Algoritmos

Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest, Ronald L. Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden

Apresentação Baseada:

- Stanford Algorithms
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt5EMI2s2WQBsLsZl7A5HEK6`
- Conjunto de Slides dos Professores Cid C. de Souza, Cândida N. da Silva, Orlando Lee, Pedro J. de Rezende
- Conjunto de Slides do Professores Cid C. de Souza para a disciplina MO420

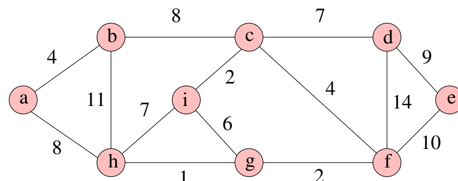
Qualquer erro é de minha responsabilidade.

O algoritmo de Kruskal para AGM

Principais passos do algoritmo de Kruskal:

1. Ordenar as arestas em ordem **não** decrescente de peso e inicializar A como estando vazio.
2. Seja (u, v) a próxima aresta na ordem **não** decrescente de peso. Se ela formar um ciclo com as arestas de A, então ela é **rejeitada**. Caso contrário, ela é **aceita** e faz-se $A = A \cup \{(u, v)\}$.
3. Repetir o passo anterior até que $|V| - 1$ arestas tenham sido **aceitas**.

Exemplo:



O algoritmo de Kruskal para AGM

- O algoritmo de Kruskal é baseado diretamente no *algoritmo genérico* discutido anteriormente.
- Note que quando uma aresta (u, v) é aceita, as componentes C_1 e C_2 contendo u e v, respectivamente, são *distintas*. É fácil ver que (u, v) é uma *aresta leve* para $\delta(C_1)$ (ou $\delta(C_2)$). Portanto, (u, v) satisfaz às condições do Corolário 23.3, o que garante a corretude do algoritmo.
- Em uma iteração qualquer, a solução parcial corrente é a floresta composta por todos vértices do grafo e as arestas em A.
- Para implementar o algoritmo de Kruskal, precisamos encontrar uma maneira eficiente de *manter as componentes* da floresta $G_A = (V, A)$.

O algoritmo de Kruskal para AGM

- Em particular, as componentes devem ser armazenadas de modo que duas operações sejam feitas muito rapidamente:
  - dado um vértice u, **encontrar** a componente contendo u;
  - dados dois vértices u e v em componentes distintas C e C', **unir** C e C' em uma única componente.
- para prosseguir com esta discussão, vamos apresentar um pseudo-código do algoritmo de Kruskal.
- Neste pseudo-código, denota-se por $a[u]$ a componente (árvore) de $G_A = (V, A)$ que contém o vértice u na iteração corrente.

Algoritmo de Kruskal: pseudo-código

KRUSKAL(G, w)

1. $W \leftarrow 0$; $A \leftarrow \emptyset$; ▷ inicializações
   ▷ Iniciar G_A com $|V|$ árvores com um vértice cada
2. **para** todo $v \in V$ **faça** $a[v] \leftarrow \{v\}$; ▷ $a[v]$ é identificado com v
   ▷ lista de arestas em ordem não decrescente de peso
3. $L \leftarrow$ **ordene**(E,w);
4. $k \leftarrow 0$; ▷ conta arestas **aceitas**
5. **enquanto** $k \neq |V| - 1$ **faça**
6.   **remove**$(L, (u, v))$; ▷ tomar primeira aresta em L
       ▷ acha componentes de u e v
7. $a[u] \leftarrow$ **encontrar**(u); $a[v] \leftarrow$ **encontrar**(v);
8.   **se** $a[u] \neq a[v]$ **então** ▷ aceita (u, v) se não forma ciclo com A
9. $A \leftarrow A \cup \{(u, v)\}$;
10. $W \leftarrow W + w(u, v)$;
11. $k \leftarrow k + 1$;
12.   **unir**$(a[u], a[v])$; ▷ unir componentes
13. **retorne** (A, W).

