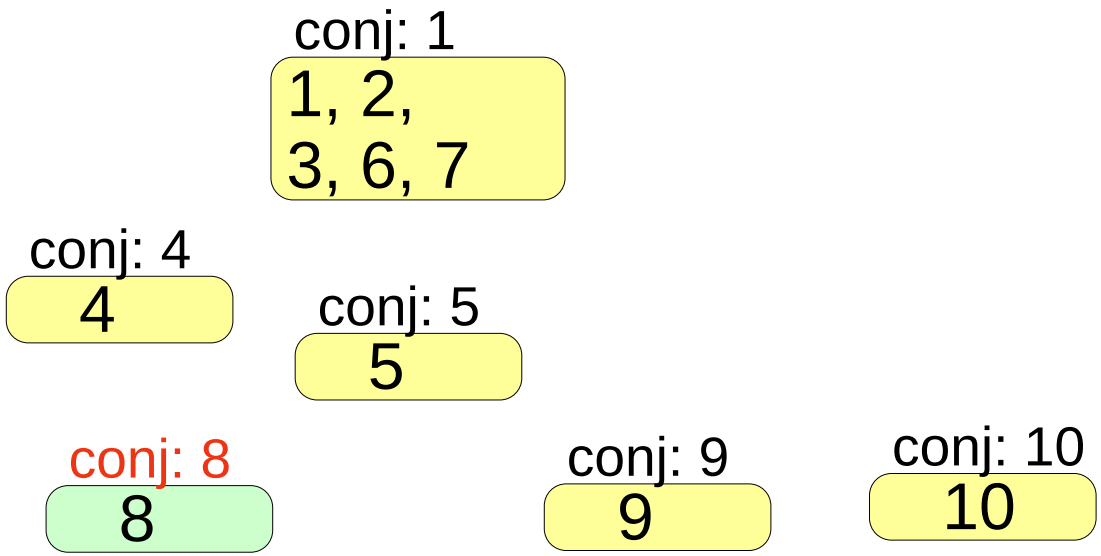
Union-Find

Union(1, 6):



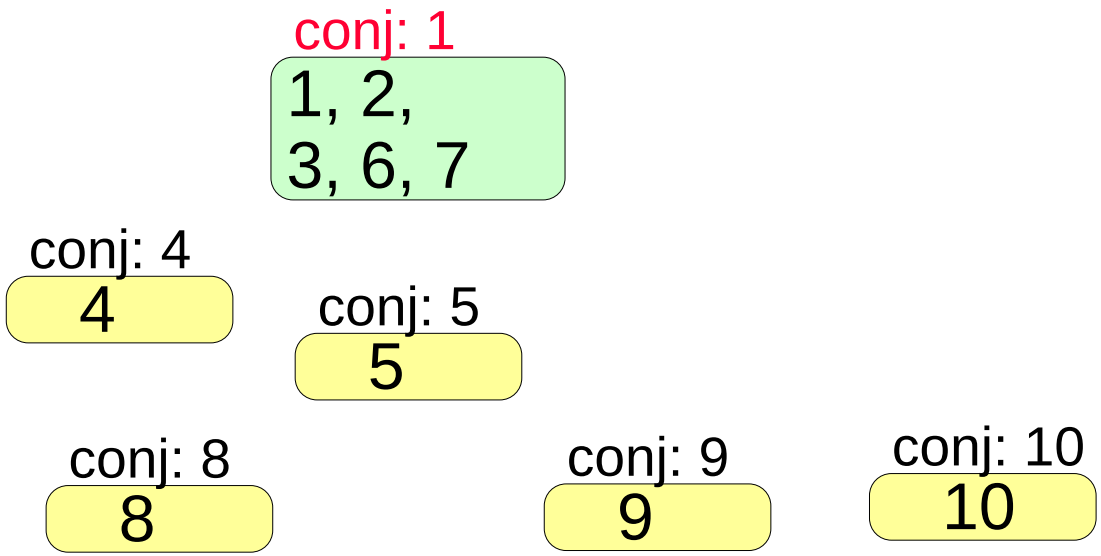
Union-Find

Find(8):



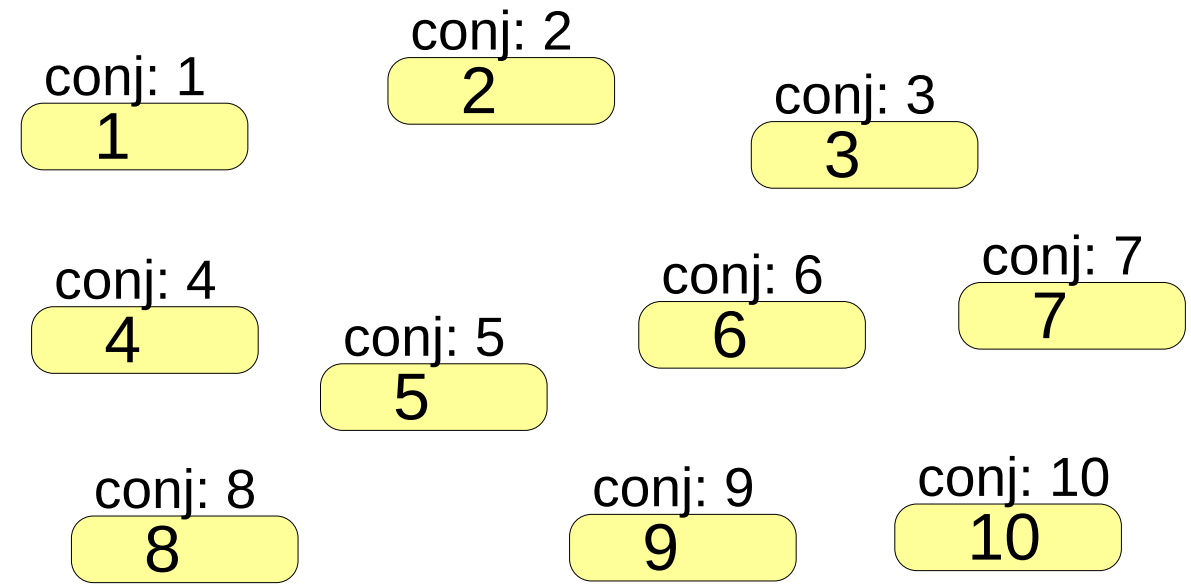
Union-Find

Find(6):



Union-Find

Para resolver computacionalmente: um vértice torna-se dono do grupo.

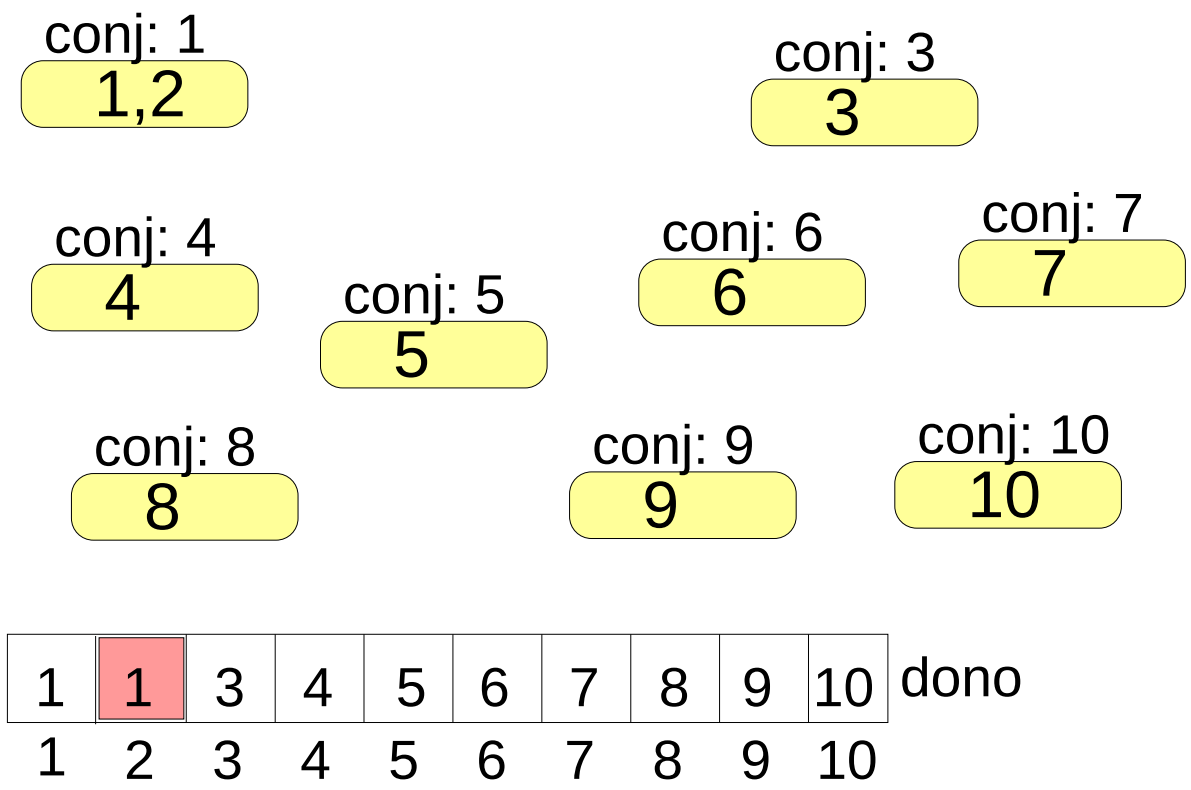


1	2	3	4	5	6	7	8	9	10	dono
1	2	3	4	5	6	7	8	9	10	

Union-Find

Para resolver computacionalmente: ao fazer Union(u, v), o dono do conjunto de u passa a ser dono do conjunto de v.

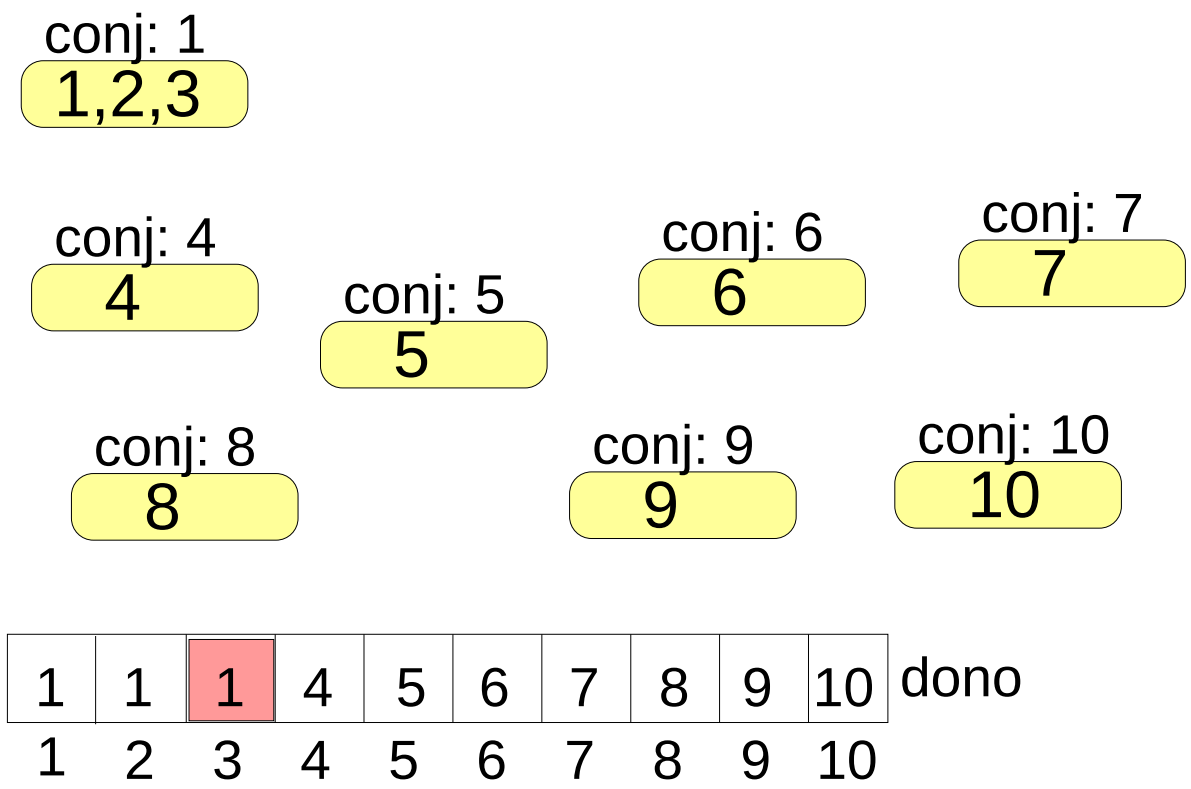
Union(1, 2):



Union-Find

Para resolver computacionalmente: ao fazer Union(u, v), o dono do conjunto de u passa a ser dono do conjunto de v.

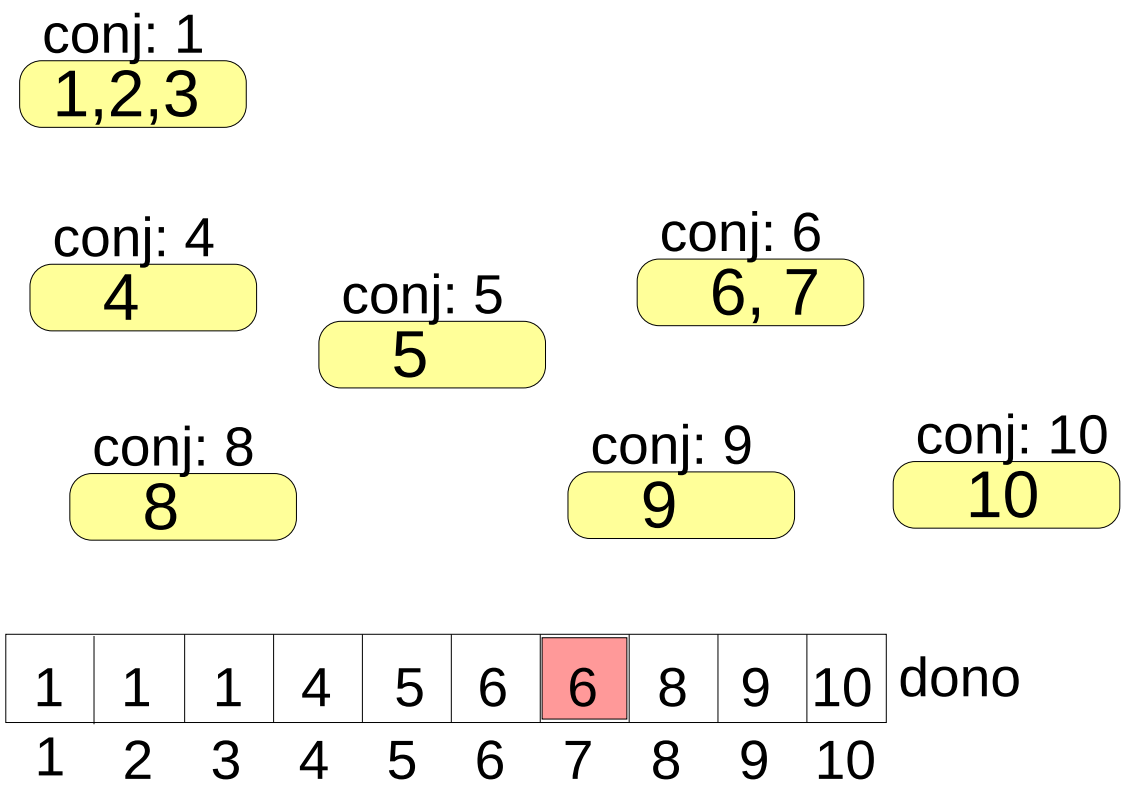
Union(1, 3):



Union-Find

Para resolver computacionalmente: ao fazer Union(u, v), o dono do conjunto de u passa a ser dono do conjunto de v.

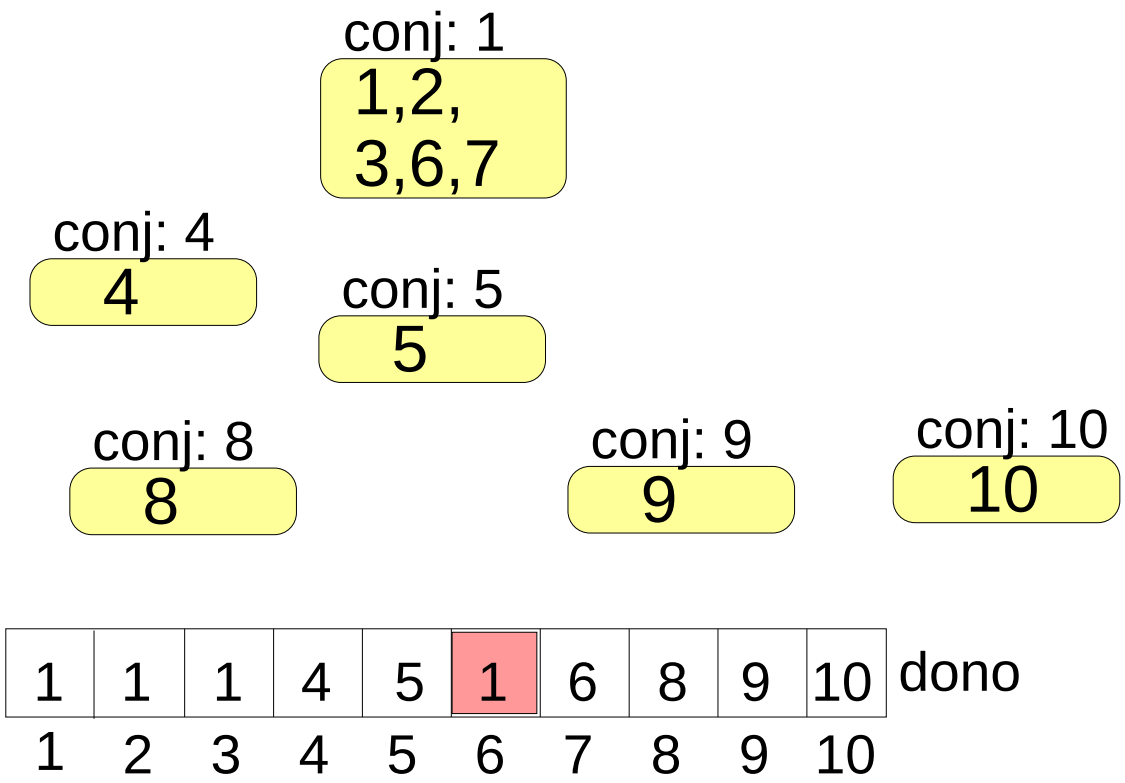
Union(6, 7):



Union-Find

Para resolver computacionalmente: ao fazer Union(u, v), o dono do conjunto de u passa a ser dono do conjunto de v.

