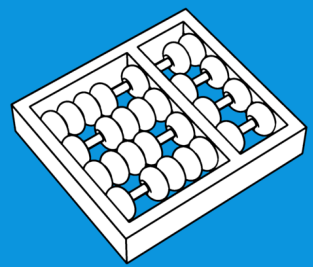
Árvore geradora mínima

MC558 - Projeto e Análise de Algoritmos II

5

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

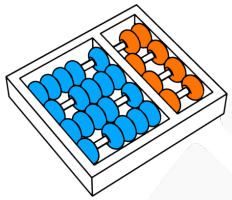
UNICAMP



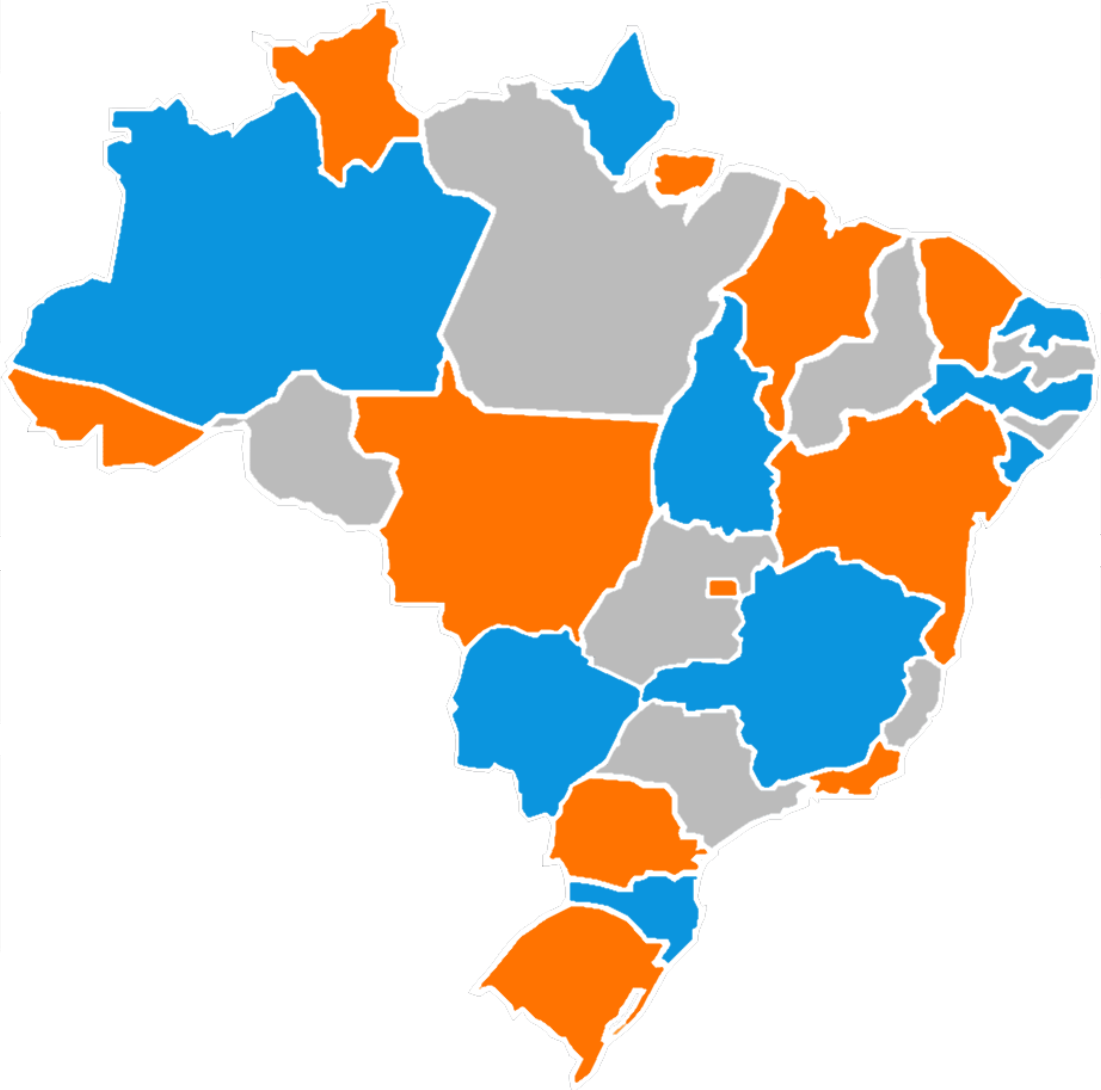
"Minimum Spanning Trees ... or how to bring the world together on a budget."

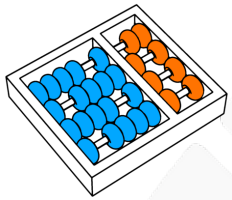
Seth James Nielson

Introdução

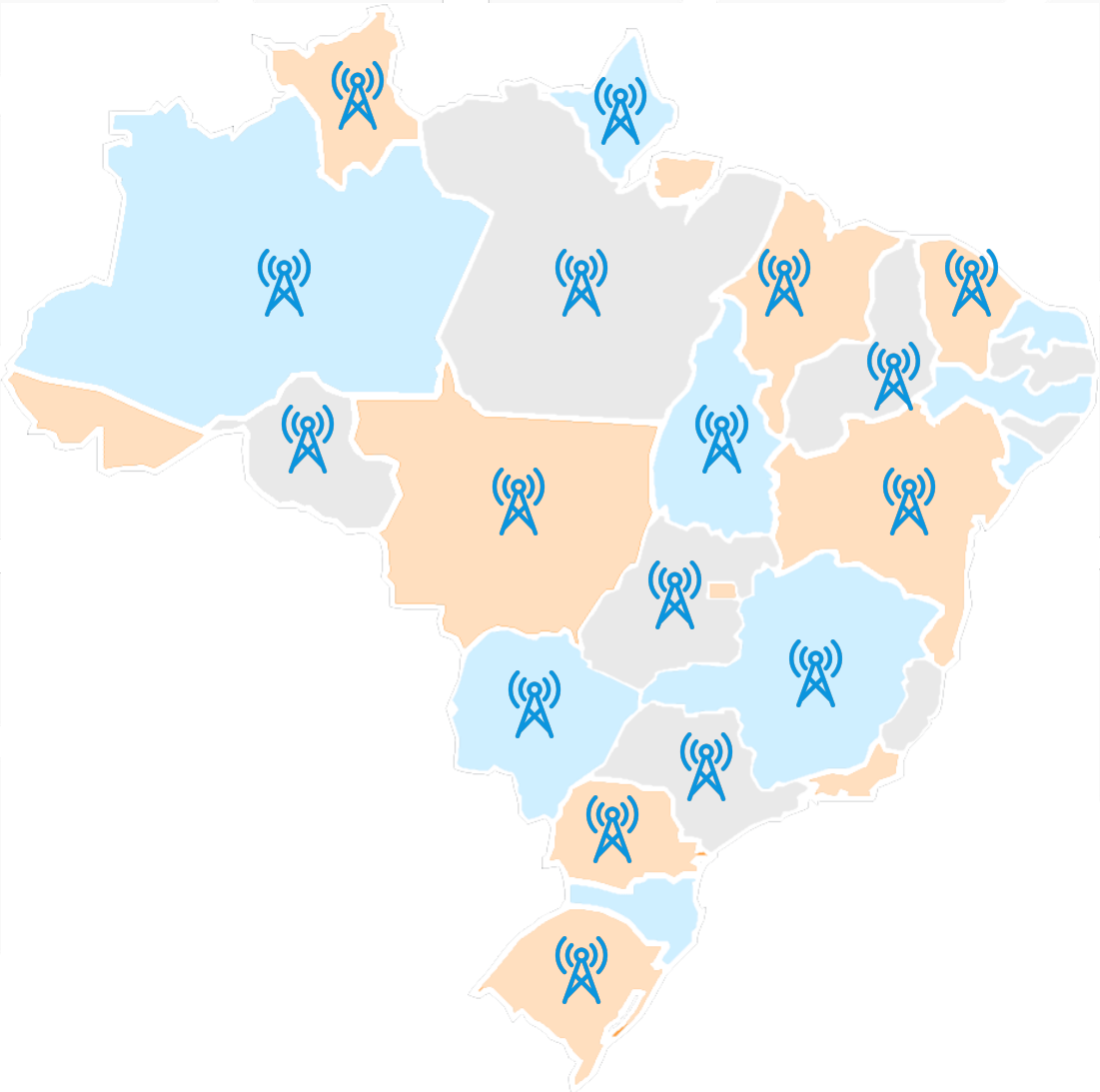


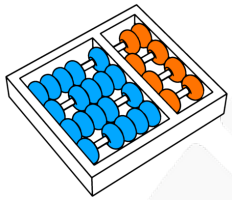
Motivação



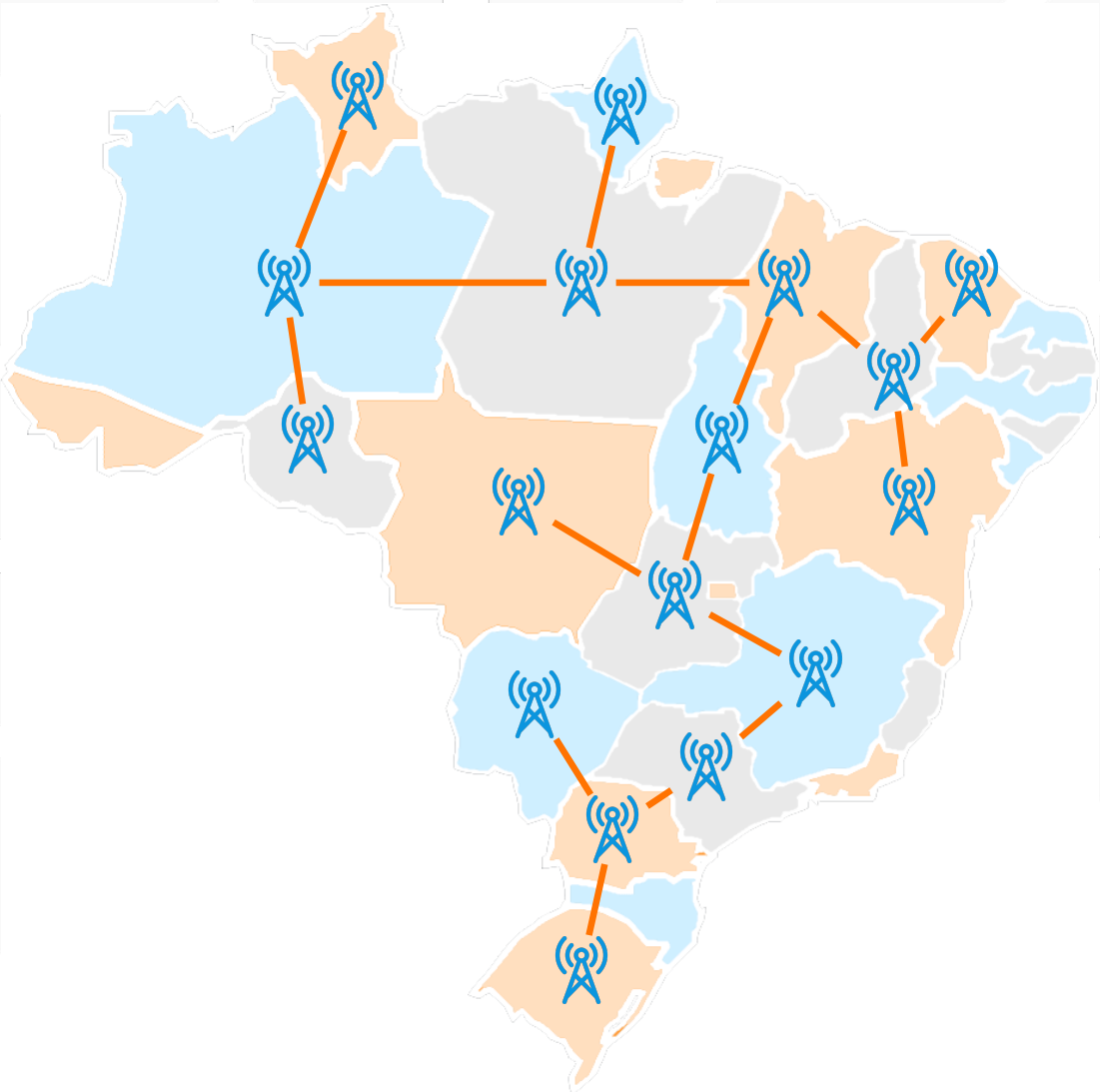


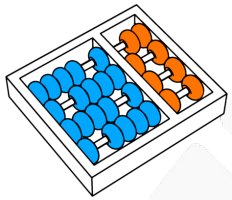
Motivação





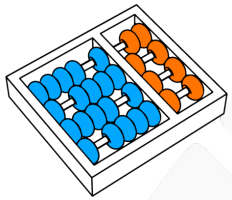
Motivação





Motivação

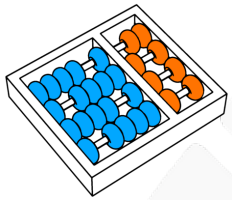
Na situação anterior:



Motivação

Na situação anterior:

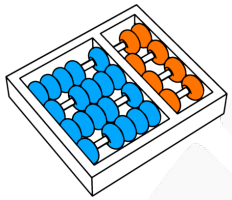
- Temos um conjunto de torres.



Motivação

Na situação anterior:

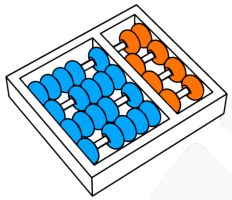
- Temos um conjunto de torres.
- Para conectar duas torres usamos um cabeamento.



Motivação

Na situação anterior:

- Temos um conjunto de torres.
- Para conectar duas torres usamos um cabeamento.
- Queremos que todas elas estejam interconectadas.

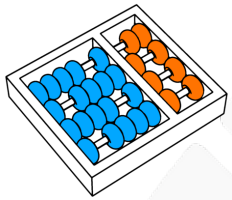


Motivação

Na situação anterior:

- Temos um conjunto de torres.
- Para conectar duas torres usamos um cabeamento.
- Queremos que todas elas estejam interconectadas.

Como **MINIMIZAR** o comprimento do cabeamento?



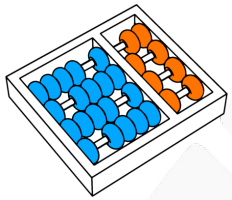
Motivação

Na situação anterior:

- Temos um conjunto de torres.
- Para conectar duas torres usamos um cabeamento.
- Queremos que todas elas estejam interconectadas.

Como **MINIMIZAR** o comprimento do cabeamento?

- Este é um problema de otimização.



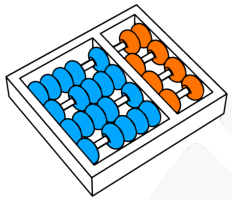
Motivação

Na situação anterior:

- Temos um conjunto de torres.
- Para conectar duas torres usamos um cabeamento.
- Queremos que todas elas estejam interconectadas.

Como **MINIMIZAR** o comprimento do cabeamento?

- Este é um problema de otimização.
- A entrada pode ser modelada como um grafo



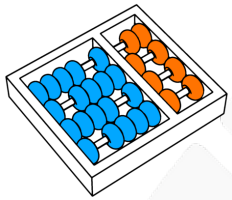
Árvore geradora mínima

Problema (Árvore geradora mínima (AGM))

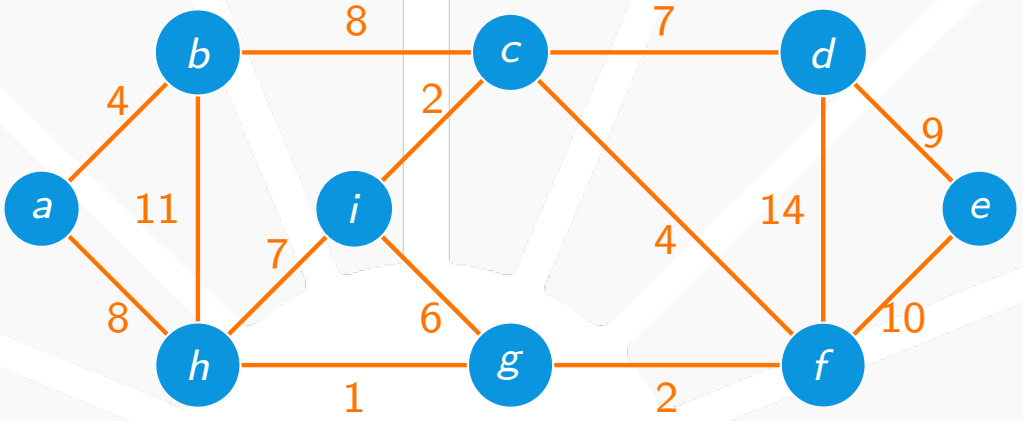
Entrada: *Um grafo conexo* $G = (V, E)$ *com peso* $w(u,v) \geq 0$ *para cada aresta* (u,v).

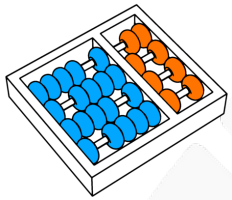
Solução: *Um subgrafo gerador conexo* T *de* G.

Objetivo: ***MINIMIZAR***: $w(T) = \sum_{(u,v) \in E[T]} w(u,v)$.

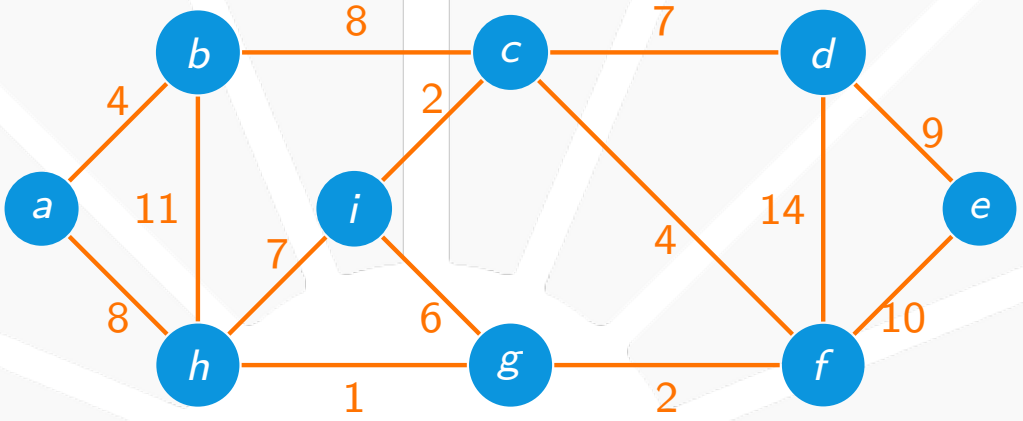


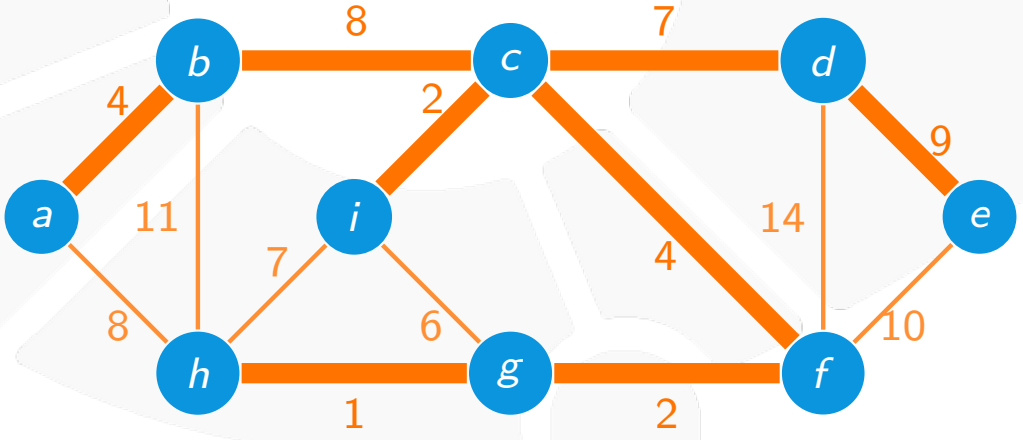
Exemplo

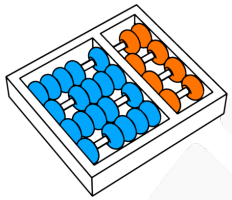




Exemplo

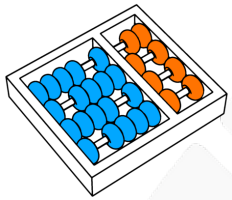

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
14
7
4
8
6
10
1
2


b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
14
7
4
8
6
10
1
2



Refletindo um pouco

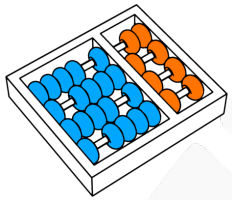
Observações:



Refletindo um pouco

Observações:

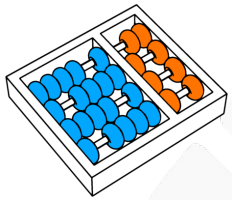
- Se o grafo fosse desconexo, então não haveria solução.



Refletindo um pouco

Observações:

- Se o grafo fosse desconexo, então não haveria solução.
- Supomos que **NÃO** há arestas de peso negativo.

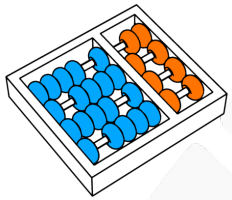


Refletindo um pouco

Observações:

- Se o grafo fosse desconexo, então não haveria solução.
- Supomos que **NÃO** há arestas de peso negativo.

Algumas perguntas:



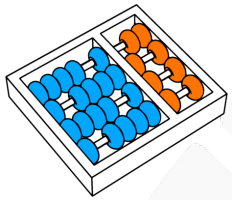
Refletindo um pouco

Observações:

- Se o grafo fosse desconexo, então não haveria solução.
- Supomos que **NÃO** há arestas de peso negativo.

Algumas perguntas:

- Por que dizemos que uma solução ótima é uma **ÁRVORE**?



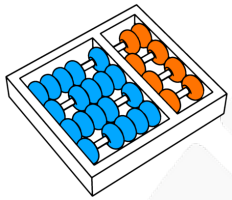
Refletindo um pouco

Observações:

- Se o grafo fosse desconexo, então não haveria solução.
- Supomos que **NÃO** há arestas de peso negativo.

