

MC448 — Análise de Algoritmos I

Cid Carvalho de Souza Cândia Nunes da Silva
Orlando Lee

18 de novembro de 2009

Árvore Geradora Mínima

- Suponha que queremos resolver o seguinte problema: dado um conjunto de computadores, onde cada par de computadores pode ser ligado usando uma quantidade de fibra ótica, encontrar uma rede interconectando-os que use a menor quantidade de fibra ótica possível.
- Este problema pode ser modelado por um problema em grafos não orientados ponderados onde os vértices representam os computadores, as arestas representam as conexões que podem ser construídas e o peso/custo de uma aresta representa a quantidade de fibra ótica necessária.

Árvore Geradora Mínima

Árvore Geradora Mínima

- Nessa modelagem, o problema que queremos resolver é encontrar um **subgrafo gerador** (que contém todos os vértices do grafo original), **conexo** (para garantir a interligação de todas as cidades) e cuja soma dos custos de suas arestas seja a menor possível.
- Obviamente, o problema só tem solução se o **grafo** for **conexo**. Daqui pra frente vamos supor que o grafo de entrada é conexo.
- Além disso, o subgrafo gerador procurado é sempre uma árvore (supondo que os pesos são positivos).

Árvore Geradora Mínima

Problema da Árvore Geradora Mínima

Entrada: grafo conexo $G = (V, E)$ com pesos $w(u, v)$ para cada aresta (u, v) .

Saída: subgrafo gerador conexo T de G cujo peso total

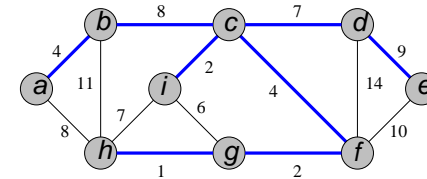
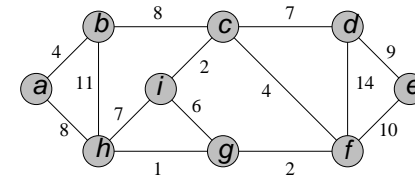
$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

seja o menor possível.

Árvore Geradora Mínima

- Veremos dois algoritmos para resolver o problema:
 - algoritmo de Prim
 - algoritmo de Kruskal
- Ambos algoritmos usam **estratégia gulosa**. Eles são exemplos clássicos de algoritmos gulosos.

Exemplo



Algoritmo genérico

- A estratégia gulosa usada baseia-se em um **algoritmo genérico** que constrói uma AGM incrementalmente.
- O algoritmo mantém um conjunto de arestas A que satisfaz o seguinte **invariante**:
No início de cada iteração, A está contido em uma AGM.
- Em cada iteração, determina-se uma aresta (u, v) tal que $A' = A \cup \{(u, v)\}$ também satisfaz o invariante.
Uma tal aresta é chamada **aresta segura** (para A).

Algoritmo genérico

AGM-GENÉRICO(G, w)

- 1 $A \leftarrow \emptyset$
- 2 **enquanto** A não é uma árvore geradora
- 3 Encontre uma aresta (u, v) segura para A
- 4 $A \leftarrow A \cup \{(u, v)\}$
- 5 **devolva** A

Obviamente o “algoritmo” está correto!

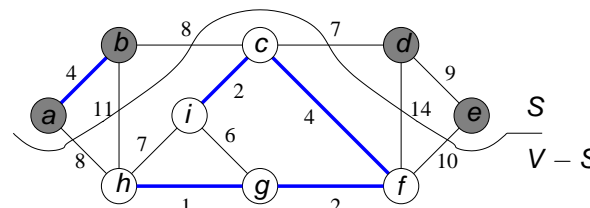
Note que nas linhas 2–4 A está propriamente contido em uma AGM, digamos T . Logo, existe uma **aresta segura** (u, v) em $T - A$.

Naturalmente, para que isso seja um algoritmo de verdade, é preciso especificar como **encontrar** uma **aresta segura**.

Como encontrar arestas seguras

Considere um grafo $G = (V, E)$ e seja $S \subset V$.

Denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V - S$. Dizemos que um tal conjunto é um **corte**.



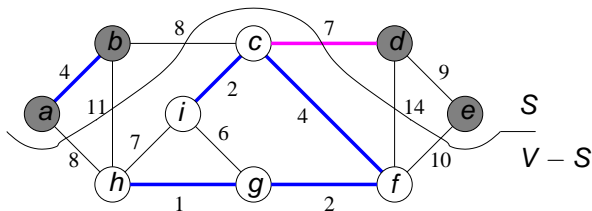
Um corte $\delta(S)$ **respeita** um conjunto A de arestas se não contém nenhuma aresta de A .

Como encontrar arestas seguras

Uma aresta de um corte $\delta(S)$ é **leve** se tem o menor peso entre as arestas do corte.

Teorema 23.1: (CLRS)

Seja G um grafo com pesos nas arestas dado por w . Seja A um subconjunto de arestas contido em uma AGM. Seja $\delta(S)$ um corte que respeita A e (u, v) uma aresta leve desse corte. Então (u, v) é uma **aresta segura**.



Como encontrar arestas seguras

Corolário 23.2 (CLRS)

Seja G um grafo com pesos nas arestas dado por w . Seja A um subconjunto de arestas contido em uma AGM. Seja C um componente (árvore) de $G_A = (V, A)$. Se (u, v) é uma aresta leve de $\delta(C)$, então (u, v) é segura para A .