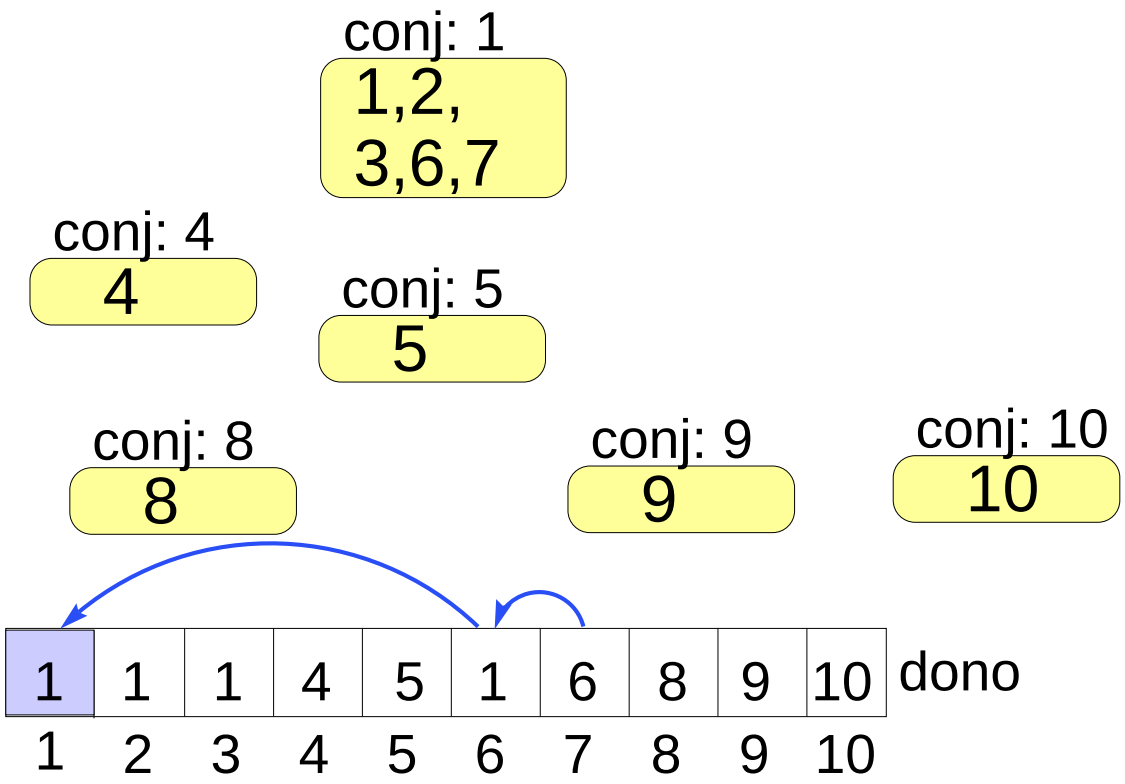
Union(1, 6):



Union-Find

Para resolver computacionalmente: ao fazer Find(u), percorre-se a lista de donos a partir de u, até encontrar um vértice dono do próprio conjunto.

Find(7):



Conjunto do 7: 1.

Union-Find

Complexidade do Find

Se, na operação de *Union*, sempre o dono do maior conjunto é quem se torna dono da união (empates resolvidos arbitrariamente), então o caminho de qualquer elemento até o dono do conjuto tem tamanho $O(\log n)$, onde n é a quantidade de elementos.

Portanto, o procedimento de *Find* percorre no máximo $O(\log n)$ elementos.

Union-Find

Union$(u, v, tam[\,], dono[\,])$

Se $tam[u] < tam[v]$, então
    $dono[u] \leftarrow v$;
    $tam[v] \leftarrow tam[v] + tam[u]$;
Senão
    $dono[v] \leftarrow u$;
    $tam[u] \leftarrow tam[u] + tam[v]$;

Complexidade do algoritmo: $O(1)$.

Find$(u, dono[\,])$

Enquanto $dono[u] \neq u$ faça:
    $u \leftarrow dono[u]$;
retorne u;

Complexidade do algoritmo: $O(\log n)$.

Algoritmo de Kruskal com Union-Find

Kruskal(G)

Ordene as arestas de G, em ordem crescente de custo
$T \leftarrow \{\ \}$
Para cada vértice $v \in V(G)$ faça:
    $dono[v] \leftarrow v$;
    $tam[v] \leftarrow 1$;
Para cada aresta $\{u, v\}$, na ordem estabelecida faça:
    $pu \leftarrow \text{Find}(u, dono)$;
    $pv \leftarrow \text{Find}(v, dono)$;
    Se $pu \neq pv$ então:
        $T \leftarrow T + \{u, v\}$;
        $\text{Union}(pu, pv, tam, dono)$;

Algoritmo de Kruskal com Union-Find

Kruskal(G)

Ordene as arestas de G, em ordem crescente de custo $O(m \log m)$
$T \leftarrow \{\ \}$ $O(1)$
Para cada vértice $v \in V(G)$ faça: $O(n)$
    $dono[v] \leftarrow v$; $O(n)$
    $tam[v] \leftarrow 1$; $O(n)$
Para cada aresta $\{u, v\}$, na ordem estabelecida faça: $O(m)$
    $pu \leftarrow \text{Find}(u, dono)$; $O(m)$
    $pv \leftarrow \text{Find}(v, dono)$; $O(m)$
    Se $pu \neq pv$ então: $O(m)$
        $T \leftarrow T + \{u, v\}$; $O(n)$
        $\text{Union}(u, v, tam, dono)$; $O(n \log n)$

Complexidade do algoritmo: $O(m \log m) + O(n \log n)$. Como $m \leq n^2$, temos: $\log m \in O(\log n)$. Como $m \geq n - 1$, $\log n \in O(\log m)$.
Complexidade do algoritmo: $O(m \log n)$.

Algoritmo de Prim

Outra forma de evitar ciclos:

Imagine que os vértices estão separados em dois conjuntos:

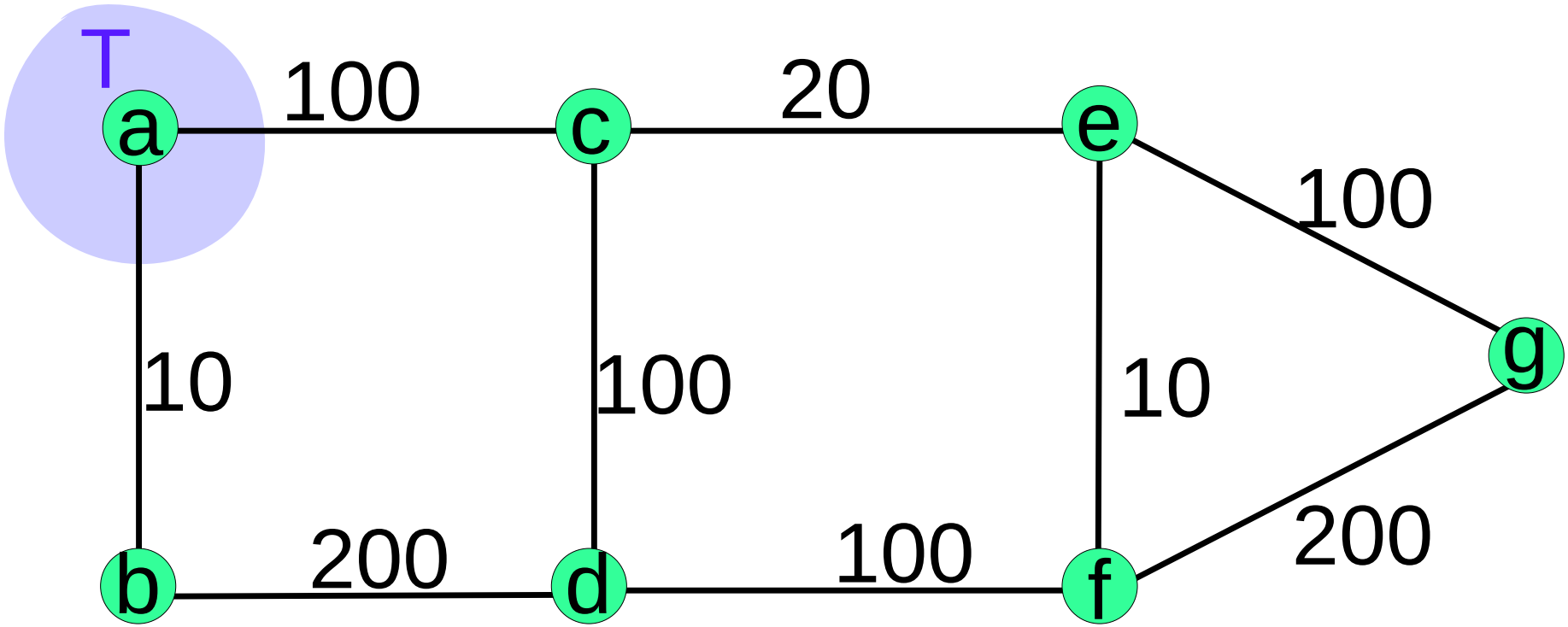
- os que já fazer parte da árvore;
- e os que ainda não foram conectados.

A cada iteração, um novo vértice é conectado à árvore.

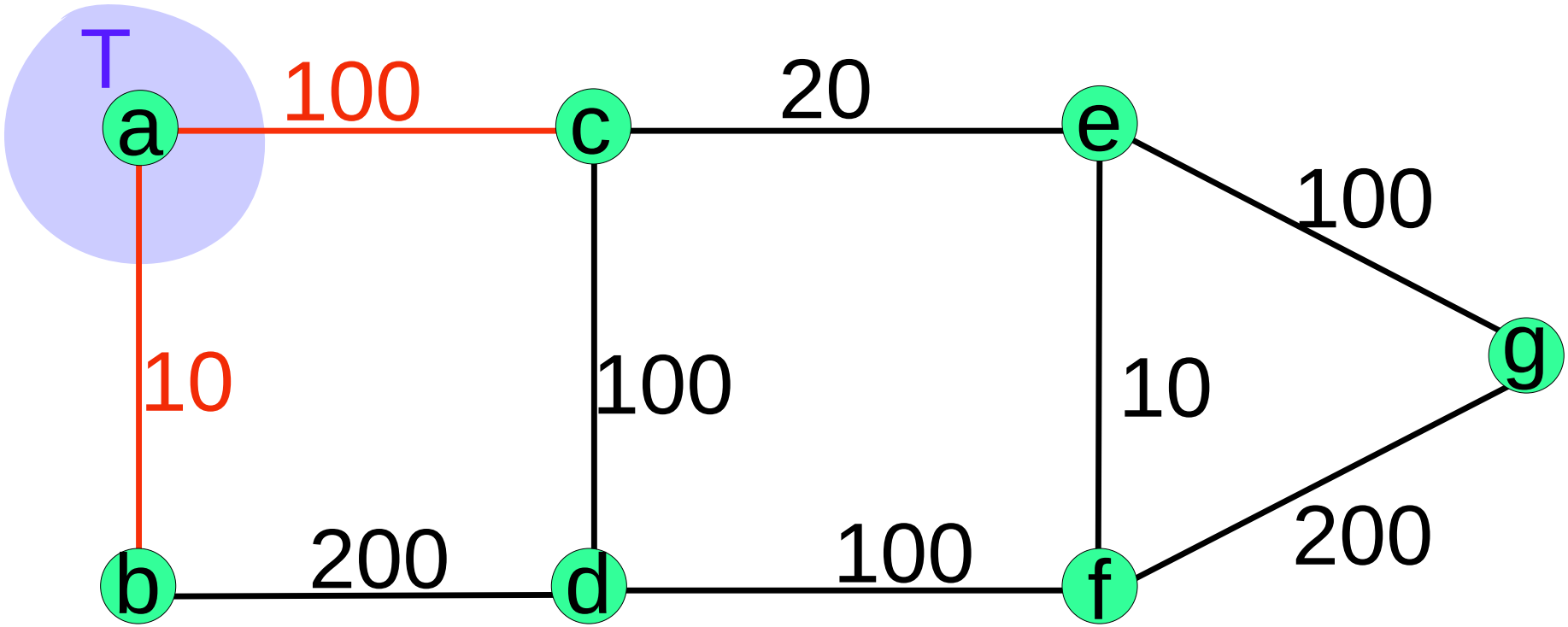
Quem será o escolhido? O que puder ser conectado com o menor custo!

Método de programação guloso!

