Projeto e Análise de Algoritmos II (MC558)
Árvores geradoras mínimas

Prof. Dr. Ruben Interian

Resumo

Resumo

Revisão do conteúdo

- Vimos o problema de obter uma **árvore geradora mínima** em **grafos ponderados**.
- Vimos o **Algoritmo de Prim**, um dos dois algoritmos usados para resolver esse problema.
- No **Algoritmo de Prim**, as arestas do conjunto A, que representa a AGM "parcial", sempre formam uma **única árvore**.

Objetivo

- Estudar o **Algoritmo de Kruskal** - outro algoritmo para encontrar uma **árvore geradora mínima**.

Resumo

O algoritmo de Kruskal

No **algoritmo de Kruskal**:

- Inicialmente, $A = \emptyset$. A cada momento, o subgrafo $G_A = (V, A)$ é uma floresta.

O algoritmo de Kruskal

No **algoritmo de Kruskal**:

- Inicialmente, $A = \emptyset$. A cada momento, o subgrafo $G_A = (V, A)$ é uma floresta.
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de **menor peso** que liga vértices de duas componentes (árvores) distintas C e C' de $G_A = (V, A)$.

  Note que (u, v) é uma aresta leve do corte $\delta(C) \Rightarrow$

O algoritmo de Kruskal

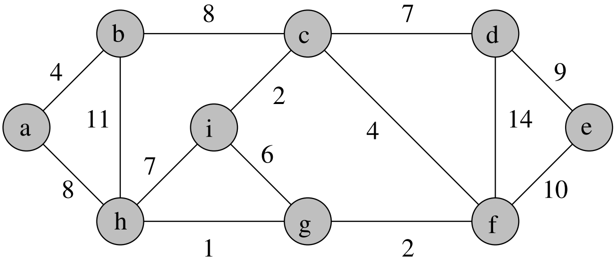
No **algoritmo de Kruskal**:

- Inicialmente, $A = \emptyset$. A cada momento, o subgrafo $G_A = (V, A)$ é uma floresta.
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de **menor peso** que liga vértices de duas componentes (árvores) distintas C e C' de $G_A = (V, A)$.

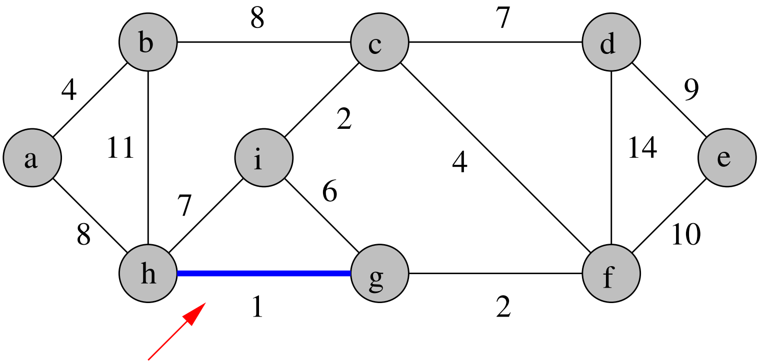
  Note que (u, v) é uma aresta leve do corte $\delta(C)$ $\Rightarrow$ é uma **aresta segura**.
- O algoritmo acrescenta (u, v) ao conjunto A e começa outra iteração até que A seja uma árvore geradora.

Um "detalhe" de implementação importante é como encontrar a aresta de menor peso ligando componentes distintos de $G_A = (V, A)$ de forma **eficiente**.