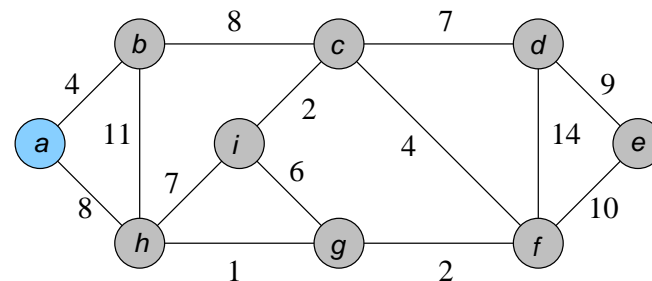
Os algoritmos de Prim e Kruskal são especializações do algoritmo genérico e fazem uso do Corolário 23.2.

O algoritmo de Prim

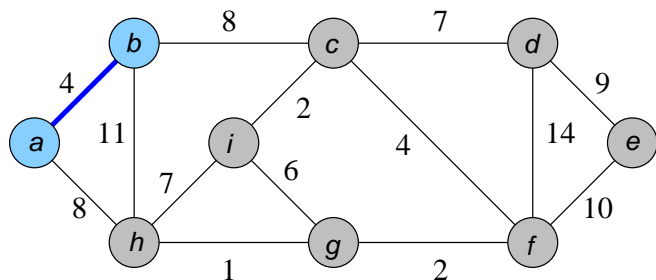
- No algoritmo de Prim, o conjunto A é uma **árvore** com raiz r (escolhido arbitrariamente no início). Inicialmente, A é vazio.
- Em cada iteração, o algoritmo considera o corte $\delta(C)$ onde C é o conjunto de vértices que são extremos de A .
- Ele encontra uma **aresta leve** (u, v) neste corte e acrescenta-a ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar **eficientemente** uma **aresta leve** no corte.