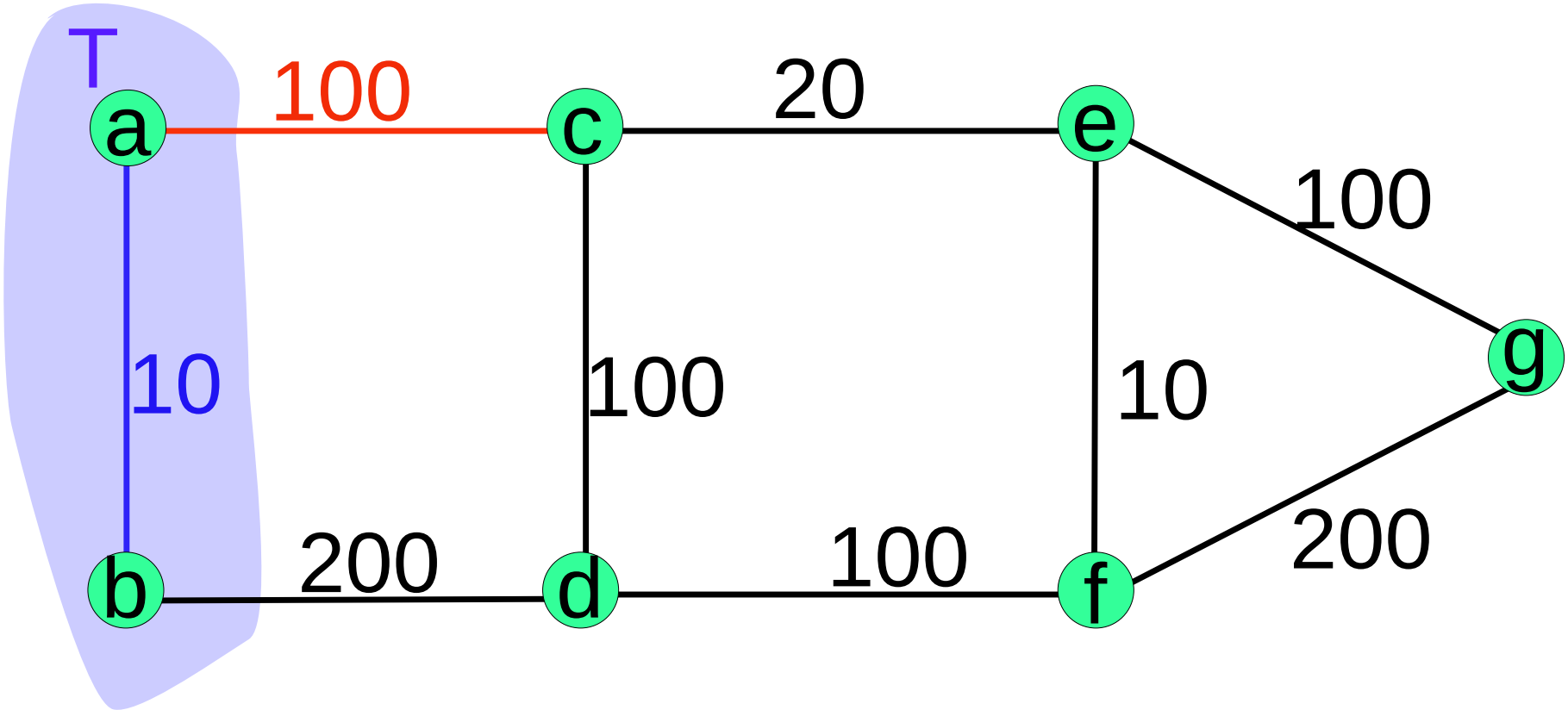
Algoritmo de Prim



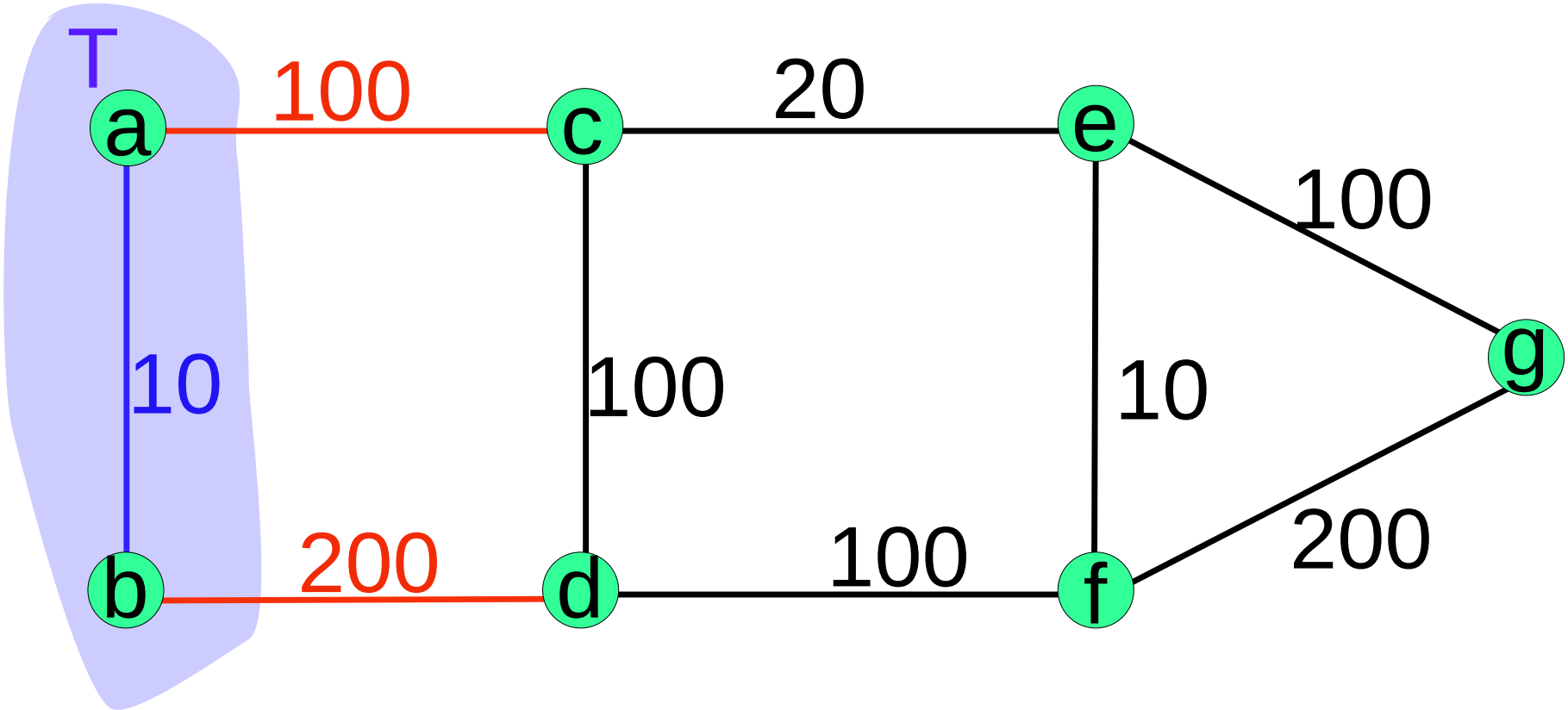
Algoritmo de Prim



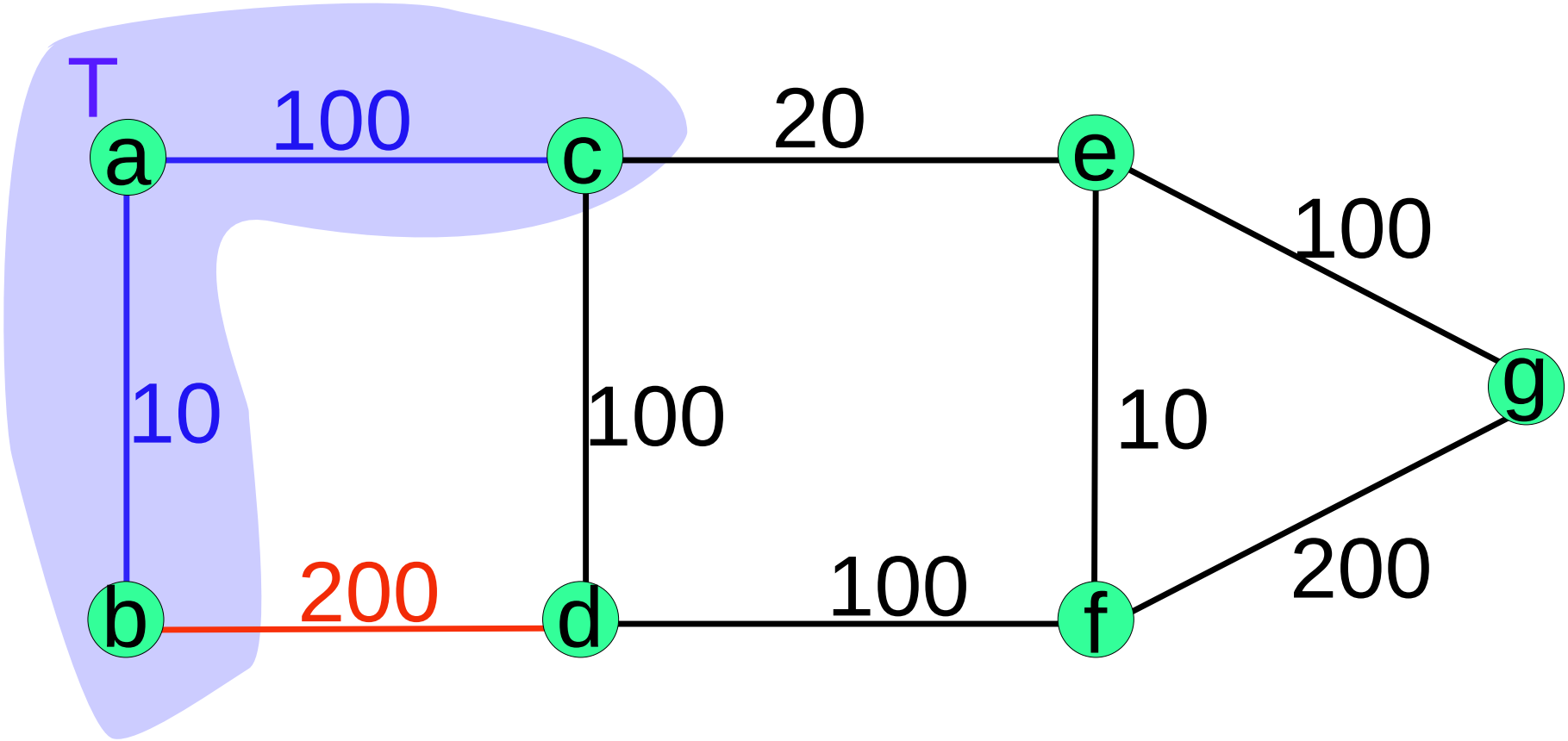
Algoritmo de Prim



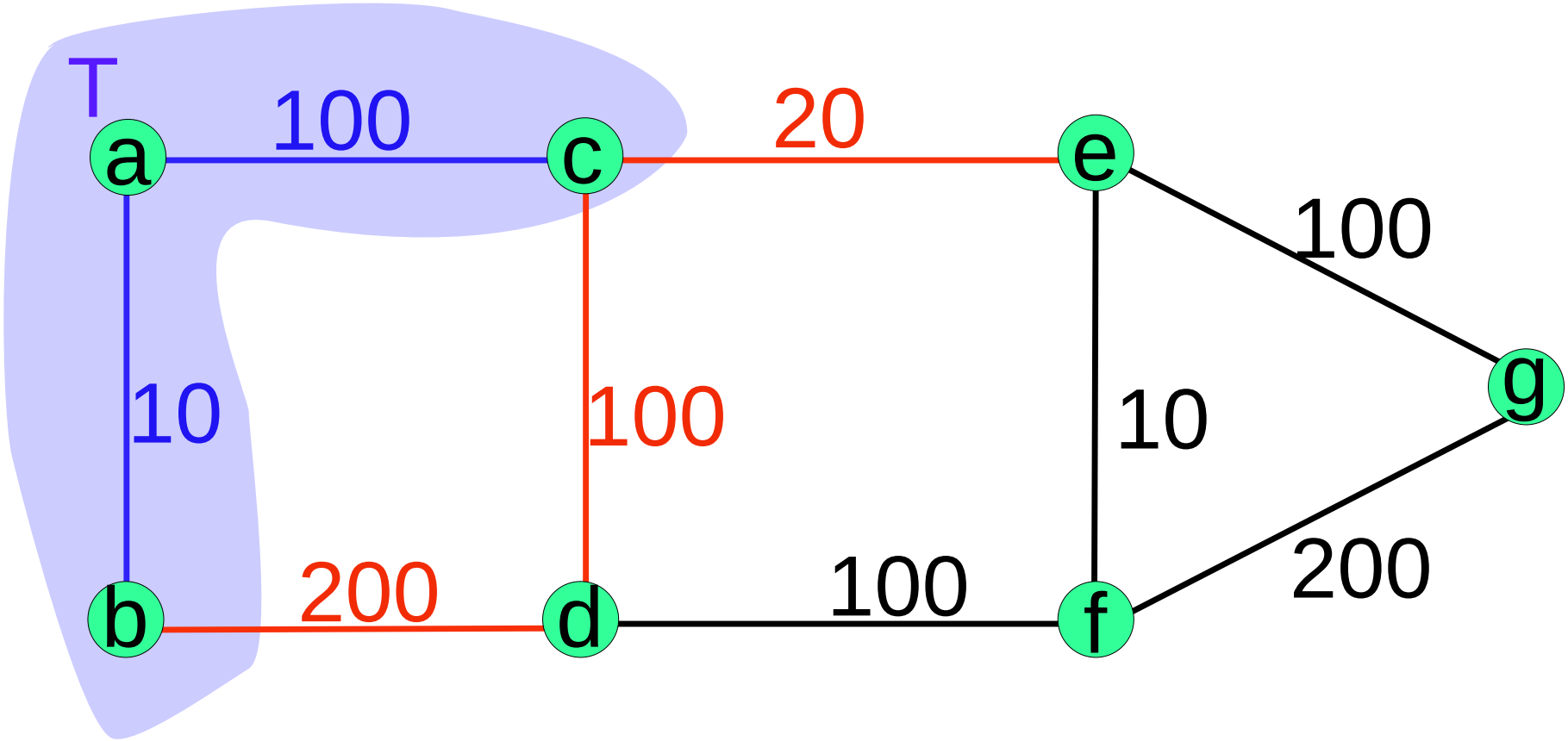
Algoritmo de Prim



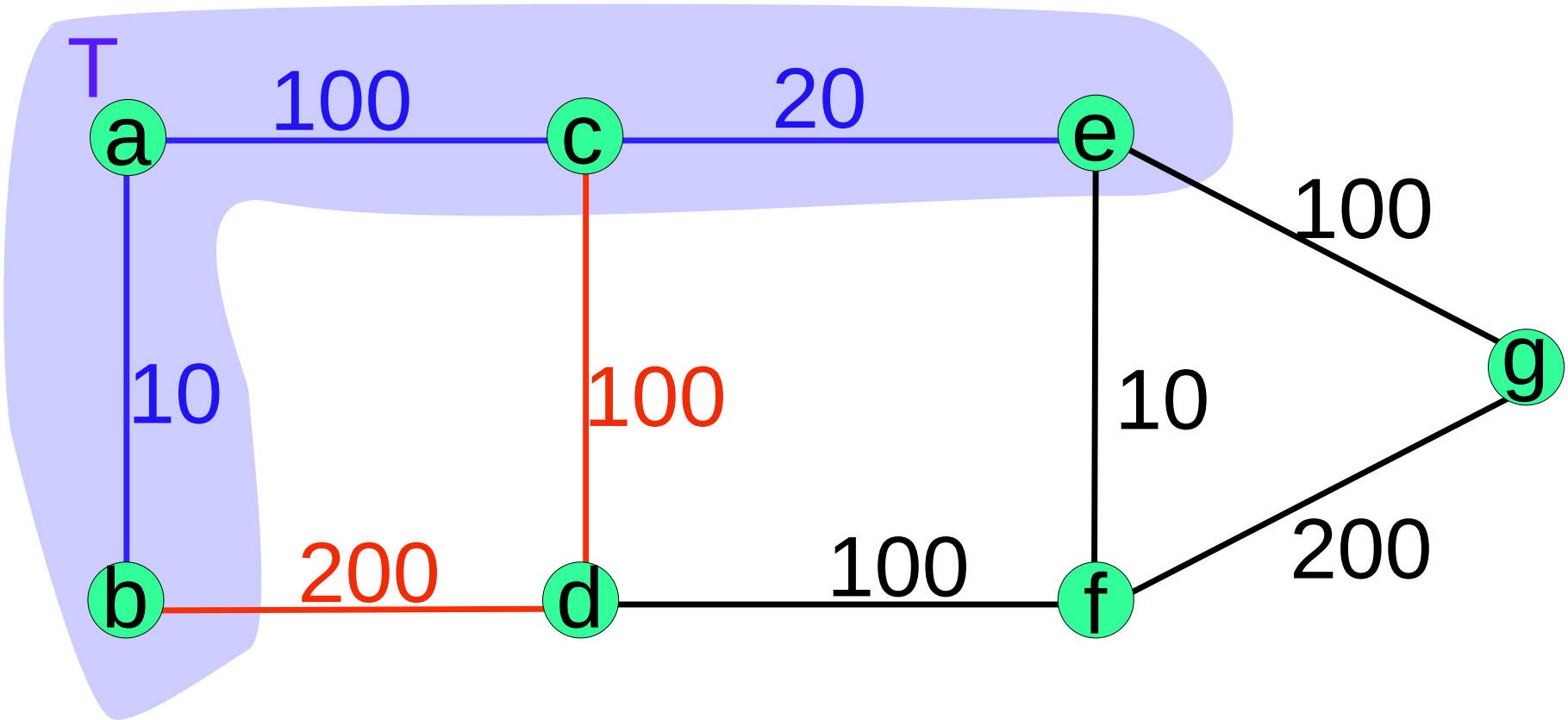
Algoritmo de Prim



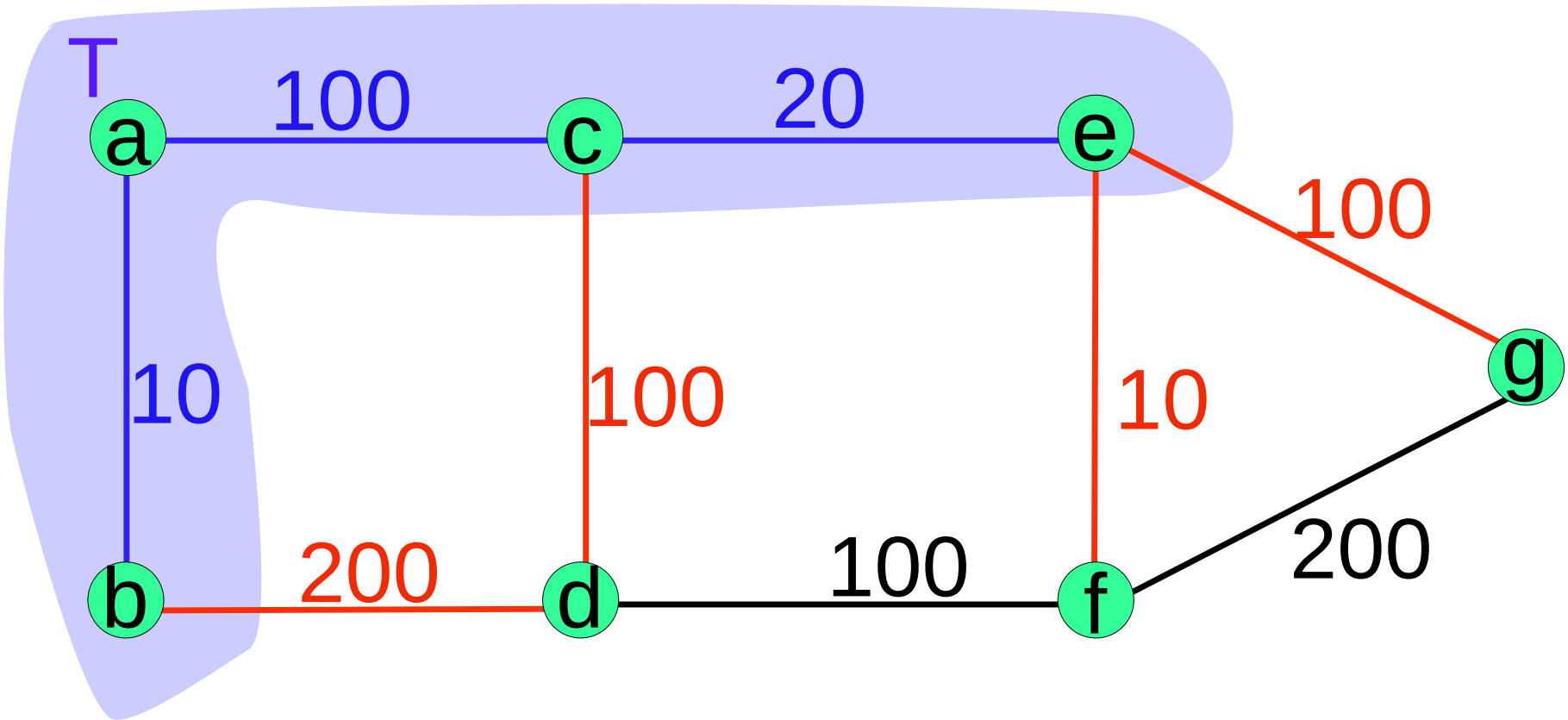
Algoritmo de Prim



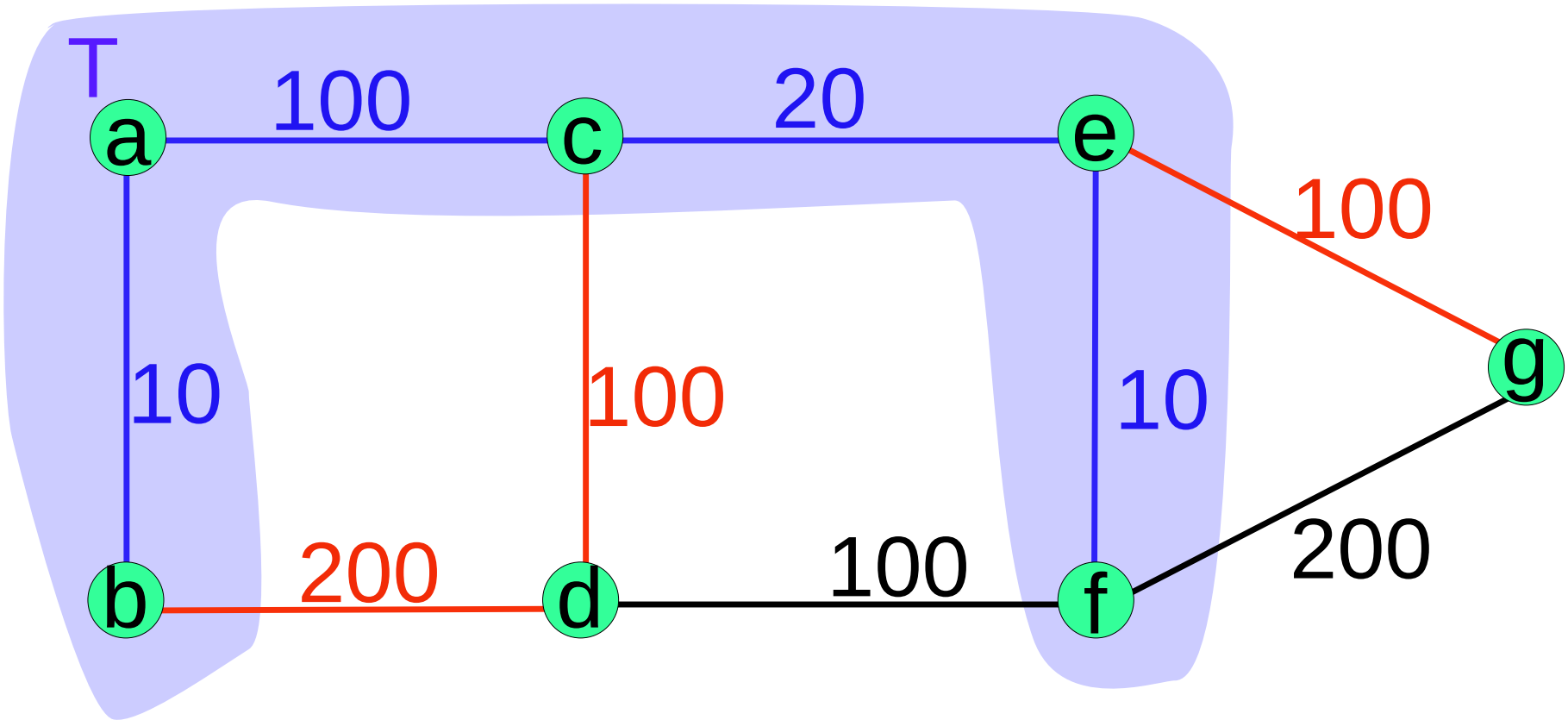
Algoritmo de Prim



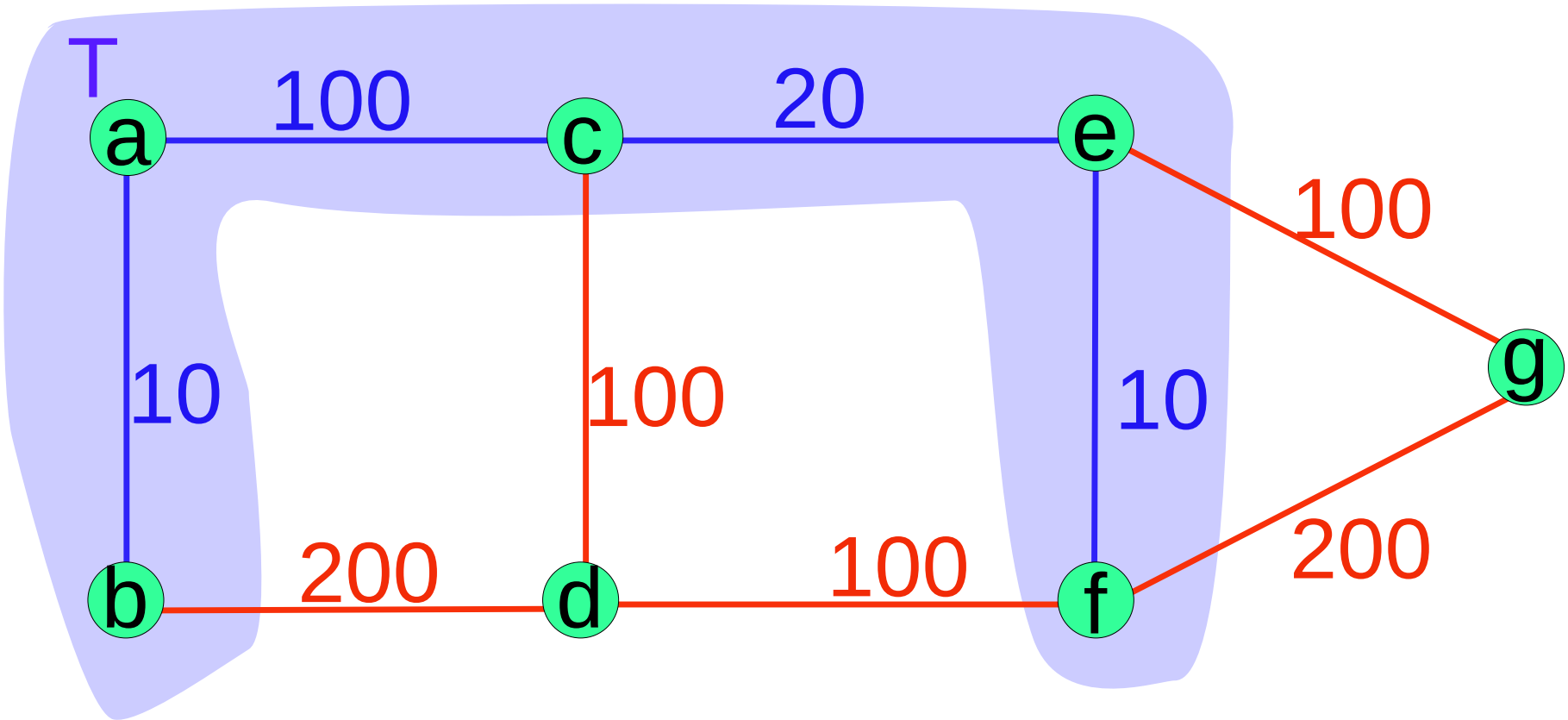
Algoritmo de Prim



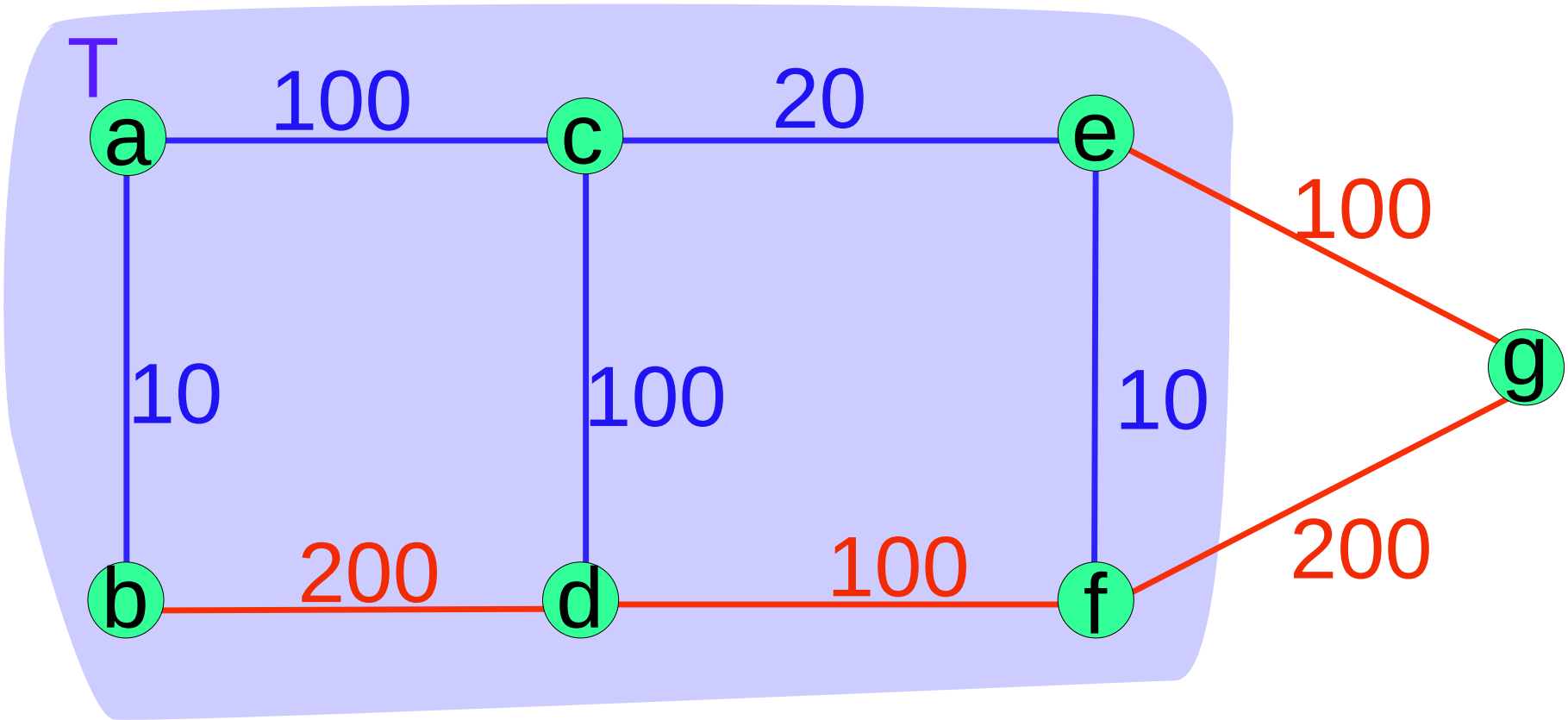
Algoritmo de Prim



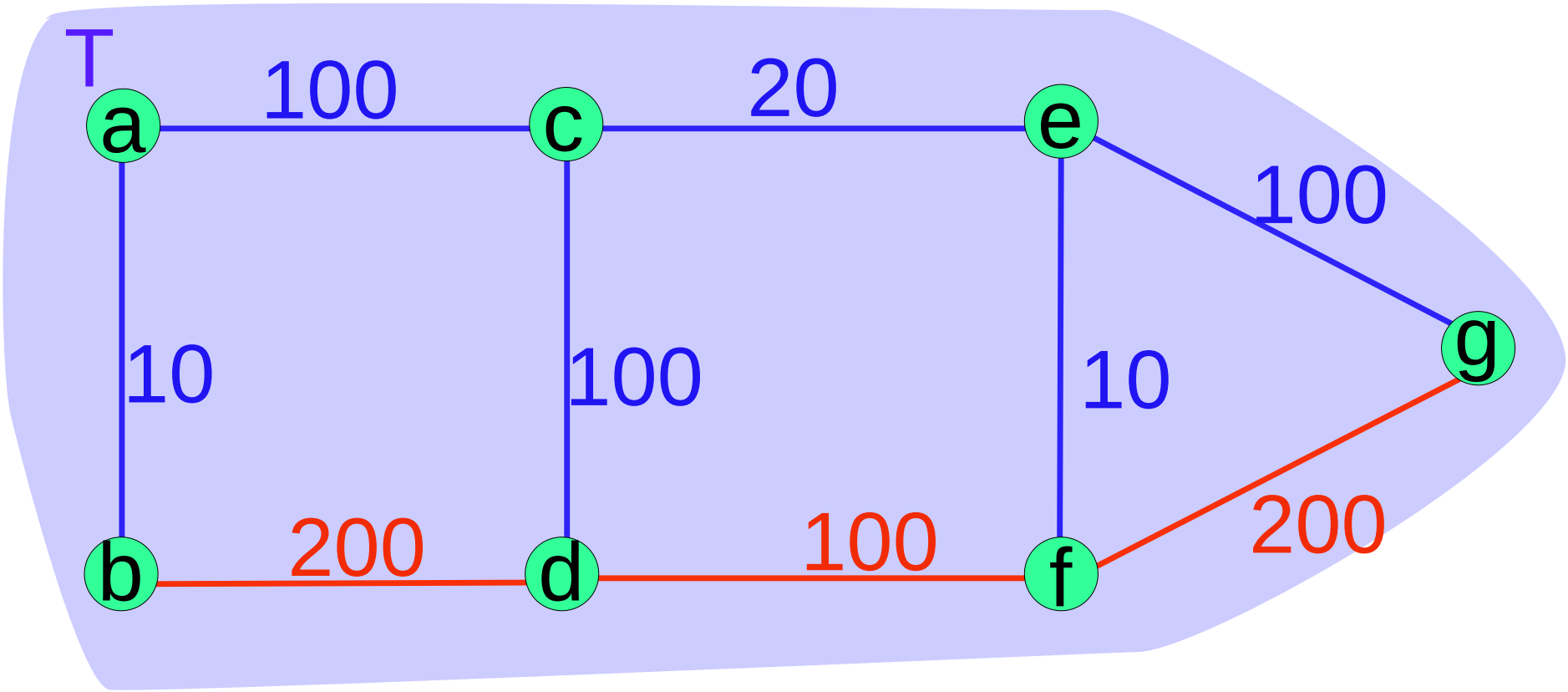
Algoritmo de Prim