Algoritmo de Kruskal: complexidade

- Supor que as componentes de G_A são mantidas de modo que as operações de **encontrar** na linha 7 e **unir** na linha 12 sejam feitas com complexidade $O(f(|V|))$ e $O(g(|V|))$, respectivamente.
- A ordenação da linha 3 tem complexidade $O(|E| \log |E|)$ e domina a complexidade das demais operações das inicializações das linhas de 1 a 4.
- O laço das linhas 5–12 será executado $O(|E|)$ no pior caso.
  Logo, a complexidade total das linhas 6 e 7 será $O(|E|.f(|V|))$.
- As linhas de 9 a 12 serão executados $|V| - 1$ vezes no total (por quê ?). Assim, a complexidade total de execução destas linhas será $O(|V|.g(|V|))$.
- A complexidade do algoritmo de Kruskal será então

$$O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$$

Fica claro que necessitamos de uma estrutura de dados que permita *manipular* eficientemente as componentes de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

- Note que os conjuntos de vértices das componentes de G_A formam uma *coleção de conjuntos disjuntos* de V. Sobre estes conjuntos é executada uma seqüência de operações **encontrar** e **unir**.
- A necessidade de representação e manipulação eficiente de coleção de conjuntos disjuntos sob estas operações ocorre em áreas diversas como construção de compiladores e problemas combinatórios, como é o caso da AGM.
- Estruturas de dados simples como vetores contendo informação sobre qual a componente contendo cada elemento (vértice) realizam a operação **encontrar** em $O(1)$ mas, no pior caso, consomem um tempo $O(|V|)$ para realizar uma operação de **união**.
- A alternativa é o uso de estruturas ligadas do tipo **árvore**

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Cuidado ! Não confunda a estrutura de dados árvore que estaremos usando para armazenar as componentes de G_A com as árvores da floresta G_A.

Ou seja, nesta estrutura os registros estão ligados numa estrutura tipo árvore **cujos apontadores ligando registros de vértices distintos não necessariamente correspondem a uma aresta de** A (pode ser inclusive que nem exista uma aresta com estas mesmas extremidades no grafo).

Para evitar confusão o termo *componente* será usado na discussão que se segue para designar vértices em uma mesma árvore de G_A. O termo *árvore* denotará uma parte da estrutura de dados que representa uma componente de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Na discussão que se segue, supomos que, no início, seja criado um **registro** para cada vértice u de V e que o endereço deste registro esteja armazenado em alguma variável de modo que possa ser acessado a qualquer momento do algoritmo.

Além disso, cada registro r da estrutura terá dois campos obrigatórios (M) e um terceiro campo opcional (O):

- `rot (M)`: contendo o rótulo do vértice que o originou;
- `prx (M)`: apontador ligando registros de uma mesma árvore;
- `num (O)`: número de registros da estrutura de dados tais que, seguindo o campo `prx` até que este seja nulo, chega-se ao registro r.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

encontrar(u):

▷ retorna o apontador para o registro que representa
a componente onde se encontra o vértice u

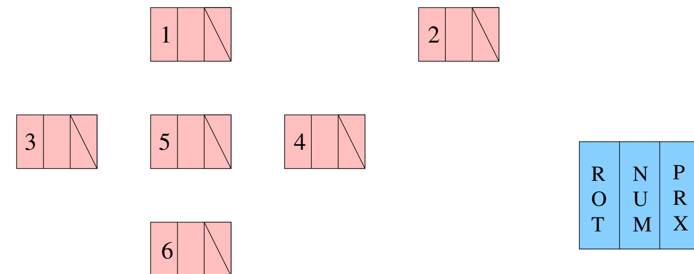
1. $p \leftarrow$ endereço do registro correspondente a u;
2. **enquanto** (p^.prx $\neq$ nulo) **faça**
3. $p \leftarrow p$^.prx;
4. **retorne** p.

unir(p,q): **versão inicial**

▷ **Entrada**: dois apontadores para registros correspondentes
a componentes distintas de G_A
▷ **Saída**: apontador para o registro correspondente ao
primeiro parâmetro da entrada