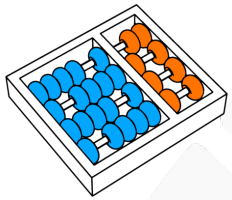
Algumas perguntas:

- Por que dizemos que uma solução ótima é uma **ÁRVORE**?
- Vale se houvesse arestas de peso negativo na entrada?



Algoritmos estudados

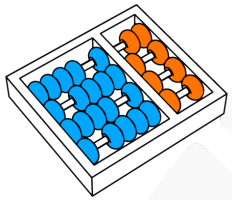
Veremos dois algoritmos **GULOSOS**:



Algoritmos estudados

Veremos dois algoritmos **GULOSOS**:

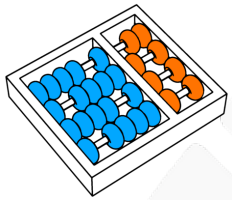
1. Algoritmo de Prim.



Algoritmos estudados

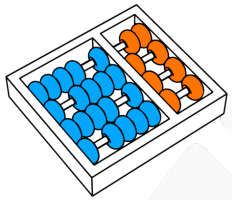
Veremos dois algoritmos **GULOSOS**:

1. Algoritmo de Prim.
2. Algoritmo de Kruskal.



Esquema dos algoritmos

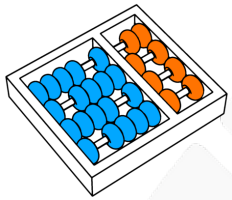
Ideia:



Esquema dos algoritmos

Ideia:

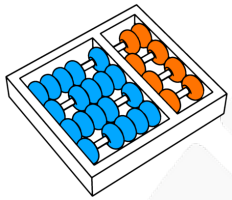
- Construiremos uma árvore incrementalmente.



Esquema dos algoritmos

Ideia:

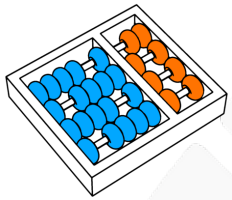
- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.



Esquema dos algoritmos

Ideia:

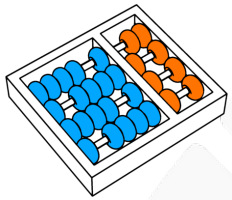
- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.



Esquema dos algoritmos

Ideia:

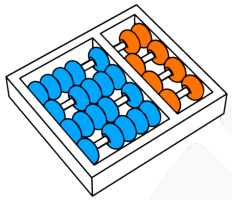
- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.
- Mantemos essa invariante em cada iteração:



Esquema dos algoritmos

Ideia:

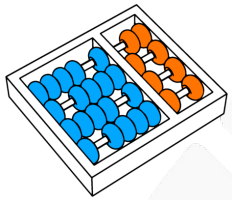
- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.
- Mantemos essa invariante em cada iteração:
  1. No início da iteração, **A** satisfaz a invariante.



Esquema dos algoritmos

Ideia:

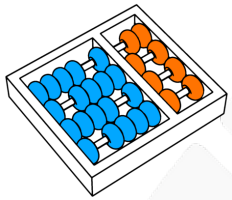
- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.
- Mantemos essa invariante em cada iteração:
  1. No início da iteração, **A** satisfaz a invariante.
  2. Selecionamos **(u,v)** tal que $\mathbf{A} \cup \{\mathbf{(u,v)}\}$ também a satisfaça.



Esquema dos algoritmos

Ideia:

- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.
- Mantemos essa invariante em cada iteração:
  1. No início da iteração, **A** satisfaz a invariante.
  2. Selecionamos **(u,v)** tal que $\mathbf{A} \cup \{\mathbf{(u,v)}\}$ também a satisfaça.
  3. Adicionamos **(u,v)** ao conjunto **A**.

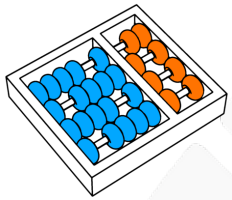


Esquema dos algoritmos

Ideia:

- Construiremos uma árvore incrementalmente.
- Denotamos por **A** um conjunto de arestas da árvore.
- Garantimos que **A** seja um subconjunto de uma AGM.
- Mantemos essa invariante em cada iteração:
  1. No início da iteração, **A** satisfaz a invariante.
  2. Selecionamos **(u,v)** tal que $\mathbf{A} \cup \{\mathbf{(u,v)}\}$ também a satisfaça.
  3. Adicionamos **(u,v)** ao conjunto **A**.

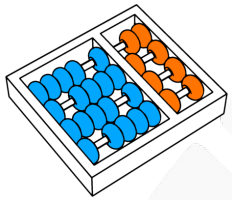
Dizemos que uma tal **(u,v)** é uma **ARESTA SEGURA**.



Algoritmo genérico

Algoritmo 1: AGM-GENERICO(G, w)

```
1 A ← ∅
2 enquanto A não é uma árvore geradora
3 |   encontre uma aresta segura (u, v) para A
4 |   A ← A ∪ {(u, v)}
5 devolva A
```

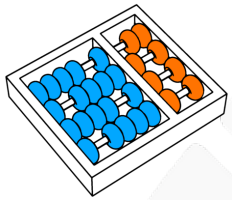


Algoritmo genérico

Algoritmo 2: AGM-GENERICO(G, w)

```
1 A ← ∅
2 enquanto A não é uma árvore geradora
3 |   encontre uma aresta segura (u, v) para A
4 |   A ← A ∪ {(u, v)}
5 devolva A
```

O algoritmo está correto:



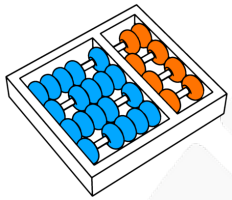
Algoritmo genérico

Algoritmo 3: AGM-GENERICO(G, w)

```
1 A ← ∅
2 enquanto A não é uma árvore geradora
3 |   encontre uma aresta segura (u, v) para A
4 |   A ← A ∪ {(u, v)}
5 devolva A
```

O algoritmo está correto:

- O algoritmo devolve uma árvore geradora **A**.



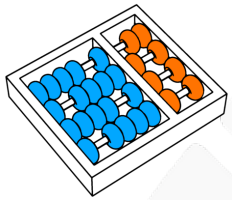
Algoritmo genérico

Algoritmo 4: AGM-GENERICO(G, w)

```
1 A ← ∅
2 enquanto A não é uma árvore geradora
3 |   encontre uma aresta segura (u, v) para A
4 |   A ← A ∪ {(u, v)}
5 devolva A
```

O algoritmo está correto:

- O algoritmo devolve uma árvore geradora **A**.
- **A** é subgrafo de alguma AGM.



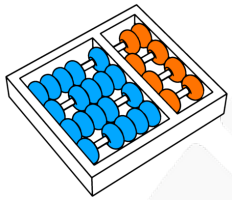
Algoritmo genérico

Algoritmo 5: AGM-GENERICO(G, w)

```
1 A ← ∅
2 enquanto A não é uma árvore geradora
3 |   encontre uma aresta segura (u, v) para A
4 |   A ← A ∪ {(u, v)}
5 devolva A
```

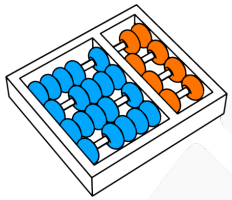
O algoritmo está correto:

- O algoritmo devolve uma árvore geradora **A**.
- **A** é subgrafo de alguma AGM.
- Logo, **A** é também mínima.



Algoritmo genérico

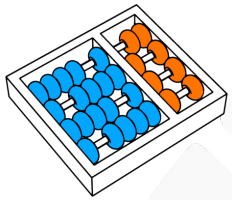
O algoritmo está bem definido?



Algoritmo genérico

O algoritmo está bem definido?

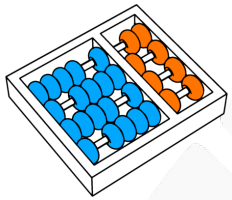
- Se a iteração executa, então **A** não é árvore geradora.



Algoritmo genérico

O algoritmo está bem definido?

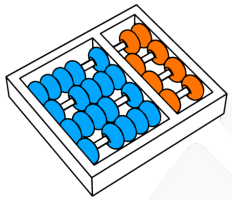
- Se a iteração executa, então **A** não é árvore geradora.
- Logo, **A** não contém todas arestas de alguma AGM T.



Algoritmo genérico

O algoritmo está bem definido?

- Se a iteração executa, então **A** não é árvore geradora.
- Logo, **A** não contém todas arestas de alguma AGM T.
- Assim qualquer aresta de **E[T]** $\setminus$ **A** é segura.

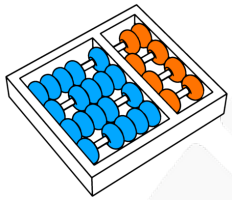


Algoritmo genérico

O algoritmo está bem definido?

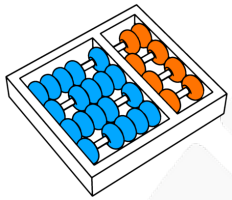
- Se a iteração executa, então **A** não é árvore geradora.
- Logo, **A** não contém todas arestas de alguma AGM T.
- Assim qualquer aresta de **E[T]** $\setminus$ **A** é segura.

Os algoritmos que veremos diferem em como encontrar uma **ARESTA SEGURA**.



Como encontrar arestas seguras?

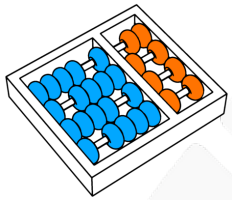
Considere um grafo $G = (\mathbf{V}, \mathbf{E})$ e seja $\mathbf{S} \subset \mathbf{V}$.



Como encontrar arestas seguras?

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

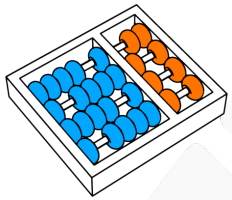
- Denote por $\delta(\mathbf{S})$ o conjunto de arestas de G com um extremo em $\mathbf{S}$ e outro em $\mathbf{V} \setminus \mathbf{S}$.



Como encontrar arestas seguras?

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

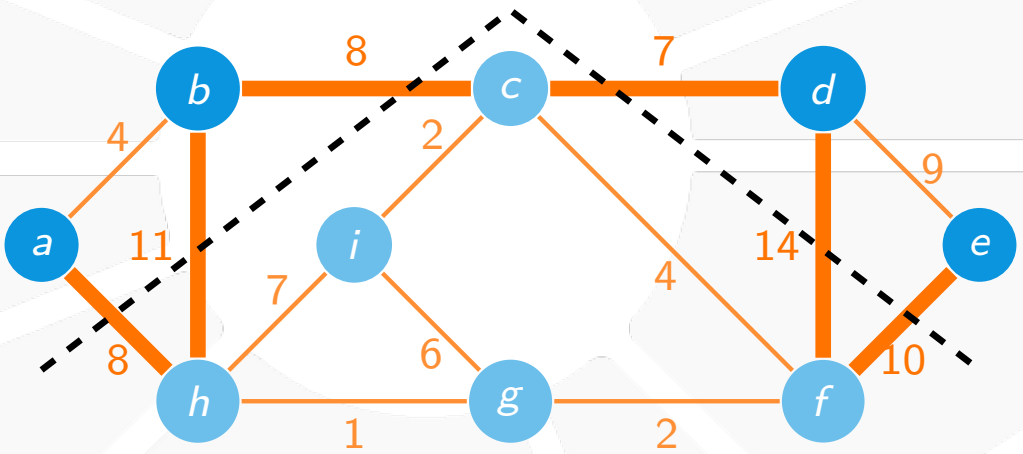
- Denote por $\delta(\mathbf{S})$ o conjunto de arestas de G com um extremo em $\mathbf{S}$ e outro em $\mathbf{V} \setminus \mathbf{S}$.
- Lembre-se de que um tal conjunto é chamado de **CORTE**.

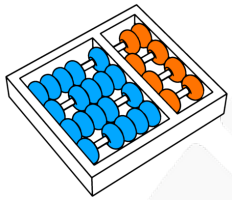


Como encontrar arestas seguras?

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

- Denote por $\delta(\mathbf{S})$ o conjunto de arestas de G com um extremo em $\mathbf{S}$ e outro em $\mathbf{V} \setminus \mathbf{S}$.
- Lembre-se de que um tal conjunto é chamado de **CORTE**.

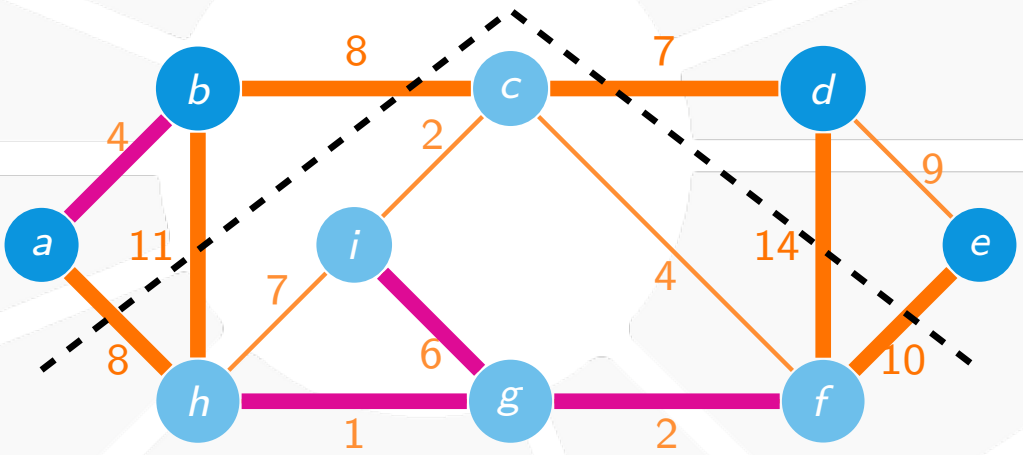




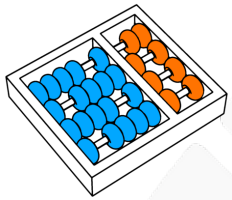
Como encontrar arestas seguras?

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

- Denote por $\delta(\mathbf{S})$ o conjunto de arestas de G com um extremo em $\mathbf{S}$ e outro em $\mathbf{V} \setminus \mathbf{S}$.
- Lembre-se de que um tal conjunto é chamado de **CORTE**.

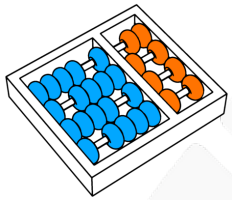


Um corte $\delta(\mathbf{S})$ **RESPEITA** um conjunto $\mathbf{A}$ de arestas se não contiver nenhuma aresta de $\mathbf{A}$.



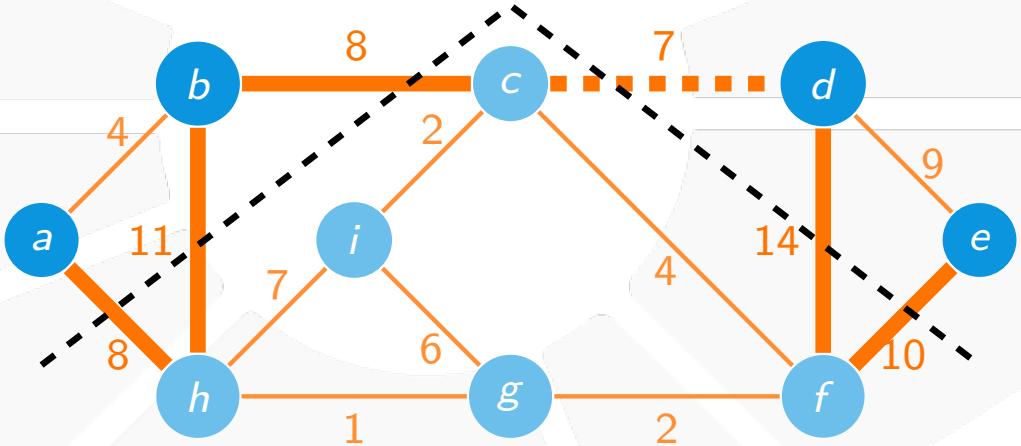
Arestas leves

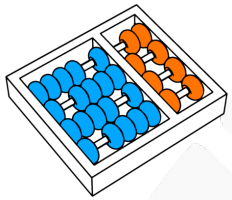
Uma aresta de um corte $\delta(\mathbf{S})$ é **LEVE** se tem o menor peso entre as arestas do corte.



Arestas leves

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

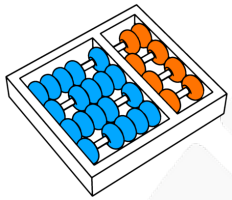




Arestas leves

Teorema

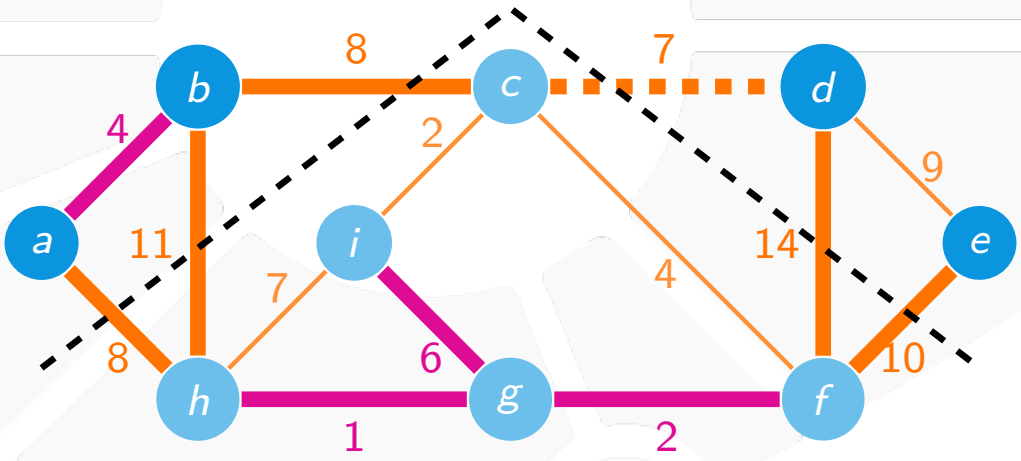
*Seja (G, w) um grafo com pesos nas arestas e suponha que **A** é um subconjunto de arestas de uma AGM de G. Se $\delta(\mathbf{S})$ é um corte que respeita **A** e **(u,v)** é uma aresta leve desse corte, então **(u,v)** é uma **ARESTA SEGURA**.*

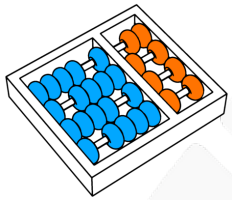


Arestas leves

Teorema

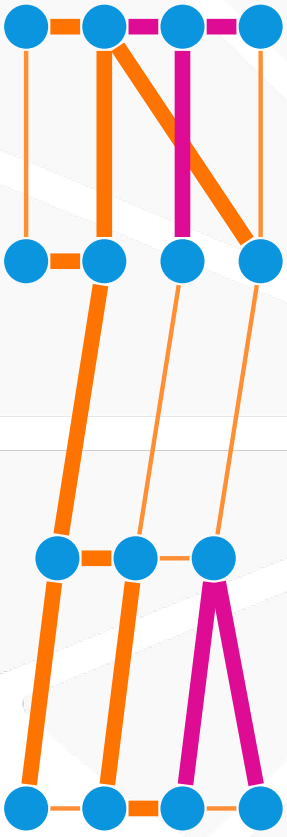
*Seja (G, w) um grafo com pesos nas arestas e suponha que **A** é um subconjunto de arestas de uma AGM de G. Se $\delta(\mathbf{S})$ é um corte que respeita **A** e **(u,v)** é uma aresta leve desse corte, então **(u,v)** é uma **ARESTA SEGURA**.*

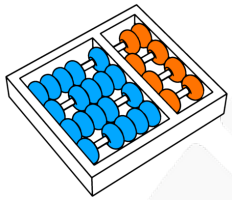




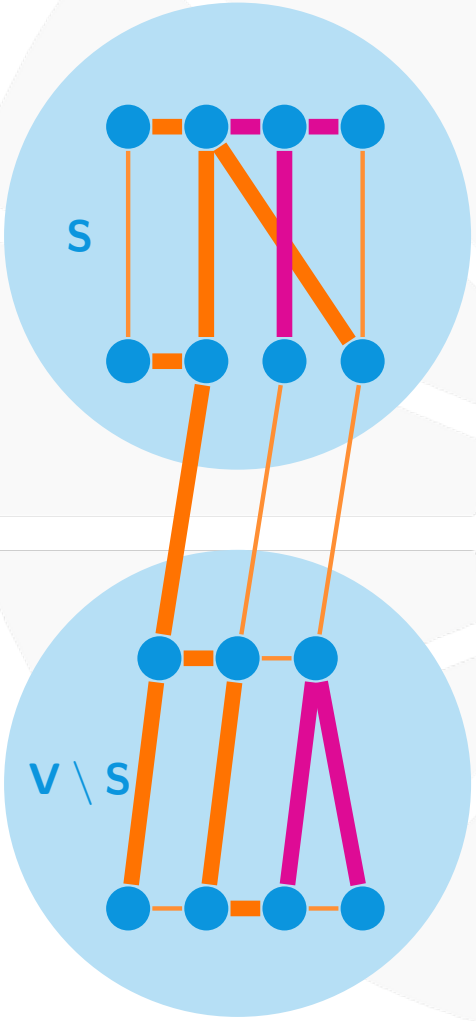
Demonstração do teorema

Seja **T** uma AGM que contém **A**:



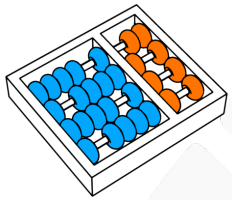


Demonstração do teorema

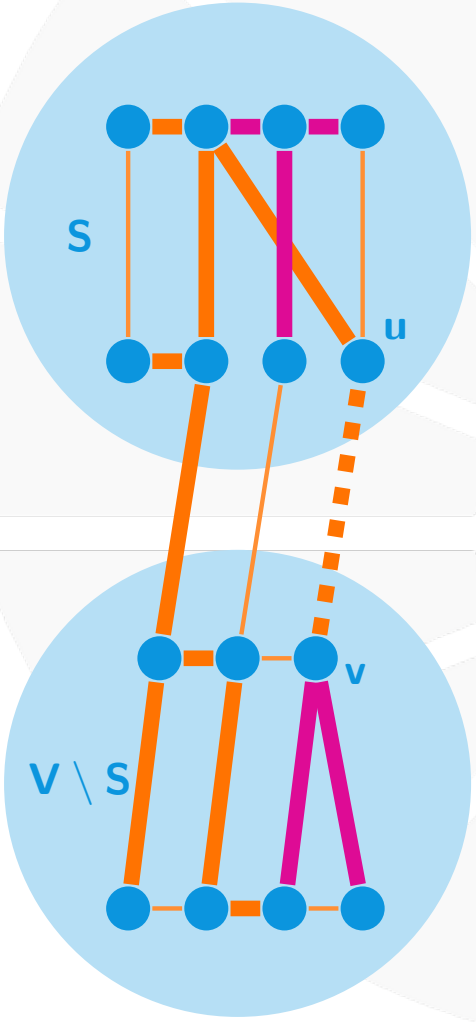


Seja **T** uma AGM que contém **A**:

- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.

