

ALGORITMO DE KRUSKAL E CONJUNTOS DISJUNTOS

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

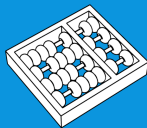
MC558 - Projeto e Análise de
Algoritmos II

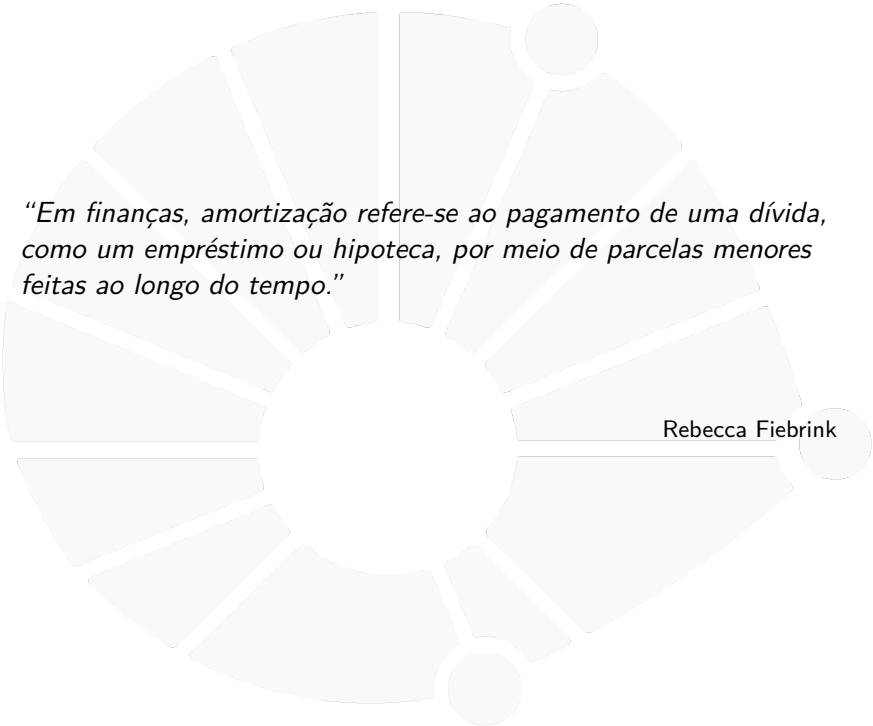
09/24

10



UNICAMP



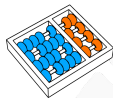


“Em finanças, amortização refere-se ao pagamento de uma dívida, como um empréstimo ou hipoteca, por meio de parcelas menores feitas ao longo do tempo.”

Rebecca Fiebrink



O ALGORITMO DE KRUSKAL



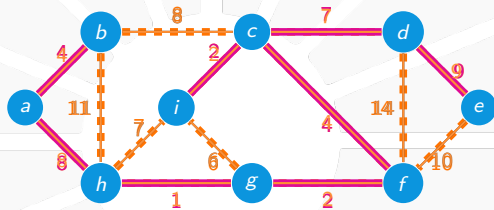
O algoritmo

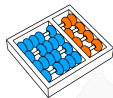
Algoritmo: AGM-KRUSKAL(G, w)

- 1 $A \leftarrow \emptyset$
 - 2 ordene as arestas em ordem não decrescente de peso
 - 3 **para cada** $(u, v) \in E[G]$ *na ordem obtida*
 - 4 **se** u e v *estão em componentes distintas de* (V, A)
 - 5 $A \leftarrow A \cup \{(u, v)\}$
 - 6 **devolva** A
-



Exemplo





O algoritmo

Algoritmo: AGM-KRUSKAL(G, w)

- 1 $A \leftarrow \emptyset$
 - 2 ordene as arestas em ordem não decrescente de peso
 - 3 **para cada** $(u, v) \in E[G]$ *na ordem obtida*
 - 4 **se** u e v *estão em componentes distintas de* (V, A)
 - 5 $A \leftarrow A \cup \{(u, v)\}$
 - 6 **devolva** A
-

Esta é uma versão preliminar do algoritmo:

- ▶ Falta detalhar a implementação da linha 4.
- ▶ Como fazer isso **EFICIENTEMENTE?**



Estrutura de dados

Como guardar a floresta sendo construída?

- ▶ Basta guardar o conjunto **A** das arestas.
- ▶ Assim, a floresta é $G_A = (\mathbf{V}, \mathbf{A})$.

Como representar as componentes de G_A ?

- ▶ Durante o algoritmo, as componentes de G_A mudam e precisamos:
 - ▶ **DETERMINAR** qual componente contém vértice **u**.
 - ▶ **FAZER A UNIÃO** das componentes que contêm **u** e **v**.