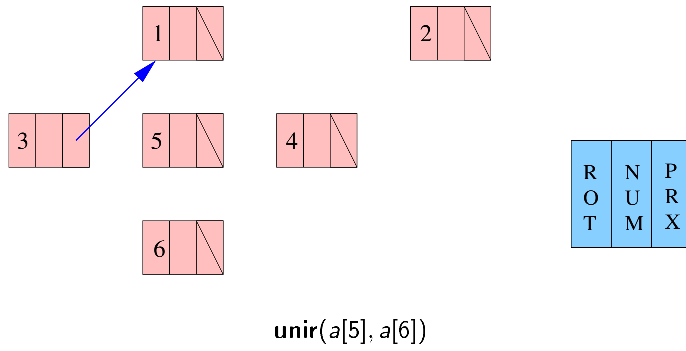
1. q^.prx $\leftarrow p$
2. **retorne** p.

Conjuntos disjuntos: exemplo



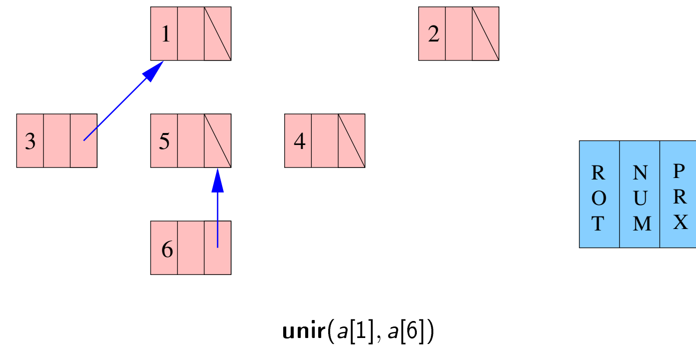
unir(a[1], a[3])

Conjuntos disjuntos: exemplo



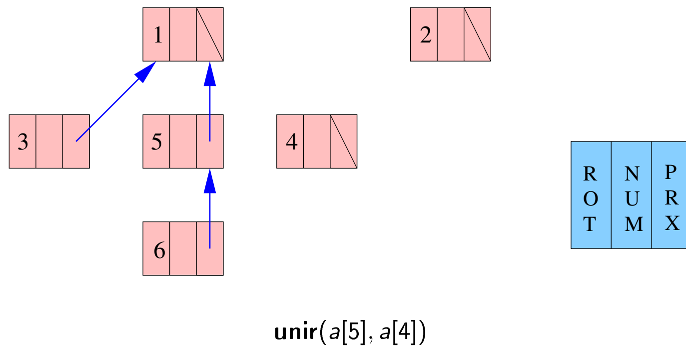
unir($a[5], a[6]$)

Conjuntos disjuntos: exemplo



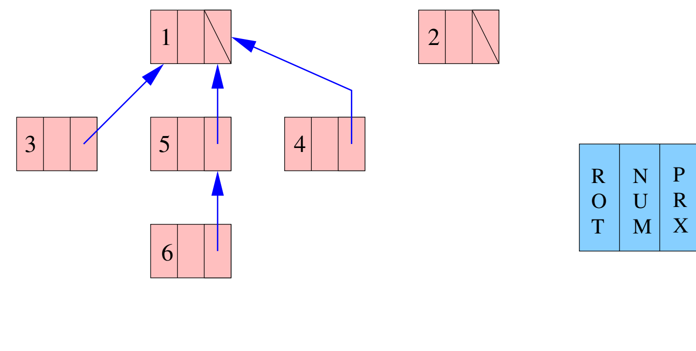
unir($a[1], a[6]$)

Conjuntos disjuntos: exemplo



unir($a[5], a[4]$)

Conjuntos disjuntos: exemplo



Conjuntos disjuntos: complexidade das operações

Complexidades:

Denote-se por T_x a árvore contendo o registro do vértice x e h_x e n_x respectivamente sua altura e seu número de elementos. No pior caso teríamos as complexidades $O(h_u)$ para **encontrar(u)** e $O(1)$ para **unir(p,q)**.

Dificuldade:

Uma das árvores pode se transformar em um longo *caminho*. Neste caso, uma operação envolvendo esta árvore teria uma complexidade de pior caso muito alta ($O(|V|)$).