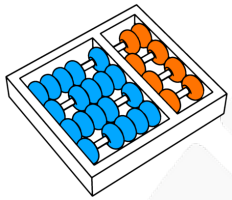


Demonstração do teorema

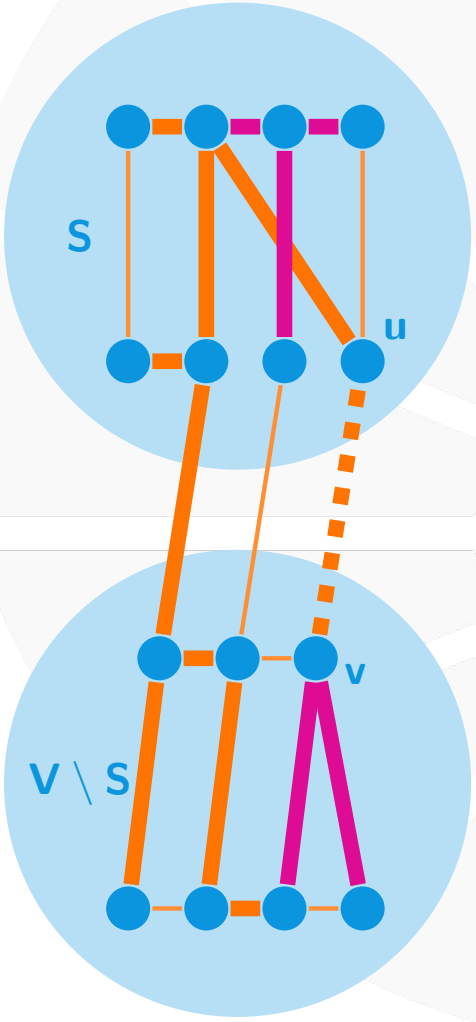


Seja **T** uma AGM que contém **A**:

- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.
- Seja **(u,v)** uma **ARESTA LEVE** deste corte.

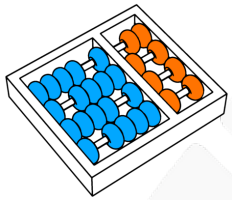


Demonstração do teorema

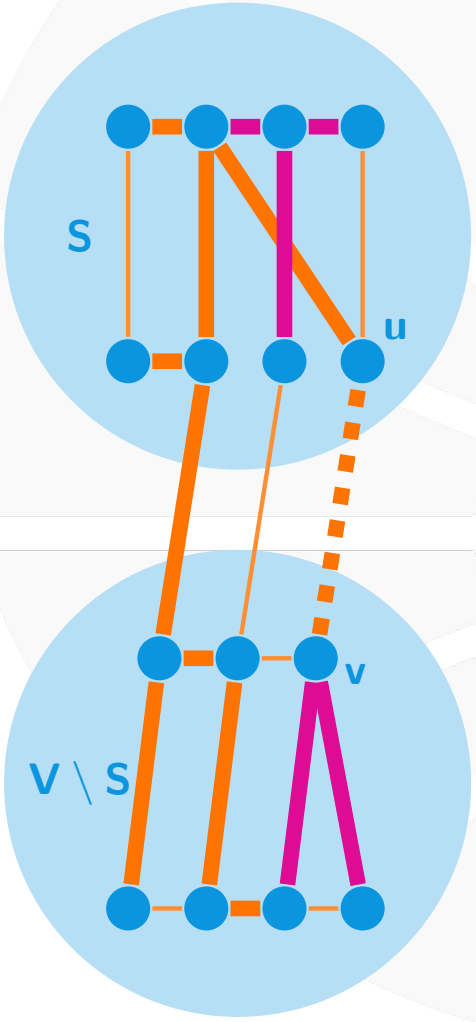


Seja **T** uma AGM que contém **A**:

- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.
- Seja **(u,v)** uma **ARESTA LEVE** deste corte.
- Se **(u,v)** estiver em T, então não há nada a mostrar.

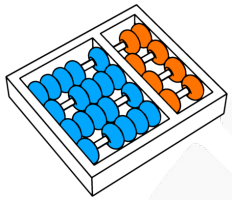


Demonstração do teorema

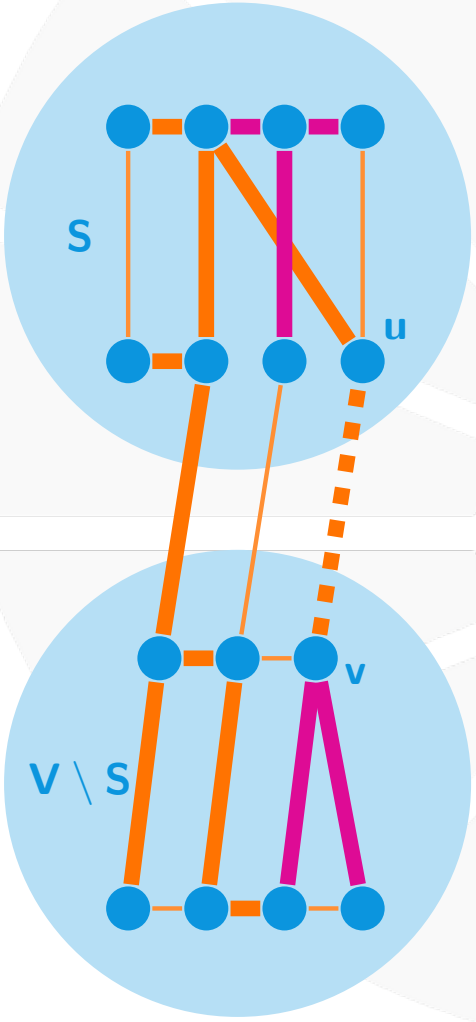


Seja **T** uma AGM que contém **A**:

- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.
- Seja **(u,v)** uma **ARESTA LEVE** deste corte.
- Se **(u,v)** estiver em T, então não há nada a mostrar.
- Se **(u,v)** **NÃO** for uma aresta de T:

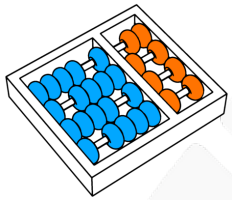


Demonstração do teorema

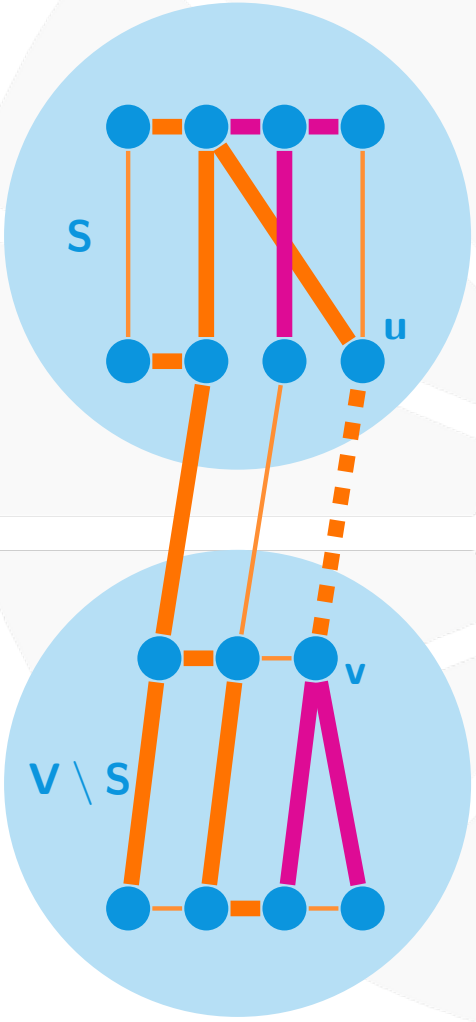


Seja **T** uma AGM que contém **A**:

- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.
- Seja **(u,v)** uma **ARESTA LEVE** deste corte.
- Se **(u,v)** estiver em T, então não há nada a mostrar.
- Se **(u,v)** **NÃO** for uma aresta de T:
  - Construiremos uma AGM T' que contém $\mathbf{A} \cup \{\mathbf{(u,v)}\}$,

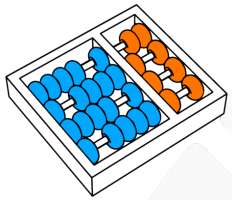


Demonstração do teorema



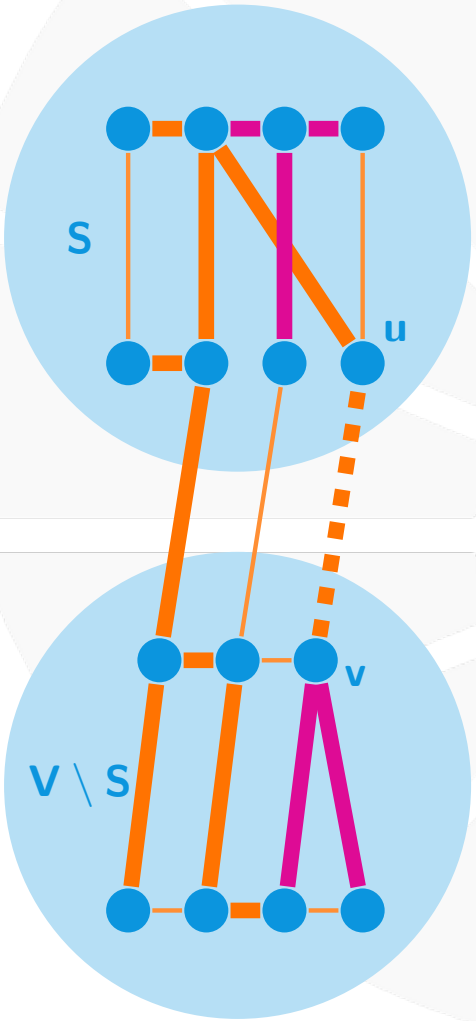
Seja **T** uma AGM que contém **A**:

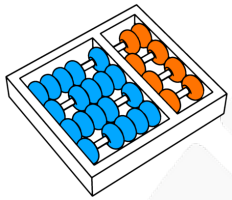
- Tome $\delta(\mathbf{S})$ um corte que respeita **A**.
- Seja **(u,v)** uma **ARESTA LEVE** deste corte.
- Se **(u,v)** estiver em T, então não há nada a mostrar.
- Se **(u,v)** **NÃO** for uma aresta de T:
  - Construiremos uma AGM T' que contém $\mathbf{A} \cup \{\mathbf{(u,v)}\}$,
  - concluindo que **(u,v)** é **SEGURA**.



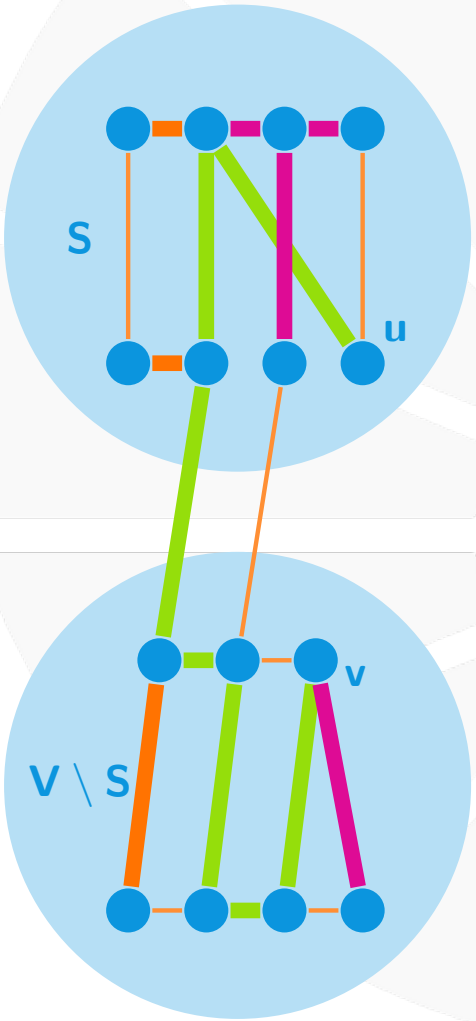
Demonstração do teorema

Suponha que **(u,v)** **NÃO** for uma aresta de T:



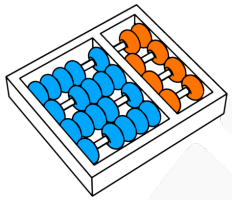


Demonstração do teorema

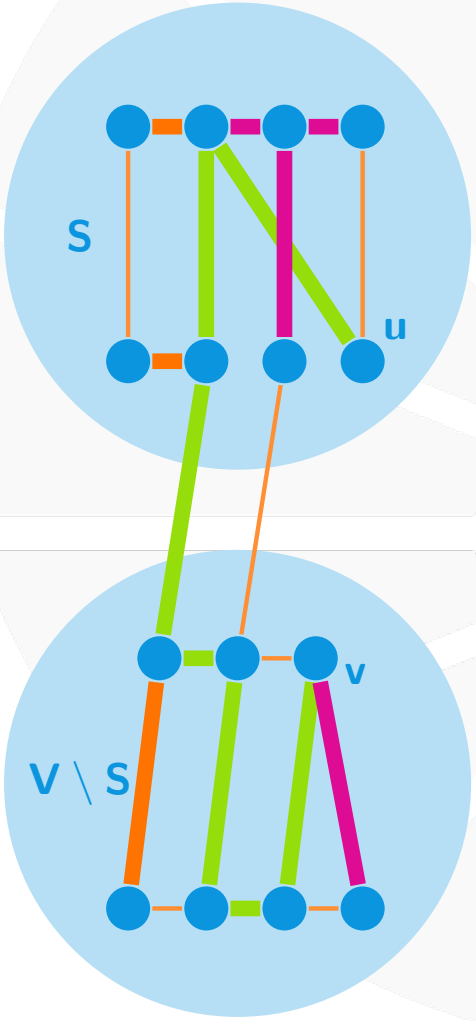


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.

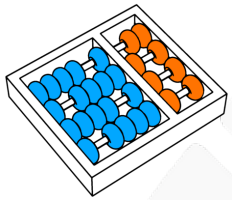


Demonstração do teorema

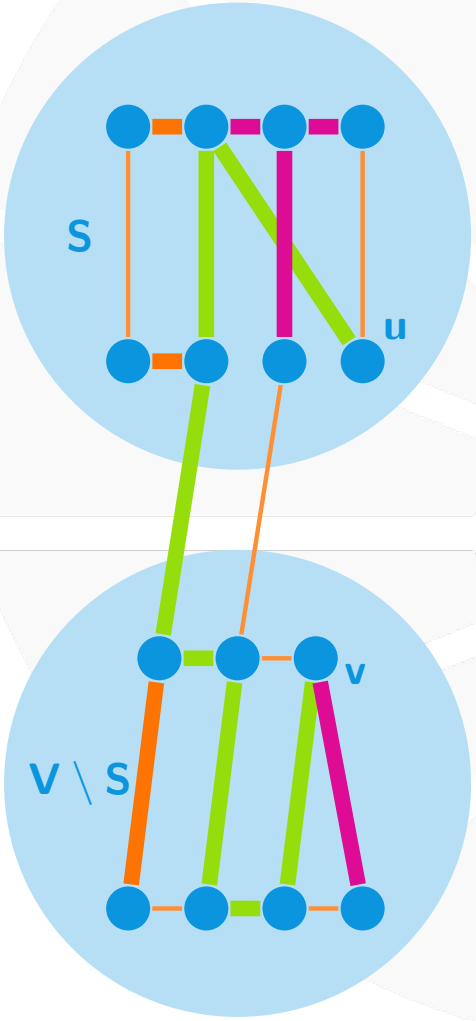


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.

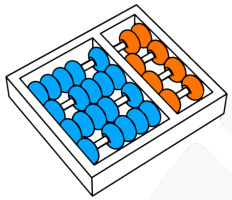


Demonstração do teorema

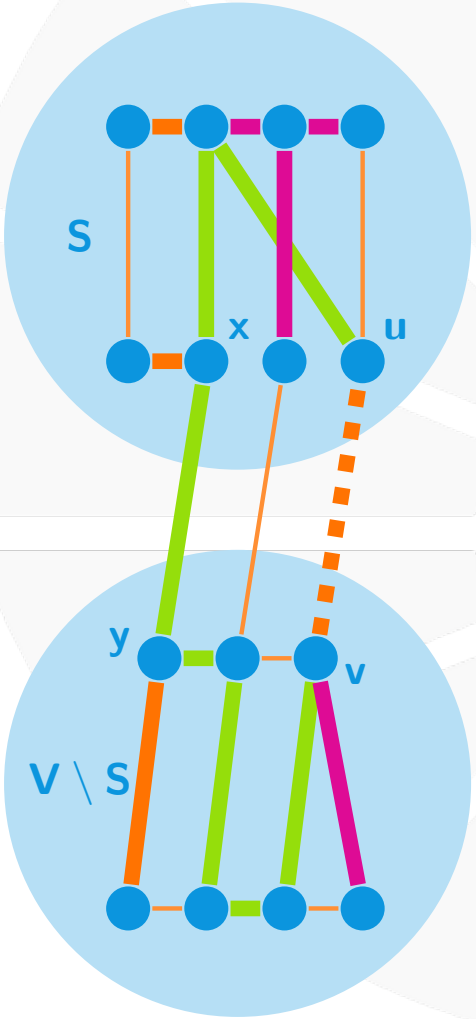


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.

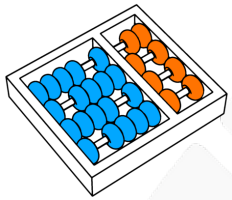


Demonstração do teorema

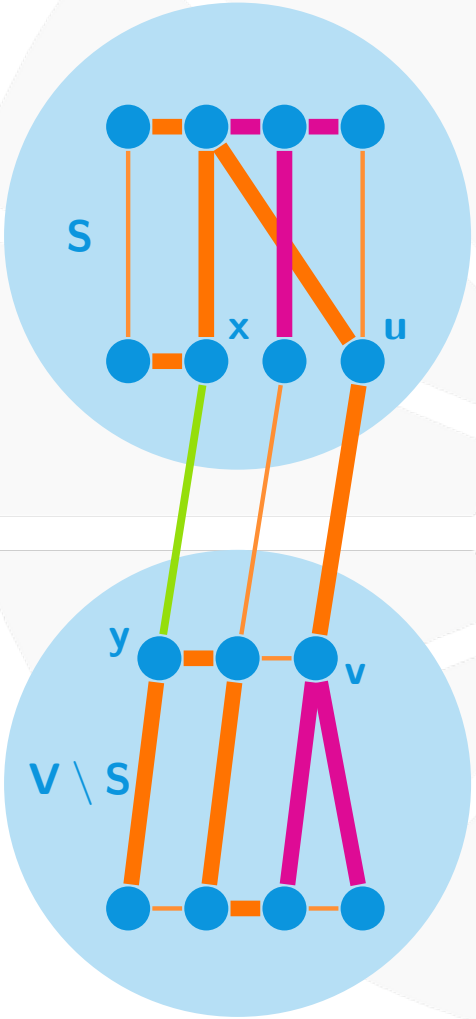


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que **(x,y)** $\notin$ **A** pois o corte respeita **A**).

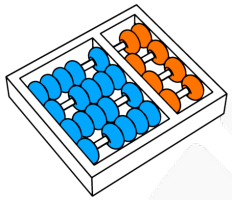


Demonstração do teorema

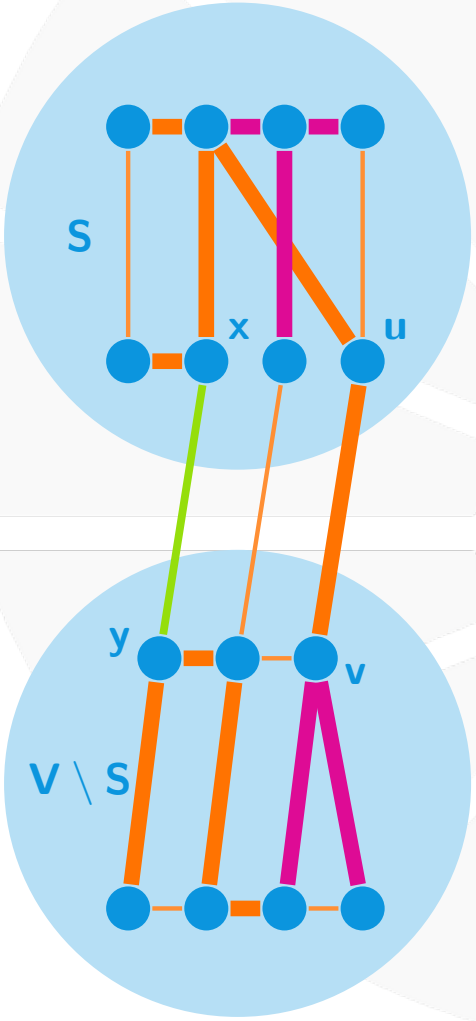


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que **(x,y)** $\notin$ **A** pois o corte respeita **A**).
- Defina $T' = T - \mathbf{(x,y)} + \mathbf{(u,v)}$.