Qual estrutura realiza essas operações eficientemente?



CONJUNTOS DISJUNTOS



Conjuntos disjuntos

Queremos uma estrutura de dados que:

- ▶ Mantenha uma coleção S_1, S_2, \dots, S_k de **CONJUNTOS DISJUNTOS**.
- ▶ Permita remover ou adicionar conjuntos à tal coleção.
- ▶ Identifique cada conjunto por um **REPRESENTANTE**:
 - ▶ O representante é um elemento do próprio conjunto.
 - ▶ A escolha do representante é irrelevante.
 - ▶ O representante de um conjunto não pode mudar.



Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

1. **MAKE-SET(x)**: cria um novo conjunto $\{x\}$.
2. **UNION(x, y)**: une os conjuntos que contêm x e y .
 - ▶ Se esses conjuntos forem S_x e S_y ,
 - ▶ então adicionamos o conjunto $S_x \cup S_y$
 - ▶ e descartamos S_x e S_y da coleção.
3. **FIND-SET(x)**: devolve o representante do conjunto que contém x .



Exemplo de aplicação

Vamos determinar as componentes conexas de um grafo G

- ▶ Primeiro, vamos utilizar a estrutura de dados para conjuntos disjuntos para representar as componentes.
- ▶ Depois, vamos utilizar essa estrutura para determinar eficientemente se dois vértices estão na mesma componente.



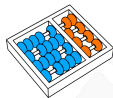
Componentes conexas

Algoritmo: CONNECTED-COMPONENTS(G)

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

Algoritmo: SAME-COMPONENT(u, v)

```
1 se FIND-SET( $u$ ) = FIND-SET( $v$ )
2   └─ devolva TRUE
3 senão
4   └─ devolva FALSE
```



Componentes conexas

A complexidade depende da implementação:

- ▶ $|V|$ chamadas a MAKE-SET.
- ▶ $2|E|$ chamadas a FIND-SET.
- ▶ Até $|V| - 1$ chamadas a UNION.



O algoritmo de Kruskal

Agora escrevemos a versão completa do algoritmo de Kruskal:

Algoritmo: AGM-KRUSKAL(G, w)

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



Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados:

- ▶ A ordenação toma tempo $O(E \log E)$.
- ▶ $|V|$ chamadas a MAKE-SET.
- ▶ $2|E|$ chamadas a FIND-SET.
- ▶ $|V| - 1$ chamadas a UNION.



Complexidade da operações realizadas

Sequência de chamadas MAKE-SET, UNION e FIND-SET:

- ▶ n chamadas a MAKE-SET.
- ▶ m chamadas no total.

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



n

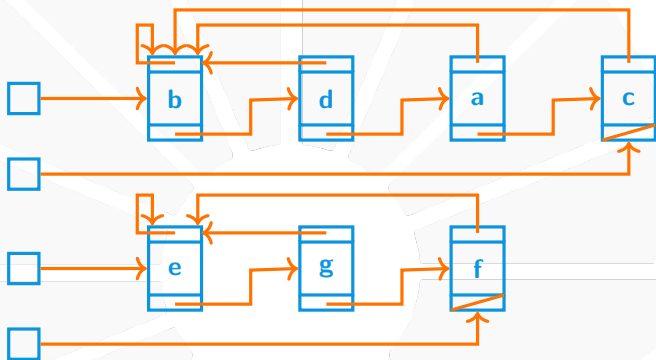


m

Queremos medir a complexidade em termos de n e m .



Representação por listas ligadas

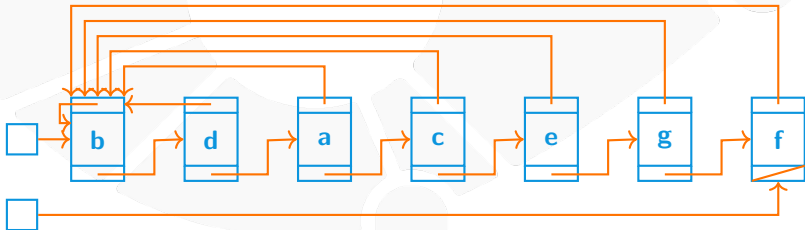


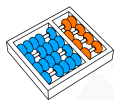
- ▶ Cada conjunto tem um representante (início da lista).
- ▶ Cada nó tem um campo que aponta para o representante.
- ▶ Guarda-se um apontador para o fim de cada lista.