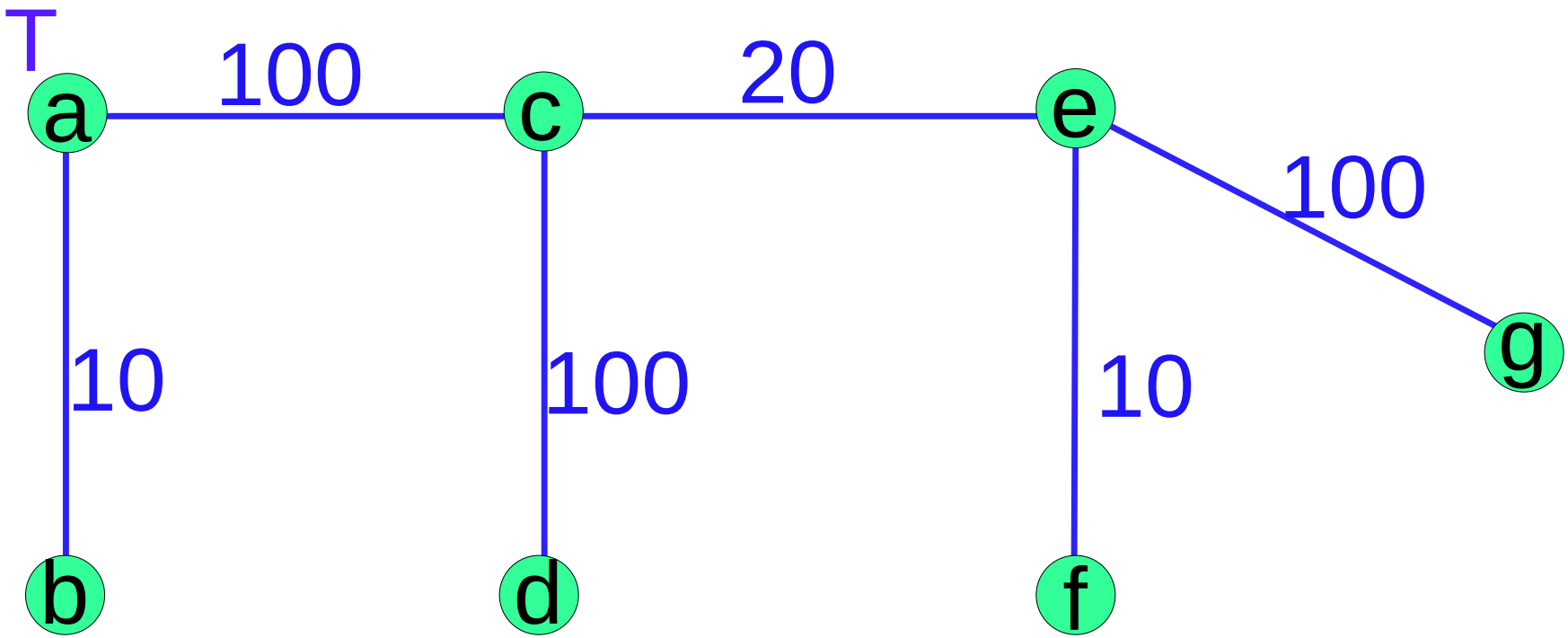
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



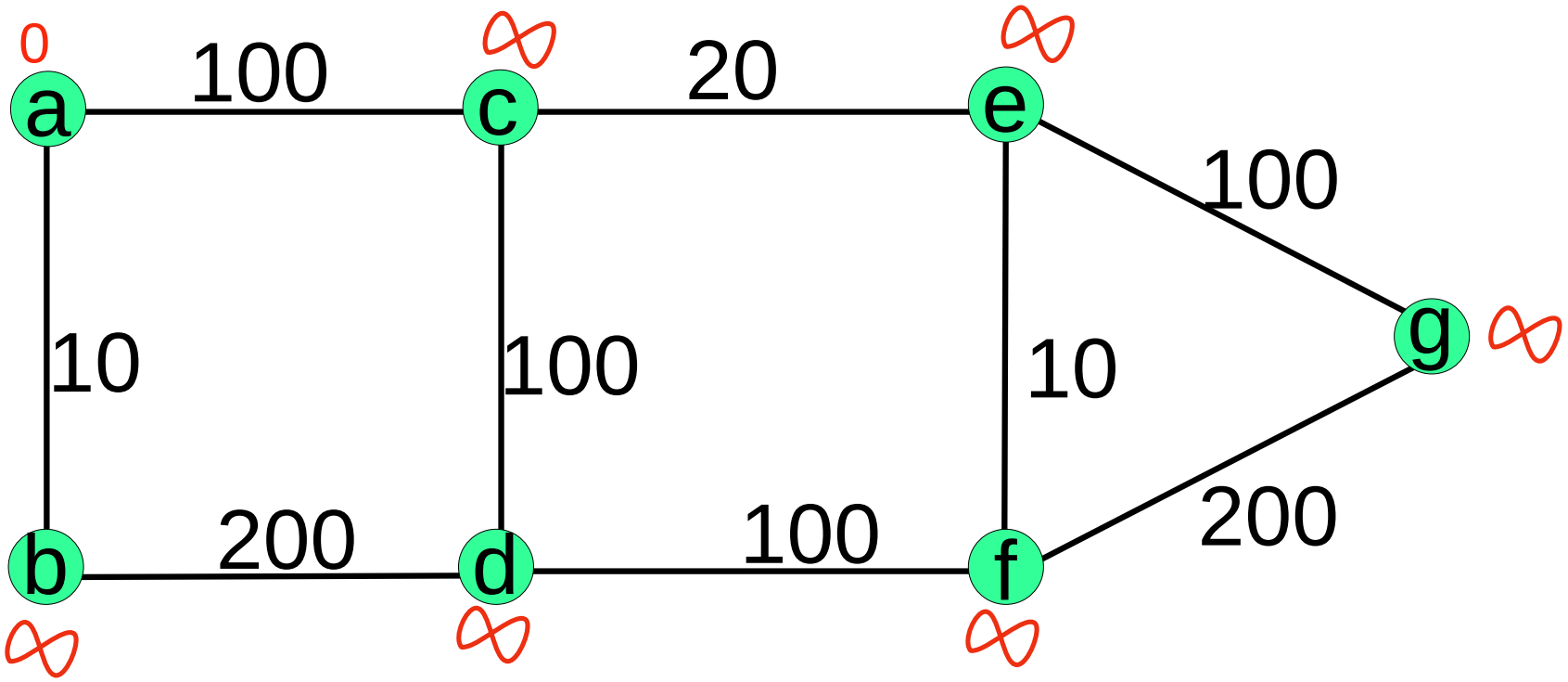
Algoritmo de Prim

Computacionalmente:

1. usamos pesos nos vértices para indicar qual o custo para conectá-los.
2. o vértice inicial tem custo zero;
3. todos os outros começam com custo infinito;
4. escolhemos o vértice com menor custo para conectar à árvore;
5. quando um vértice é conectado à árvore, atualizamos o custo dos seus vizinhos, caso diminuam;
6. retornamos ao passo 4 até todos os vértices se conectarem à árvore.

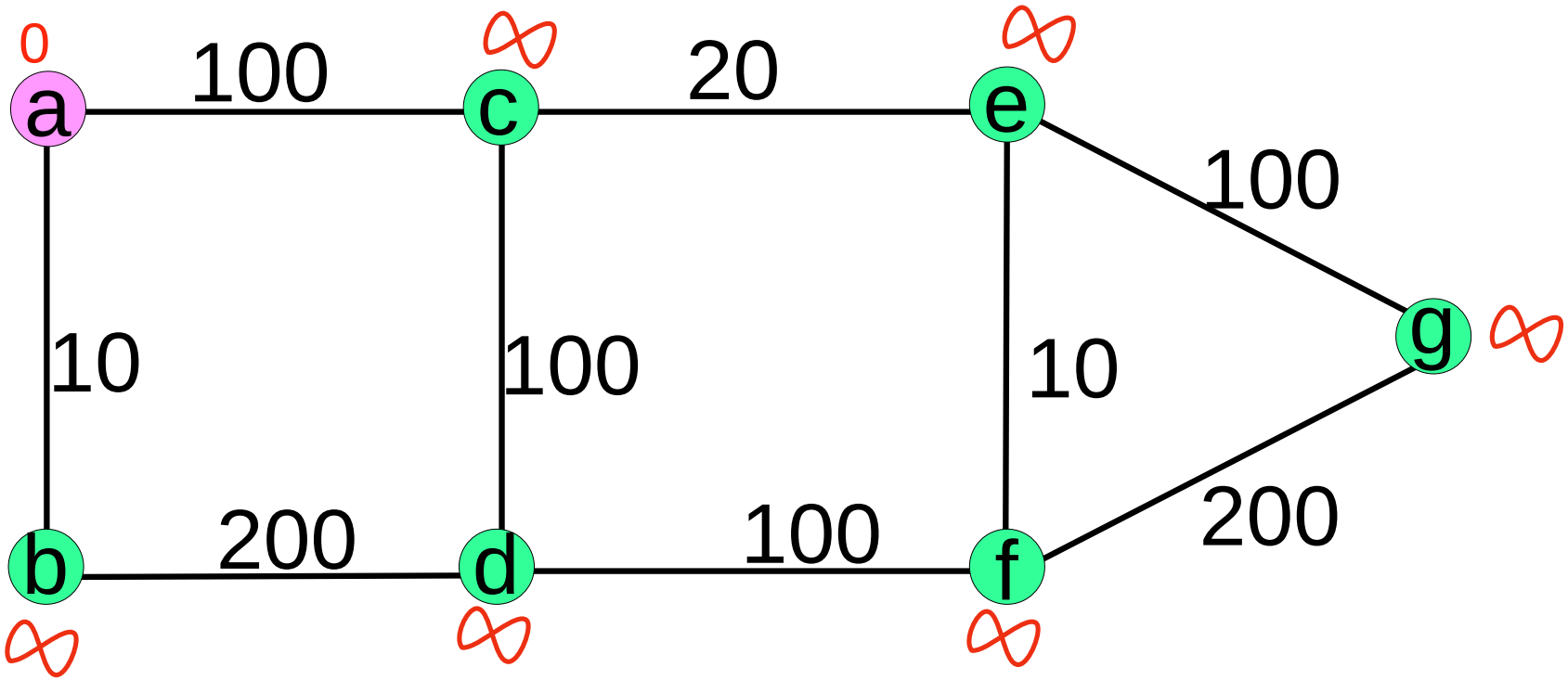
Algoritmo de Prim

Início:



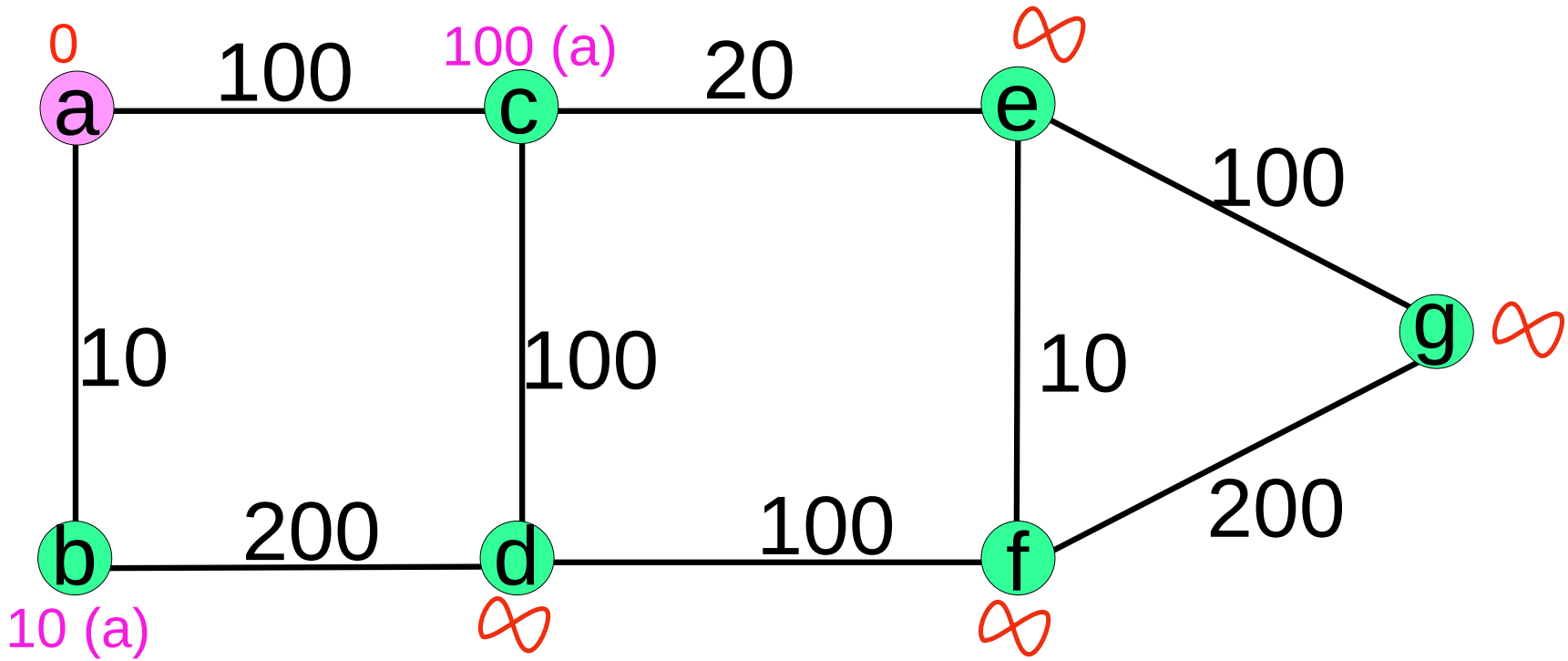
Algoritmo de Prim

Inclusão do vértice de menor custo:



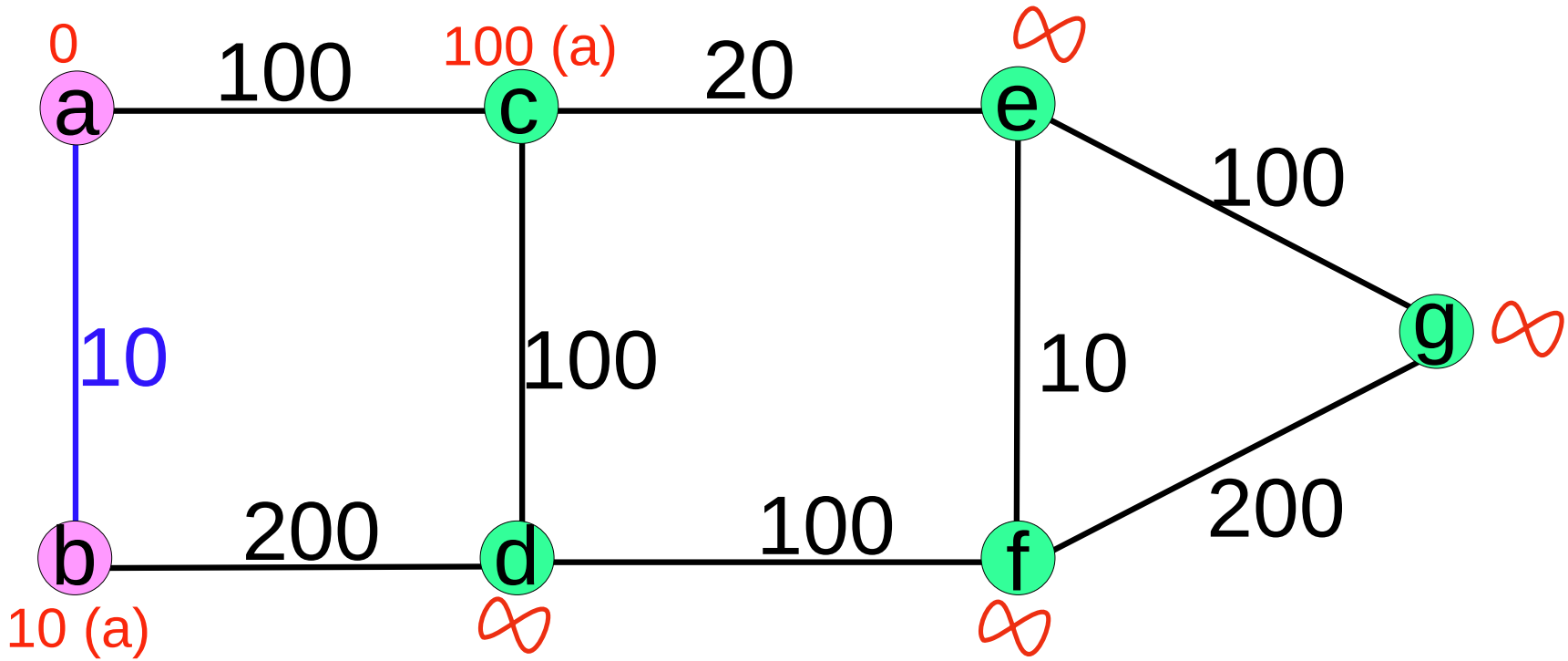
Algoritmo de Prim

Atualização do custo dos vizinhos:



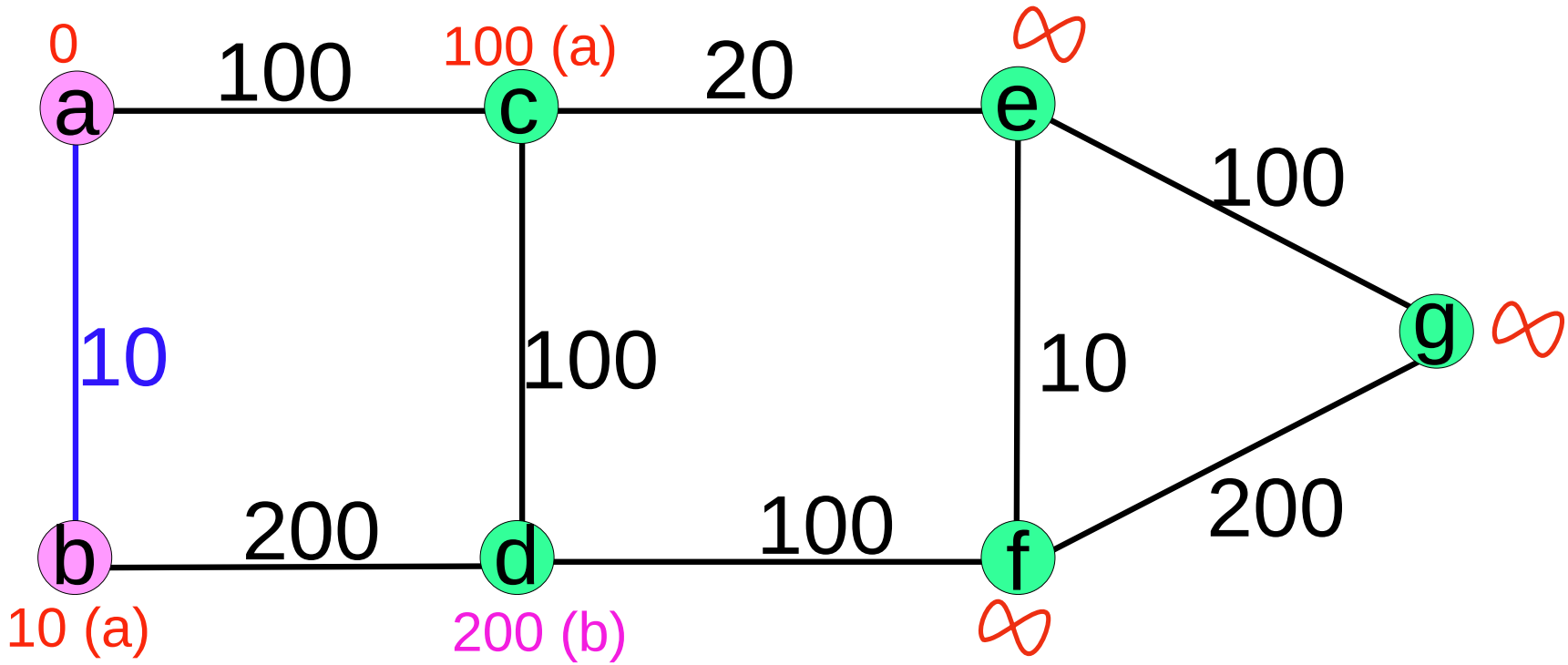
Algoritmo de Prim

Inclusão do vértice de menor custo:



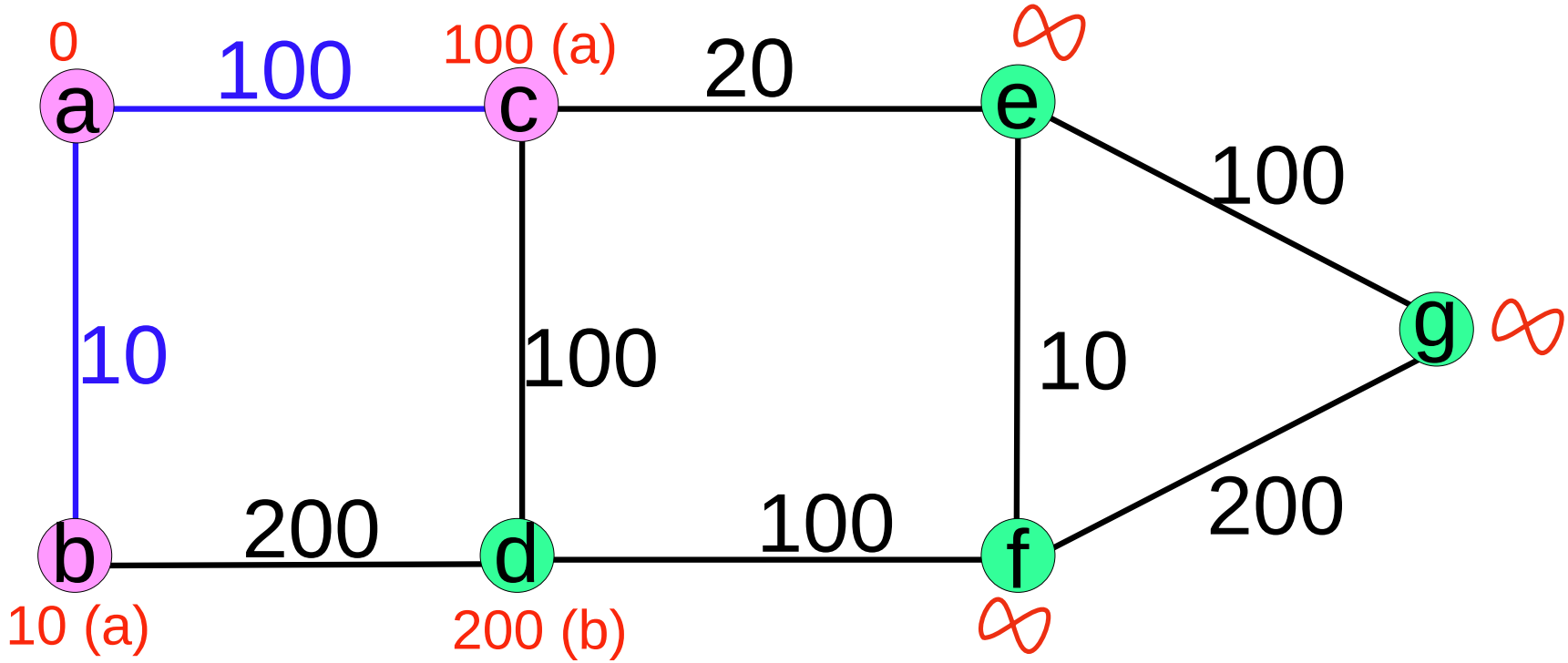
Algoritmo de Prim

Atualização do custo dos vizinhos:



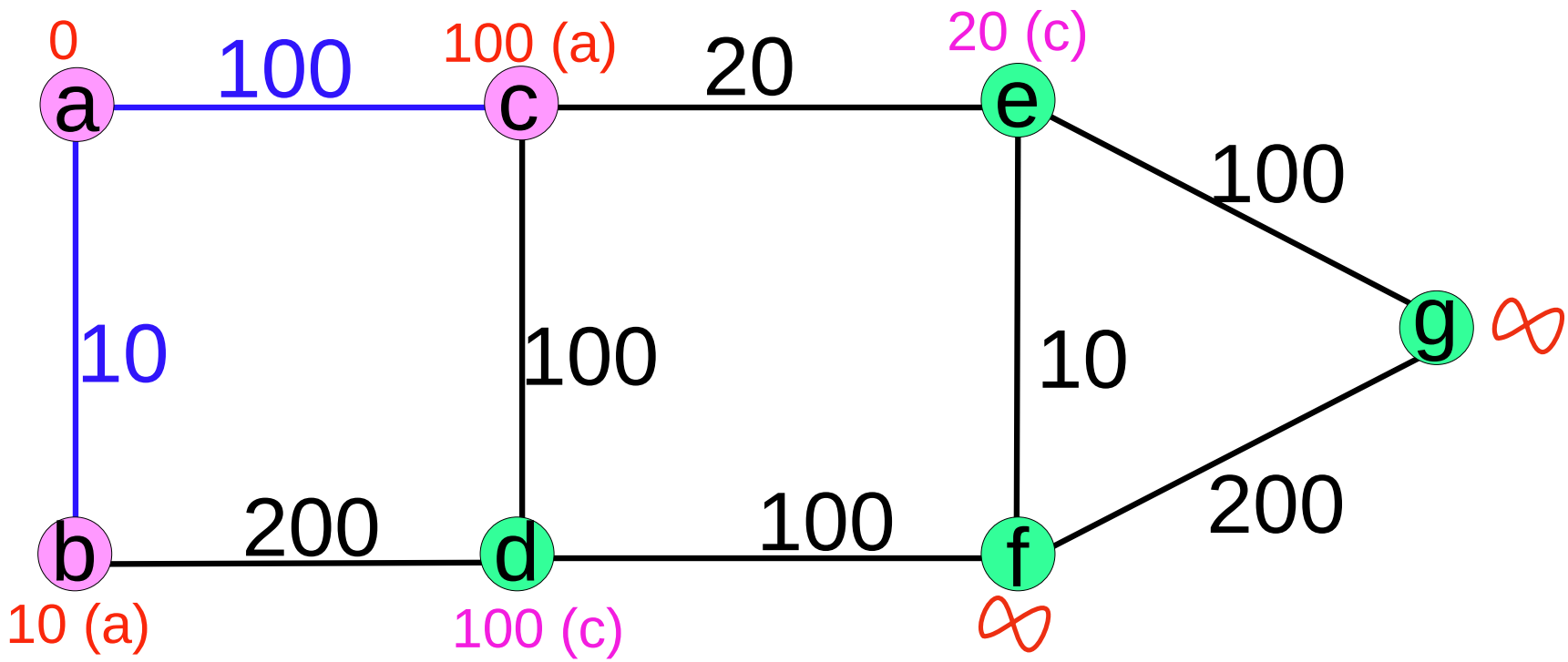
Algoritmo de Prim

Inclusão do vértice de menor custo:



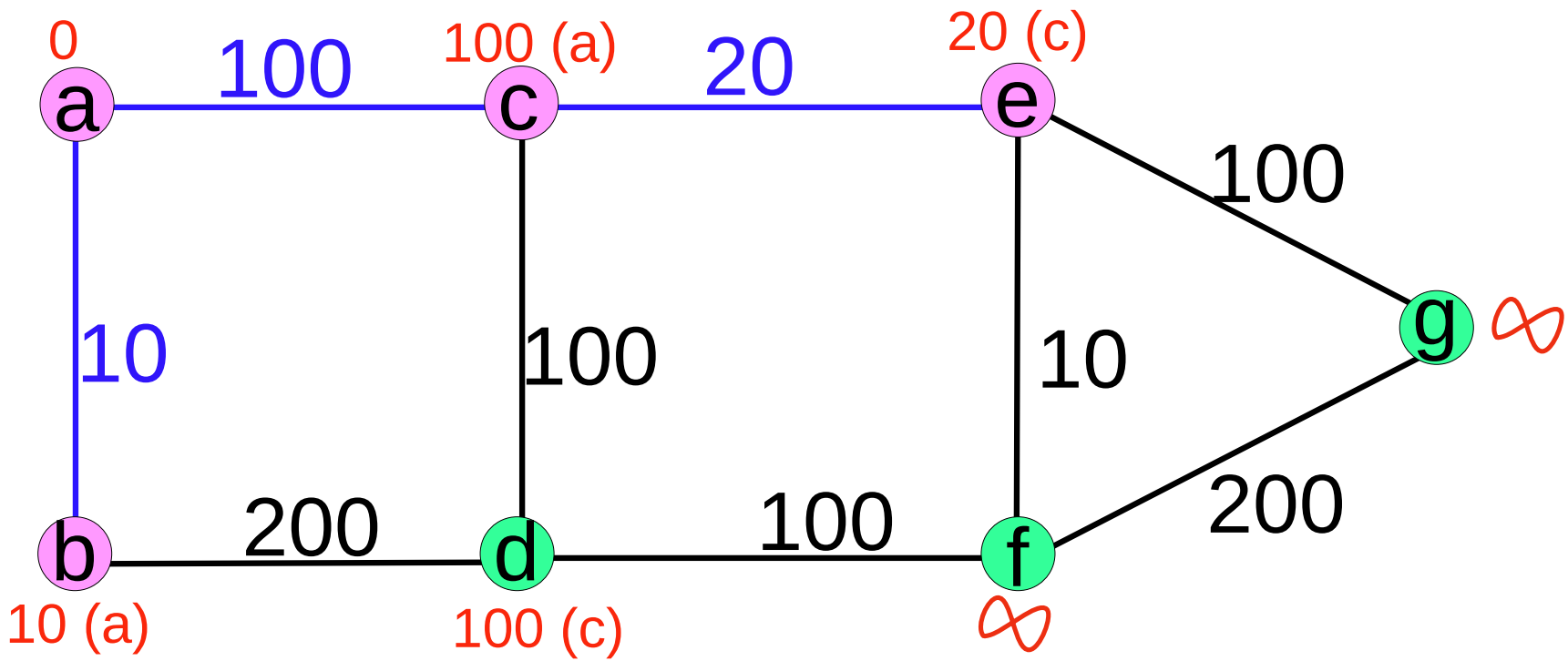
Algoritmo de Prim

Atualização do custo dos vizinhos:



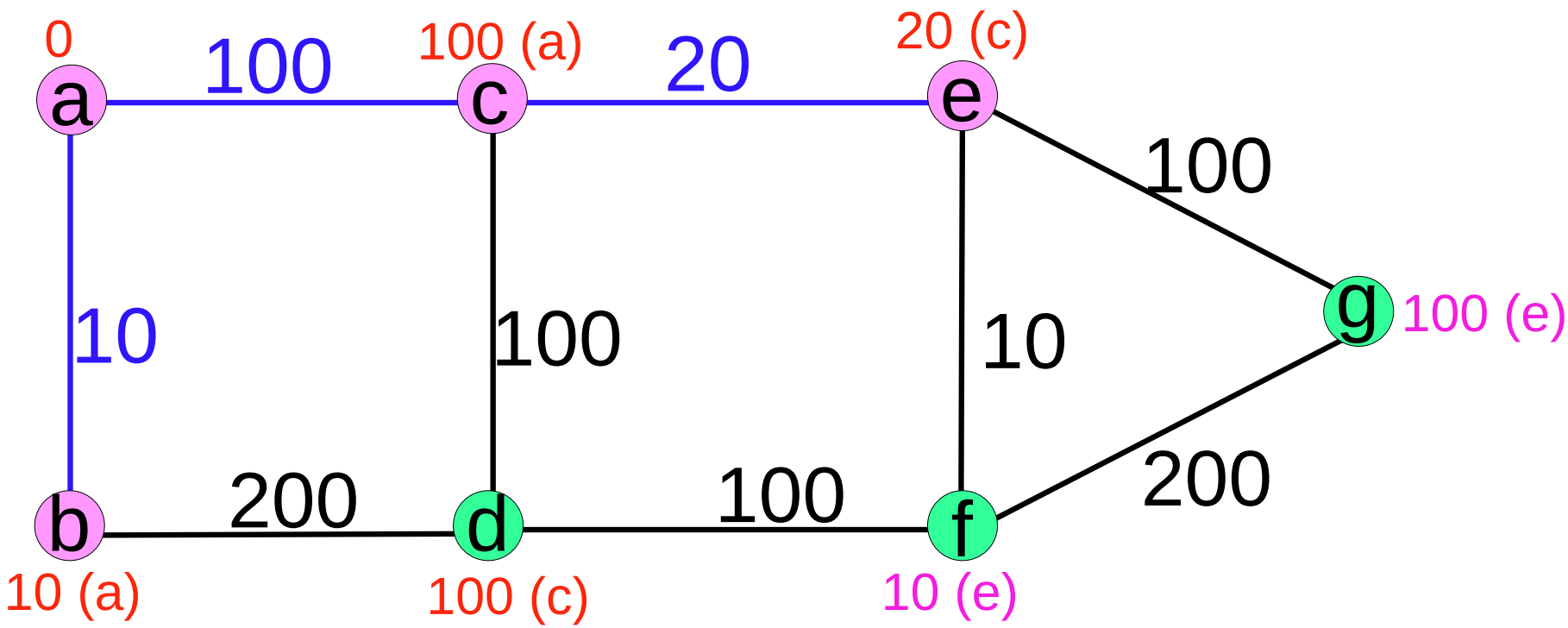
Algoritmo de Prim

Inclusão do vértice de menor custo:



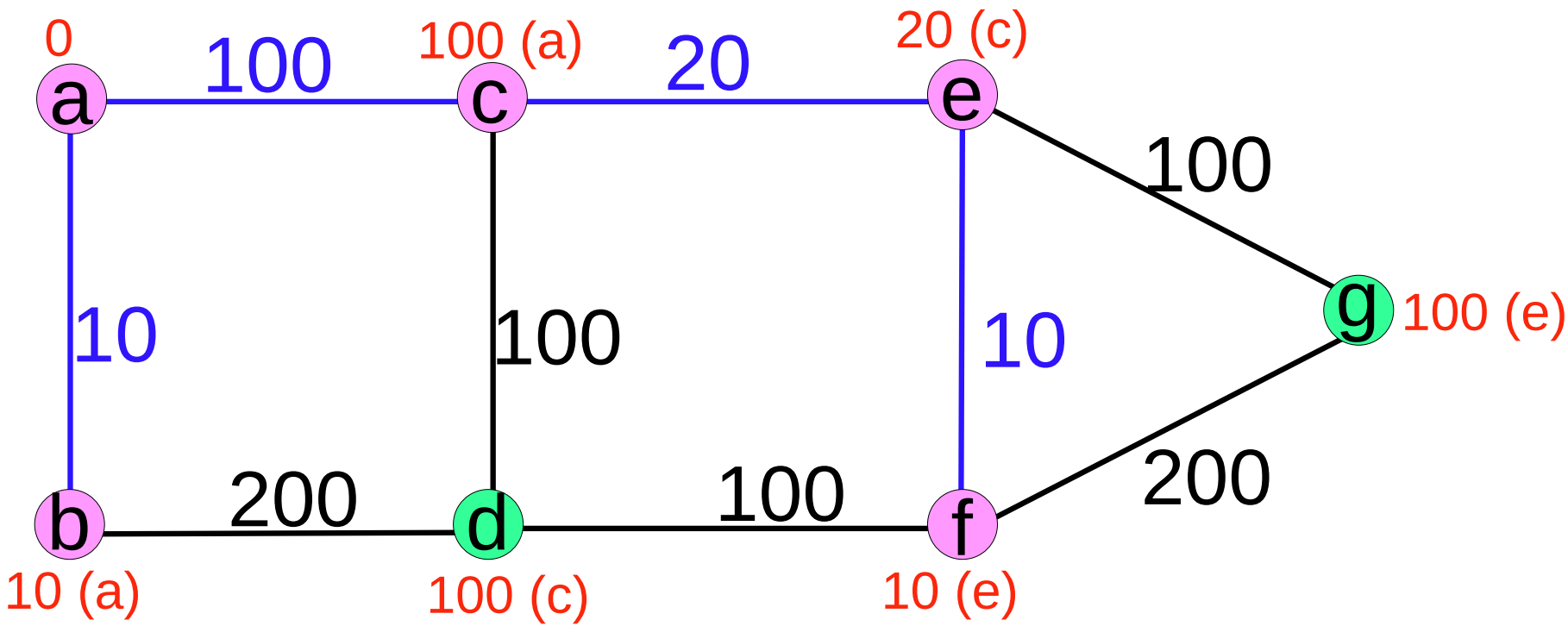
Algoritmo de Prim

Atualização do custo dos vizinhos:



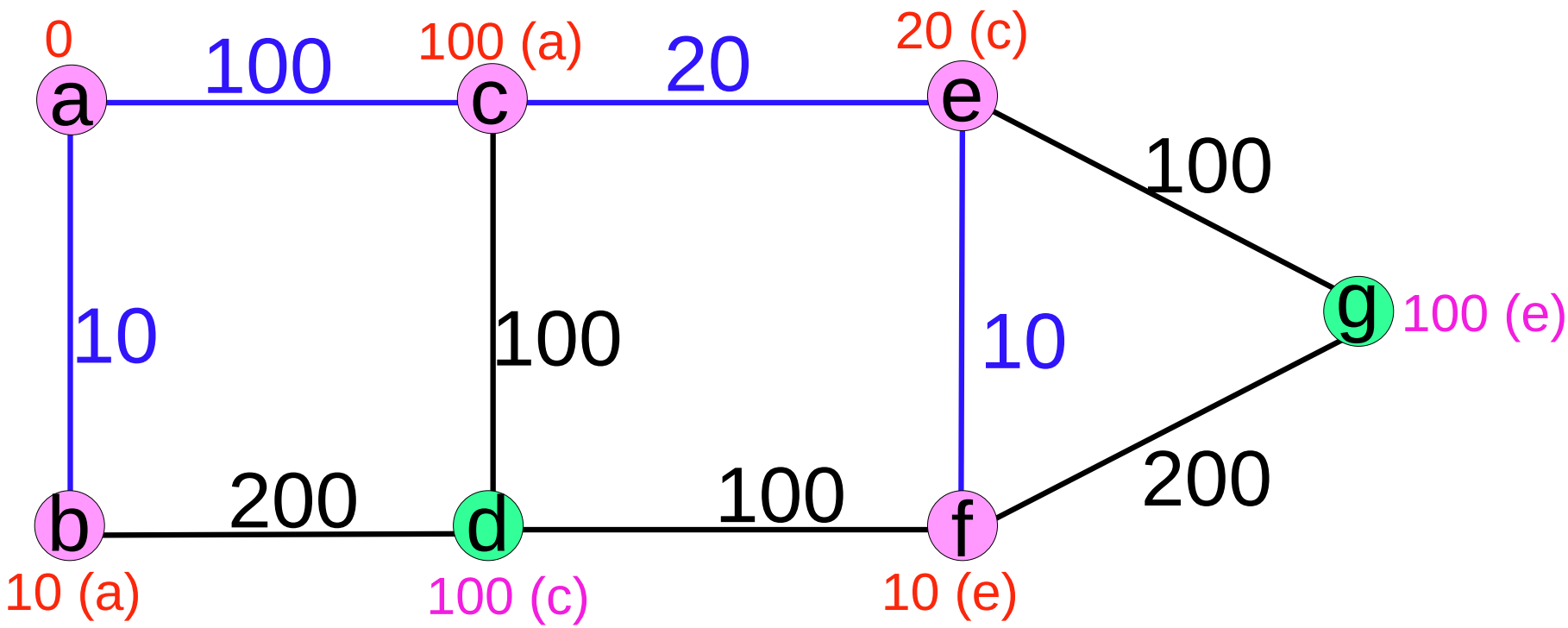
Algoritmo de Prim

Inclusão do vértice de menor custo:



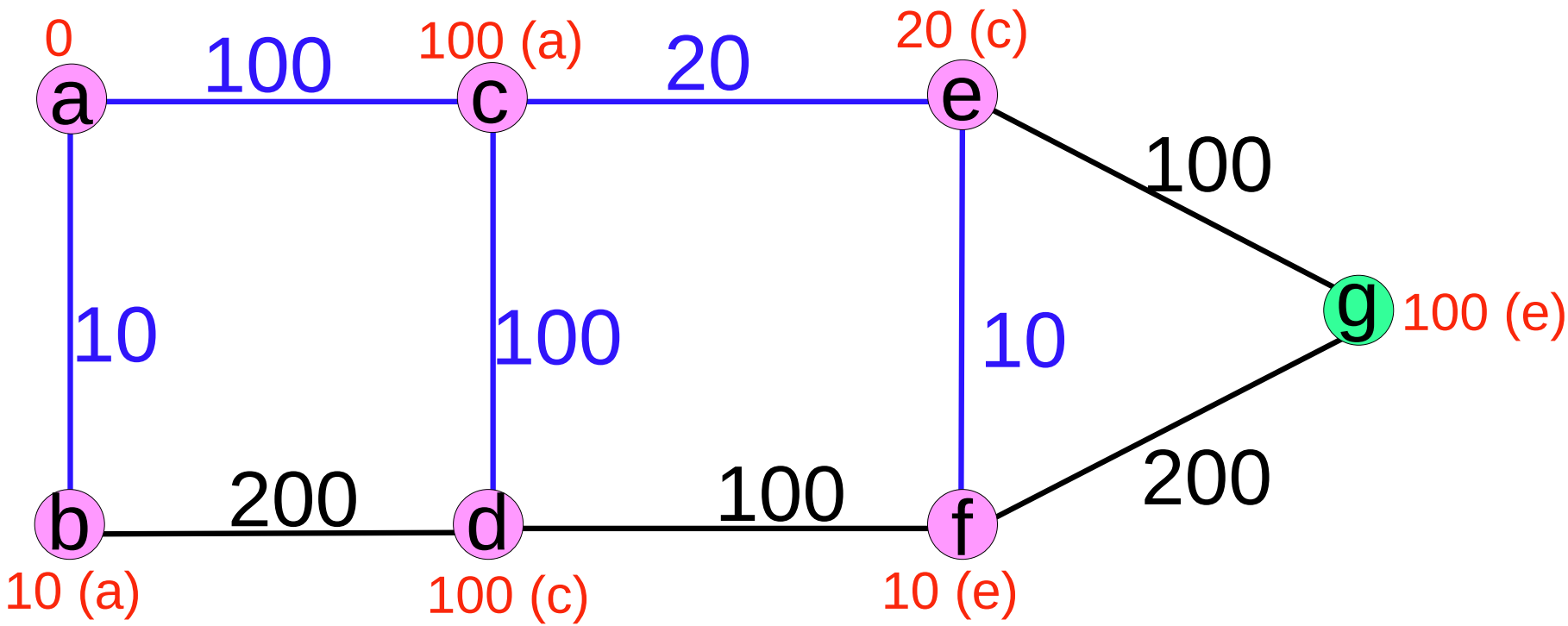
Algoritmo de Prim

Atualização do custo dos vizinhos:



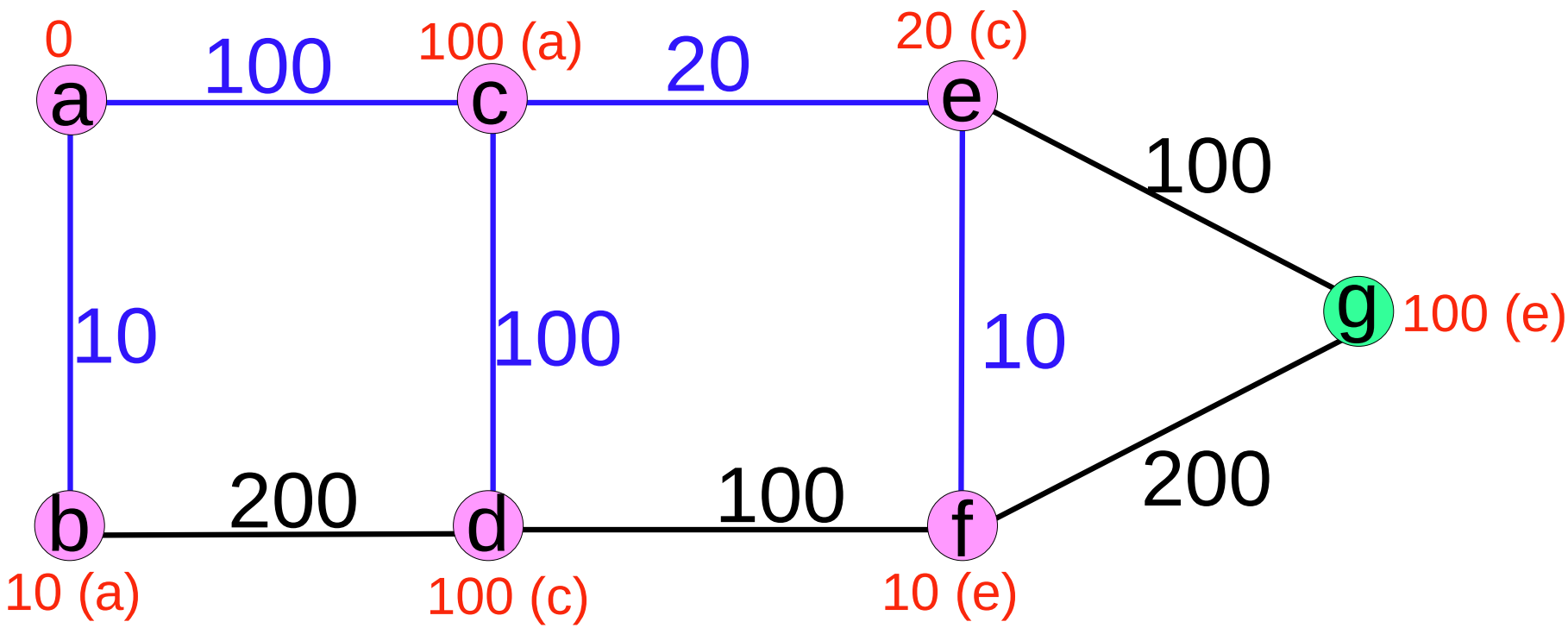
Algoritmo de Prim

Inclusão do vértice de menor custo:



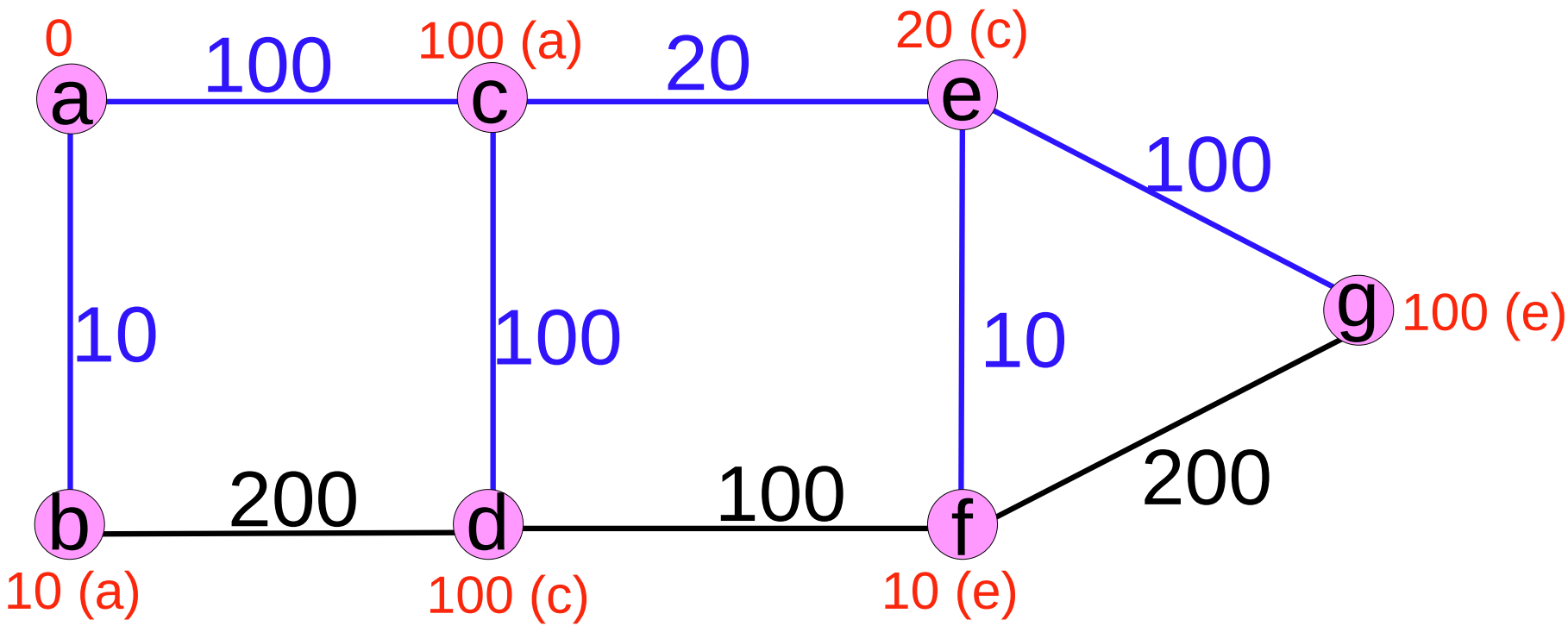
Algoritmo de Prim

Atualização do custo dos vizinhos:



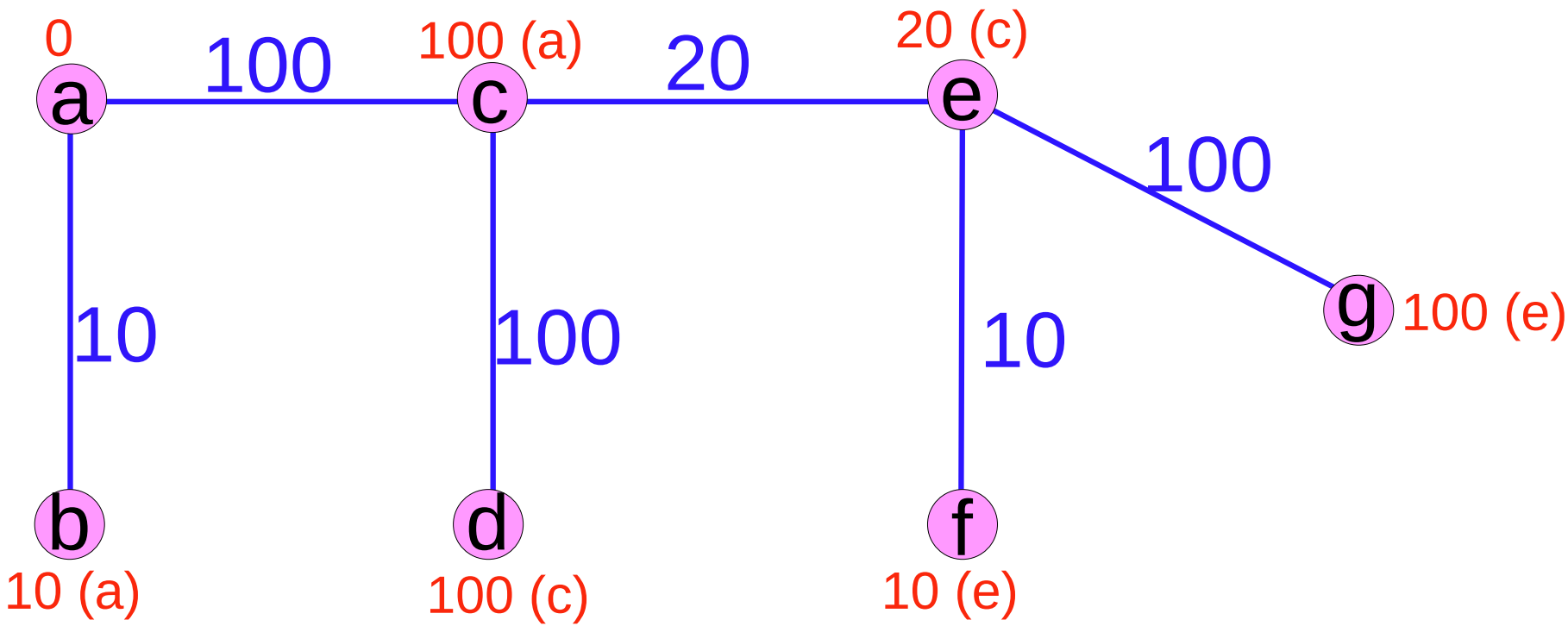
Algoritmo de Prim

Inclusão do vértice de menor custo:



Algoritmo de Prim

Árvore Geradora de Custo Mínimo:



Algoritmo de Prim

Prim(G)

$T \leftarrow \{\ \}$ (conjunto de vértices conectados à árvore.)
Para cada vértice $v \in V(G)$ faça:
    $custo[v] \leftarrow \infty$;
    $pred[v] \leftarrow NULL$;
Escolha um vértice de origem v_0;
$custo[v_0] \leftarrow 0$;
Enquanto $T \neq V(G)$ faça:
    $v \leftarrow$ vértice de custo mínimo em $V(G) \setminus T$;
    $T \leftarrow T \cup \{v\}$
    Para cada vizinho u do vértice v faça:
        Se $peso[v, u] < custo[u]$ então:
            $custo[u] \leftarrow peso[v, u]$;
            $pred[u] \leftarrow v$;

Prim(G)

$T \leftarrow \{\ \}$ $O(1)$

Para cada vértice $v \in V(G)$ faça: $O(n)$

  $custo[v] \leftarrow \infty$; $O(n)$

  $pred[v] \leftarrow NULL$; $O(n)$

Escolha um vértice de origem v_0; $O(1)$

$custo[v_0] \leftarrow 0$; $O(1)$

Enquanto $T \neq V(G)$ faça: $O(n)$ vértices são inseridos em T uma vez

  $v \leftarrow$ vértice de custo mínimo em $V(G) \setminus T$; $O(n^2)$ percorre $custo[\ ]$

  $T \leftarrow T \cup \{v\}$ $O(n)$ vértices são inseridos em T uma vez

  Para cada vizinho u do vértice v faça: $O(m)$ arestas vistas duas vezes

    Se $peso[v, u] < custo[u]$ então: $O(m)$

      $custo[u] \leftarrow peso[v, u]$; $O(m)$

      $pred[u] \leftarrow v$; $O(m)$

Complexidade do Algoritmo: $O(n^2)$.

Algoritmo de Prim

Pode-se reduzir a complexidade do Algoritmo de Prim:

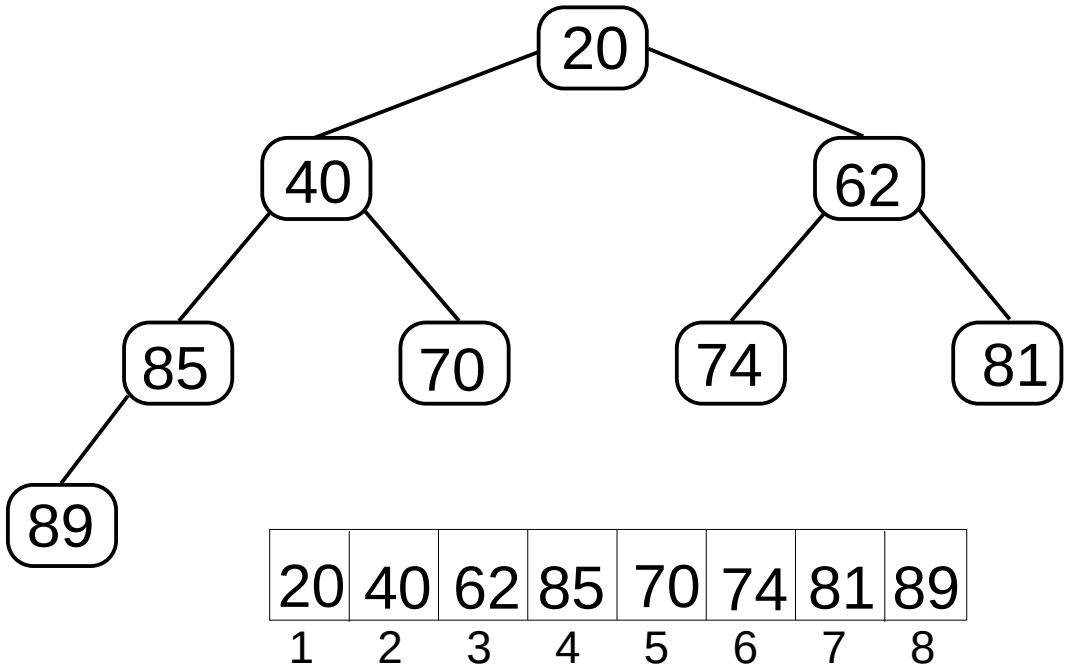
usar estruturas mais eficientes para encontrar o vértice com custo mínimo!

Filas de prioridade: heap.

Heap

Definição

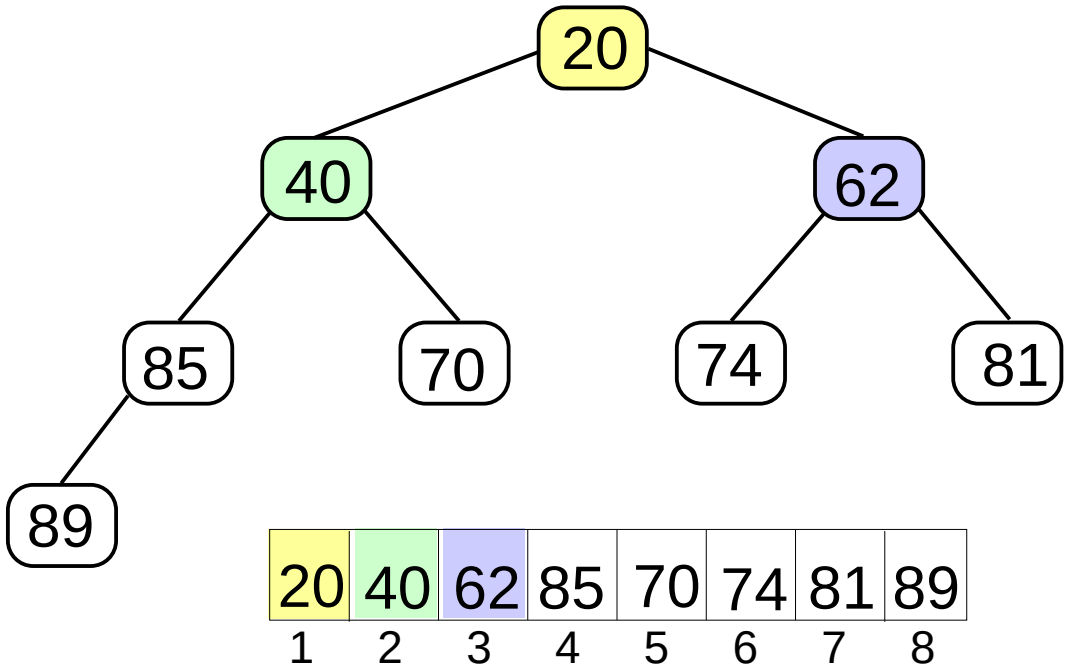
Um **heap de mínimo** é uma lista linear $c[1], c[2], c[3] \ldots c[n]$ tal que $c[i] \geq c[\lfloor \frac{i}{2} \rfloor]$, para todo valor i no intervalo $[2, n]$.



Heap

Definição

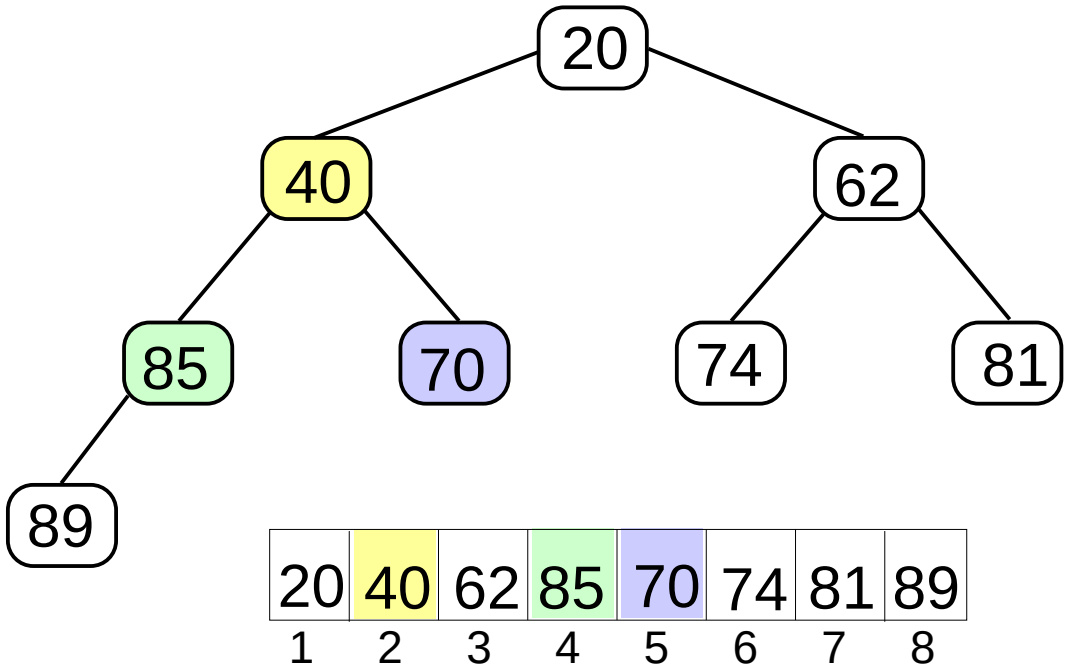
Um **heap de mínimo** é uma lista linear $c[1], c[2], c[3] \ldots c[n]$ tal que $c[i] \geq c[\lfloor \frac{i}{2} \rfloor]$, para todo valor i no intervalo $[2, n]$.



Heap

Definição

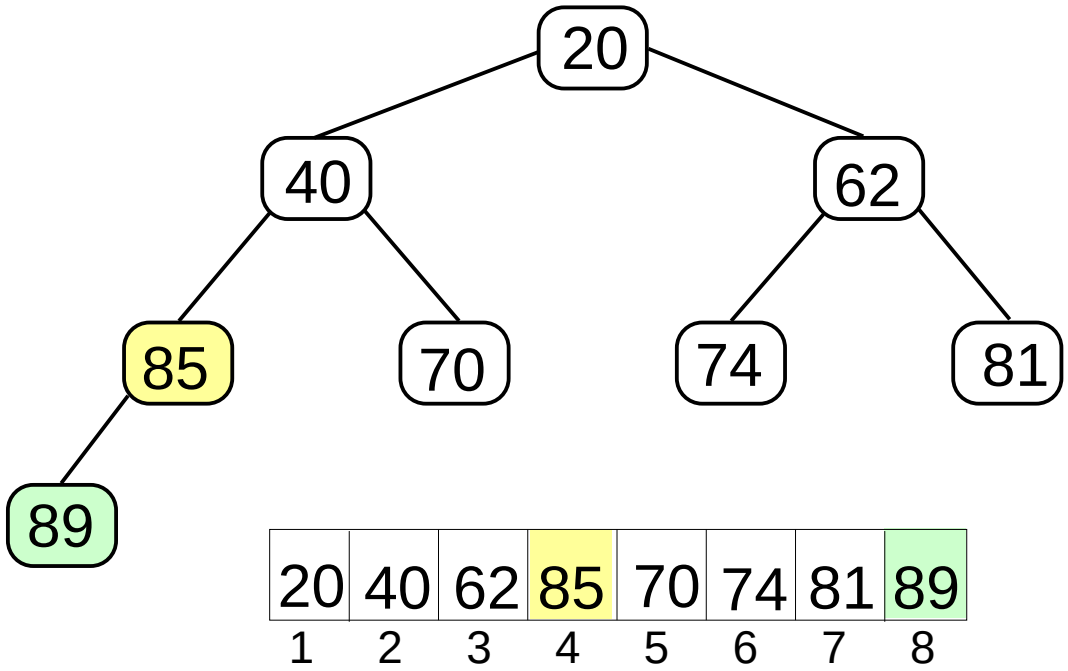
Um **heap de mínimo** é uma lista linear $c[1], c[2], c[3] \ldots c[n]$ tal que $c[i] \geq c[\lfloor \frac{i}{2} \rfloor]$, para todo valor i no intervalo $[2, n]$.



Heap

Definição

Um **heap de mínimo** é uma lista linear $c[1], c[2], c[3] \ldots c[n]$ tal que $c[i] \geq c[\lfloor \frac{i}{2} \rfloor]$, para todo valor i no intervalo $[2, n]$.



Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

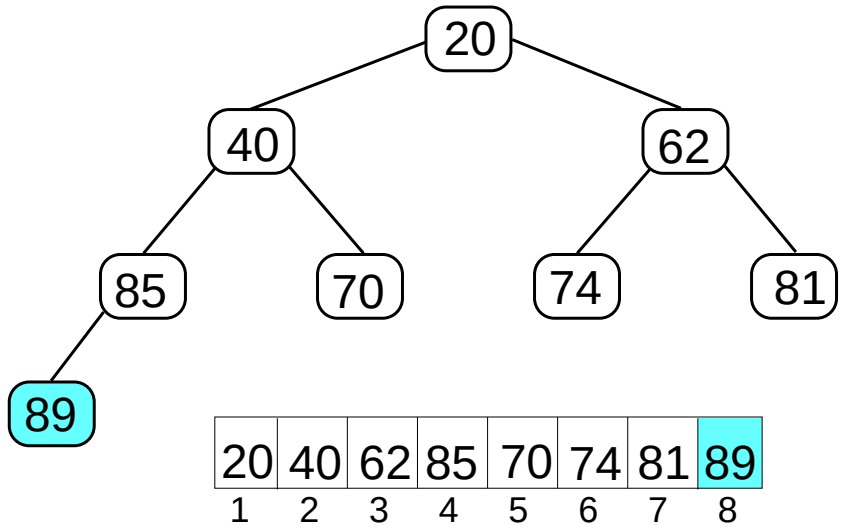
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);



Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

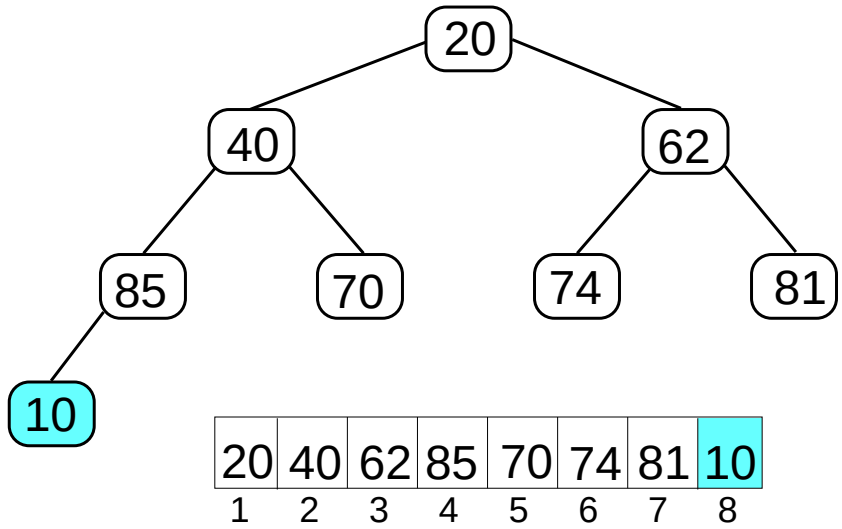
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);



Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

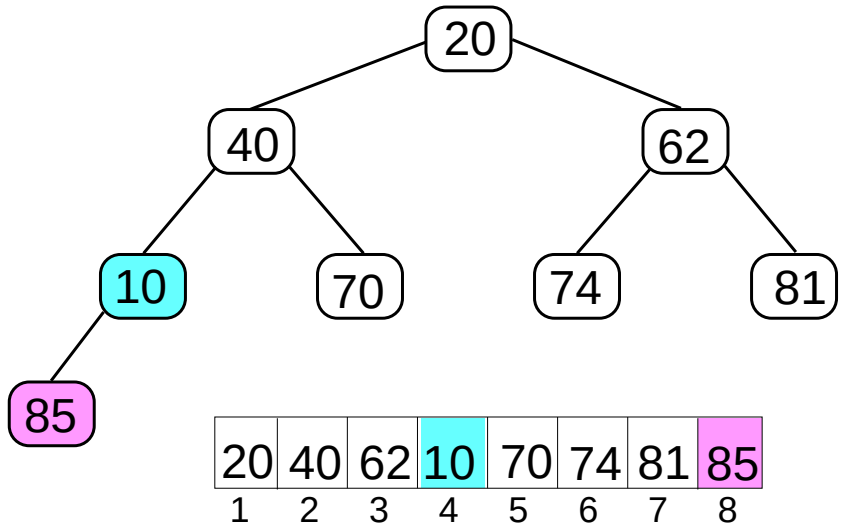
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);



Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

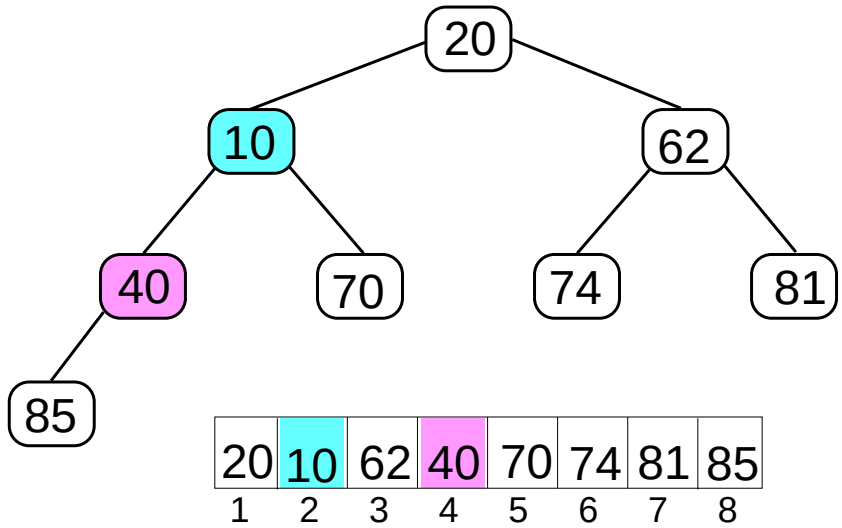
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);



Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

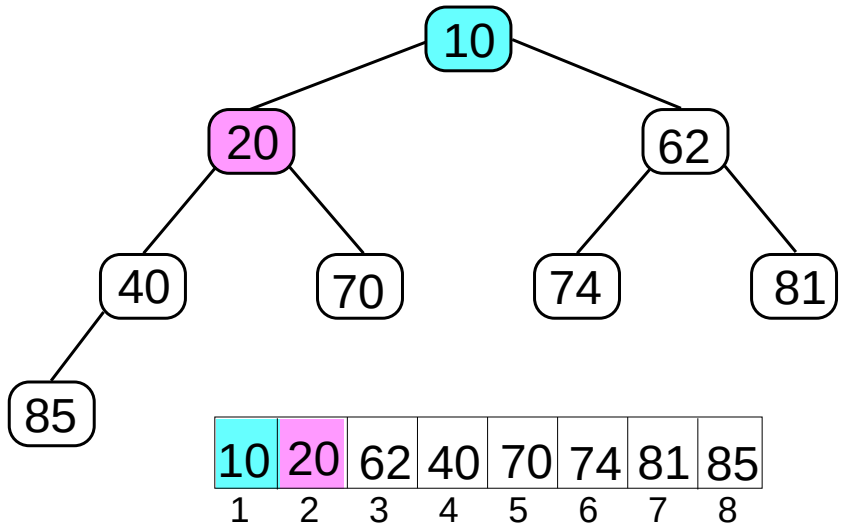
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);



Heap

Operações do heap:

SubirHeap(i)

$j \leftarrow \lfloor \frac{i}{2} \rfloor$

Se $j \geq 1$, então:

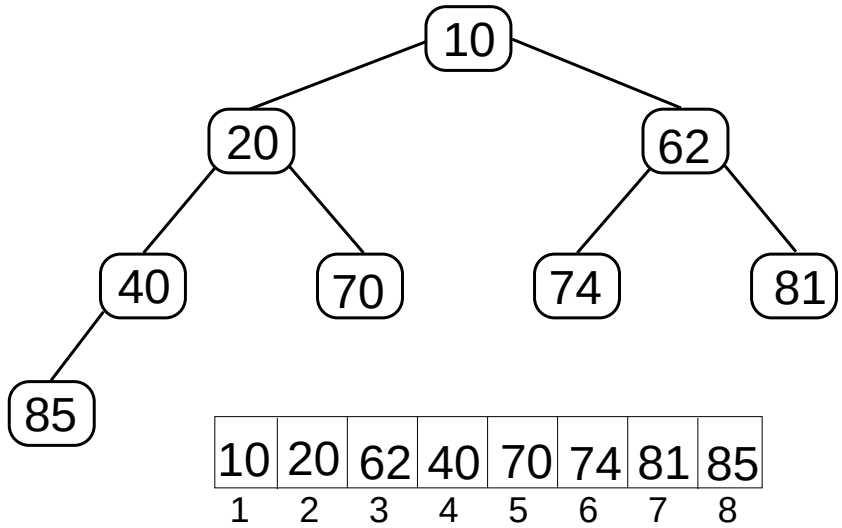
Se $c[i] < c[j]$, então:

$tmp \leftarrow c[i]$;

$c[i] \leftarrow c[j]$;

$c[j] \leftarrow tmp$;

SubirHeap(j);

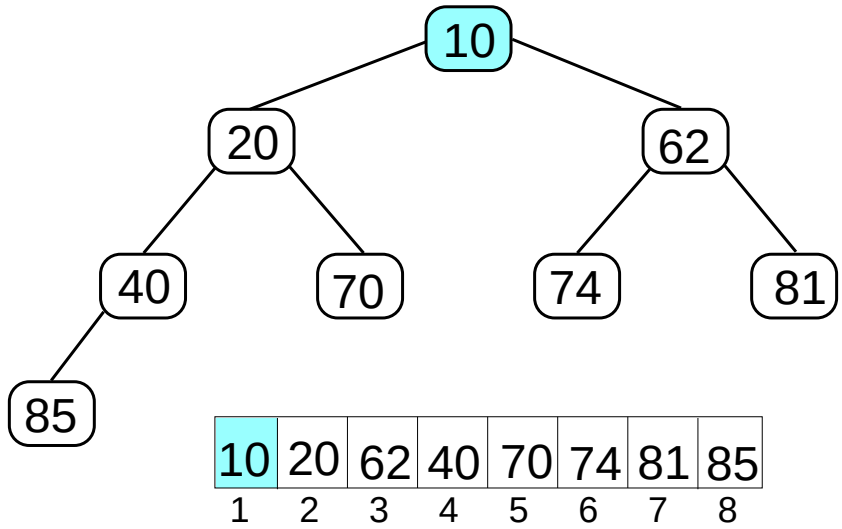


Complexidade: número de trocas na altura de árvore binária completa: $O(\log n)$.

Operações do heap:

DescerHeap(i, n)

$j \leftarrow 2 * i$;
Se $j \leq n$, então:
    Se $j < n$, então:
        Se $j + 1 \leq n$ e $c[j] > c[j + 1]$, então:
            $j \leftarrow j + 1$;
        Se $c[i] > c[j]$, então:
            $tmp \leftarrow c[i]$;
            $c[i] \leftarrow c[j]$;
            $c[j] \leftarrow tmp$;
            DescerHeap(j, n);

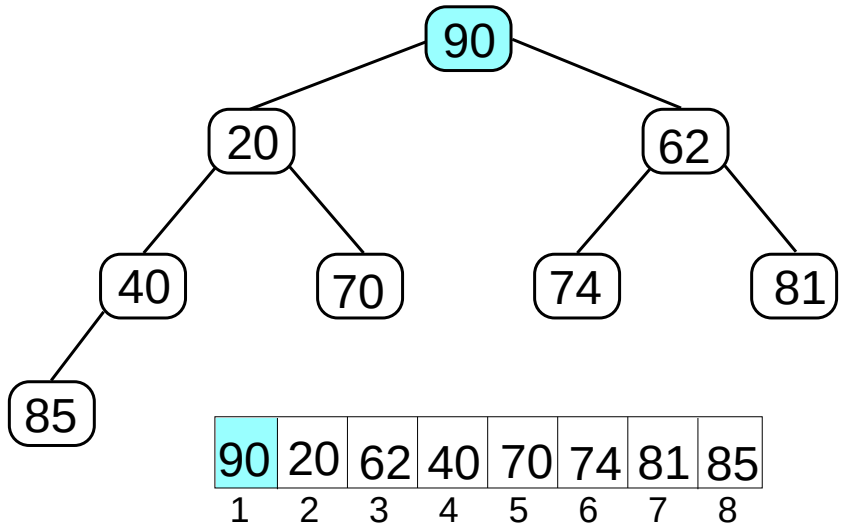


Heap

Operações do heap:

DescerHeap(i, n)

$j \leftarrow 2 * i$;
Se $j \leq n$, então:
    Se $j < n$, então:
        Se $j + 1 \leq n$ e $c[j] > c[j + 1]$,
        então:
            $j \leftarrow j + 1$;
        Se $c[i] > c[j]$, então:
            $tmp \leftarrow c[i]$;
            $c[i] \leftarrow c[j]$;
            $c[j] \leftarrow tmp$;
            DescerHeap(j, n);

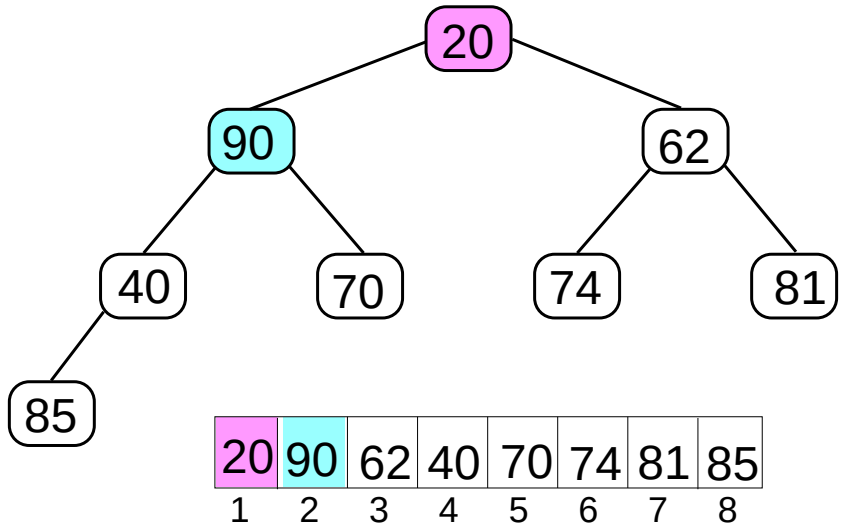


Heap

Operações do heap:

DescerHeap(i, n)

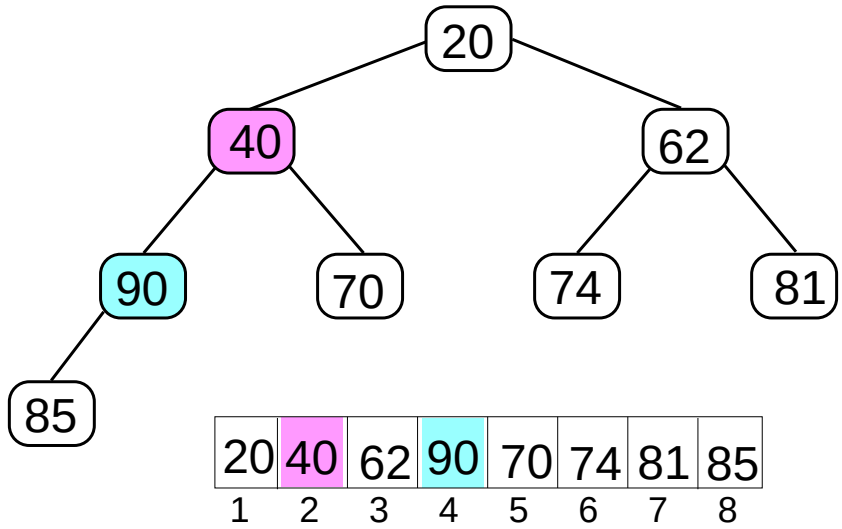
$j \leftarrow 2 * i$;
Se $j \leq n$, então:
    Se $j < n$, então:
        Se $j + 1 \leq n$ e $c[j] > c[j + 1]$,
        então:
            $j \leftarrow j + 1$;
        Se $c[i] > c[j]$, então:
            $tmp \leftarrow c[i]$;
            $c[i] \leftarrow c[j]$;
            $c[j] \leftarrow tmp$;
            DescerHeap(j, n);



Operações do heap:

DescerHeap(i, n)

$j \leftarrow 2 * i$;
Se $j \leq n$, então:
    Se $j < n$, então:
        Se $j + 1 \leq n$ e $c[j] > c[j + 1]$,
        então:
            $j \leftarrow j + 1$;
        Se $c[i] > c[j]$, então:
            $tmp \leftarrow c[i]$;
            $c[i] \leftarrow c[j]$;
            $c[j] \leftarrow tmp$;
            DescerHeap(j, n);

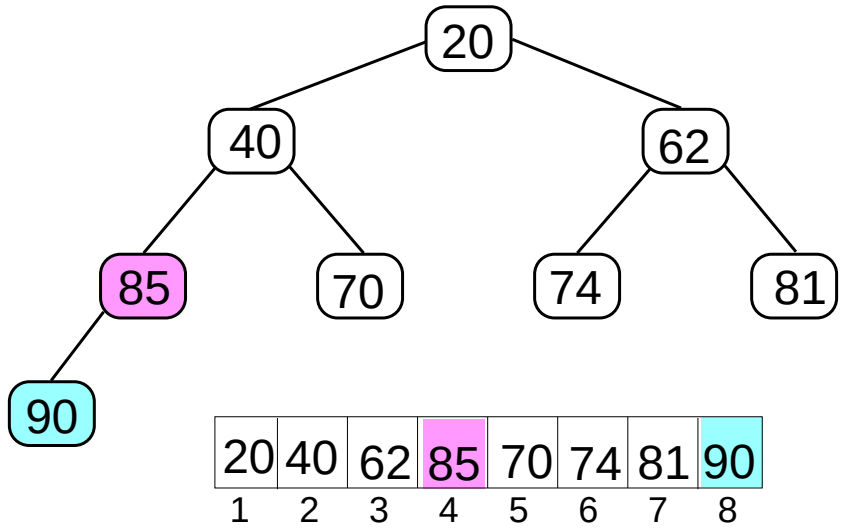


Heap

Operações do heap:

DescerHeap(i, n)

$j \leftarrow 2 * i$;
Se $j \leq n$, então:
    Se $j < n$, então:
        Se $j + 1 \leq n$ e $c[j] > c[j + 1]$,
        então:
            $j \leftarrow j + 1$;
        Se $c[i] > c[j]$, então:
            $tmp \leftarrow c[i]$;
            $c[i] \leftarrow c[j]$;
            $c[j] \leftarrow tmp$;
            DescerHeap(j, n);

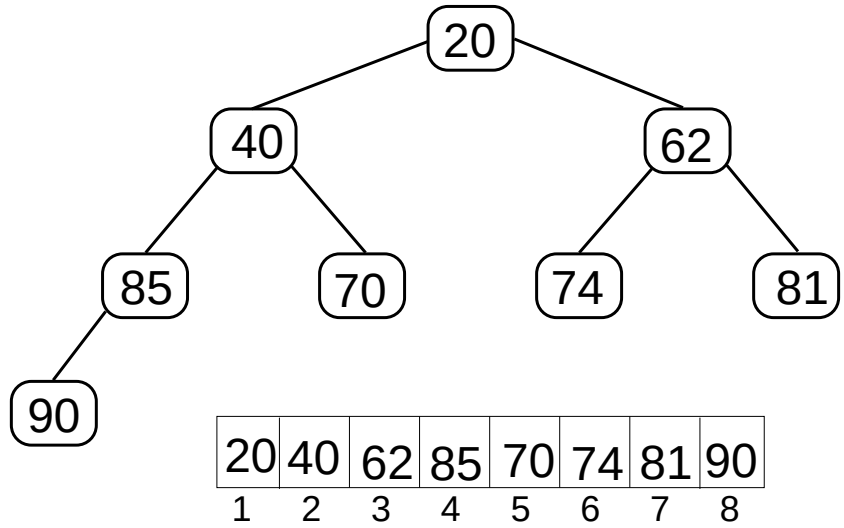


Heap

Operações do heap:

DescerHeap(i, n)

$j \leftarrow 2 * i$;
Se $j \leq n$, então:
  Se $j < n$, então:
    Se $j + 1 \leq n$ e $c[j] > c[j + 1]$, então:
      $j \leftarrow j + 1$;
    Se $c[i] > c[j]$, então:
      $tmp \leftarrow c[i]$;
      $c[i] \leftarrow c[j]$;
      $c[j] \leftarrow tmp$;
      DescerHeap(j, n);



Complexidade: número de trocas na altura de árvore binária completa: $O(\log n)$.

Heap

Operações do heap:

- **Inserir:** insere em $c[n+1]$ e executa SubirHeap$(n+1)$.
  Complexidade: $O(\log n)$.

- **Remover:** $c[1] \leftarrow c[n]$ e executa DescerHeap$(1, n-1)$.
  Complexidade: $O(\log n)$.

- **Construção do heap:** para i de $\lfloor \frac{n}{2} \rfloor$ até 1, executa DescerHeap(i, n).
  Complexidade: $O(n \log n)$.[1]

[1] É possível garantir a construção do heap em $O(n)$.

Algoritmo de Prim com Heap

Prim(G)

Para cada vértice $v \in V(G)$ faça:
  $custo[v] \leftarrow \infty$;
  $pred[v] \leftarrow NULL$;
Escolha um vértice de origem v_0;
$custo[v_0] \leftarrow 0$;
Construa um Heap H com o vetor $custo[\,]$;
Enquanto $H \neq \emptyset$ faça:
  $v \leftarrow RemoverHeap(H)$;
  Para cada vizinho u do vértice v faça:
    Se $u \in H$ e $peso[v,u] < custo[u]$ então:
      $custo[u] \leftarrow peso[v,u]$;
      $pred[u] \leftarrow v$;
      $SubirHeap(H,u)$;

Algoritmo de Prim com Heap

Prim(G)

Para cada vértice $v \in V(G)$ faça: $O(n)$

$custo[v] \leftarrow \infty$; $O(n)$

$pred[v] \leftarrow NULL$; $O(n)$

Escolha um vértice de origem v_0; $O(1)$

$custo[v_0] \leftarrow 0$; $O(1)$

Construa um Heap H com o vetor $custo[\,]$; $O(n \log n)$

Enquanto $H \neq \emptyset$ faça:

$v \leftarrow RemoverHeap(H)$; $O(\log n)$

Para cada vizinho u do vértice v faça: $O(m)$

Se $u \in H$ e $peso[v,u] < custo[u]$ então: $O(m)$

$custo[u] \leftarrow peso[v,u]$; $O(m)$

$pred[u] \leftarrow v$; $O(m)$

$SubirHeap(H,u)$; $O(m \log n)$

Complexidade do algoritmo: $O(m \log n)$.