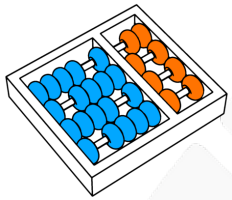


Demonstração do teorema

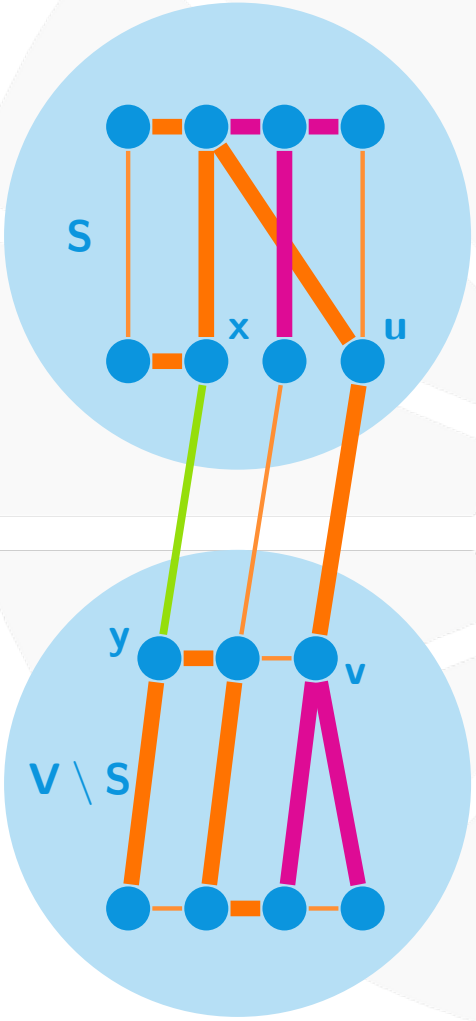


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que **(x,y)** $\notin$ **A** pois o corte respeita **A**).
- Defina $T' = T - \mathbf{(x,y)} + \mathbf{(u,v)}$.
- Observe que T' é uma árvore geradora.

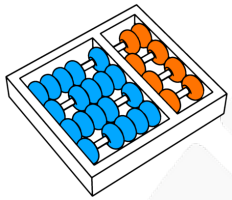


Demonstração do teorema

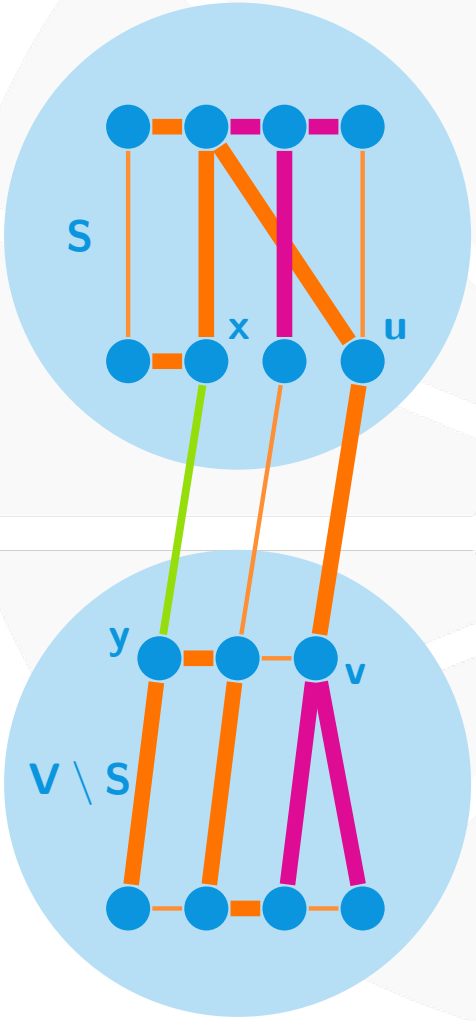


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que **(x,y)** $\notin$ **A** pois o corte respeita **A**).
- Defina $T' = T - \mathbf{(x,y)} + \mathbf{(u,v)}$.
- Observe que T' é uma árvore geradora.
- Como **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{S})$, temos que $w(\mathbf{u,v}) \leq w(\mathbf{x,y})$.

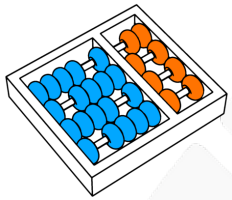


Demonstração do teorema

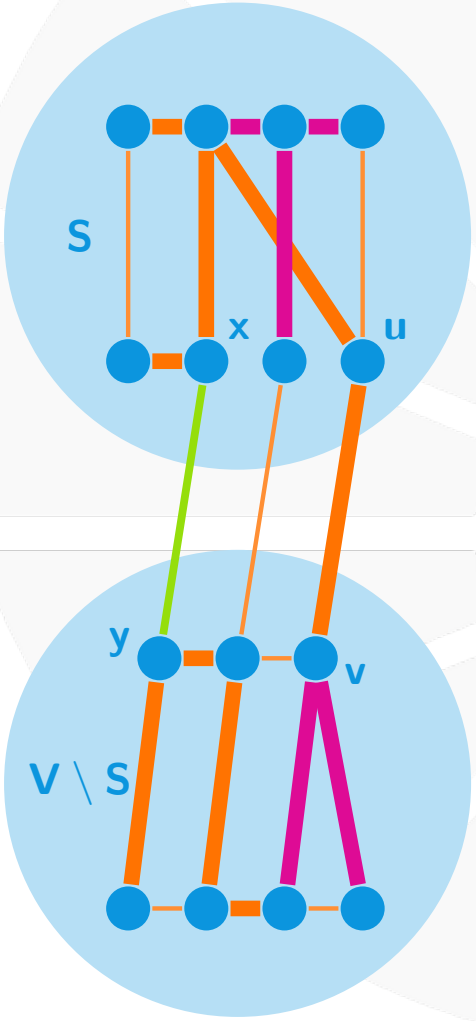


Suponha que **(u,v) NÃO** for uma aresta de T:

- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que $\mathbf{(x,y)} \notin \mathbf{A}$ pois o corte respeita **A**).
- Defina $T' = T - \mathbf{(x,y)} + \mathbf{(u,v)}$.
- Observe que T' é uma árvore geradora.
- Como **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{S})$, temos que $w(\mathbf{u,v}) \leq w(\mathbf{x,y})$.
- Logo,
  $w(T') = w(T) - w(\mathbf{x,y}) + w(\mathbf{u,v}) \leq w(T)$.

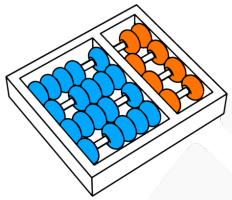


Demonstração do teorema



Suponha que **(u,v) NÃO** for uma aresta de T:

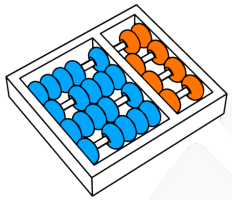
- Existe um único caminho **P** de **u** a **v** em T.
- **u** a **v** estão em lados opostos do corte $\delta(\mathbf{S})$.
- Então, alguma aresta de **P** pertence ao corte.
- Seja **(x,y)** uma tal aresta (note que **(x,y)** $\notin$ **A** pois o corte respeita **A**).
- Defina $T' = T - \mathbf{(x,y)} + \mathbf{(u,v)}$.
- Observe que T' é uma árvore geradora.
- Como **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{S})$, temos que $w(\mathbf{u,v}) \leq w(\mathbf{x,y})$.
- Logo,
  $w(T') = w(T) - w(\mathbf{x,y}) + w(\mathbf{u,v}) \leq w(T)$.
- Portanto, T' é uma **AGM**.



Consequência para os algoritmos

Corolário

Seja (G, w) um grafo com pesos nas arestas e suponha que ***A*** *é um subconjunto de arestas de uma AGM de G. Se* ***C*** *são os vértices de uma componente de $G_A = (V, A)$ e* ***(u,v)*** *é uma aresta leve do corte $\delta(C)$, então* ***(u,v)*** *é uma* ***ARESTA SEGURA****.*

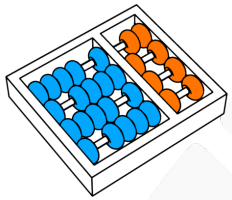


Consequência para os algoritmos

Corolário

*Seja (G, w) um grafo com pesos nas arestas e suponha que **A** é um subconjunto de arestas de uma AGM de G. Se **C** são os vértices de uma componente de $G_A = (V, A)$ e **(u,v)** é uma aresta leve do corte $\delta(C)$, então **(u,v)** é uma **ARESTA SEGURA**.*

Isso sugere uma estratégia iterativa:



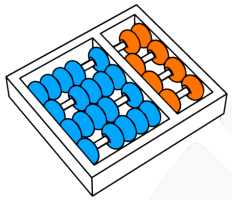
Consequência para os algoritmos

Corolário

*Seja (G, w) um grafo com pesos nas arestas e suponha que **A** é um subconjunto de arestas de uma AGM de G. Se **C** são os vértices de uma componente de $G_A = (V, A)$ e **(u,v)** é uma aresta leve do corte $\delta(C)$, então **(u,v)** é uma **ARESTA SEGURA**.*

Isso sugere uma estratégia iterativa:

- Prim e Kruskal implementam essa ideia.



Consequência para os algoritmos

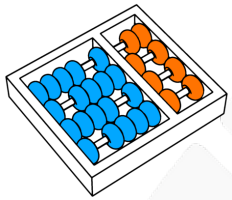
Corolário

*Seja (G, w) um grafo com pesos nas arestas e suponha que **A** é um subconjunto de arestas de uma AGM de G. Se **C** são os vértices de uma componente de $G_A = (V, A)$ e **(u,v)** é uma aresta leve do corte $\delta(C)$, então **(u,v)** é uma **ARESTA SEGURA**.*

Isso sugere uma estratégia iterativa:

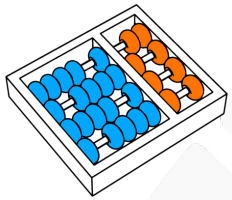
- Prim e Kruskal implementam essa ideia.
- Tais algoritmos farão uso desse corolário.

Algoritmo de Prim



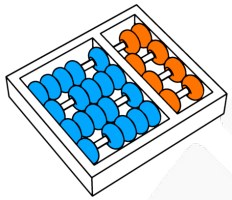
Ideia

- Escolhemos um vértice **r** arbitrariamente no início.



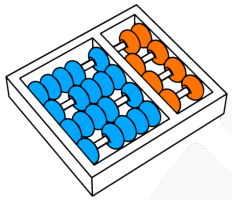
Ideia

- Escolhemos um vértice **r** arbitrariamente no início.
- O conjunto **A** são as arestas de uma árvore com raiz **r**.



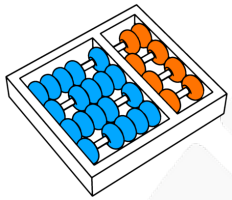
Ideia

- Escolhemos um vértice **r** arbitrariamente no início.
- O conjunto **A** são as arestas de uma árvore com raiz **r**.
- O conjunto **S** são os vértices dessa árvore.



Ideia

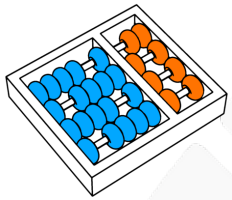
- Escolhemos um vértice **r** arbitrariamente no início.
- O conjunto **A** são as arestas de uma árvore com raiz **r**.
- O conjunto **S** são os vértices dessa árvore.
- Em cada iteração, adicionamos uma **ARESTA LEVE** de $\delta(\mathbf{S})$.



Ideia

- Escolhemos um vértice **r** arbitrariamente no início.
- O conjunto **A** são as arestas de uma árvore com raiz **r**.
- O conjunto **S** são os vértices dessa árvore.
- Em cada iteração, adicionamos uma **ARESTA LEVE** de $\delta(\mathbf{S})$.

Detalhe de implementação importante:

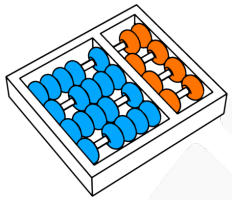


Ideia

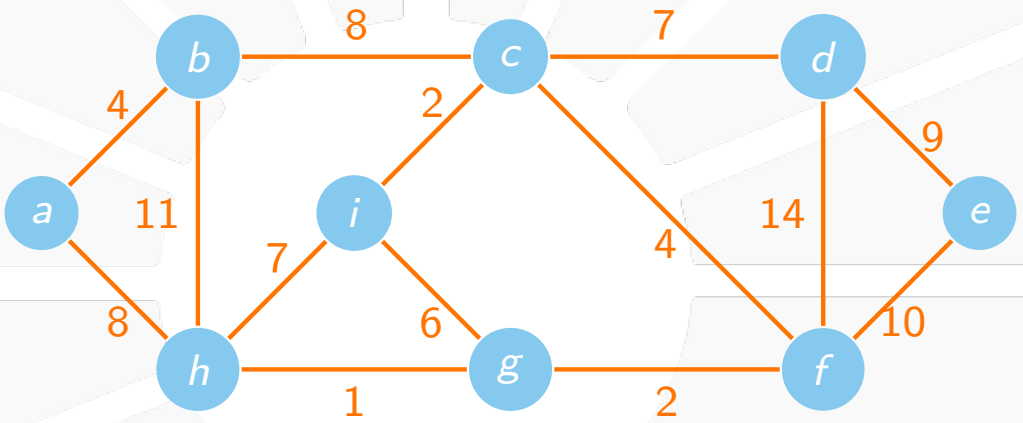
- Escolhemos um vértice **r** arbitrariamente no início.
- O conjunto **A** são as arestas de uma árvore com raiz **r**.
- O conjunto **S** são os vértices dessa árvore.
- Em cada iteração, adicionamos uma **ARESTA LEVE** de $\delta(\mathbf{S})$.

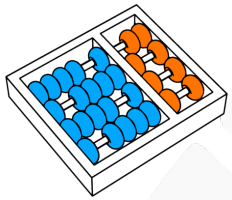
Detalhe de implementação importante:

- Como encontrar essa aresta leve **EFICIENTEMENTE**?