Alternativa:

Construir *árvores balanceadas* colocando no campo `num` do registro r, o número de registros na sub-árvore "abaixo" de r.

Conjuntos disjuntos: complexidade das operações

unir(p,q): versão melhorada

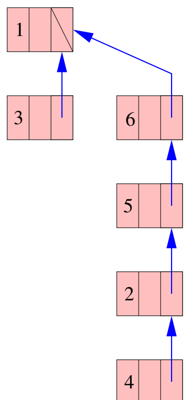
▷ **Entrada**: dois apontadores para registros correspondentes a componentes distintas de G_A

▷ **Saída**: apontador para o registro com maior valor no campo `num`

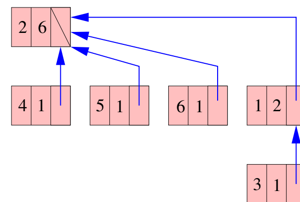
1. **se** q^.num $\leq$ p^.num **então**
2. q^.prx $\leftarrow$ p;
3. p^.num $\leftarrow$ p^.num + q^.num;
4. **retorne** p.
5. **se não**
6. p^.prx $\leftarrow$ q;
7. q^.num $\leftarrow$ q^.num + p^.num;
8. **retorne** q.

Seqüência de operações: unir(a[1], a[3]), unir(a[2], a[4]), unir(a[5], a[2]), unir(a[6], a[5]) e unir(a[1], a[6]).

união não-balanceda:



união balanceda:



Conjuntos disjuntos: complexidade das operações

Teorema:

Considere uma seqüência $s_0, s_1, \ldots, s_m$ de operações sobre uma estrutura de dados representando conjuntos disjuntos. A operação s_0 simplesmente cria os registros representando os conjuntos unitários formados por cada um dos elementos. Em $s_1, \ldots, s_m$ todas as operações são do tipo **encontrar** ou **unir**, sendo esta última feita de modo *balanceada*.

Seja x um registro qualquer da estrutura e T_x a árvore que armazena x. Após a operação s_m, T_x é uma **árvore balanceada**.

Conjuntos disjuntos: complexidade das operações

Prova: seja n_x o número de registros em T_x e h_x a sua altura. Deve se mostrar que $2^{h_x-1} \leq n_x$ e, consequentemente, fica provado que $h_x \in O(\log n_x)$.

A prova é feita por indução no número de operações. Na base, o resultado é trivialmente verdadeiro para s_0 já que toda árvore tem um único elemento.

Suponha que o resultado é verdadeiro até o término de s_{m-1}. Deve-se analisar dois casos: *(i)* s_m é uma operação **encontrar**(u) e *(ii)* s_m é uma operação **unir**(v, w).

O caso *(i)* é trivial pois nenhuma árvore é alterada. No caso *(ii)*, sem perda de generalidade, vamos supor que $n_v \leq n_w$.

Conjuntos disjuntos: complexidade das operações

Prova (cont.):
Sejam T_v e T_w as árvores contendo os registros dos vértices v e w, respectivamente. Seja T' a árvore resultante da operação **unir**(v, w) com altura e número de registros dados por h' e n', respectivamente. Note que $n' = n_v + n_w$ e que deve ser provado que $2^{h'-1} \leq n'$.

Claramente $h' = \max\{1 + h_v, h_w\}$. Tem-se que

$$\begin{aligned} 2^{h'-1} &= 2^{\max\{1+h_v, h_w\}-1} = \max\{2^{h_v-1+1}, 2^{h_w-1}\} \\ &\leq \max\{2n_v, n_w\} \qquad \text{(pela H.I.)}. \end{aligned}$$

Como $n' = n_v + n_w$ e $n_v \leq n_w$, então $2n_v \leq n'$ e $n_w \leq n'$.