Complexidade usando listas ligadas

- ▶ $\text{MAKE-SET}(x) - O(1)$.
- ▶ $\text{FIND-SET}(x) - O(1)$.
- ▶ $\text{UNION}(x, y) - O(n)$: Temos que concatenar a lista de y no final da lista de x e atualizar os apontadores para o representante.





Um exemplo de pior caso

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

- ▶ O número de chamadas a operações é $2n - 1$.
- ▶ O tempo total é $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$.
- ▶ O custo amortizado por operação é $\frac{\Theta(n^2)}{2n-1} = \Theta(n)$.



Uma heurística simples

Entendendo o pior caso

- ▶ Cada chamada a `UNION` gasta em média tempo $\Theta(n)$.
- ▶ Isso porque concatenamos a maior lista no final da menor.
- ▶ Para evitar isso, podemos concatenar a **MENOR** lista no final.
- ▶ Essa ideia é chamada de ***WEIGHTED-UNION HEURISTIC***.

Implementação:

- ▶ Basta guardar o tamanho de cada lista.
- ▶ Pode ser que uma chamada a `UNION` leve tempo $\Theta(n)$.
- ▶ Mas isso não pode acontecer sempre.



Uma heurística simples

Teorema

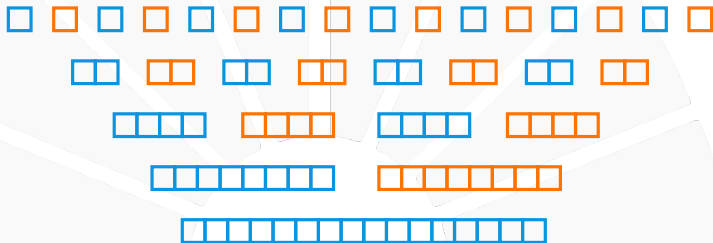
*Suponha que executamos uma sequência de m chamadas a MAKE-SET, UNION e FIND-SET. Se utilizarmos a representação por listas ligadas com a weighted-union heuristic, então o **TEMPO TOTAL** gasto será $O(m + n \log n)$.*

Demonstração:

- ▶ O tempo total das chamadas MAKE-SET e FIND-SET é $O(m)$.
- ▶ Ao atualizarmos um nó, a lista que o continha pelo menos dobra.
- ▶ Mas uma lista só pode dobrar no máximo $O(\log n)$ vezes.
- ▶ Assim, cada nó só é atualizado $O(\log n)$ vezes.
- ▶ Portanto, o tempo total com chamadas a UNION é $O(n \log n)$.



Um exemplo de pior caso



O custo total de UNION nesse exemplo é $\Theta(n \log n)$:

- ▶ Há $\Theta(\log n)$ níveis, cada um representando a coleção de conjuntos disjuntos em determinado instante.
- ▶ Entre um nível e o próximo, as listas em **laranja** são concatenadas às listas em **azul** da esquerda.
- ▶ Assim, em cada nível, $n/2$ apontadores são atualizados.



Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por:

- ▶ Ordenação, que toma tempo $O(E \log E)$.
- ▶ $|V|$ chamadas a MAKE-SET.
- ▶ $2|E|$ chamadas a FIND-SET.
- ▶ $|V| - 1$ chamadas a UNION.

O tempo total utilizando a representação por listas ligadas é:

$$O(E \log E) + O(V + E + V \log V) = O(E \log E) = O(E \log V).$$

- ▶ O tempo é dominado pela ordenação das arestas.
- ▶ Se já estiverem ordenadas, então gastamos $O(E + V \log V)$.



CONJUNTOS DISJUNTOS COM FLORESTAS DE CONJUNTOS



Outra representação

Representar uma coleção por uma floresta:

- ▶ Cada conjunto corresponde a uma árvore enraizada.
- ▶ O representante de um conjunto é a raiz.
- ▶ Tal floresta é a chamada **DISJOINT-SET FOREST**.

Veremos duas implementações:

1. Uma simples:

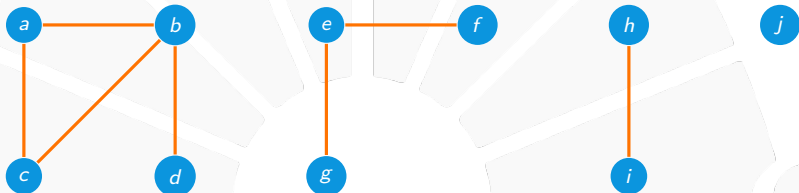
- ▶ Só altera a floresta durante **UNION**.
- ▶ O tempo não melhor que listas (assintoticamente).