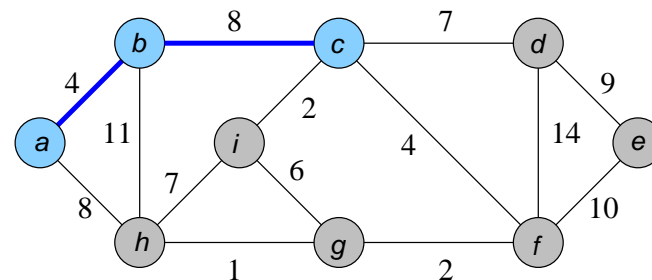
O algoritmo de Prim



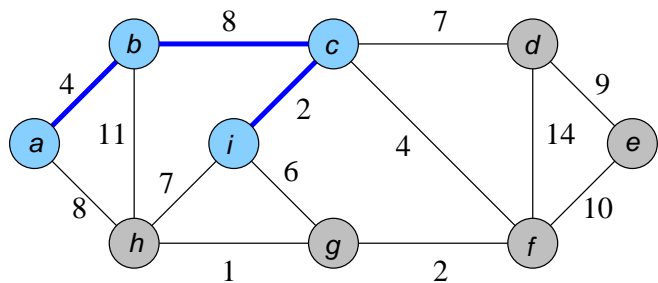
O algoritmo de Prim



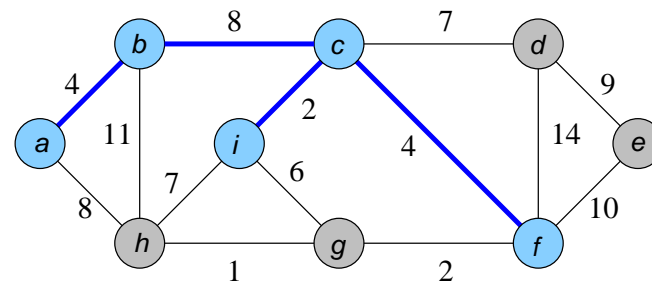
O algoritmo de Prim



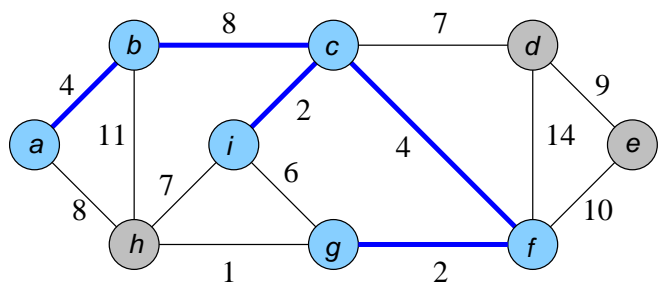
O algoritmo de Prim



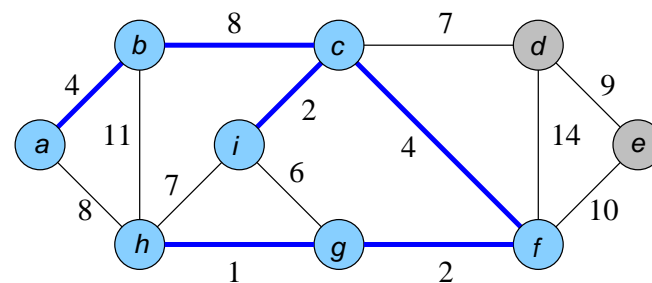
O algoritmo de Prim



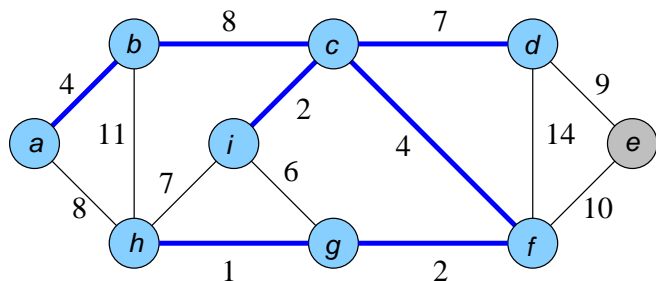
O algoritmo de Prim



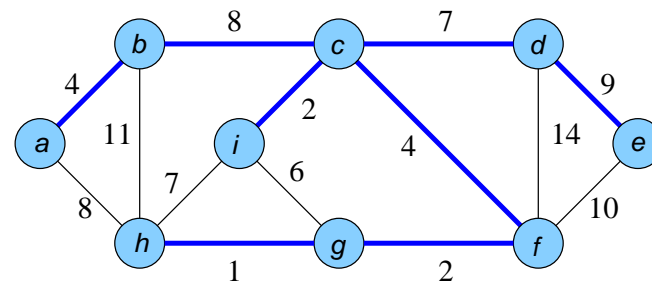
O algoritmo de Prim



O algoritmo de Prim



O algoritmo de Prim



O algoritmo de Prim

O algoritmo mantém durante sua execução as seguintes informações:

- Todos os vértices que **não** estão na árvore estão em uma fila de prioridade (de mínimo) Q .
- Cada vértice v em Q tem uma **chave** $key[v]$ que indica o menor peso de qualquer aresta ligando v a algum vértice da árvore. Se não existir nenhuma aresta, então $key[v] = \infty$.
- A variável $\pi[u]$ indica o **pai** de u na árvore. Então

$$A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}.$$

O algoritmo de Prim

AGM-PRIM(G, w, r)

```

1  para cada  $u \in V[G]$ 
2    faça  $key[u] \leftarrow \infty$ 
3     $\pi[u] \leftarrow \text{NIL}$ 
4   $key[r] \leftarrow 0$ 
5   $Q \leftarrow V[G]$ 
6  enquanto  $Q \neq \emptyset$  faça
7     $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8    para cada  $v \in \text{Adj}[u]$ 
9      se  $v \in Q$  e  $w(u, v) < key[v]$ 
10     então  $\pi[v] \leftarrow u$ 
11            $key[v] \leftarrow w(u, v)$ 

```

Corretude do algoritmo de Prim

O algoritmo mantém os seguintes invariantes.

No início de cada iteração das linhas 6–11:

- $A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}$.
- O conjunto de vértices da árvore é exatamente $V[G] - Q$.
- Para cada $v \in Q$, se $\pi[v] \neq \text{NIL}$, então $\text{key}[v]$ é o peso de uma aresta $(v, \pi[v])$ de menor peso ligando v a um vértice $\pi[v]$ na árvore.

Esse invariantes garantem que o algoritmo sempre escolhe uma **aresta segura** para acrescentar a A e portanto, o algoritmo está correto.

Complexidade do algoritmo de Prim

Pode-se fazer melhor usando uma estrutura de dados chamada *heap de Fibonacci* que guarda $|V|$ elementos e suporta as seguintes operações:

- **EXTRACT-MIN** – $O(\lg V)$,
- **DECREASE-KEY** – tempo amortizado $O(1)$.
- **INSERT** – tempo amortizado $O(1)$.
- Outras operações eficientes que um **min-heap** não suporta. Por exemplo, **UNION**.
- Maiores detalhes no CLRS.

Usando um *heap de Fibonacci* para implementar Q melhoramos o tempo para $O(E + V \lg V)$.

Este é um resultado interessante do **ponto de vista teórico**. Na prática, a implementação anterior comporta-se muito melhor.

Complexidade do algoritmo de Prim

Obviamente, a complexidade de **AGM-PRIM** depende de como a fila de prioridade Q é implementada.

Vejamos o que acontece se implementarmos Q como um **min-heap**.

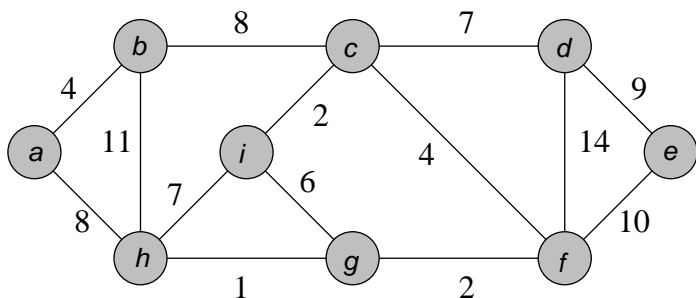
- As linhas 1–5 podem ser executadas em tempo $O(V)$ usando **BUILD-MIN-HEAP**.
- O laço da linha 6 é executado $|V|$ vezes e cada operação **EXTRACT-MIN** consome tempo $O(\lg V)$. Logo, o tempo gasto para todas as chamadas a **EXTRACT-MIN** é $O(V \lg V)$.
- O laço das linhas 8–11 é executado $O(E)$ vezes no total. O teste de pertinência a Q pode ser feito em tempo constante com um **vetor booleano**. Ao atualizar uma chave na linha 11 é feita uma *chamada implícita* a **DECREASE-KEY** que consome tempo $O(\lg V)$.
- O tempo total é $O(V \lg V + E \lg V) = O(E \lg V)$.

O algoritmo de Kruskal

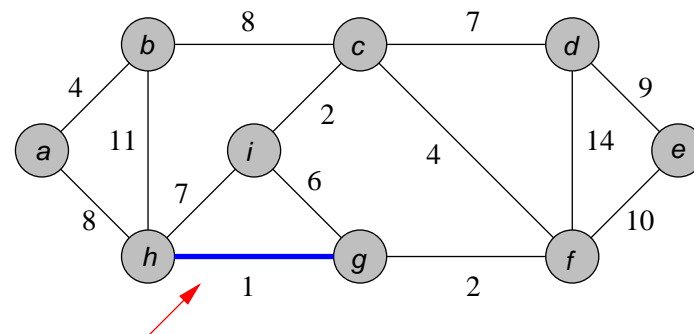
- No algoritmo de Kruskal o subgrafo $F = (V, A)$ é uma **floresta**. Inicialmente, A é vazio.
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de **menor peso** que liga vértices de componentes (árvores) distintos C e C' de $F = (V, A)$. Note que (u, v) é uma **aresta leve** do corte $\delta(C)$.
- Ele acrescenta (u, v) ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar a **aresta de menor peso** ligando componentes distintos.