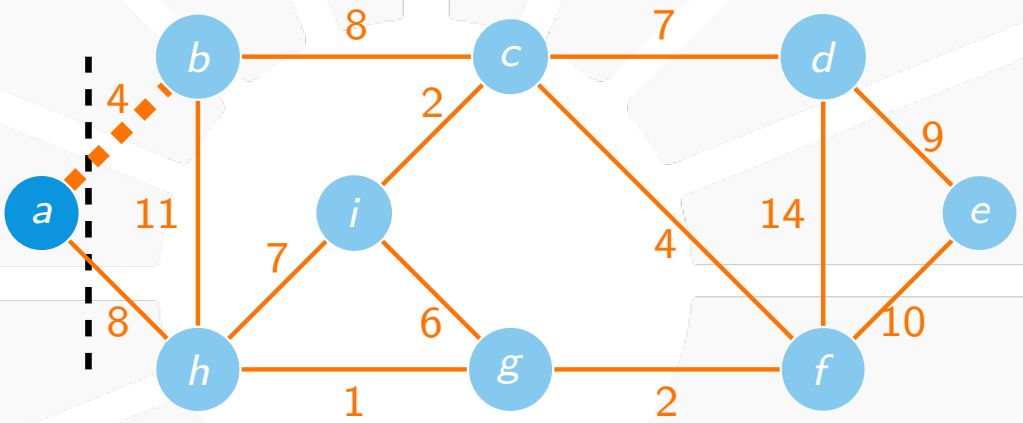


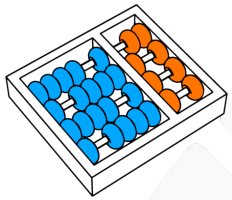
Exemplo



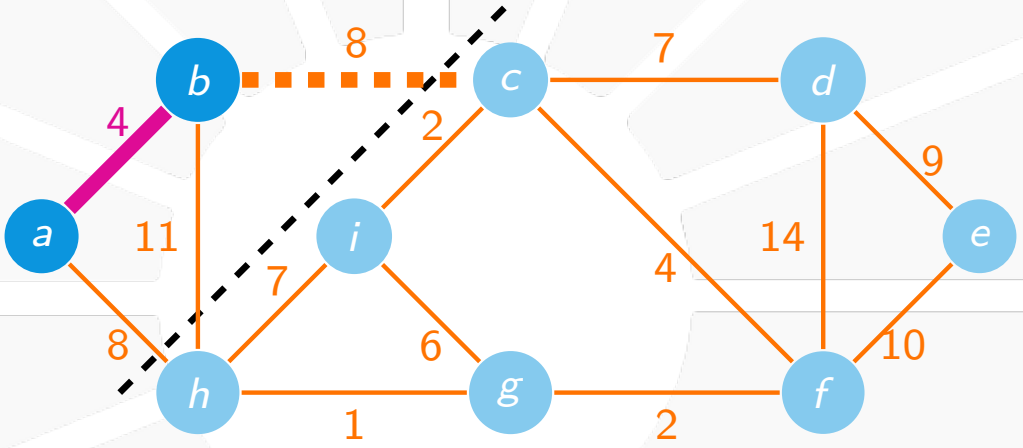


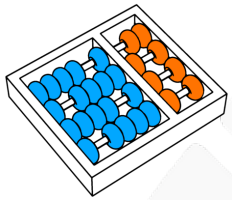
Exemplo



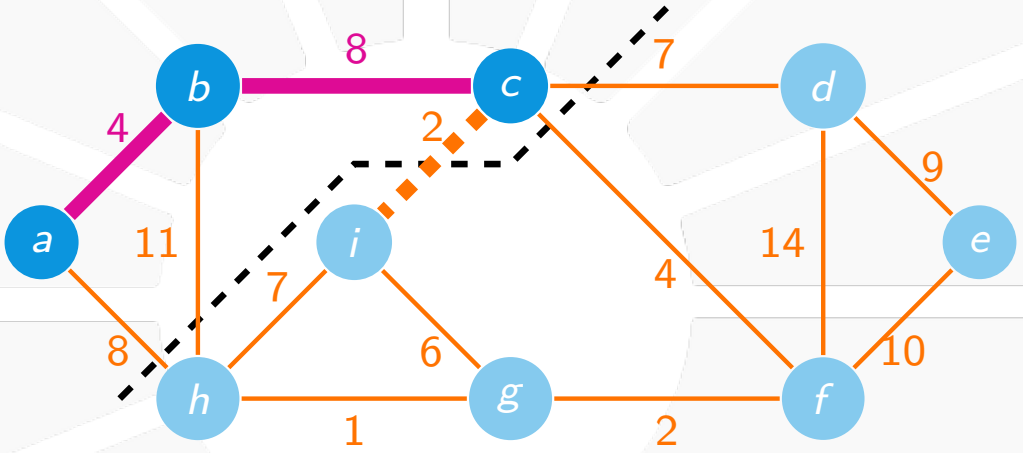


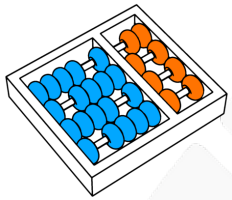
Exemplo



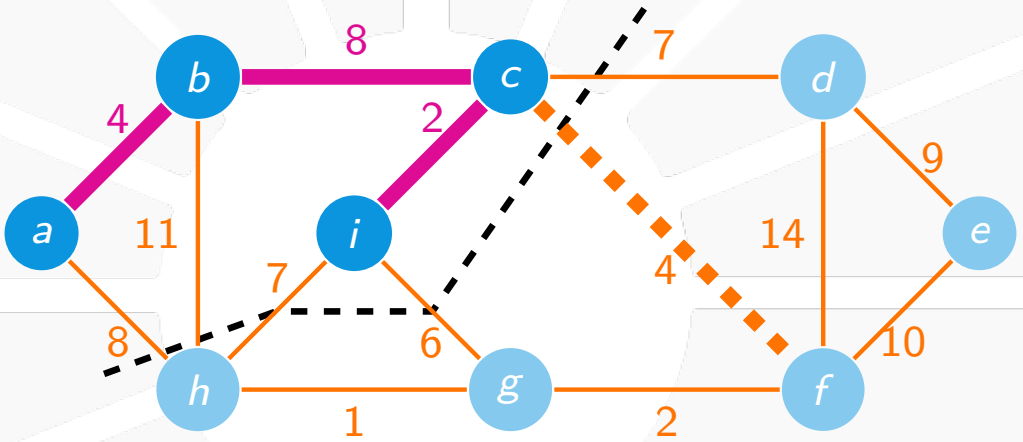


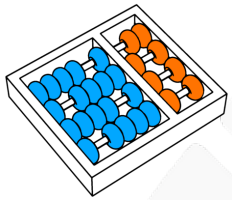
Exemplo



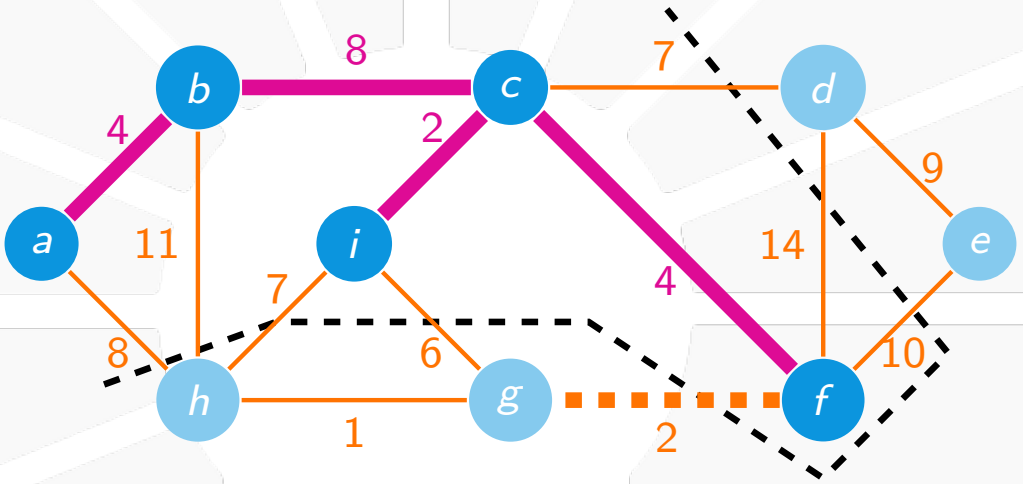


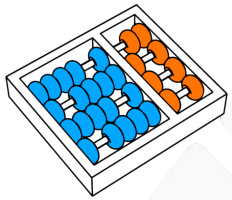
Exemplo



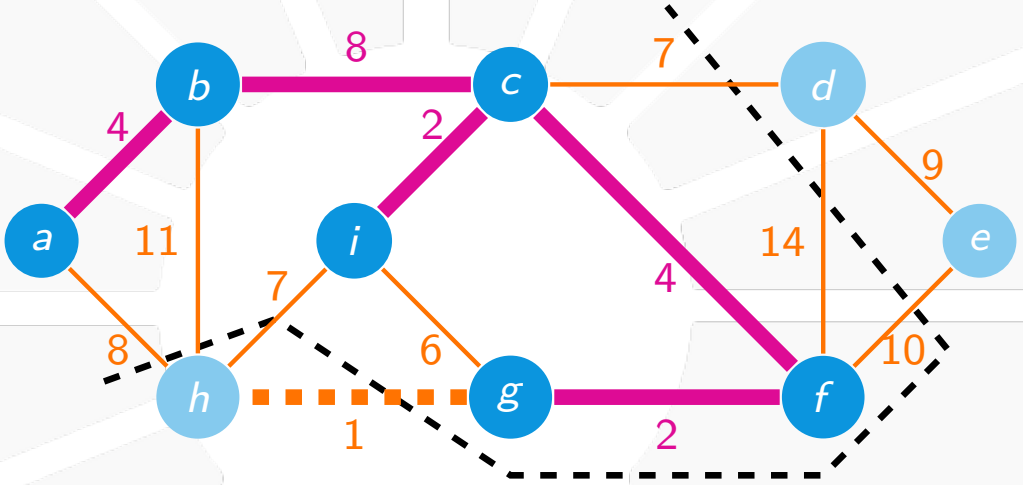


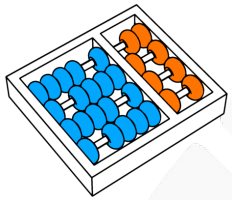
Exemplo



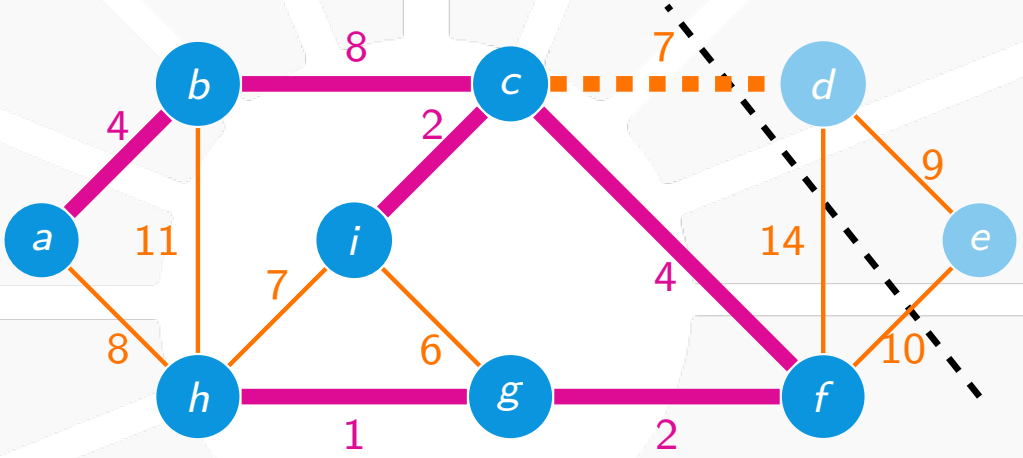


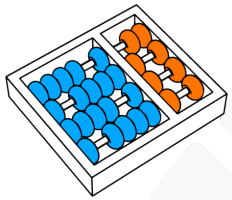
Exemplo



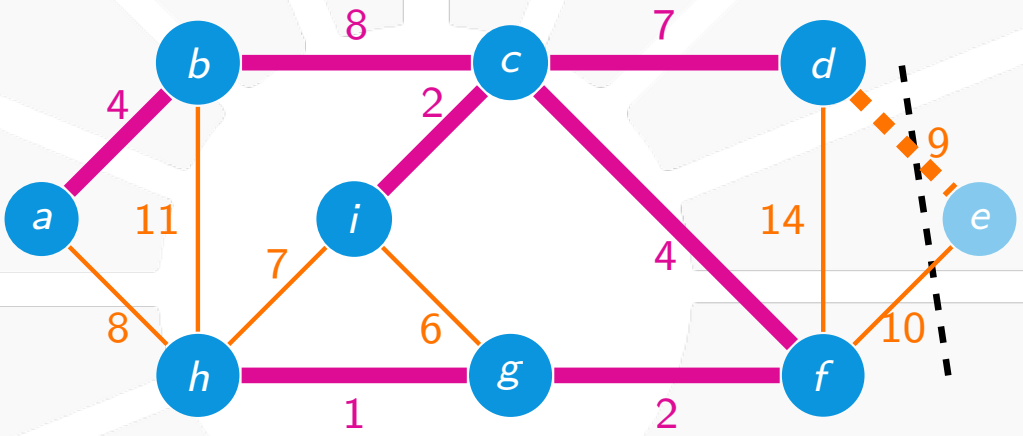


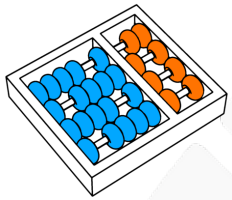
Exemplo



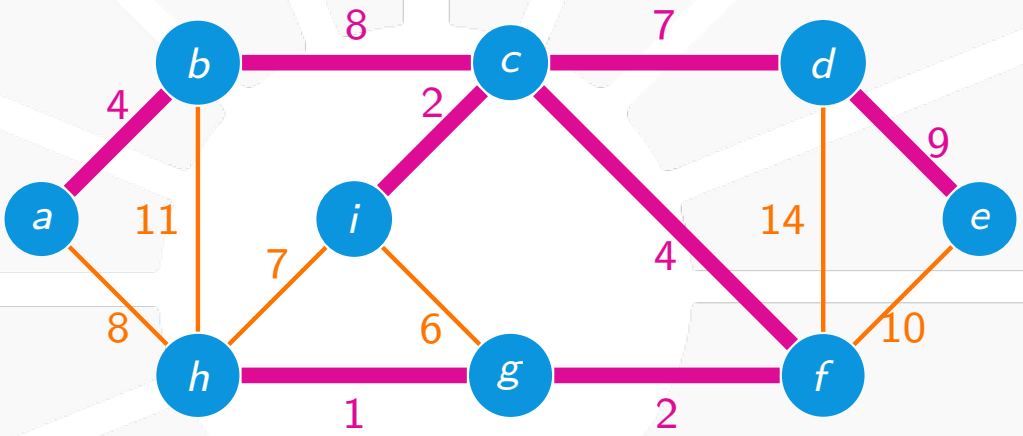


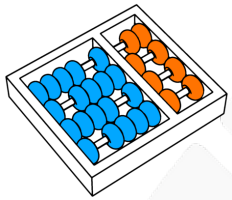
Exemplo





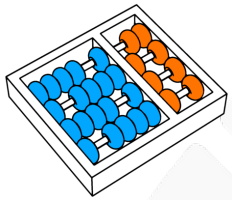
Exemplo





Estruturas de dados

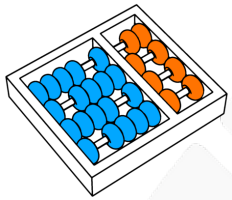
Como representar os vértices a serem adicionados?



Estruturas de dados

Como representar os vértices a serem adicionados?

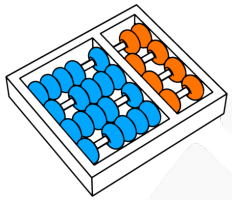
- Mantemos uma fila de prioridade (de mínimo) Q.



Estruturas de dados

Como representar os vértices a serem adicionados?

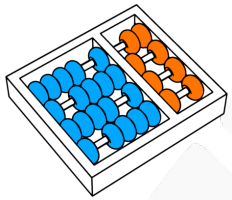
- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.



Estruturas de dados

Como representar os vértices a serem adicionados?

- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

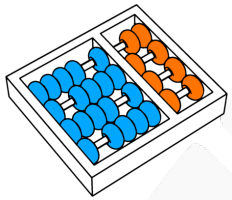


Estruturas de dados

Como representar os vértices a serem adicionados?

- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

Qual a prioridade de escolher um vértice **v**?



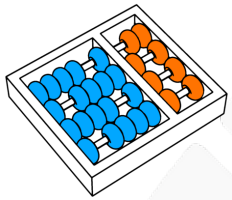
Estruturas de dados

Como representar os vértices a serem adicionados?

- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

Qual a prioridade de escolher um vértice **v**?

- key[**v**] guarda o peso da menor aresta ligando **v** à árvore, ou vale ∞ se não houver uma tal aresta.



Estruturas de dados

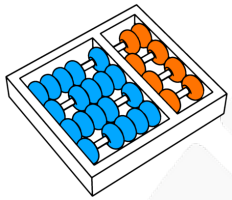
Como representar os vértices a serem adicionados?

- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

Qual a prioridade de escolher um vértice **v**?

- key[**v**] guarda o peso da menor aresta ligando **v** à árvore, ou vale ∞ se não houver uma tal aresta.

Como representar a árvore sendo construída?



Estruturas de dados

Como representar os vértices a serem adicionados?

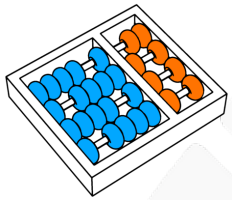
- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

Qual a prioridade de escolher um vértice **v**?

- key[**v**] guarda o peso da menor aresta ligando **v** à árvore, ou vale ∞ se não houver uma tal aresta.

Como representar a árvore sendo construída?

- Mantemos um vetor π de pais de todos vértices.



Estruturas de dados

Como representar os vértices a serem adicionados?

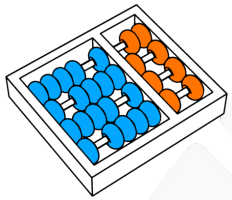
- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice **v** na fila tem prioridade key[**v**] de ser inserido.

Qual a prioridade de escolher um vértice **v**?

- key[**v**] guarda o peso da menor aresta ligando **v** à árvore, ou vale ∞ se não houver uma tal aresta.

Como representar a árvore sendo construída?

- Mantemos um vetor π de pais de todos vértices.
- Os vértices da árvore são: $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$.



Estruturas de dados

Como representar os vértices a serem adicionados?

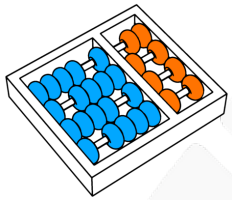
- Mantemos uma fila de prioridade (de mínimo) Q.
- Ela contém todos vértices que **NÃO** estão na árvore.
- Cada vértice $\mathbf{v}$ na fila tem prioridade key[$\mathbf{v}$] de ser inserido.

Qual a prioridade de escolher um vértice $\mathbf{v}$?

- key[$\mathbf{v}$] guarda o peso da menor aresta ligando $\mathbf{v}$ à árvore, ou vale ∞ se não houver uma tal aresta.

Como representar a árvore sendo construída?

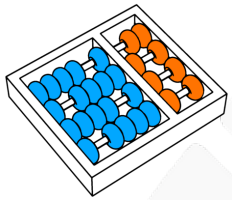
- Mantemos um vetor π de pais de todos vértices.
- Os vértices da árvore são: $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$.
- As arestas da árvore são: $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$.



O algoritmo

Algoritmo 6: AGM-PRIM(G, w, r)

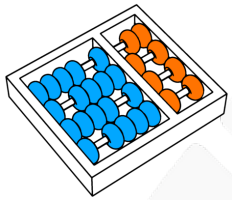
```
1  para cada u ∈ V[G]
2      key[u] ← ∞
3      π[u] ← NIL
4  key[r] ← 0
5  Q ← V[G]
6  enquanto Q ≠ ∅
7      u ← EXTRACT-MIN(Q)
8      para cada v ∈ Adj[u]
9          se v ∈ Q e w(u, v) < key[v]
10             π[v] ← u
11             key[v] ← w(u, v)
```



Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$, *então:*

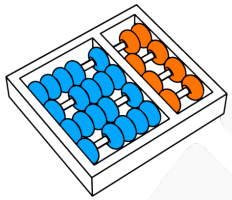


Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$*, então:*

1. ***A*** *contém arestas de uma árvore* T *com vértices* ***S*** *e raiz* ***r****.*

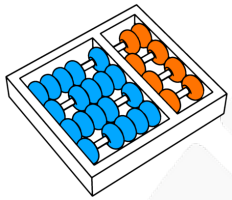


Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$*, então:*

1. ***A*** *contém arestas de uma árvore* T *com vértices* ***S*** *e raiz* ***r****.*
2. *Para cada* $\boldsymbol{v} \in \boldsymbol{Q}$*:*

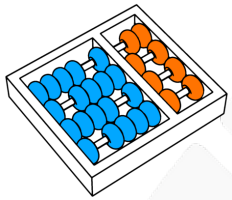


Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$*, então:*

1. $\boldsymbol{A}$ *contém arestas de uma árvore* T *com vértices* $\boldsymbol{S}$ *e raiz* $\boldsymbol{r}$*.*
2. *Para cada* $\boldsymbol{v} \in \boldsymbol{Q}$*:*
   - *Se* $\pi[\boldsymbol{v}] \neq$ NIL*, então* key$[\boldsymbol{v}]$ *é o peso de uma aresta com menor peso ligando* $\boldsymbol{v}$ *a algum vértice de* T*.*

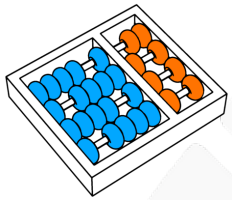


Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$*, então:*

1. $\boldsymbol{A}$ *contém arestas de uma árvore* T *com vértices* $\boldsymbol{S}$ *e raiz* $\boldsymbol{r}$*.*
2. *Para cada* $\boldsymbol{v} \in \boldsymbol{Q}$*:*
   - *Se* $\pi[\boldsymbol{v}] \neq$ NIL*, então* key$[\boldsymbol{v}]$ *é o peso de uma aresta com menor peso ligando* $\boldsymbol{v}$ *a algum vértice de* T*.*
   - *Se* $\pi[\boldsymbol{v}] =$ NIL*, então não existe aresta ligando* $\boldsymbol{v}$ *a algum vértice de* T*.*



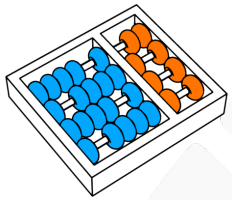
Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$, *então:*

1. $\boldsymbol{A}$ *contém arestas de uma árvore* T *com vértices* $\boldsymbol{S}$ *e raiz* $\boldsymbol{r}$.
2. *Para cada* $\boldsymbol{v} \in \boldsymbol{Q}$:
   - *Se* $\pi[\boldsymbol{v}] \neq$ NIL, *então* key$[\boldsymbol{v}]$ *é o peso de uma aresta com menor peso ligando* $\boldsymbol{v}$ *a algum vértice de* T.
   - *Se* $\pi[\boldsymbol{v}] =$ NIL, *então não existe aresta ligando* $\boldsymbol{v}$ *a algum vértice de* T.

- As invariantes implicam que no início da iteração do laço, $(\mathbf{u}, \pi[\mathbf{u}])$ é uma **ARESTA SEGURA**.



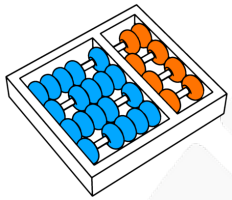
Correção do algoritmo

Teorema (Invariantes)

Considere a execução no início do laço **enquanto** *e defina* $\mathbf{S} = \mathbf{V} \setminus \mathbf{Q}$ *e* $\mathbf{A} = \{(\mathbf{u}, \pi[\mathbf{u}]) : \mathbf{u} \in \mathbf{S} \setminus \{\mathbf{r}\}\}$*, então:*

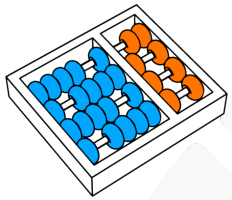
1. $\boldsymbol{A}$ *contém arestas de uma árvore* T *com vértices* $\boldsymbol{S}$ *e raiz* $\boldsymbol{r}$*.*
2. *Para cada* $\boldsymbol{v} \in \boldsymbol{Q}$*:*
   - *Se* $\pi[\boldsymbol{v}] \neq$ NIL*, então* key$[\boldsymbol{v}]$ *é o peso de uma aresta com menor peso ligando* $\boldsymbol{v}$ *a algum vértice de* T*.*
   - *Se* $\pi[\boldsymbol{v}] =$ NIL*, então não existe aresta ligando* $\boldsymbol{v}$ *a algum vértice de* T*.*

- As invariantes implicam que no início da iteração do laço, $(\mathbf{u}, \pi[\mathbf{u}])$ é uma **ARESTA SEGURA**.
- Portanto, o algoritmo está correto.



Complexidade

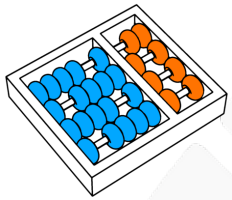
A complexidade depende da fila de prioridade Q:



Complexidade

A complexidade depende da fila de prioridade Q:

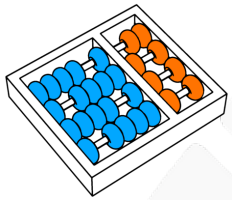
- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).



Complexidade

A complexidade depende da fila de prioridade Q:

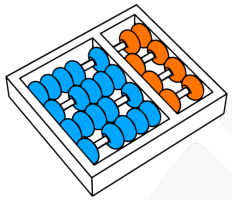
- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:



Complexidade

A complexidade depende da fila de prioridade Q:

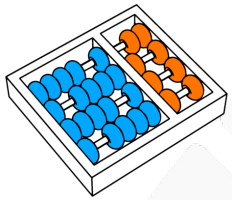
- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:
  - INSERT é executada $|V|$ vezes (linhas 1–5).



Complexidade

A complexidade depende da fila de prioridade Q:

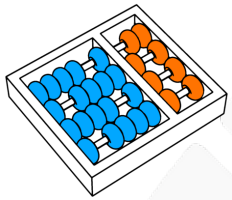
- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:
  - INSERT é executada $|V|$ vezes (linhas 1–5).
  - EXTRACT-MIN é executada $|V|$ vezes (linha 6).



Complexidade

A complexidade depende da fila de prioridade Q:

- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:
  - Insert é executada $|V|$ vezes (linhas 1–5).
  - Extract-Min é executada $|V|$ vezes (linha 6).
  - Decrease-Key é executada até $|E|$ vezes (linha 11).

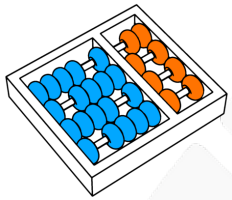


Complexidade

A complexidade depende da fila de prioridade Q:

- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:
  - INSERT é executada $|V|$ vezes (linhas 1–5).
  - EXTRACT-MIN é executada $|V|$ vezes (linha 6).
  - DECREASE-KEY é executada até $|E|$ vezes (linha 11).

Portanto, o **TEMPO TOTAL** de execução é:



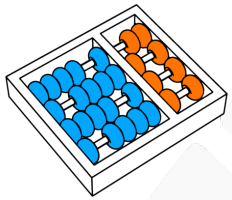
Complexidade

A complexidade depende da fila de prioridade Q:

- Cada teste $v \in Q$ (linha 9) leva tempo constante (por quê?).
- Vamos contar quantas vezes executamos cada operação:
  - INSERT é executada $|V|$ vezes (linhas 1–5).
  - EXTRACT-MIN é executada $|V|$ vezes (linha 6).
  - DECREASE-KEY é executada até $|E|$ vezes (linha 11).

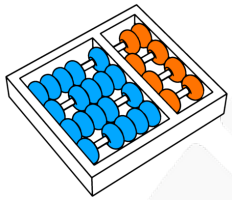
Portanto, o **TEMPO TOTAL** de execução é:

$$O(V) \cdot \textsc{Insert} + O(V) \cdot \textsc{Extract-Min} + O(E) \cdot \textsc{Decrease-Key}.$$



Complexidade usando min-heap

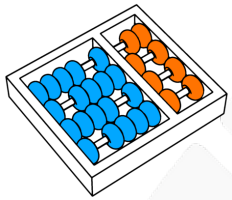
Se implementarmos Q como um min-heap, então:



Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

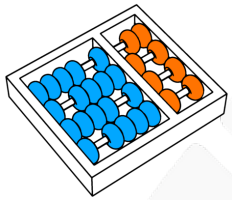
- INSERT consome tempo $O(\log V)$.



Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

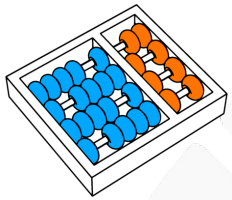
- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.



Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

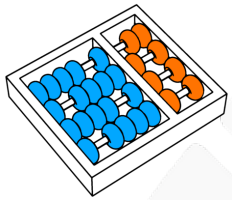


Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

Então, o tempo total será:



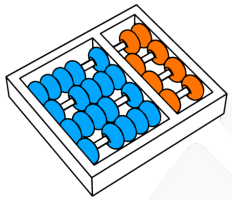
Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

Então, o tempo total será:

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



Complexidade usando min-heap

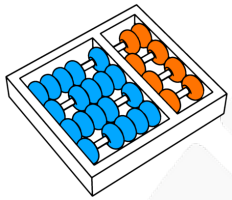
Se implementarmos Q como um min-heap, então:

- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

Então, o tempo total será:

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

Observações:



Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

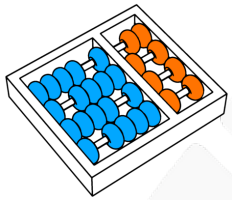
- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

Então, o tempo total será:

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

Observações:

- Podemos inicializar o min-heap em tempo $O(V)$.



Complexidade usando min-heap

Se implementarmos Q como um min-heap, então:

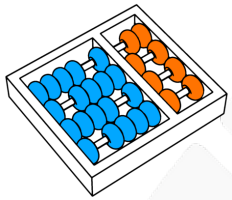
- INSERT consome tempo $O(\log V)$.
- EXTRACT-MIN consome tempo $O(\log V)$.
- DECREASE-KEY consome tempo $O(\log V)$.

Então, o tempo total será:

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

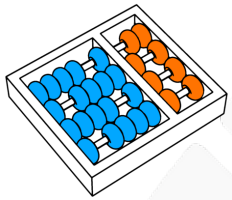
Observações:

- Podemos inicializar o min-heap em tempo $O(V)$.
- Usamos $V = O(E)$ pois sabemos que G é conexo.