2. Uma mais elaborada:

- ▶ Utiliza as heurísticas **UNION BY RANK** e **PATH COMPRESSION**.
- ▶ Também altera a floresta durante **FIND-SET**.
- ▶ É a melhor implementação de conhecida.



Exemplo de grafo

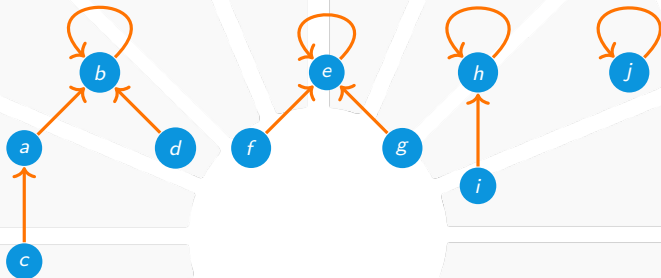


Veja o grafo:

- ▶ Considere um conjunto para cada componente.
- ▶ Como representar os conjuntos com *disjoint-set forest*?



Exemplo de representação

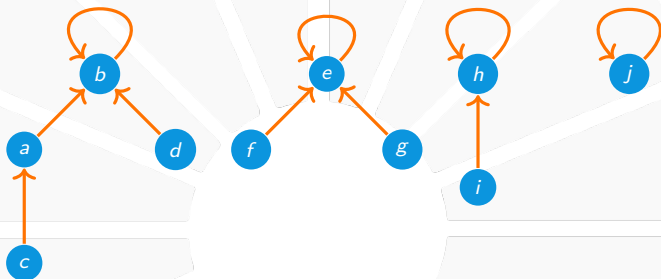


Convenções:

- ▶ Cada conjunto é uma árvore enraizada.
- ▶ Cada elemento aponta para seu pai.
- ▶ A **RAIZ** aponta para si mesma.
- ▶ A **RAIZ** é o representante do conjunto.

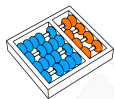


Implementação simples

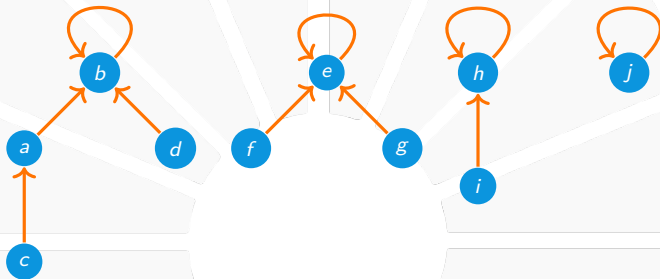


Algoritmo: MAKE-SET(x)

1 pai[x] $\leftarrow x$

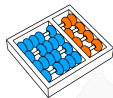


Implementação simples

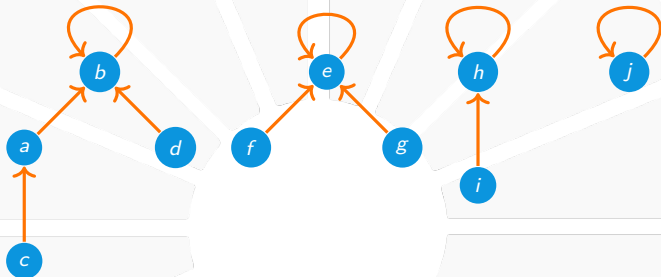


Algoritmo: FIND-SET(x)

- 1 se $x = \text{pai}[x]$
 - 2 | devolva x
 - 3 senão
 - 4 | devolva FIND-SET($\text{pai}[x]$)
-



Implementação simples



Algoritmo: UNION(x, y)

- 1 $x' \leftarrow \text{FIND-SET}(x)$
 - 2 $y' \leftarrow \text{FIND-SET}(y)$
 - 3 $\text{pai}[y'] \leftarrow x'$
-



Complexidade da implementação simples

Tempo das operações:

- ▶ $\text{MAKE-SET}(x) - O(1)$.
- ▶ $\text{FIND-SET}(x) - O(n)$.
- ▶ $\text{UNION}(x, y) - O(n)$.

Não é melhor do que a representação por listas ligadas:

- ▶ Considere uma sequência de $n - 1$ chamadas a UNION , que resulta em uma cadeia linear com n nós.
- ▶ Então, n chamadas a FIND-SET podem levar tempo total $\Theta(n^2)$.

Podemos melhorar isso usando duas heurísticas:

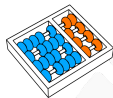
- ▶ *union by rank*.
- ▶ *path compression*.