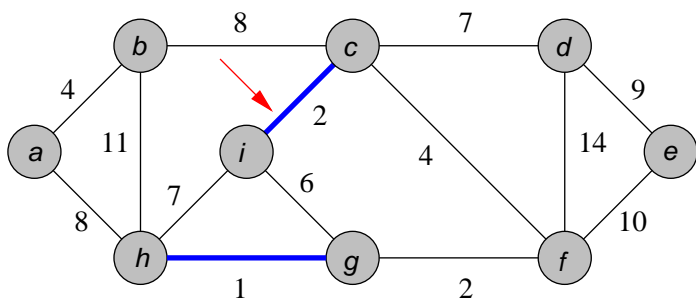
O algoritmo de Kruskal



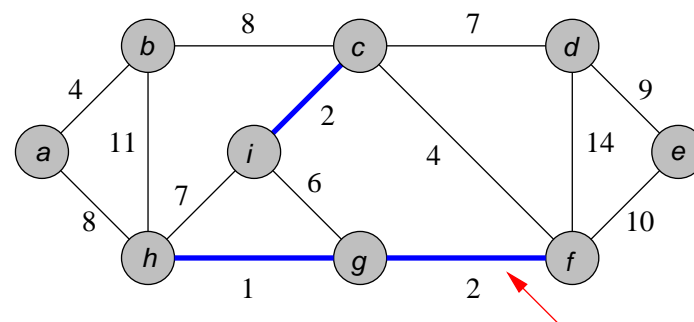
O algoritmo de Kruskal



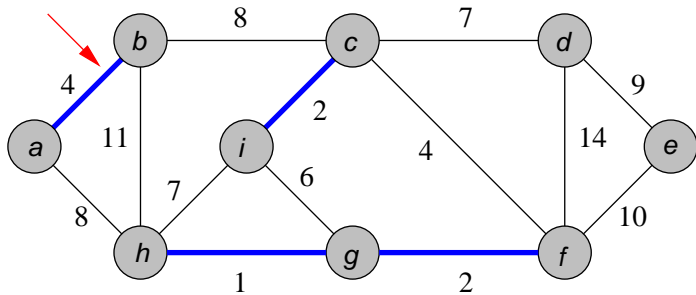
O algoritmo de Kruskal



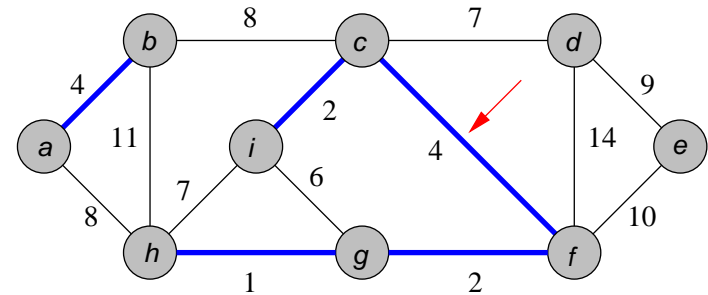
O algoritmo de Kruskal



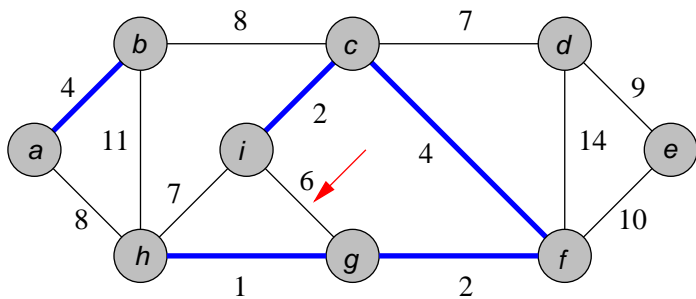
O algoritmo de Kruskal



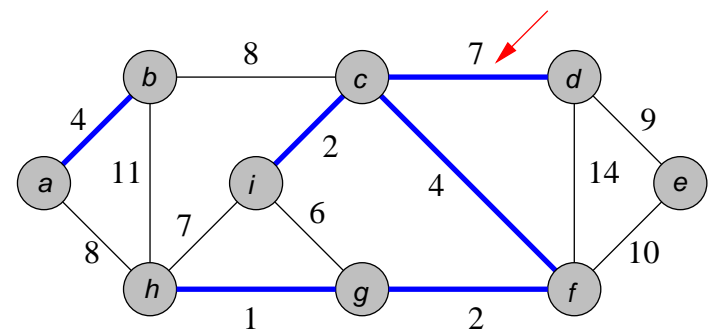
O algoritmo de Kruskal



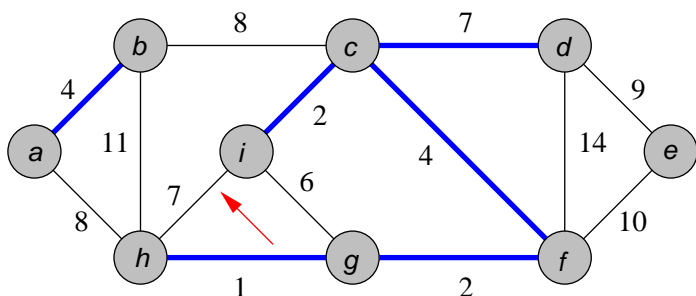
O algoritmo de Kruskal



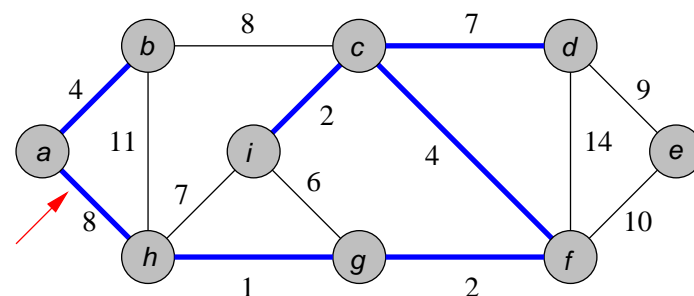
O algoritmo de Kruskal



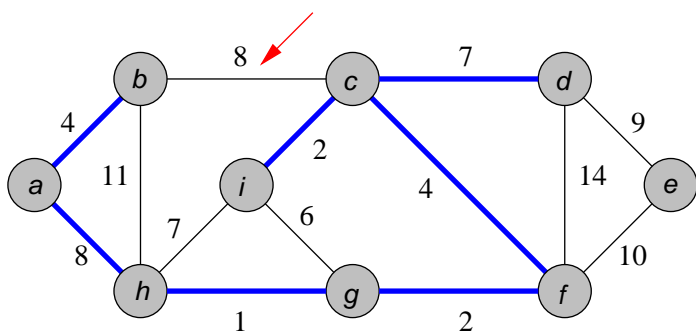
O algoritmo de Kruskal



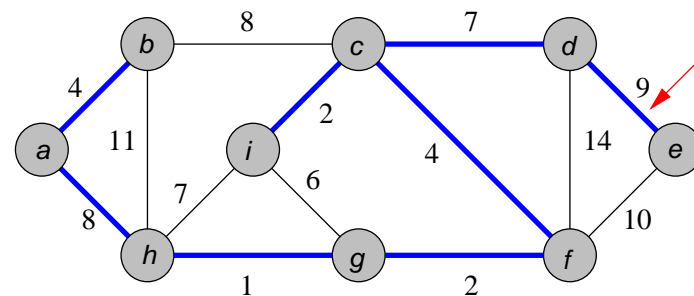
O algoritmo de Kruskal



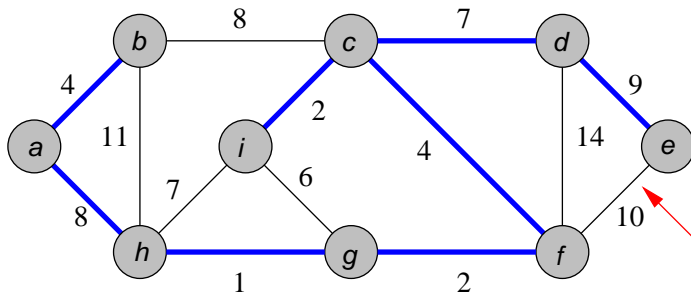
O algoritmo de Kruskal



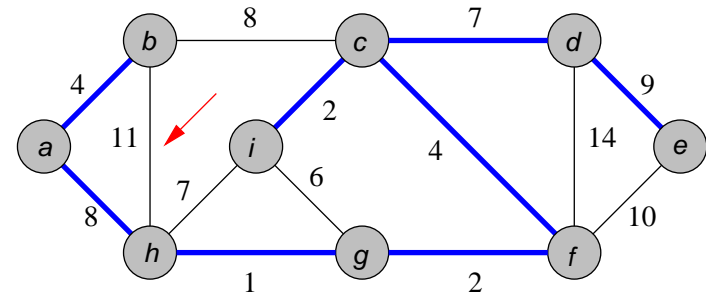
O algoritmo de Kruskal



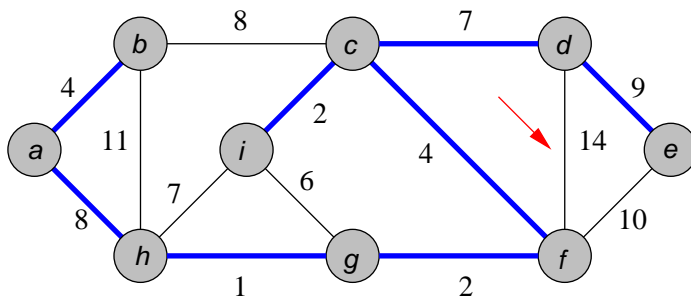
O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal

Eis uma versão preliminar do algoritmo de Kruskal.

AGM-KRUSKAL(G, w)

- 1 $A \leftarrow \emptyset$
- 2 Ordene as arestas em ordem não-decrescente de peso
- 3 **para cada** $(u, v) \in E$ nessa ordem **faça**
- 4 **se** u e v estão em componentes distintos de (V, A)
- 5 **então** $A \leftarrow A \cup \{(u, v)\}$
- 6 **devolva** A

Problema: Como verificar eficientemente se u e v estão no mesmo componente da floresta $G_A = (V, A)$?

O algoritmo de Kruskal

Inicialmente $G_A = (V, \emptyset)$, ou seja, G_A corresponde à floresta onde cada componente é um vértice isolado.

Ao longo do algoritmo, esses componentes são modificados pela inclusão de arestas em A .

Uma estrutura de dados para representar $G_A = (V, A)$ deve ser capaz de executar eficientemente as seguintes operações:

- Dado um vértice u , **determinar** o componente de G_A que contém u e
- dados dois vértices u e v em componentes distintos C e C' , fazer a **união** desses em um novo componente.