Análise amortizada

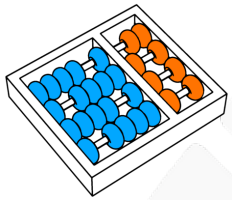
Refletindo sobre como analisamos uma operação:



Análise amortizada

Refletindo sobre como analisamos uma operação:

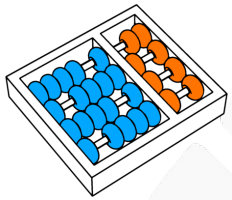
- Supomos que todas as chamadas levam o mesmo tempo.



Análise amortizada

Refletindo sobre como analisamos uma operação:

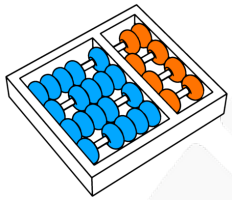
- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.



Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

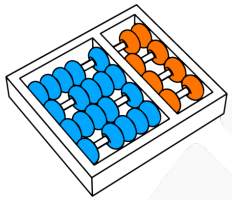


Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

Custo amortizado:



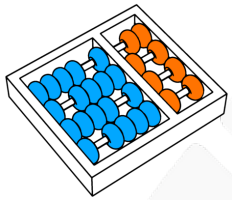
Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

Custo amortizado:

- Considere uma estrutura de dados abstrata S.



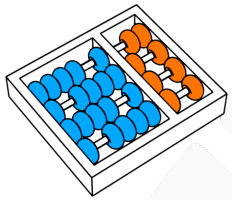
Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

Custo amortizado:

- Considere uma estrutura de dados abstrata S.
- Suponha que podemos realizar uma operação $p(S)$.



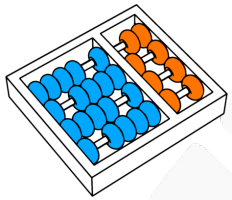
Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

Custo amortizado:

- Considere uma estrutura de dados abstrata S.
- Suponha que podemos realizar uma operação $p(S)$.
- Pode haver operações distintas (inserir, remover, etc.).



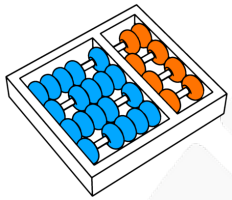
Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

Custo amortizado:

- Considere uma estrutura de dados abstrata S.
- Suponha que podemos realizar uma operação $p(S)$.
- Pode haver operações distintas (inserir, remover, etc.).
- Se executamos essas operações diversas vezes:



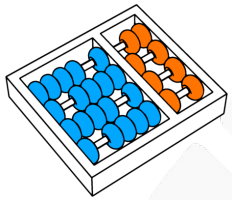
Análise amortizada

Refletindo sobre como analisamos uma operação:

- Supomos que todas as chamadas levam o mesmo tempo.
- Consideramos **SEMPRE** o tempo de pior caso.
- Na prática, o tempo de uma chamada pode ser bem menor.

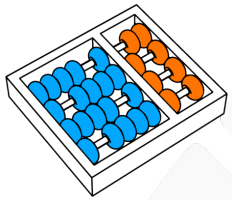
Custo amortizado:

- Considere uma estrutura de dados abstrata S.
- Suponha que podemos realizar uma operação $p(S)$.
- Pode haver operações distintas (inserir, remover, etc.).
- Se executamos essas operações diversas vezes:
- Quanto tempo leva cada chamada em **MÉDIA**?



Análise amortizada

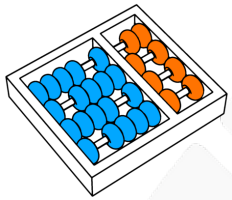
Ideia da análise amortizada:



Análise amortizada

Ideia da análise amortizada:

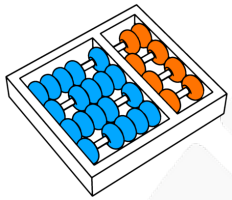
- Suponha que na execução fazemos m chamadas a $p(S)$.



Análise amortizada

Ideia da análise amortizada:

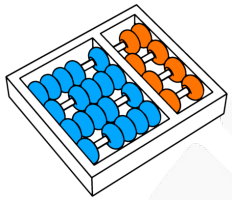
- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.



Análise amortizada

Ideia da análise amortizada:

- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.
- Então, o custo amortizado de p é $T(n)/m$.

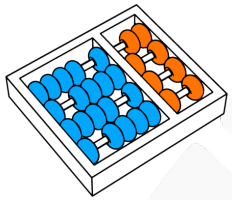


Análise amortizada

Ideia da análise amortizada:

- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.
- Então, o custo amortizado de p é $T(n)/m$.

Exemplo:



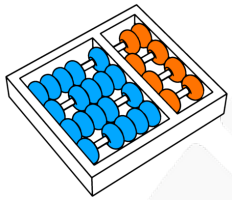
Análise amortizada

Ideia da análise amortizada:

- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.
- Então, o custo amortizado de p é $T(n)/m$.

Exemplo:

- se $T(n) = 4n$ e $m = 2n$, então o custo amortizado é 2.



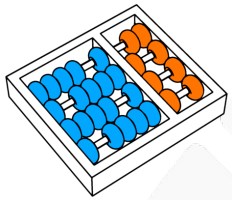
Análise amortizada

Ideia da análise amortizada:

- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.
- Então, o custo amortizado de p é $T(n)/m$.

Exemplo:

- se $T(n) = 4n$ e $m = 2n$, então o custo amortizado é 2.
- Isso **NÃO** significa que a operação leva tempo constante.



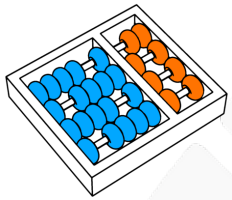
Análise amortizada

Ideia da análise amortizada:

- Suponha que na execução fazemos m chamadas a $p(S)$.
- Que o **TEMPO TOTAL** das operações p é $T(n)$.
- Então, o custo amortizado de p é $T(n)/m$.

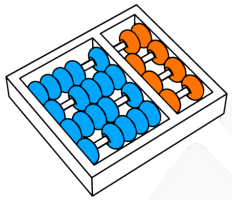
Exemplo:

- se $T(n) = 4n$ e $m = 2n$, então o custo amortizado é 2.
- Isso **NÃO** significa que a operação leva tempo constante.
- Apenas que em média o tempo gasto por p é constante.



Revisitando a complexidade de Prim

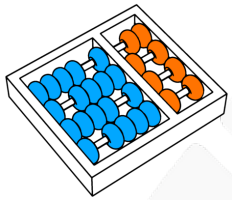
Um **HEAP DE FIBONACCI** é uma estrutura de dados que:



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

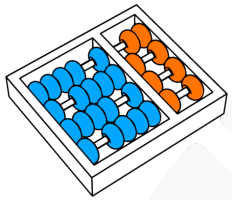
- É utilizada para guardar um conjunto de $|V|$ elementos.



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

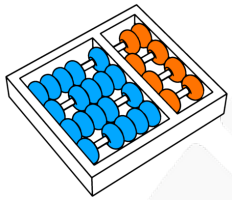
- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

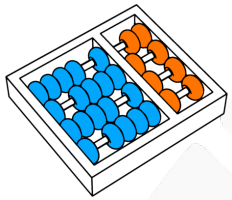
- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

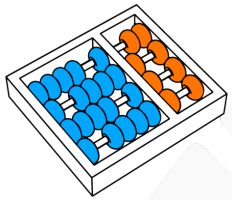
- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

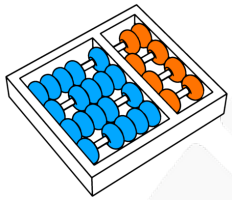
- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$
  - INSERT – tempo amortizado $O(1)$



Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$
  - INSERT – tempo amortizado $O(1)$
- Além de outras operações como UNION, etc. (veja CLRS)

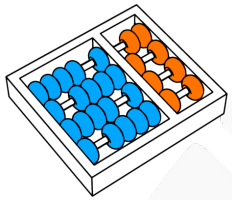


Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$
  - INSERT – tempo amortizado $O(1)$
- Além de outras operações como UNION, etc. (veja CLRS)

Se usarmos um heap de Fibonacci para implementar Q:



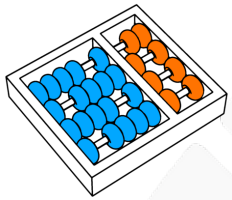
Revisitando a complexidade de Prim

Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$
  - INSERT – tempo amortizado $O(1)$
- Além de outras operações como UNION, etc. (veja CLRS)

Se usarmos um heap de Fibonacci para implementar Q:

- O tempo total melhora para $O(V + E + V \log V) = O(E + V \log V)$.



Revisitando a complexidade de Prim

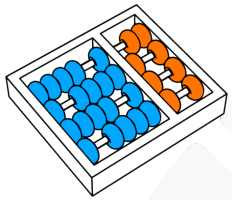
Um **HEAP DE FIBONACCI** é uma estrutura de dados que:

- É utilizada para guardar um conjunto de $|V|$ elementos.
- Implementa as operações de fila de prioridade:
  - EXTRACT-MIN – tempo $O(\log V)$
  - DECREASE-KEY – tempo amortizado $O(1)$
  - INSERT – tempo amortizado $O(1)$
- Além de outras operações como UNION, etc. (veja CLRS)

Se usarmos um heap de Fibonacci para implementar Q:

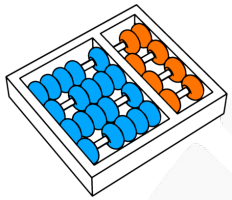
- O tempo total melhora para $O(V + E + V \log V) = O(E + V \log V)$.
- Na prática, a implementação com min-heap é melhor.

O algoritmo de Kruskal



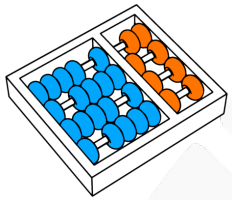
Ideia

- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.



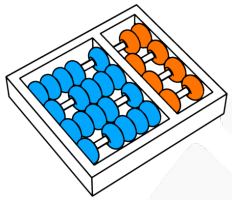
Ideia

- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.
- Consideramos cada uma das arestas em ordem de peso.



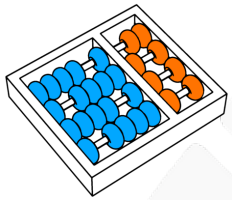
Ideia

- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.
- Consideramos cada uma das arestas em ordem de peso.
- Em cada iteração, adicionamos uma aresta **(u,v)** se ela ligar duas componentes distintas **C, C'** da floresta.



Ideia

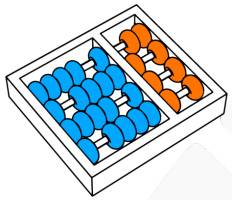
- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.
- Consideramos cada uma das arestas em ordem de peso.
- Em cada iteração, adicionamos uma aresta **(u,v)** se ela ligar duas componentes distintas **C, C'** da floresta.
- Note que **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{C})$



Ideia

- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.
- Consideramos cada uma das arestas em ordem de peso.
- Em cada iteração, adicionamos uma aresta **(u,v)** se ela ligar duas componentes distintas **C, C'** da floresta.
- Note que **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{C})$

Detalhe de implementação importante:

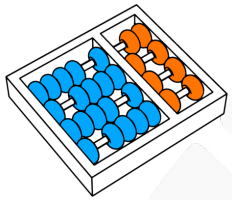


Ideia

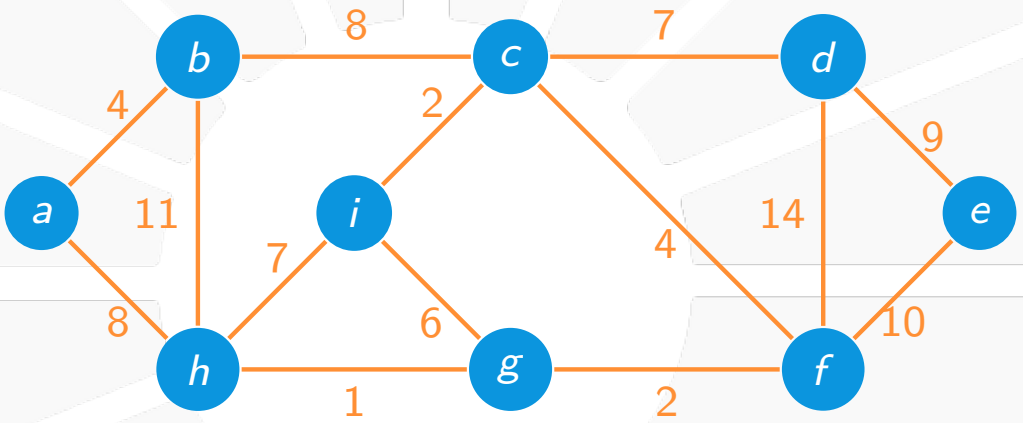
- O subgrafo $G_A = (\mathbf{V}, \mathbf{A})$ é uma floresta.
- Consideramos cada uma das arestas em ordem de peso.
- Em cada iteração, adicionamos uma aresta **(u,v)** se ela ligar duas componentes distintas **C, C'** da floresta.
- Note que **(u,v)** é uma **ARESTA LEVE** de $\delta(\mathbf{C})$

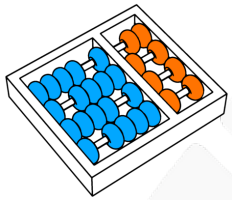
Detalhe de implementação importante:

- Como saber se **(u,v)** liga componentes distintas eficientemente?