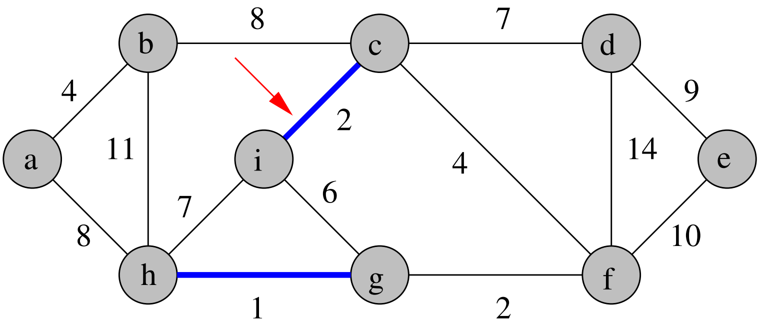
Componentes conexas



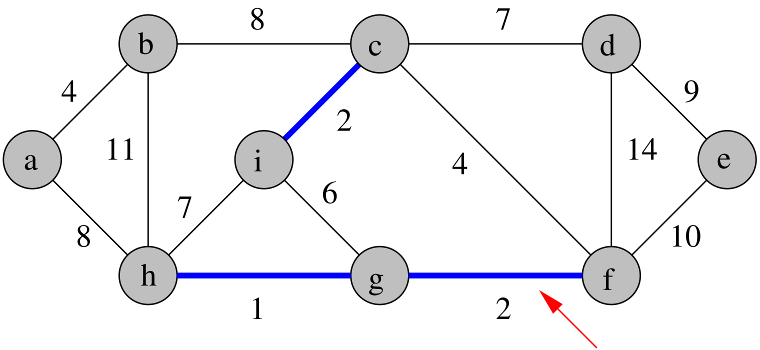
Componentes conexas



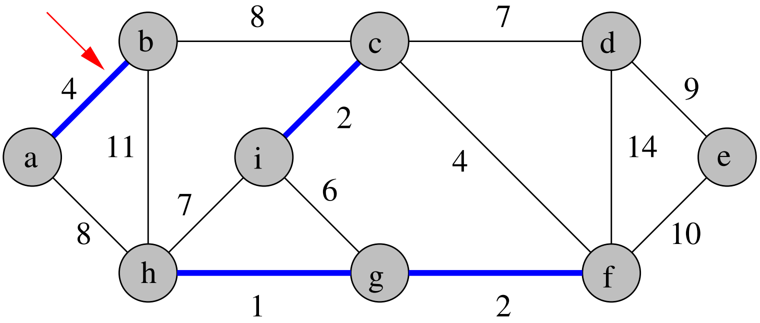
Componentes conexas



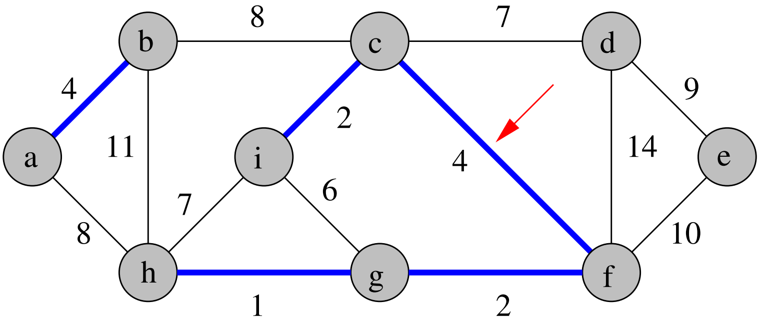
Componentes conexas



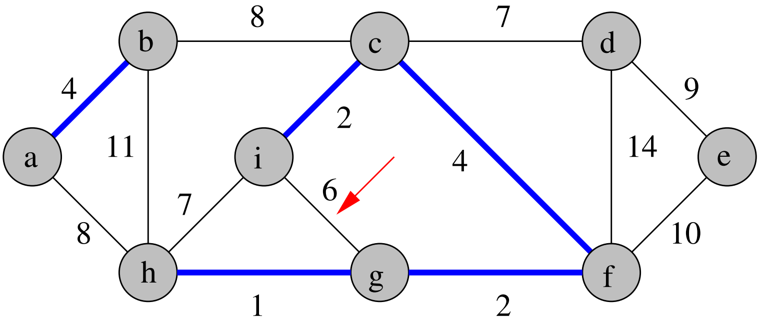
Componentes conexas



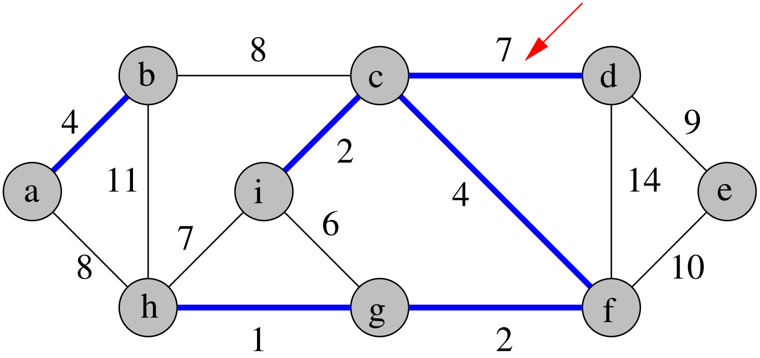
Componentes conexas



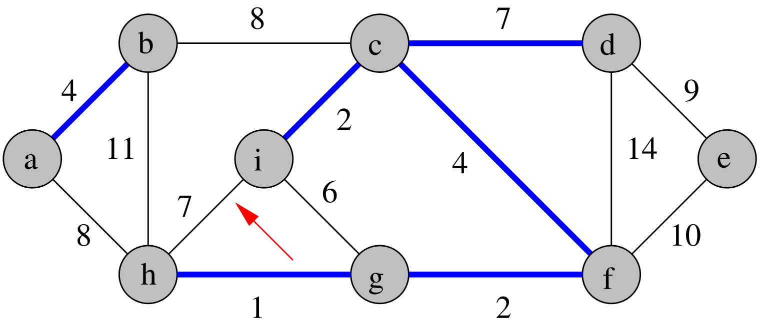
Componentes conexas



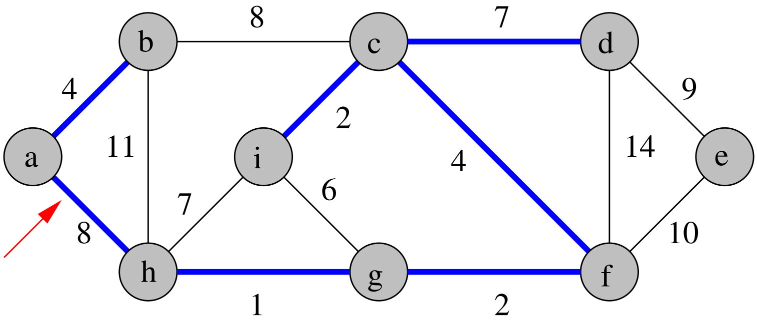
Componentes conexas



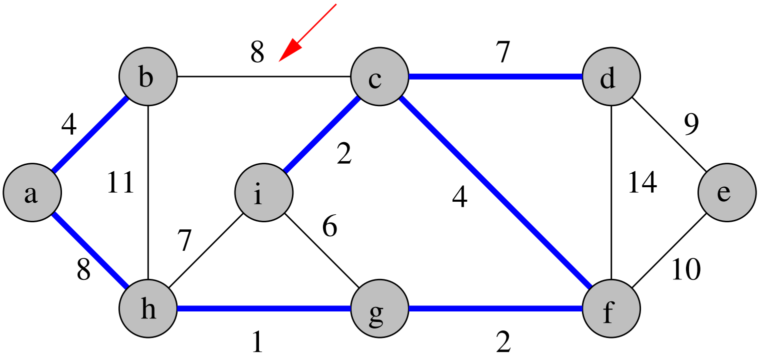
Componentes conexas



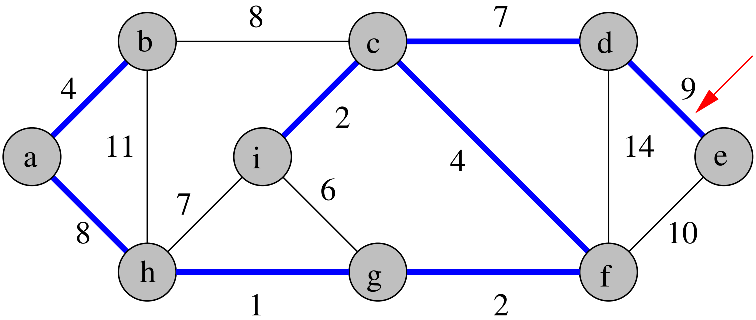
Componentes conexas



Componentes conexas



Componentes conexas



O algoritmo de Kruskal

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, \ \text{nessa ordem}, $ **faça**
4:     **se** u e v estão em componentes distintas de (V, A) **então**
5:         $A \leftarrow A \cup \{(u, v)\}$
devolva A

O algoritmo de Kruskal

Uma versão preliminar do algoritmo de Kruskal.

AGM-Kruskal (G, w)

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

Problema: Como verificar eficientemente se u e v estão em componentes diferentes da floresta $G_A = (V, A)$?

O algoritmo de Kruskal

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

Ao longo do algoritmo, as componentes são modificadas incluindo novas arestas em A.

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

- Dado um vértice u, **determinar** a componente que contém u, e
- Dados dois vértices u e v em componentes distintas C e C', juntar elas em uma nova componente ("**união**").

Estrutura de dados para conjuntos disjuntos

Estrutura de dados para conjuntos disjuntos:

- Mantém uma coleção $\{S_1, S_2, \ldots, S_k\}$ de conjuntos disjuntos **dinâmicos** (que mudam no tempo).