Union by rank

Ideia emprestada da **WEIGHTED-UNION HEURISTIC**:

- ▶ Cada nó x está associado a um número $\text{rank}[x]$:
 - ▶ Pode ser a altura de x na árvore,
 - ▶ ou pode ser um número **MENOR**.
- ▶ A raiz com menor rank aponta para a raiz com maior rank .



Union by rank

Algoritmo: MAKE-SET(x)

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

Algoritmo: UNION(x, y)

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

Algoritmo: LINK(x, y)

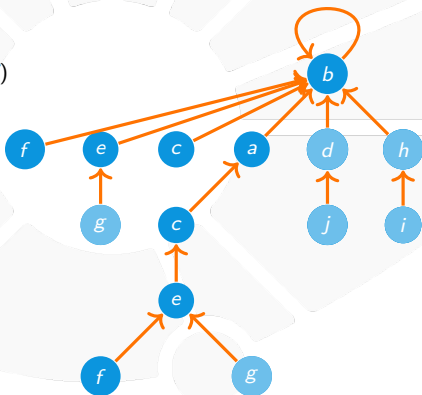
- 1 se $\text{rank}[x] > \text{rank}[y]$
 - 2 $\text{pai}[y] \leftarrow x$
 - 3 senão
 - 4 $\text{pai}[x] \leftarrow y$
 - 5 se $\text{rank}[x] = \text{rank}[y]$
 - 6 $\text{rank}[y] \leftarrow \text{rank}[y] + 1$
-

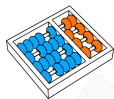


Path compression

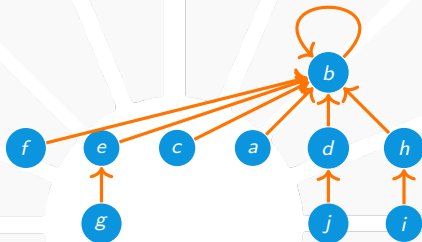
A ideia é muito simples: ao tentar determinar o representante (**RAIZ** da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.

FIND-SET(*f*)





Path compression



Algoritmo: FIND-SET(x)

- 1 se $x \neq \text{pai}[x]$
 - 2 $\lfloor \text{pai}[x] \leftarrow \text{FIND-SET}(\text{pai}[x])$
 - 3 devolva $\text{pai}[x]$
-



Análise das heurísticas

Analisando separadamente:

1. Se utilizarmos somente **UNION BY RANK**:
 - ▶ Suponha que realizamos m chamadas no total.
 - ▶ O tempo total será $O(m \log n)$. Por quê?
2. Se utilizarmos apenas **PATH COMPRESSION**:
 - ▶ Suponha que realizamos f chamadas a FIND-SET.
 - ▶ Mostra-se que o tempo total é $O(n + f \cdot (1 + \log_{2+f/n} n))$.

Combinando as duas duas heurísticas:

- ▶ Mostra-se que o tempo total é $O(m\alpha(n))$.
- ▶ Onde, $\alpha(n)$ é uma função que cresce **MUITO MUITO** lentamente.



Complexidade com as duas heurísticas

Teorema (Tarjan)

*Uma sequência de m operações MAKE-SET, UNION e FIND-SET pode ser executada com **DISJOINT-SET FOREST** com union by rank e path compression em tempo $O(m\alpha(n))$ no pior caso.*

Não vamos demonstrar este teorema:

- ▶ Ele implica que o custo amortizado por chamada é $\alpha(n)$.
- ▶ O valor de $\alpha(n)$ cresce arbitrariamente com n .
- ▶ Contudo, o crescimento é num ritmo realmente devagar.
- ▶ Uma demonstração está em CLRS.

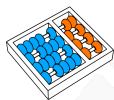


Estimando o tempo amortizado

Quão pequeno é o custo de cada operação?

$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq 16^{512} \end{cases}$$

- ▶ Observe que 16^{512} é muito muito maior que 10^{80} , o número estimado de átomos do universo!
- ▶ Isso significa que na prática o custo amortizado de cada chamada é limitado pela **CONSTANTE**, digamos 4.
- ▶ Vamos utilizar essa estrutura no algoritmo de Kruskal.



O algoritmo de Kruskal (de novo)

Complexidade:

- ▶ Ordenação das arestas: **tempo $O(E \log E)$** .
- ▶ $O(E)$ chamadas a MAKE-SET, FIND-SET e UNION: **tempo $O(E\alpha(V))$** .
- ▶ O tempo total é dominado pelo tempo de ordenação.
- ▶ Contudo, as demais operações têm tempo praticamente linear!

ALGORITMO DE KRUSKAL E CONJUNTOS DISJUNTOS

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

MC558 - Projeto e Análise de
Algoritmos II

09/24

10



UNICAMP