ED para conjuntos disjuntos

- Uma **estrutura de dados para conjuntos disjuntos** mantém uma coleção $\{S_1, S_2, \dots, S_k\}$ de **conjuntos disjuntos dinâmicos** (isto é, eles mudam ao longo do tempo).

- Cada conjunto é identificado por um **representante** que é um elemento do conjunto.

Quem é o representante é irrelevante, mas se o conjunto não for modificado, então o representante não pode mudar.

ED para conjuntos disjuntos

Uma **estrutura de dados para conjuntos disjuntos** deve ser capaz de executar as seguintes operações:

- **MAKE-SET**(x): cria um novo conjunto $\{x\}$.
- **UNION**(x, y): une os conjuntos (disjuntos) que contém x e y , digamos S_x e S_y , em um novo conjunto $S_x \cup S_y$.
Os conjuntos S_x e S_y são descartados da coleção.
- **FIND-SET**(x) devolve um **apontador** para o representante do (único) conjunto que contém x .

Componentes conexos

Vamos ilustrar uma aplicação simples da estrutura de dados para conjuntos disjuntos para resolver o seguinte problema.

Dado um grafo não-orientado G determinar seus componentes conexos.

Após determinar seus componentes conexos, gostaríamos também de ser capazes de verificar eficientemente se quaisquer dois vértices dados pertencem ao mesmo componente.

Componentes conexos

CONNECTED-COMPONENTS(G)

```
1 para cada vértice  $v \in V[G]$  faça
2   MAKE-SET( $v$ )
3 para cada aresta  $(u, v) \in E[G]$  faça
4   se FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
5     então UNION( $u, v$ )
```

SAME-COMPONENT(u, v)

```
1 se FIND-SET( $u$ ) = FIND-SET( $v$ )
2   então devolva VERDADEIRO
3   senão devolva FALSO
```

O algoritmo de Kruskal

Eis a versão completa do algoritmo de Kruskal.

AGM-KRUSKAL(G, w)

```
1  $A \leftarrow \emptyset$ 
2 para cada  $v \in V[G]$  faça
3   MAKE-SET( $v$ )
4 Ordene as arestas em ordem não-decrescente de peso
5 para cada  $(u, v) \in E$  nessa ordem faça
6   se FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7     então  $A \leftarrow A \cup \{(u, v)\}$ 
8     UNION( $u, v$ )
9 devolva  $A$ 
```

Componentes conexos

“Complexidade” de **CONNECTED-COMPONENTS**

- $|V|$ chamadas a **MAKE-SET**
- $2|E|$ chamadas a **FIND-SET**
- $|V| - 1$ chamadas a **UNION**

Usando a ED para conjuntos disjuntos também é fácil listar os vértices de cada componente ([Exercício](#)).

O algoritmo de Kruskal

“Complexidade” de **AGM-KRUSKAL**

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a **MAKE-SET**
- $2|E|$ chamadas a **FIND-SET**
- $|V| - 1$ chamadas a **UNION**

A complexidade depende de como essas operações são implementadas.

ED para conjuntos disjuntos

Seqüência de operações **MAKE-SET**, **UNION** e **FIND-SET**

M M M U F U U F U F F F U F

n

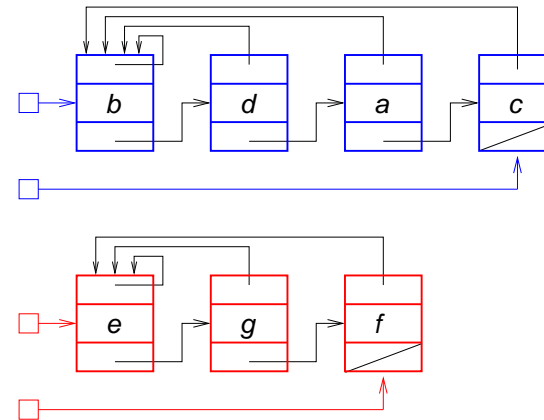
m

Vamos medir a complexidade das operações em termos de n e m .

Que estrutura de dados usar?

Ou seja, como representar os conjuntos?

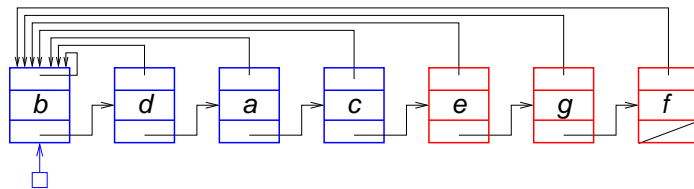
Representação por listas ligadas



- Cada conjunto tem um representante (início da lista)
- Cada nó tem um campo que aponta para o representante
- Guarda-se um apontador para o fim da lista

Representação por listas ligadas

- **MAKE-SET**(x) – $O(1)$
- **FIND-SET**(x) – $O(1)$
- **UNION**(x, y) – concatena a lista de x no final da lista de y



$O(n)$ no pior caso

É preciso atualizar os apontadores para o representante.

Um exemplo de pior caso

Operação	Número de atualizações
MAKE-SET (x_1)	1
MAKE-SET (x_2)	1
⋮	⋮
MAKE-SET (x_n)	1
UNION (x_1, x_2)	1
UNION (x_2, x_3)	2
UNION (x_3, x_4)	3
⋮	⋮
UNION (x_{n-1}, x_n)	$n-1$

Número total de operações: $2n - 1$

Custo total: $\Theta(n^2)$

Custo amortizado de cada operação: $\Theta(n)$

Uma heurística muito simples

No exemplo anterior, cada chamada de **UNION** requer em média tempo $\Theta(n)$ pois concatenamos a maior lista no final da menor.