Estrutura de dados para conjuntos disjuntos

Estrutura de dados para conjuntos disjuntos:

- Mantém uma coleção $\{S_1, S_2, \ldots, S_k\}$ de conjuntos disjuntos **dinâmicos** (que mudam no tempo).
- Cada conjunto é identificado por um representante que é um elemento do conjunto.

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

Estrutura de dados para conjuntos disjuntos

Estrutura de dados para conjuntos disjuntos – operações:

- **MakeSet**(x): cria um novo conjunto $\{x\}$.
- **Union**(x, y): une os conjuntos disjuntos S_x, que contém x, e S_y, que contém y, em um novo conjunto $S_x \cup S_y$ (S_x e S_y desaparecem).
- **FindSet**(x): devolve um representante do conjunto que contém x.

O algoritmo de Kruskal

Versão completa do algoritmo de Kruskal:

AGM-Kruskal(G, w)

$A \leftarrow \emptyset$
para cada $v \in V$ **faça MakeSet**(v)
Ordene as arestas em ordem não-decrescente de peso.
para cada $(u, v) \in E$ nessa ordem **faça**
    se FindSet$(u) \neq$ **FindSet**(v) **então**
        $A \leftarrow A \cup \{(u, v)\}$
        Union(u, v)
devolva A

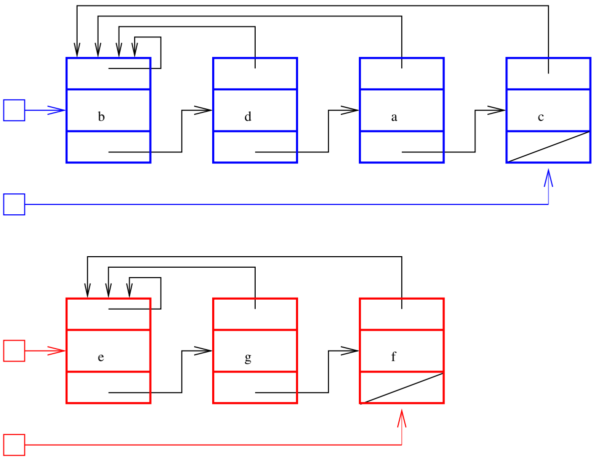
O algoritmo de Kruskal

"Complexidade" de **AGM-Kruskal**:

- **Ordenação**: $O(E \log E)$.
- O algoritmo executa uma sequência de operações **MakeSet**, **FindSet** e **Union**.
  - **MakeSet**: $|V| = n$ chamadas.
  - **FindSet**: $2|E| = 2m$ chamadas.
  - **Union**: $|V| - 1 = n - 1 < n$ chamadas.
- A complexidade depende de como essas operações são implementadas.

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.

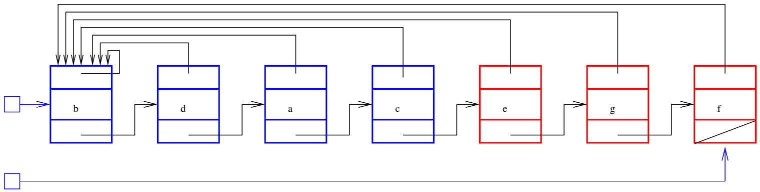


Representação por listas ligadas

- **MakeSet**(x) – $O(1)$.
- **FindSet**(x) – $O(1)$.
- **Union**(x, y) – concatena a lista de y no final da lista de x.

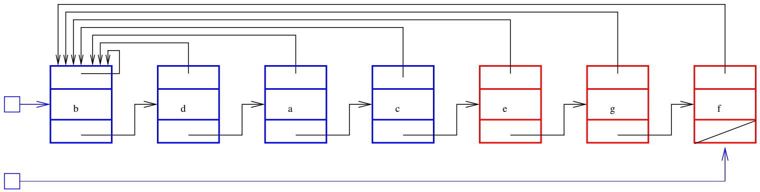
Representação por listas ligadas

- **MakeSet**(x) – $O(1)$.
- **FindSet**(x) – $O(1)$.
- **Union**(x, y) – concatena a lista de y no final da lista de x.



Representação por listas ligadas

- **MakeSet**(x) – $O(1)$.
- **FindSet**(x) – $O(1)$.
- **Union**(x, y) – concatena a lista de y no final da lista de x.



Complexidade: $O(n)$, pois precisamos atualizar os apontadores para o representante.