Logo, $2^{h'-1} \leq n'$. □

Algoritmo de Kruskal: complexidade

- Vimos anteriormente que se as operações de **encontrar** e **unir** fossem feitas com complexidades $O(f(|V|))$ e $O(g(|V|))$, respectivamente, a complexidade do algoritmo de Kruskal seria dada por $O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$.
- O resultado do Teorema garante que usando a **união por tamanho** as árvore de registros na estrutura de dados tem altura limitada a $O(\log |V|)$.
- Portanto, neste caso, a complexidade do algoritmo de Kruskal é dada por $O(|E| \log |E| + |E| \log |V| + |V|)$ ou, como estamos supondo que o grafo é conexo ($|E| \in \Omega(|V|)$), a complexidade é $O(|E| \log |E|) = O(|E| \log |V|)$.
- Melhorias na complexidade ainda podem ser alcançadas usando a técnica de **compressão de caminhos** ao se executar uma operação **encontrar**.

Algoritmo de Kruskal: compressão de caminhos

Na técnica de compressão de caminhos, todo registro visitado durante uma operação **encontrar** que termina em um registro q tem seu campo `prx` alterado de modo a apontar para q.

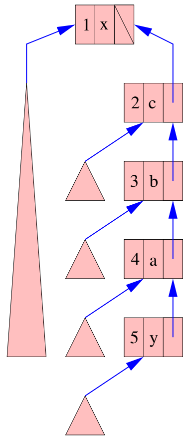
encontrar(u)**: adaptada para compressão de caminhos**

1. $p \leftarrow$ endereço do registro correspondente a u;
2. $q \leftarrow p$; ▷ guardará o último registro da busca
3. **enquanto** q^.prx $\neq$ nulo **faça** $q \leftarrow q$^.prx;
   ▷ refaz a busca fazendo todos registros apontarem para q
4. **enquanto** p^.prx $\neq q$ **faça**
5. $r \leftarrow p$^.prx;
6. p^.prx $\leftarrow q$;
7. $p \leftarrow r$;
8. **retorne** q.

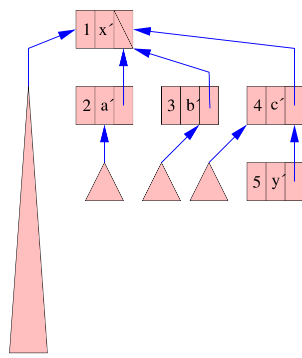
Nota: no algoritmo acima, a informação do campo `num` só é correta para o registro "raiz" da árvore.

Compressão de caminhos: exemplo **encontrar**(4)

Antes:



Depois:



Ajustando o valor de `num`: $x' = x$, $a' = a - y$, $b' = b - a$, $c' = c - b$, $y' = y$

Compressão de caminhos: complexidade

- Pode ser provado que, usando compressão de caminhos, a complexidade de se realizar $|V| - 1$ operações **unir** e $|E|$ operações **encontrar** tem complexidade dada por $O(|V| + |E|\alpha(|V| + |E|, |V|))$.
- Para compreender este resultado, é necessário conhecer a função de Ackerman A e o seu inverso α definidas por:

$$A(p,q) = \begin{cases} 2^q & p = 1,\ q \geq 1 \\ A(p-1, 2) & p \geq 2,\ q = 1 \\ A(p-1, A(p, q-1)) & p \geq 2,\ q \geq 2 \end{cases}$$

$$\alpha(m,n) = \min\{p \geq 1 : A(p, \lfloor m/n \rfloor) > \log n\}, m \geq n.$$

- É fácil ver que a função A tem crescimento muito rápido. Por exemplo:

$$A(2,1) = A(1,2) = 2^2$$

$$A(2,x) = A(1, A(2, x-1)) = 2^{A(2,x-1)} = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{(x+1) vezes}}$$

Compressão de caminhos: complexidade

- A função α satisfaz $\alpha(m,n) \leq 4$ para *todos efeitos práticos*.
- Ou seja, a complexidade $O(|V| + |E|\alpha(|V| + |E|, |V|))$, embora não seja linear, se comporta como tal para valores práticos de $|V|$ e $|E|$.
- Note que $A(4,1) = A(2,16)$, ou seja, 17 potências sucessivas de 2. Então, se $m \approx n$, na expressão $\alpha(m,n)$, pode-se aproximar o valor de p para 4 em qualquer aplicação prática, já que

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{17 vezes}} > \log n,$$

para qualquer valor razoável de n.