Uma idéia simples para evitar esta situação é sempre **concatenar a menor lista no final da maior** (*weighted-union heuristic*.)

Para implementar isto basta guardar o tamanho de cada lista.

Uma única execução de **UNION** pode gastar tempo $\Theta(n)$, mas na média o tempo é bem menor (próximo slide).

Representação por *disjoint-set forests*

- Veremos agora a representação por *disjoint-set forests*.
- Implementações ingênuas não são assintoticamente melhores do que a representação por listas ligadas.
- Usando duas heurísticas — **union by rank** e **path compression** — obtemos a representação por *disjoint-set forests* mais eficiente que se conhece até hoje.

Uma heurística muito simples

Teorema. Usando a representação por listas ligadas e *weighted-union heuristic*, uma seqüência de m operações **MAKE-SET**, **UNION** e **FIND-SET** gasta tempo $O(m + n \lg n)$.

Prova.

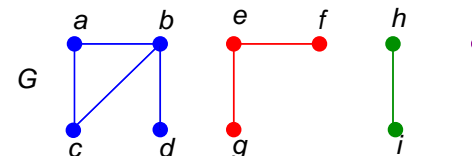
O tempo total em chamadas a **MAKE-SET** e **FIND-SET** é $O(m)$.

Sempre que o apontador para o representante de um elemento x é atualizado, o tamanho da lista que contém x (pelo menos) dobra.

Após ser atualizado $\lceil \lg k \rceil$ vezes, a lista tem tamanho pelo menos k . Como k tem que ser menor que n , cada apontador é atualizado no máximo $O(\lg n)$ vezes.

Assim, o tempo total em chamadas a **UNION** é $O(n \lg n)$.

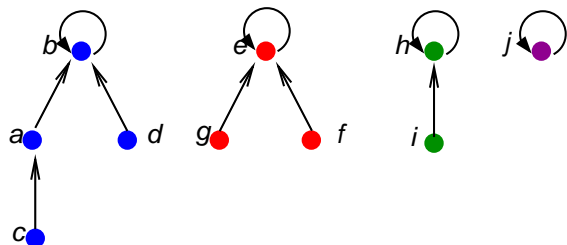
Representação por *disjoint-set forests*



Grafo com vários componentes.

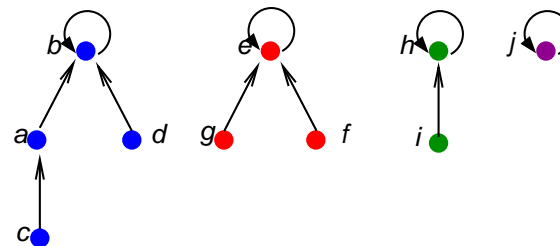
Como é a representação dos componentes na estrutura de dados *disjoint-set forests*?

Representação por *disjoint-set forests*



- Cada conjunto corresponde a uma **árvore enraizada**.
- Cada elemento aponta para seu **pai**.
- A **raiz** é o **representante** do conjunto e aponta para si mesma.

Representação por *disjoint-set forests*



MAKE-SET(x)

1 $\text{pai}[x] \leftarrow x$

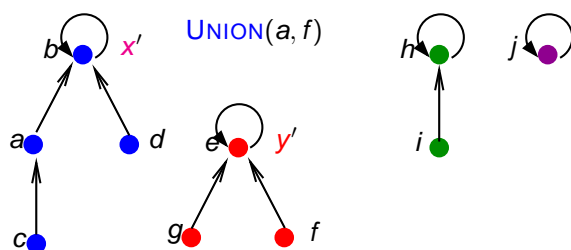
FIND-SET(x)

1 **se** $x = \text{pai}[x]$

2 **então devolva** x

3 **senão devolva** **FIND-SET**($\text{pai}[x]$)

Representação por *disjoint-set forests*



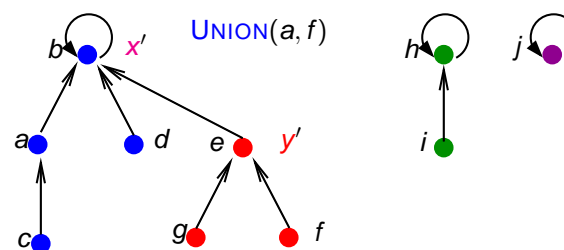
UNION(x, y)

1 $x' \leftarrow \text{FIND-SET}(x)$

2 $y' \leftarrow \text{FIND-SET}(y)$

3 $\text{pai}[y'] \leftarrow x'$

Representação por *disjoint-set forests*



UNION(x, y)

1 $x' \leftarrow \text{FIND-SET}(x)$

2 $y' \leftarrow \text{FIND-SET}(y)$

3 $\text{pai}[y'] \leftarrow x'$

Representação por *disjoint-set forests*

Com a implementação descrita até agora, **não** há **melhoria assintótica** em relação à representação por listas ligadas.

É fácil descrever uma seqüência de $n - 1$ chamadas a **UNION** que resultam em uma cadeia linear com n nós.

Pode-se melhorar (muito) isso usando duas heurísticas:

- union by rank
- path compression

Union by rank

- A idéia é emprestada do **weighted-union heuristic**.
- Cada nó x possui um “posto” $\text{rank}[x]$ que é um limitante superior para a altura de x .
- Em **union by rank** a raiz com menor rank aponta para a raiz com maior rank.