Um exemplo de pior caso

Operação	Atualizações	Listas
MakeSet(x_1)	1	x_1
MakeSet(x_2)	1	$x_2 \quad x_1$
$\vdots$	$\vdots$	
MakeSet(x_n)	1	$x_n \quad x_{n-1} \quad \cdots \quad x_4 \quad x_3 \quad x_2 \quad x_1$
Union(x_2, x_1)	1	$x_n \quad x_{n-1} \quad \cdots \quad x_4 \quad x_3 \quad x_2 x_1$
Union(x_3, x_2)	2	$x_n \quad x_{n-1} \quad \cdots \quad x_4 \quad x_3 x_2 x_1$
Union(x_4, x_3)	3	$x_n \quad x_{n-1} \quad \cdots \quad x_4 x_3 x_2 x_1$
$\vdots$	$\vdots$	
Union(x_n, x_{n-1})	$n-1$	$x_n x_{n-1} \cdots x_4 x_3 x_2 x_1$

Número total de operações: $2n - 1$.

Custo total: $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$. Custo amortizado de cada **Union**: $\frac{\Theta(n^2)}{n-1} = \Theta(n)$.

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 essa ideia, 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).

Uma heurística muito simples

Teorema.

Usando a representação por listas ligadas e *weighted-union heuristic*, uma sequência de n operações **MakeSet** e **Union**, e $2m$ operações **FindSet**, gasta tempo $O(m + n \log n)$.

Prova.

Uma heurística muito simples

Teorema.

Usando a representação por listas ligadas e *weighted-union heuristic*, uma sequência de n operações **MakeSet** e **Union**, e $2m$ operações **FindSet**, gasta tempo $O(m + n \log n)$.

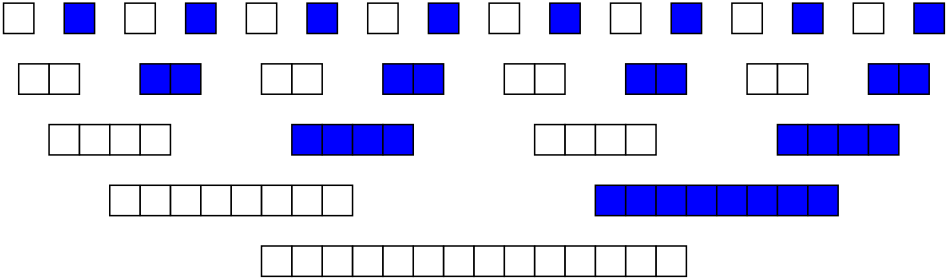
Prova. O tempo total em chamadas a **MakeSet** e **FindSet** é $O(n + m) = O(m)$.

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

Para cada x, se seu apontador foi atualizado k vezes, a sua lista ligada tem tamanho pelo menos 2^k. Mas $2^k \leq n$, o apontador de x é atualizado no máximo $O(\log n)$ vezes.

Assim, o tempo total em chamadas a **Union** (número total de atualizações) é $O(n \log n)$. O tempo total é $O(m + n \log n)$. ∎

Um exemplo de pior caso



Em cada nível, a lista em azul é concatenada com a lista a sua esquerda e assim $n/2$ apontadores são atualizados. O custo total de **Union** é $\Theta(n \log n)$

$\rightarrow$ lembrando **MergeSort**.

Complexidade do algoritmo de Kruskal

Lembrando: Complexidade de **AGM-Kruskal**:

- **Ordenação**: $O(E \log E) = O(E \log V)$ – Por quê?
- **MakeSet**: $|V|$ chamadas.
- **FindSet**: $2|E|$ chamadas.
- **Union**: $|V| - 1$ chamadas.

Complexidade do algoritmo de Kruskal

Lembrando: Complexidade de **AGM-Kruskal**:

- **Ordenação**: $O(E \log E) = O(E \log V)$ – Por quê?
- **MakeSet**: $|V|$ chamadas.
- **FindSet**: $2|E|$ chamadas.
- **Union**: $|V| - 1$ chamadas.

Usando a representação de conjuntos disjuntos por listas ligadas + *weighted-union heuristic*, o custo total é: Ordenação + $O(m + n \log n)$.

Custo total: $O(E \log V) + O(E + V \log V) = O(E \log V + V \log V) = O(E \log V)$.

- O tempo e dominado pela ordenação das arestas.

Resumo

Representação por *disjoint-set forests*

Representação por **disjoint-set forests**:

- Disjoint-set forest: floresta de conjuntos disjuntos.
- Usando essa representação, podemos obter a representação mais eficiente que se conhece até hoje.