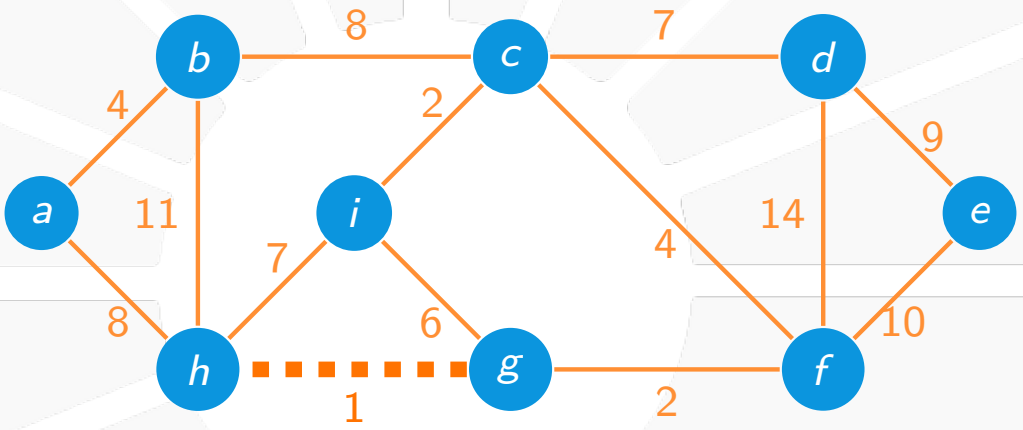


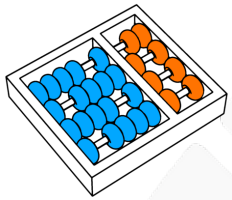
Exemplo



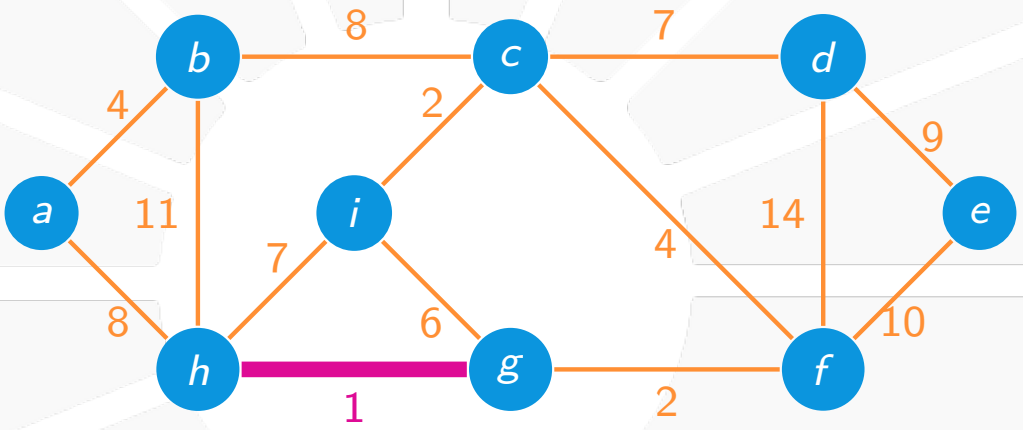


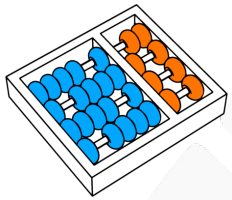
Exemplo



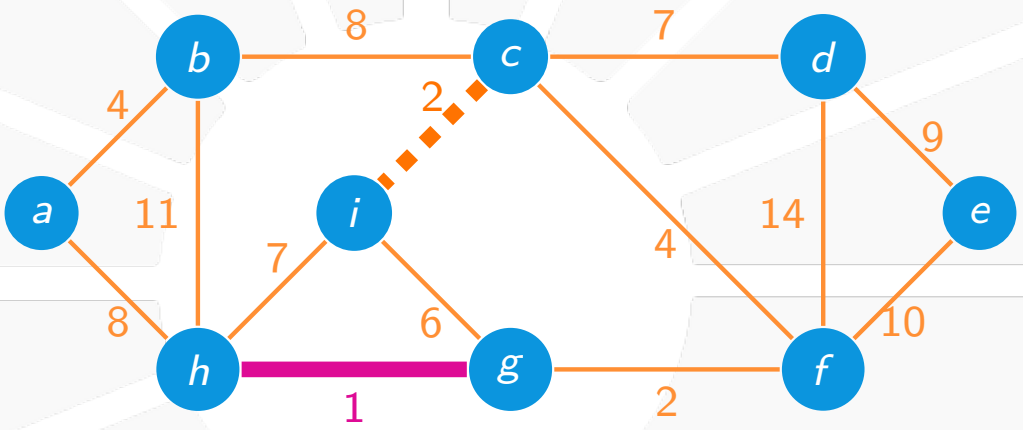


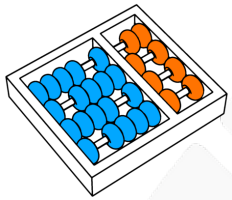
Exemplo



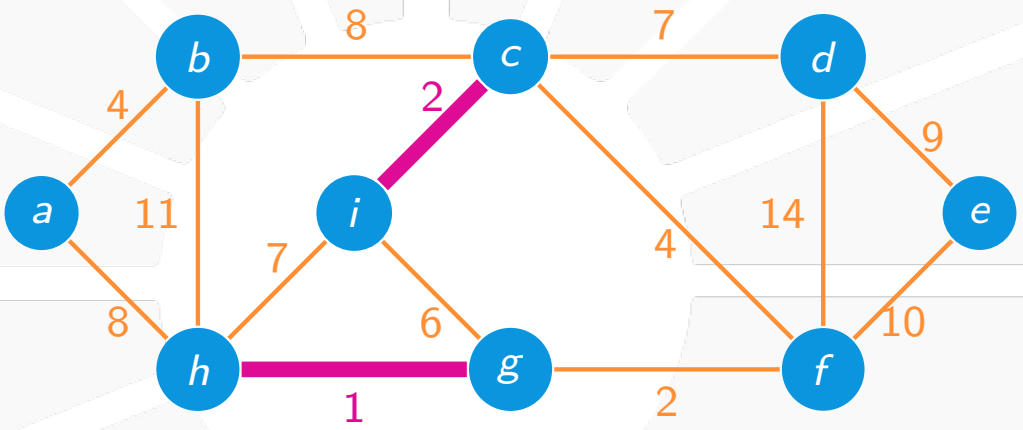


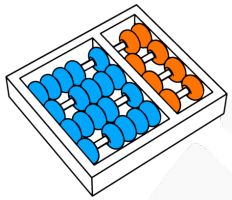
Exemplo



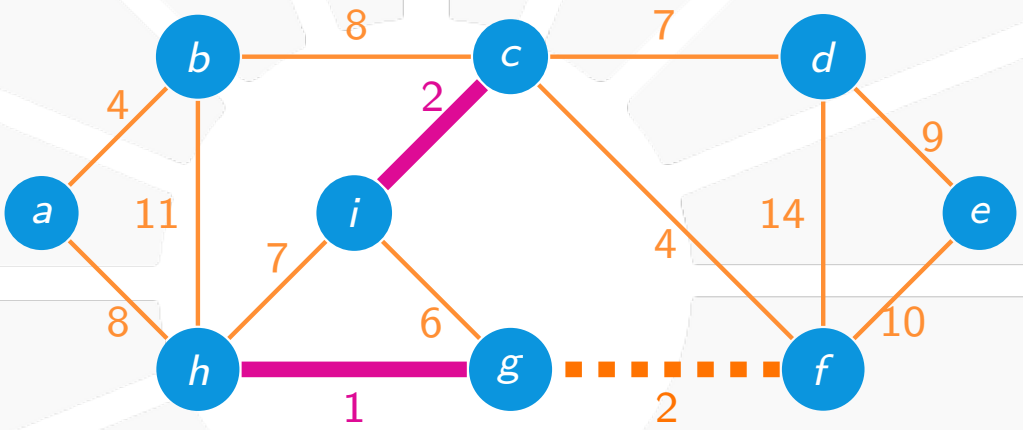


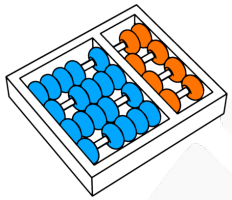
Exemplo



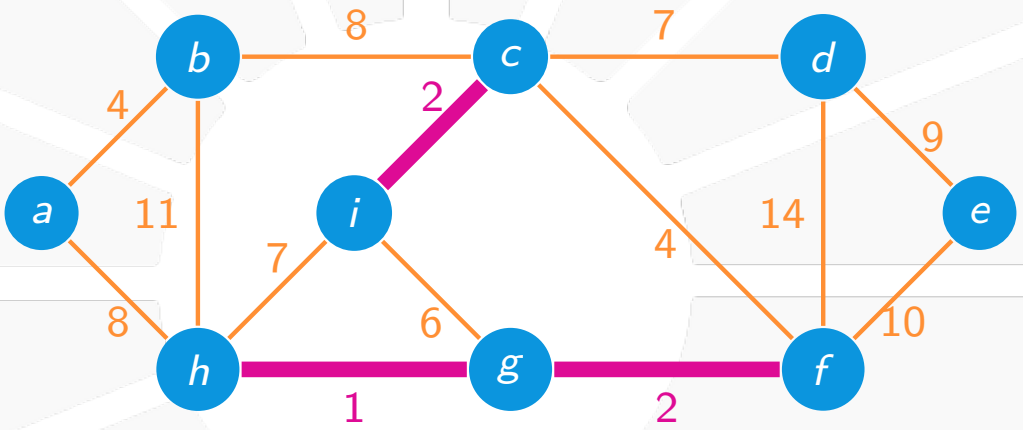


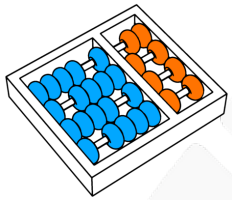
Exemplo



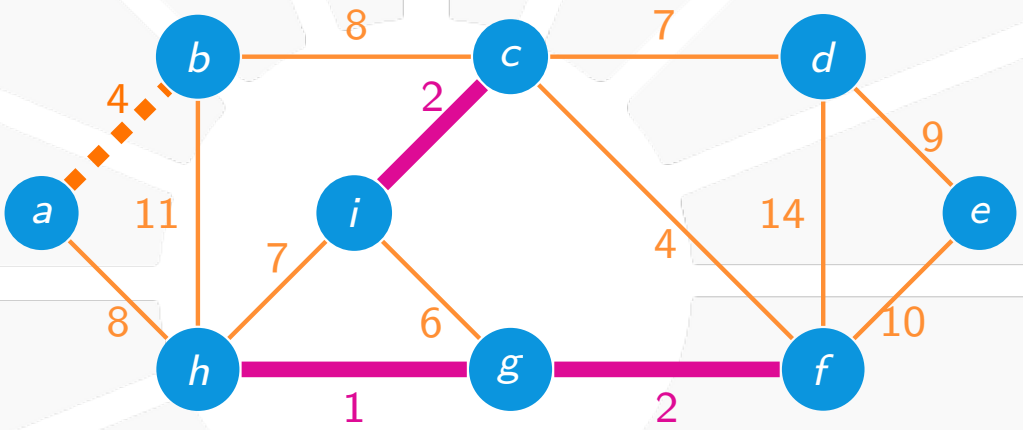


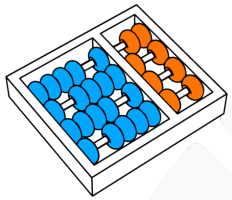
Exemplo



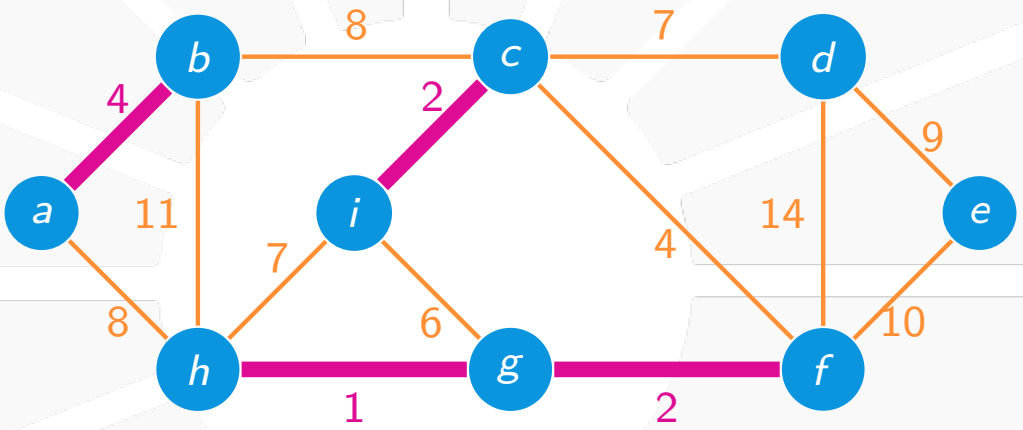


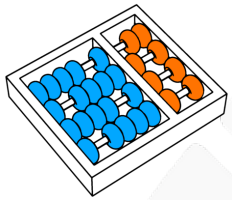
Exemplo



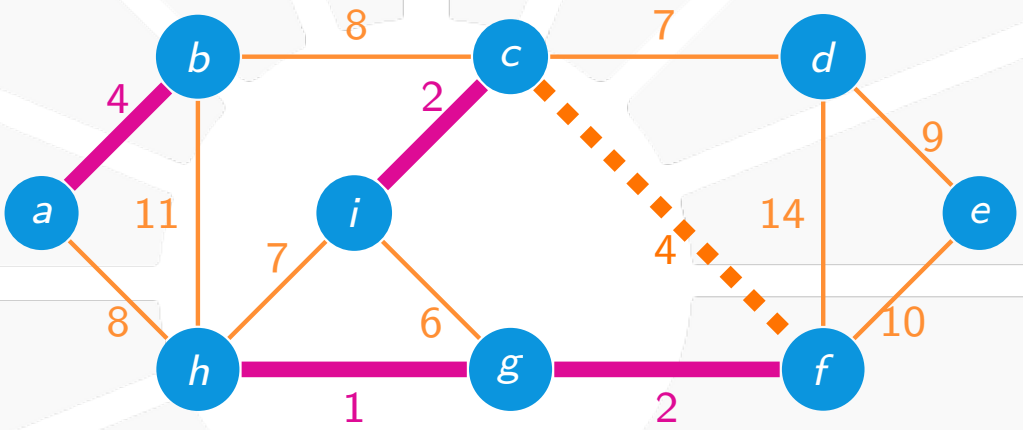


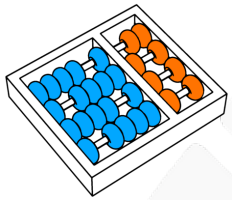
Exemplo



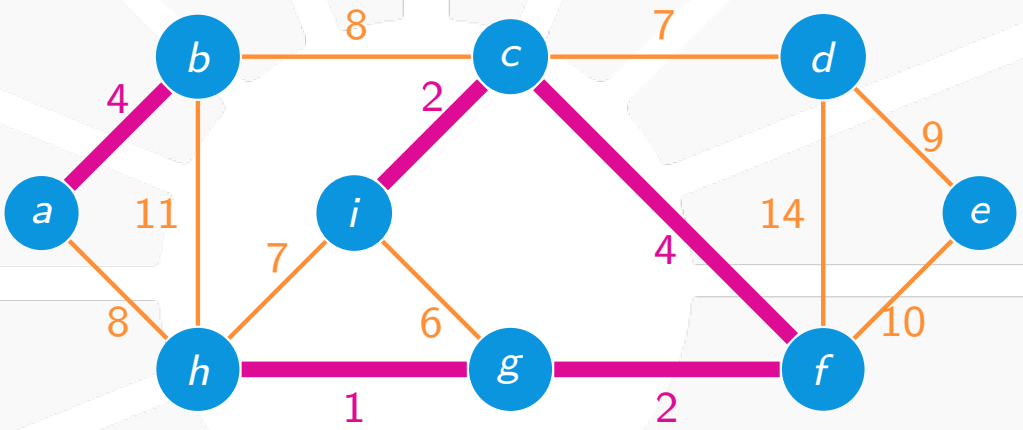


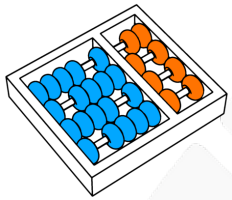
Exemplo



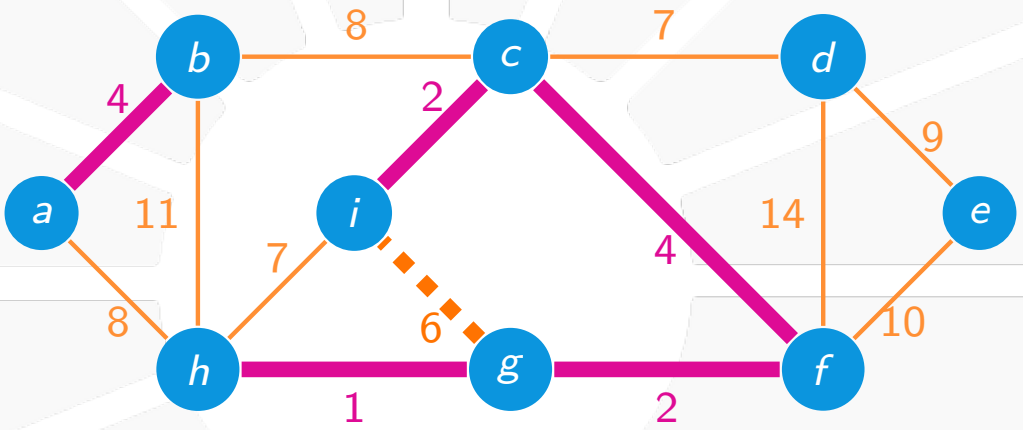


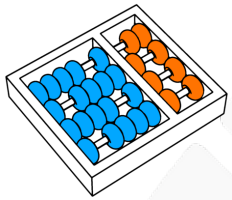
Exemplo



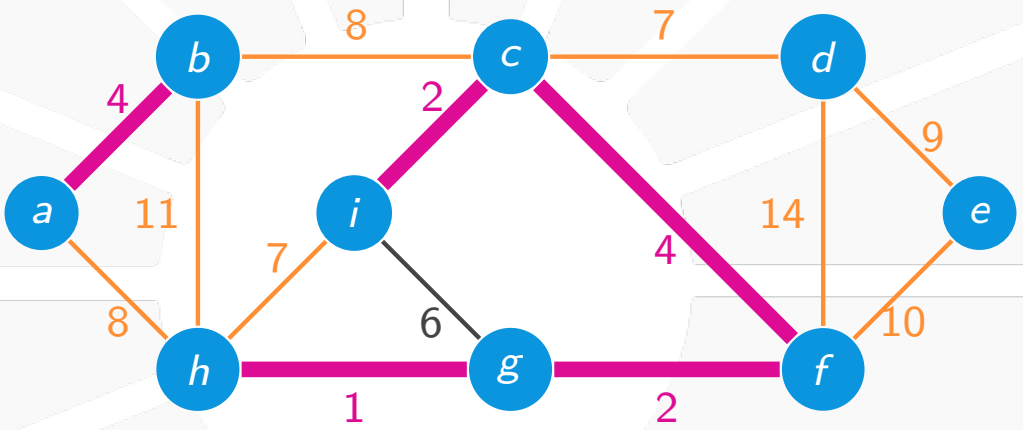


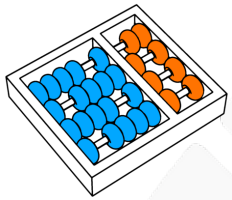
Exemplo



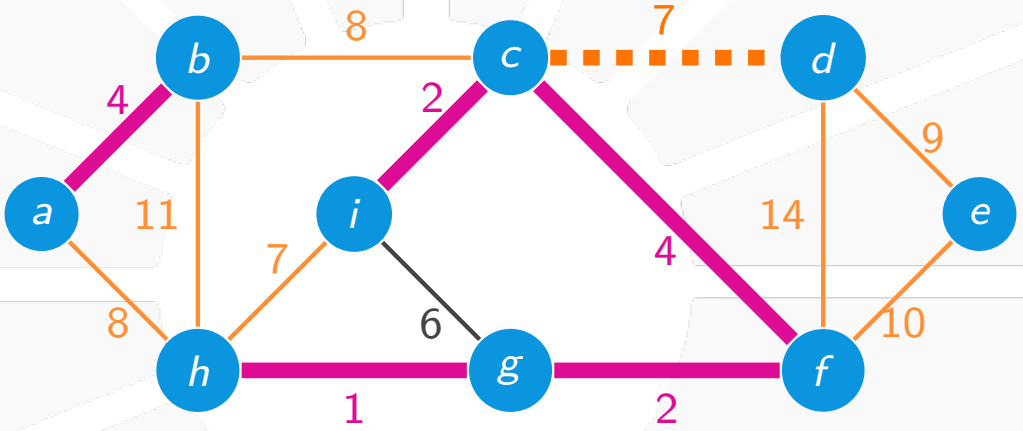


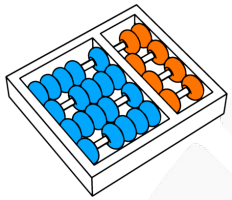
Exemplo



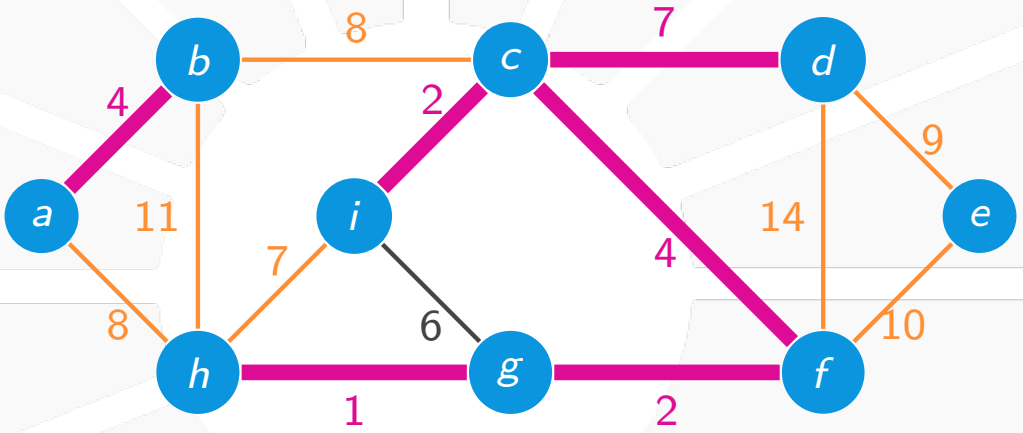


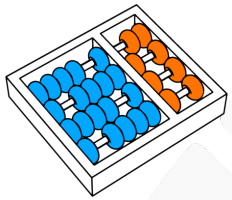
Exemplo



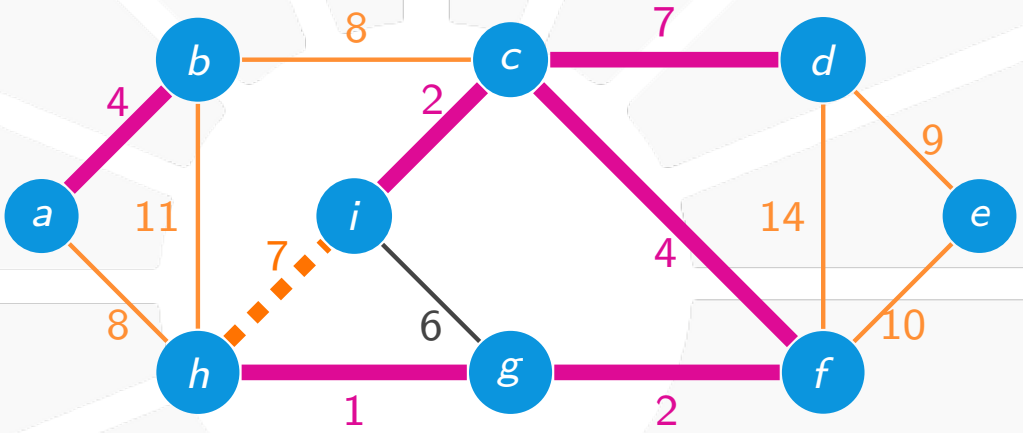


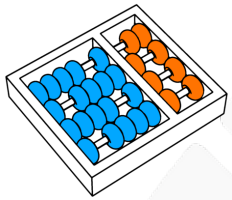
Exemplo



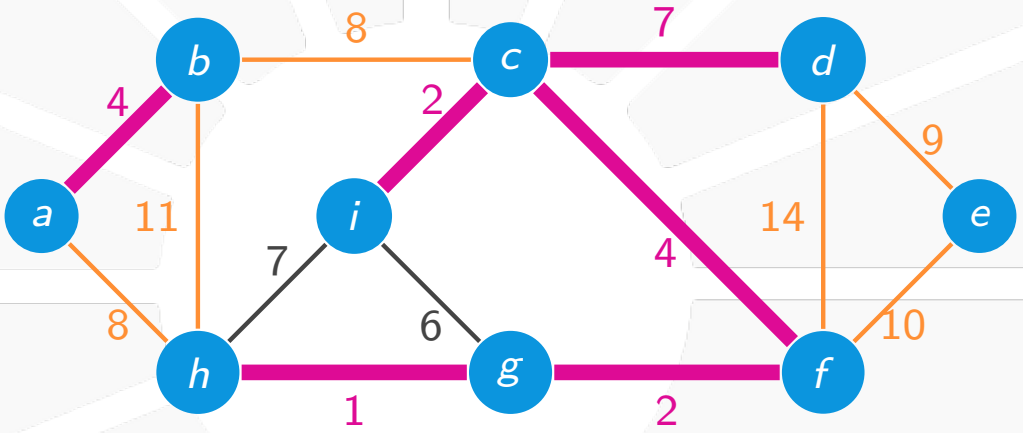


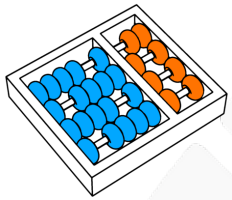
Exemplo



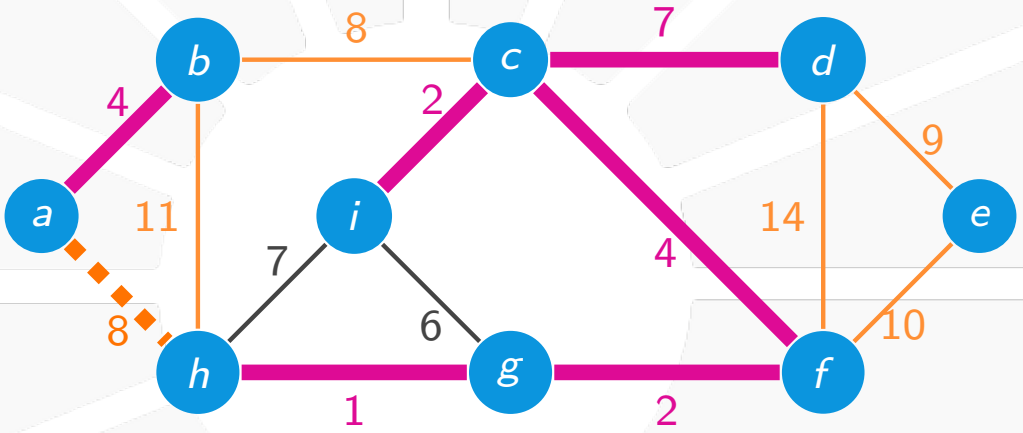


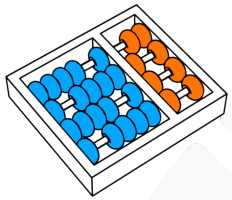
Exemplo



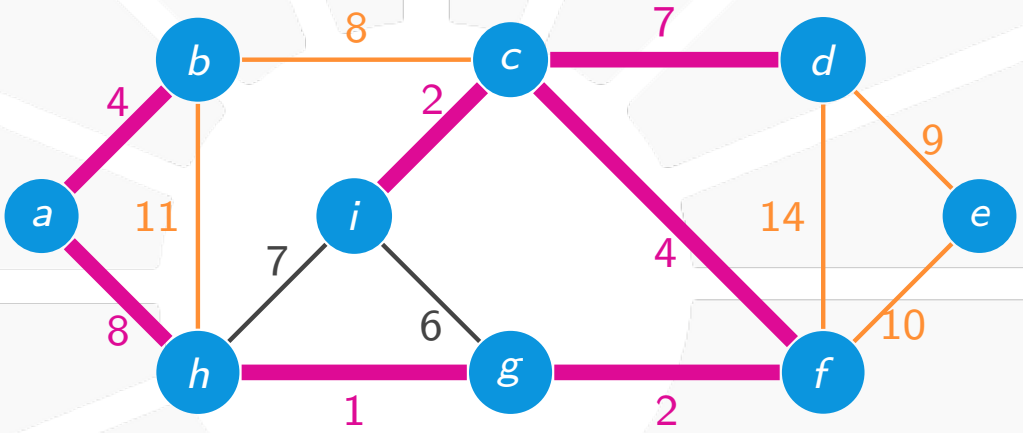


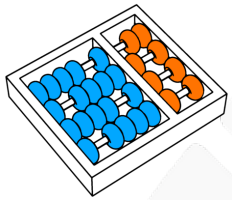
Exemplo



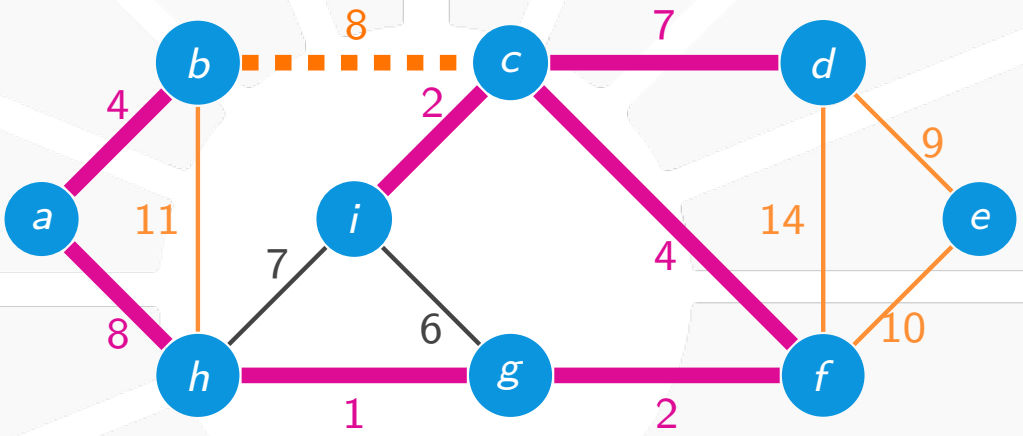


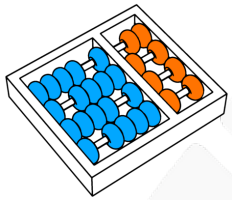
Exemplo



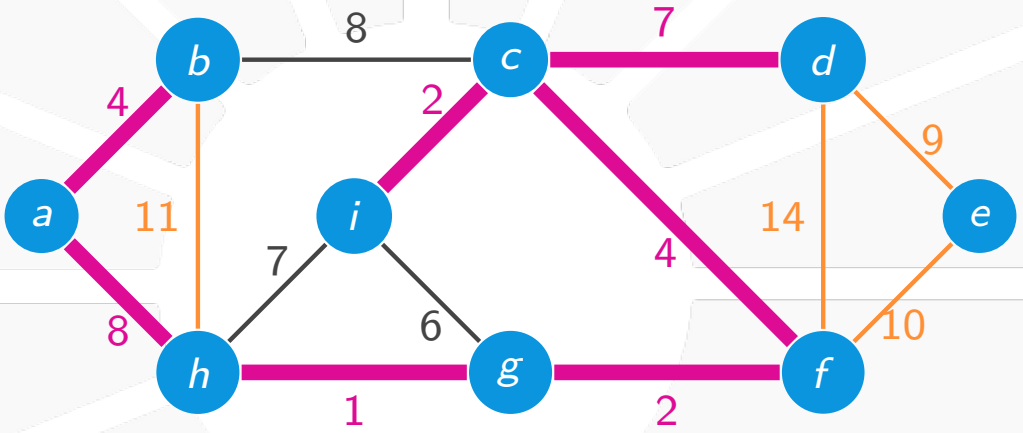


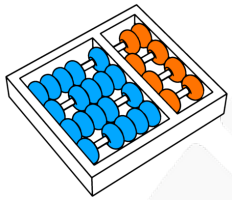
Exemplo



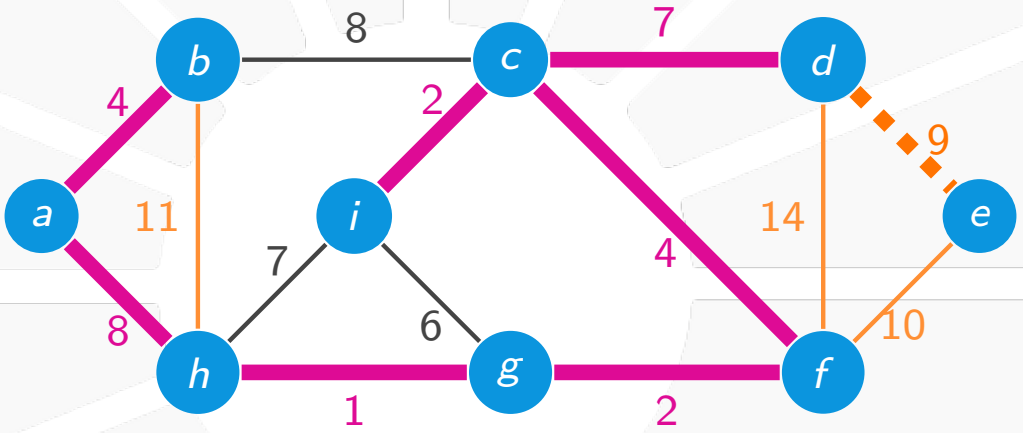


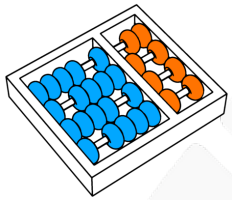
Exemplo



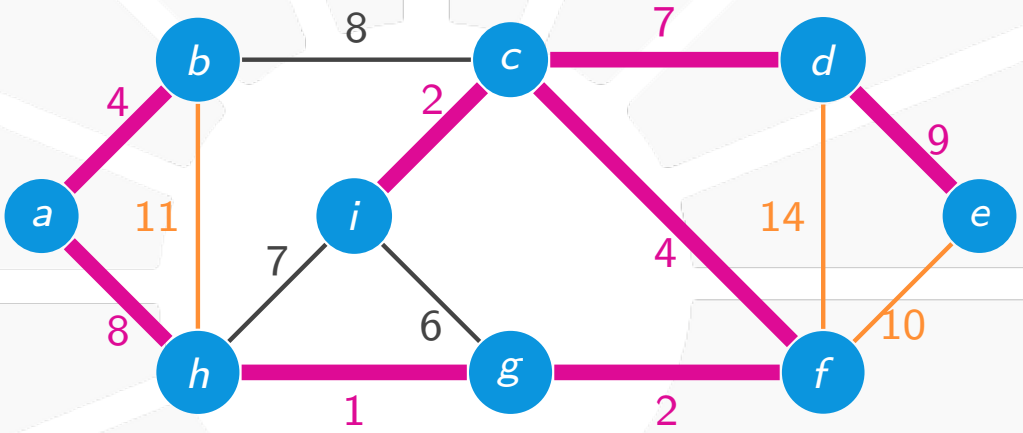


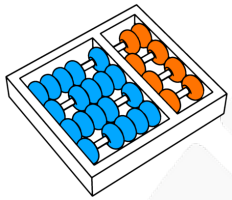
Exemplo



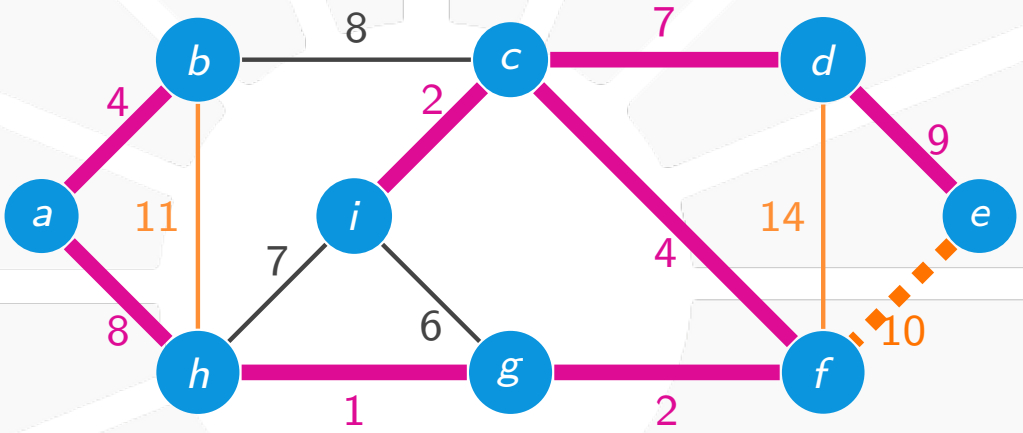


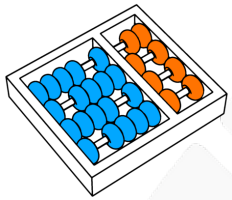
Exemplo



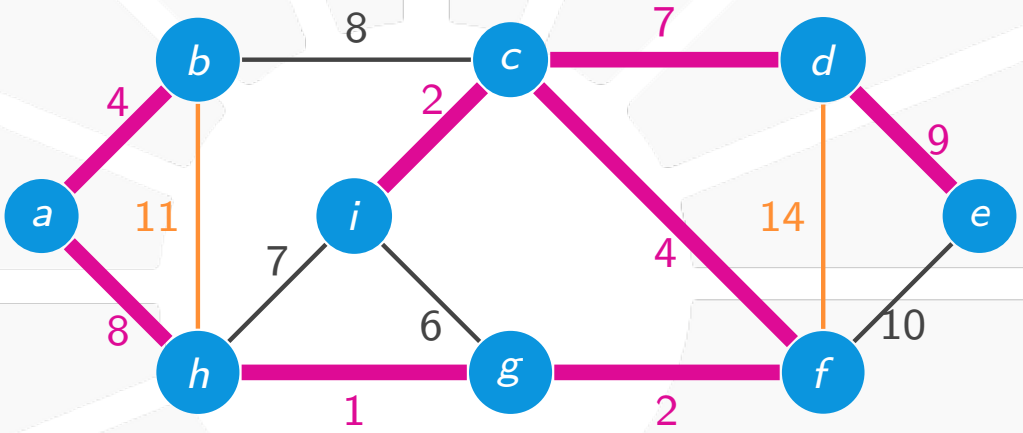


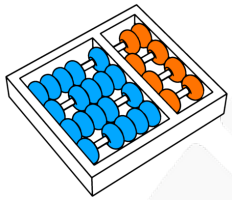
Exemplo



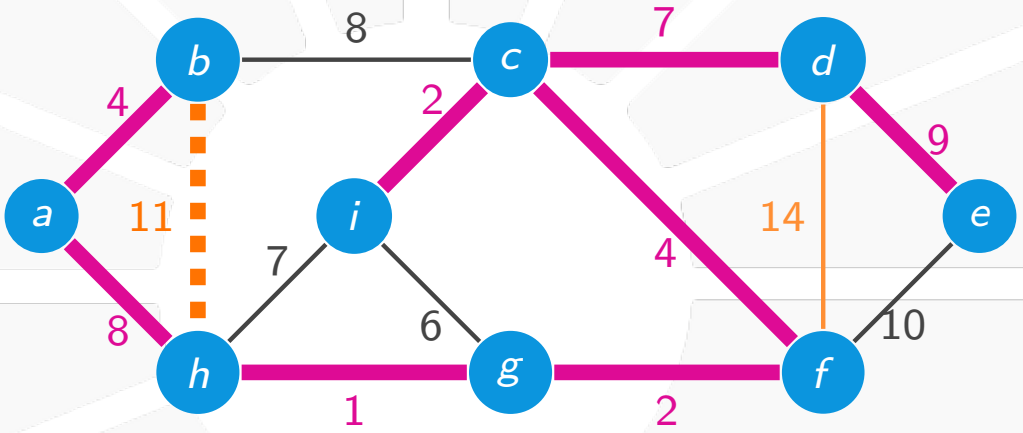


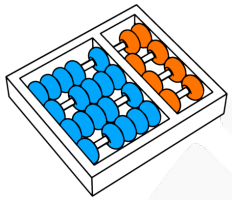
Exemplo



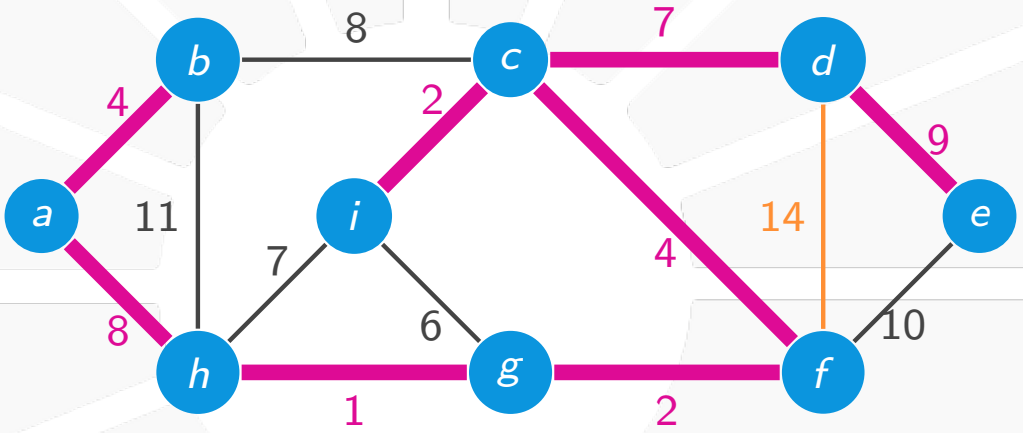


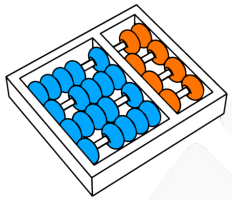
Exemplo



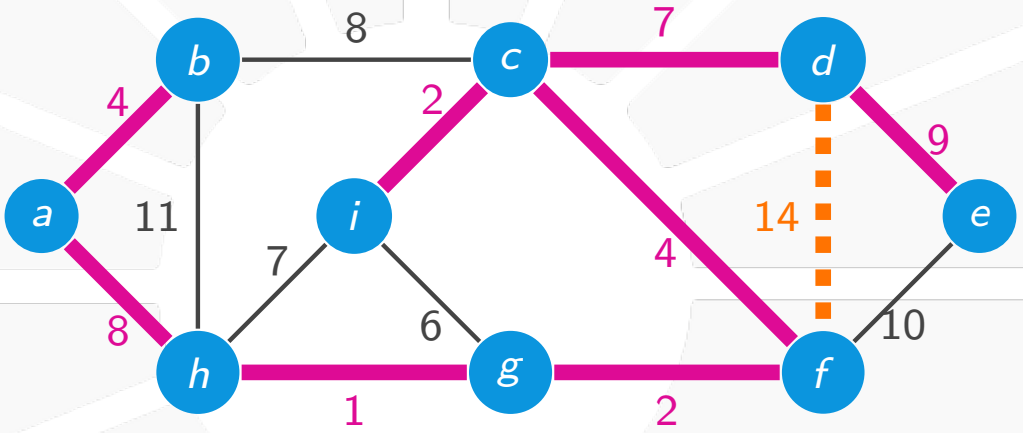


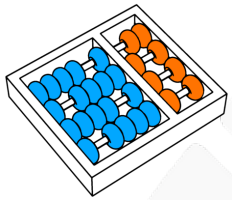
Exemplo



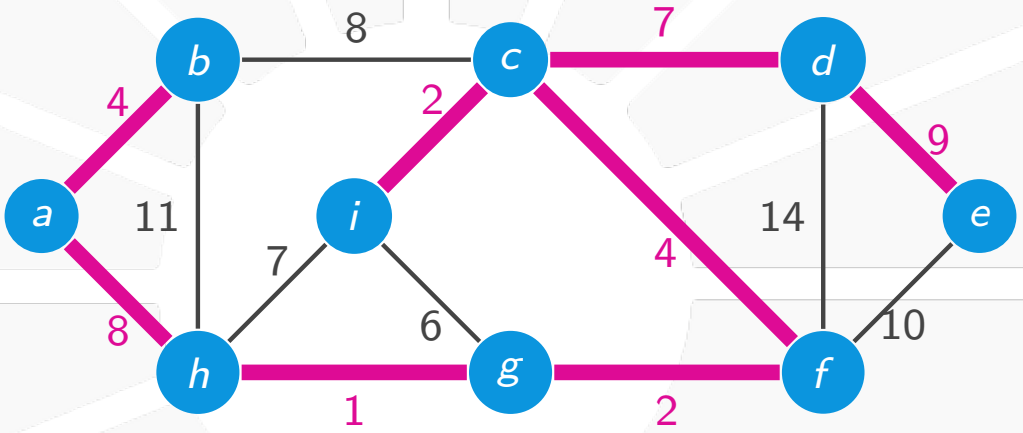


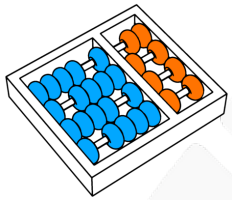
Exemplo





Exemplo



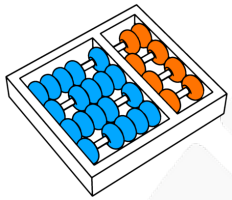


O algoritmo

Algoritmo 7: AGM-KRUSKAL(G, w)

```
1 A ← ∅
2 ordene as arestas em ordem não decrescente de peso
3 para cada (u, v) ∈ E[G] na ordem obtida
4     se u e v estão em componentes distintas de (V, A)
5         A ← A ∪ {(u, v)}
6 devolva A
```

Esta é uma versão preliminar do algoritmo:



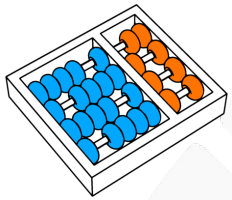
O algoritmo

Algoritmo 8: AGM-KRUSKAL(G, w)

```
1 A ← ∅
2 ordene as arestas em ordem não decrescente de peso
3 para cada (u, v) ∈ E[G] na ordem obtida
4     se u e v estão em componentes distintas de (V, A)
5         A ← A ∪ {(u, v)}
6 devolva A
```

Esta é uma versão preliminar do algoritmo:

- Falta detalhar a implementação da linha 4.



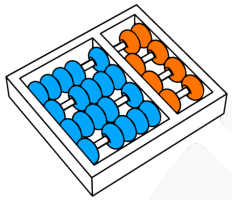
O algoritmo

Algoritmo 9: AGM-KRUSKAL(G, w)

```
1 A ← ∅
2 ordene as arestas em ordem não decrescente de peso
3 para cada (u, v) ∈ E[G] na ordem obtida
4     se u e v estão em componentes distintas de (V, A)
5         A ← A ∪ {(u, v)}
6 devolva A
```

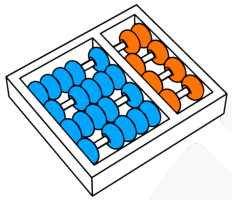