Union by rank

MAKE-SET(x)

```
1 pai[x] ← x
2 rank[x] ← 0
```

UNION(x, y)

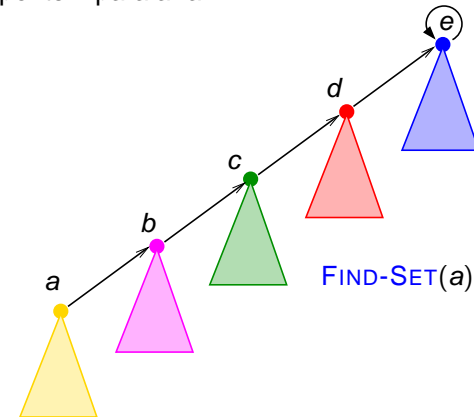
```
1 LINK(FIND-SET( $x$ ), FIND-SET( $y$ ))
```

LINK(x, y) \triangleright x e y são raízes

```
1 se rank[x] > rank[y]
2   então pai[y] ← x
3   senão pai[x] ← y
4   se rank[x] = rank[y]
5     então rank[y] ← rank[y] + 1
```

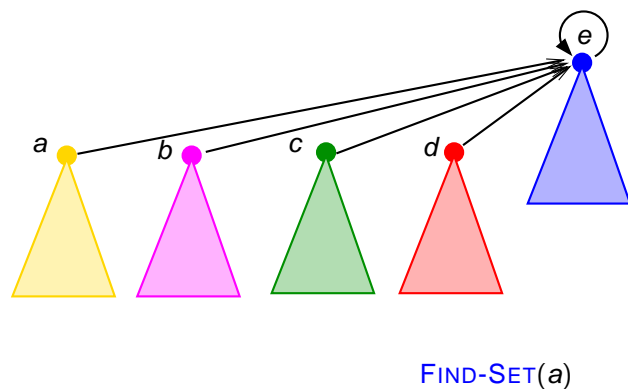
Path compression

A idéia é muito simples: ao tentar determinar o representante (**raiz** da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.

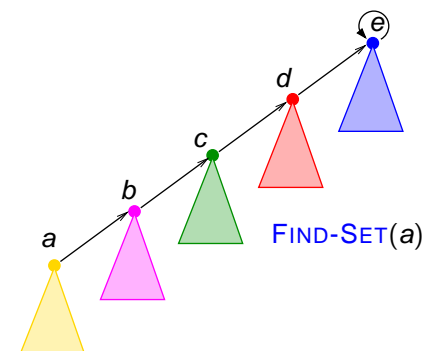


Path compression

A idéia é muito simples: ao tentar determinar o representante (raiz da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.



Path compression



```
FIND-SET(x)
1 se x ≠ pai[x]
2   então pai[x] ← FIND-SET(pai[x])
3 devolva pai[x]
```

Análise de union by rank com path compression

Vamos descrever (sem provar) a complexidade de uma seqüência de operações MAKE-SET, UNION e FIND-SET quando union by rank e path compression são usados.

Para $k \geq 0$ e $j \geq 1$ considere a função

$$A_k(j) = \begin{cases} j + 1 & \text{se } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1, \end{cases}$$

onde $A_{k-1}^{(j+1)}(j)$ significa que $A_{k-1}(j)$ foi iterada $j + 1$ vezes.

Análise de union by rank com path compression

Ok. Você não entendeu o que esta função faz. . .

Tudo que você precisa saber é que ela cresce **muito** rápido.

$$\begin{aligned} A_0(1) &= 2 \\ A_1(1) &= 3 \\ A_2(1) &= 7 \\ A_3(1) &= 2047 \\ A_4(1) &= 16^{512} \end{aligned}$$

Em particular, $A_4(1) = 16^{512} \gg 10^{80}$ que é número estimado de átomos do universo. . .

Análise de union by rank com path compression

Considere agora inversa da função $A_k(n)$ definida como

$$\alpha(n) = \min\{k : A_k(1) \geq n\}.$$

Usando a tabela anterior temos

$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq A_4(1). \end{cases}$$

Ou seja, do **ponto de vista prático**, para qualquer valor razoável de n , temos $\alpha(n) \leq 4$, ou seja, $\alpha(n)$ é uma **constante**.

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

```
1   $A \leftarrow \emptyset$ 
2  para cada  $v \in V[G]$  faça
3      MAKE-SET( $v$ )
4  Ordene as arestas em ordem não-decrescente de peso
5  para cada  $(u, v) \in E$  nessa ordem faça
6      se FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7          então  $A \leftarrow A \cup \{(u, v)\}$ 
8              UNION( $u, v$ )
9  devolva  $A$ 
```

Complexidade:

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a **MAKE-SET**
- $|E| + |V| - 1 = O(E)$ chamadas a **UNION** e **FIND-SET**

Análise de union by rank com path compression

Teorema. Uma seqüência de m operações **MAKE-SET**, **UNION** e **FIND-SET** pode ser executada em uma **ED para disjoint-set forests** com union by rank e path compression em tempo $O(m\alpha(n))$ no pior caso.

Dada a afirmação anterior de que $\alpha(n)$ é **constante** para qualquer valor razoável de n , isto significa que na prática o **tempo total** é **linear** e que o **custo amortizado por operação** é uma **constante**.

Vamos voltar agora à implementação do algoritmo de Kruskal.

O algoritmo de Kruskal (de novo)

- Ordenação: $O(E \lg E)$
- $|V|$ chamadas a **MAKE-SET**
- $O(E)$ chamadas a **UNION** e **FIND-SET**

Usando a representação *disjoint-set forests* com union by rank e path compression, o tempo gasto com as operações é $O((V + E)\alpha(V)) = O(E\alpha(V))$.

Como $\alpha(V) = O(\lg V) = O(\lg E)$ o passo que consome mais tempo no algoritmo de Kruskal é a ordenação.

Logo, a complexidade do algoritmo é $O(E \lg E) = O(E \lg V)$.