Representação por *disjoint-set forests*

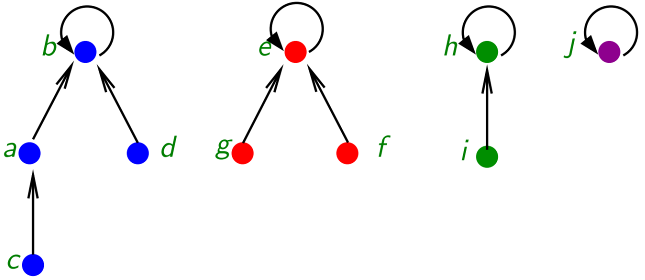
Representação por **disjoint-set forests**:

- Disjoint-set forest: floresta de conjuntos disjuntos.
- Usando essa representação, podemos obter a representação mais eficiente que se conhece até hoje.

Essa variante não diminui a complexidade assintótica de **AGM-Kruskal**, por causa do uso da ordenação. Mas podemos fazer mais eficiente a segunda "fase" do algoritmo.

Representação por *disjoint-set forests*

- Uma coleção de conjuntos é representada por uma floresta;
- Um conjunto é a uma árvore enraizada, cada elemento aponta para seu pai;
- O representante do conjunto é a raiz. A raiz aponta para si mesma.



Representação por *disjoint-set forests*

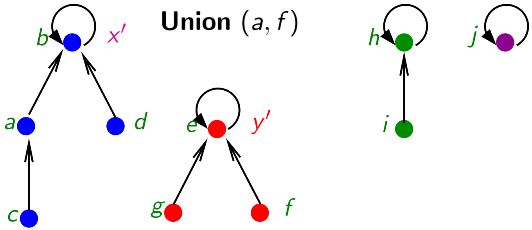
MakeSet (x)

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

FindSet (x)

1: **se** $x = \text{pai}[x]$ **então**
2: **devolva** x
3: **senão**
4: **devolva FindSet**$(\text{pai}[x])$

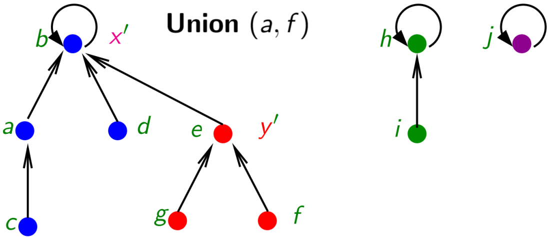
Representação por *disjoint-set forests*



Union (x, y)

1: $x' \leftarrow$ **FindSet**(x)
2: $y' \leftarrow$ **FindSet**(y)
3: $\text{pai}[y'] \leftarrow x'$

Representação por *disjoint-set forests*



Union (x, y)

1: $x' \leftarrow$ **FindSet**(x)

2: $y' \leftarrow$ **FindSet**(y)

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

Representação por *disjoint-set forests*

- **MakeSet**(x) – $O(1)$
- **FindSet**(a)

Representação por *disjoint-set forests*

- **MakeSet**(x) – $O(1)$
- **FindSet**(a) – $O(n)$
- **Union**(x, y)

Representação por *disjoint-set forests*

- **MakeSet**(x) – $O(1)$
- **FindSet**(a) – $O(n)$
- **Union**(x, y) – $O(n)$

Por enquanto, não há melhoria assintótica em relação às listas ligadas. Existem sequências de $n - 1$ chamadas a **Union** resultam em uma cadeia linear com n nós. Isto torna **FindSet** custoso demais.

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

- Union by rank,
- Path compression.

Union by rank

Union by rank:

- Ideia: limitar a altura das árvores.
- Cada nó x possui associado um valor rank[x].
- rank[x] pode ser igual à altura de x na arvore, mas pode ser um número menor.
- Idéia parecida a weighted-union heuristic: a raiz com menor **rank** irá apontar para a raiz com maior **rank**.

Union by rank

MakeSet (x)

1: $\text{pai}[x] \leftarrow x$
2: $\text{rank}[x] \leftarrow 0$

Union (x, y)

1: $\textbf{Link}(\textbf{FindSet}(x), \textbf{FindSet}(y))$

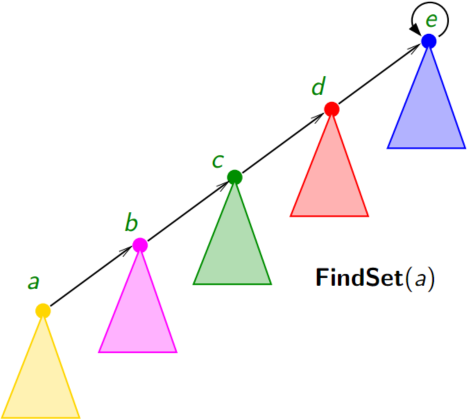
Union by rank

Link (x, y) x e y são raízes

se $\text{rank}[x] > \text{rank}[y]$ **então**
    $\text{pai}[y] \leftarrow x$
senão
    $\text{pai}[x] \leftarrow y$
    se $\text{rank}[x] = \text{rank}[y]$ **então**
        $\text{rank}[y] \leftarrow \text{rank}[y] + 1$

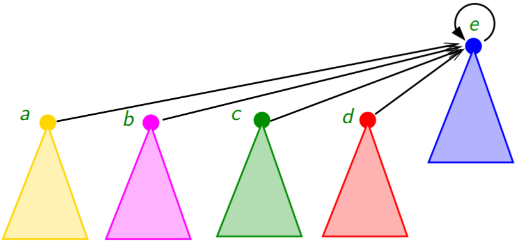
Path compression

Idéia de *path compression* (compressão de caminhos): durante o **FindSet**, fazemos com que todos os nós no caminho apontem para a raiz. Ou seja **colocamos atalhos**.



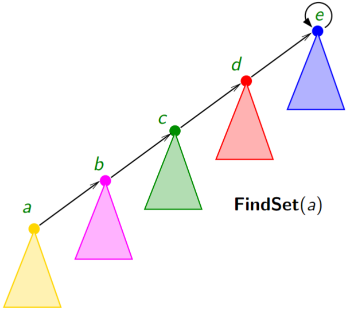
Path compression

Idéia de *path compression* (compressão de caminhos): durante o **FindSet**, fazemos com que todos os nós no caminho apontem para a raiz. Ou seja **colocamos atalhos**.



FindSet(a)

Path compression



FindSet (x)

1: **se** $x \neq$ pai[x] **então** pai[x] $\leftarrow$ **FindSet**(pai[x])
devolva pai[x]

Análise de union by rank e path compression

- Usando a estrutura de dados disjoint-set forest somente com union by rank, pode-se mostrar que o custo total é $O(m \log n)$.
- Usando a estrutura de dados disjoint-set forest somente com path compression, se são feitas f chamadas a **FindSet**, é possível mostrar que o custo total é $O(n + f \cdot (1 + \log_{2+f/n} n))$.
- Quando combinamos as duas heurísticas juntas, o custo total é $O(m \cdot \alpha(n))$ onde $\alpha = A^{-1}$ é a Função Inversa de Ackermann.
- Esta é a melhor implementação conhecida.

Função de Ackermann e sua Inversa

Função de Ackermann e sua Inversa

- A Função de Ackermann A é uma função computável: existe um "algoritmo" para computar seu valor.

Função de Ackermann e sua Inversa

Função de Ackermann e sua Inversa

- A Função de Ackermann A é uma função computável: existe um "algoritmo" para computar seu valor.
- O que ela tem de especial: ela **não é** uma função primitiva recursiva. **Todas** as funções matemáticas vistas até agora são primitivas recursivas: adição, divisão, **min**, **max**, fatorial, exponencial, logaritmo, n-ésimo primo, operações lógicas ...
- Foi a primeira função computável que não é primitiva recursiva descoberta por Wilhelm Ackermann, que foi aluno de **David Hilbert**.

Função de Ackermann e sua Inversa

Função de Ackermann e sua Inversa

- A Função de Ackermann A é uma função computável: existe um "algoritmo" para computar seu valor.
- O que ela tem de especial: ela **não é** uma função primitiva recursiva. **Todas** as funções matemáticas vistas até agora são primitivas recursivas: adição, divisão, **min**, **max**, fatorial, exponencial, logaritmo, n-ésimo primo, operações lógicas ...
- Foi a primeira função computável que não é primitiva recursiva descoberta por Wilhelm Ackermann, que foi aluno de **David Hilbert**.
- **Importante**: é possível mostrar que **toda função primitiva recursiva cresce mais devagar** do que a Função de Ackermann.
- Portanto, a Função Inversa de Ackermann é uma função que cresce muito, muito, **muito** lentamente. Mais devagar do que $\log n$, e mais devagar do que $\log \log n$.

Função de Ackermann e sua Inversa

Função de Ackermann e sua Inversa

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0, \\ A(m-1,1) & \text{se } m > 0, n = 0, \\ A(m-1, A(m, n-1)) & \text{se } m > 0, n > 0. \end{cases}$$

Função de Ackermann e sua Inversa

Função de Ackermann e sua Inversa

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0, \\ A(m-1,1) & \text{se } m > 0, n = 0, \\ A(m-1, A(m, n-1)) & \text{se } m > 0, n > 0. \end{cases}$$

Valores de A:

$A(0,0) = 1$, $A(1,1) = 3$, $A(2,2) = 7$,

$A(3,3) = 61$, $A(4,4) = 2^{2^{10^{19729}}}$

$A(4,4) >$ # estimado átomos no Universo

Valores de $\alpha = A^{-1}$:

$\alpha(61) = 3$,

$\alpha(x) = 4$,

se $62 \leq x \leq 2^{2^{10^{19729}}}$.

Análise de union by rank com path compression

Quando a estrutura de dados disjoint-set forest é usada com union by rank e path compression, a complexidade de tempo de uma sequência de m operações **MakeSet**, **Union** e **FindSet** é $O(m \cdot \alpha(n))$ no pior caso.

- Não iremos mostrar este resultado, que se chama também Teorema de Tarjan.

Análise de union by rank com path compression

Quando a estrutura de dados disjoint-set forest é usada com union by rank e path compression, a complexidade de tempo de uma sequência de m operações **MakeSet**, **Union** e **FindSet** é $O(m \cdot \alpha(n))$ no pior caso.

- Não iremos mostrar este resultado, que se chama também Teorema de Tarjan.
- A função $m \cdot \alpha(n)$ é superlinear, mas para qualquer valor **razoável** (usado na prática) de n, temos $\alpha(n) \leq 4$, e $\alpha(n)$ é uma constante.
- Na prática, o tempo total das operações com os conjuntos disjuntos é linear e o custo amortizado por operação é uma constante.
- Discussão mais detalhada da estrutura de dados disjoint-set forests no Capítulo 21 do Cormen.

O algoritmo de Kruskal (de novo)

Voltando à implementação do algoritmo de Kruskal. Podemos supor que o grafo é conexo e assim $V = O(E)$:

AGM-Kruskal (G, w)

$A \leftarrow \emptyset$
para cada $v \in V$ **faça**
    MakeSet(v)
Ordene as arestas em ordem não-decrescente de peso.
para cada $(u, v) \in E$ nessa ordem **faça**
    se FindSet$(u) \neq$ **FindSet**(v) **então**
        $A \leftarrow A \cup \{(u, v)\}$
        Union(u, v)
devolva A

O algoritmo de Kruskal (de novo)

Complexidade:

- **Ordenação**: $O(E \log E) = O(E \log V)$
- **MakeSet**: $|V|$ chamadas.
- **FindSet**: $2|E|$ chamadas.
- **Union**: $|V| - 1$ chamadas.

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

Complexidade: $O(E \log(V) + E\alpha(V))$.

O algoritmo de Kruskal (de novo)

Complexidade:

- **Ordenação**: $O(E \log E) = O(E \log V)$
- **MakeSet**: $|V|$ chamadas.
- **FindSet**: $2|E|$ chamadas.
- **Union**: $|V| - 1$ chamadas.

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

Complexidade: $O(E \log(V) + E\alpha(V))$.

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

Resumo

Síntese

- Estudamos o **Algoritmo de Kruskal**, que constroi uma **AGM** mantendo uma floresta com árvores que possuem conjuntos disjuntos de vértices.
- A estrutura de dados disjoint-set forests com as heurísticas union by rank e path compression, permite que o tempo total gasto com as operações com conjuntos disjuntos seja, **na prática**, linear.
- O custo amortizado por operação, neste caso, é uma constante.

Material bibliográfico e exercícios

T. Cormen et al. Algoritmos - Teoria e Prática (3a ed.). – **Cap. 21, 23**

Exercícios: ver exercícios no final dos (sub)capítulos 23.1 e 23.2.

Observação: A função $A_k(j)$ definida no Cormen (Cap. 21.4) é semelhante à função de Ackermann $A(m, n)$, e a função inversa do Cormen é semelhante à inversa $\alpha(n)$ apresentada aqui. Ambas são no máximo 4 para todos os valores práticos de m e n.

Dúvidas

Dúvidas?