Esta é uma versão preliminar do algoritmo:

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



Estrutura de dados

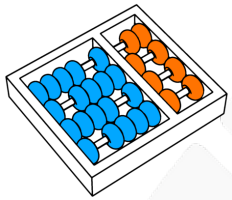
Como guardar a floresta sendo construída?



Estrutura de dados

Como guardar a floresta sendo construída?

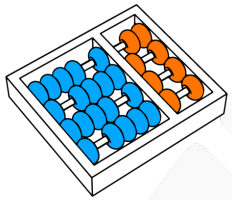
- Basta guardar o conjunto **A** das arestas.



Estrutura de dados

Como guardar a floresta sendo construída?

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

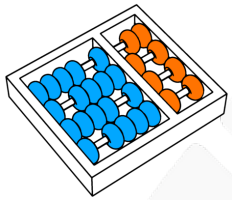


Estrutura de dados

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?



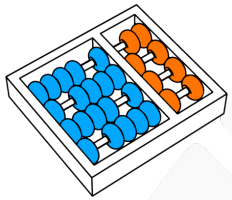
Estrutura de dados

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:.



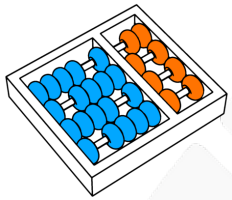
Estrutura de dados

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**.



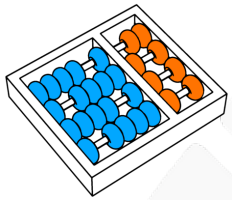
Estrutura de dados

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**.



Estrutura de dados

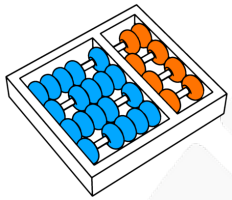
Como guardar a floresta sendo construída?

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

Como representar as componentes de G_A?

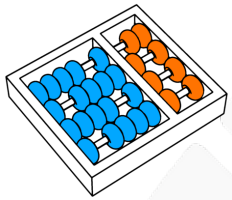
- Durante o algoritmo, as componentes de G_A mudam e precisamos:.
  - **DETERMINAR** qual componente contém vértice **u**.
  - **FAZER A UNIÃO** das componentes que contêm **u** e **v**.

Qual estrutura realiza essas operações eficientemente?



Conjuntos disjuntos

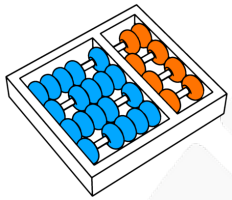
Queremos uma estrutura de dados que:



Conjuntos disjuntos

Queremos uma estrutura de dados que:

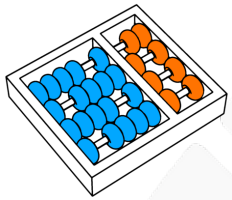
- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ de **CONJUNTOS DISJUNTOS**.



Conjuntos disjuntos

Queremos uma estrutura de dados que:

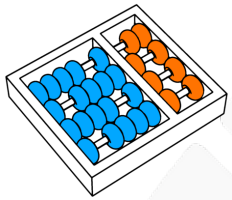
- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ de **CONJUNTOS DISJUNTOS**.
- Permita remover ou adicionar conjuntos à tal coleção.



Conjuntos disjuntos

Queremos uma estrutura de dados que:

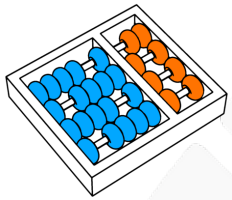
- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ de **CONJUNTOS DISJUNTOS**.
- Permita remover ou adicionar conjuntos à tal coleção.
- Identifique cada conjunto por um **REPRESENTANTE**:



Conjuntos disjuntos

Queremos uma estrutura de dados que:

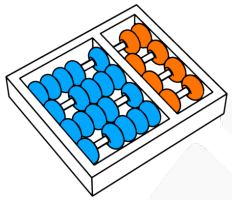
- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ de **CONJUNTOS DISJUNTOS**.
- Permita remover ou adicionar conjuntos à tal coleção.
- Identifique cada conjunto por um **REPRESENTANTE**:
  - O representante é um elemento do próprio conjunto.



Conjuntos disjuntos

Queremos uma estrutura de dados que:

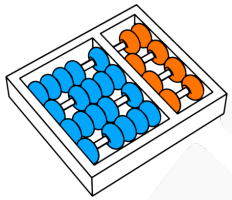
- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ de **CONJUNTOS DISJUNTOS**.
- Permita remover ou adicionar conjuntos à tal coleção.
- Identifique cada conjunto por um **REPRESENTANTE**:
  - O representante é um elemento do próprio conjunto.
  - A escolha do representante é irrelevante.



Conjuntos disjuntos

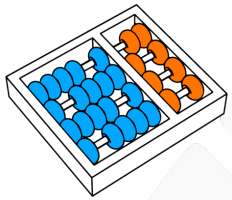
Queremos uma estrutura de dados que:

- Mantenha uma coleção $S_1, S_2, \ldots, S_k$ 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

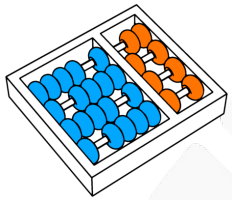
A estrutura de dados deve permitir as seguintes operações:



Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

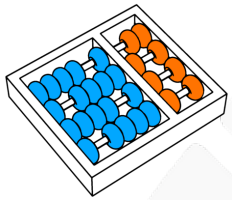
1. MAKE-SET(x): cria um novo conjunto $\{x\}$.



Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

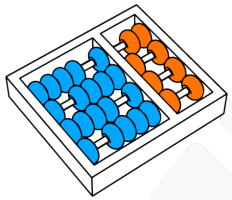
1. MAKE-SET(x): cria um novo conjunto $\{x\}$.
2. UNION(x, y): une os conjuntos que contêm x e y.



Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

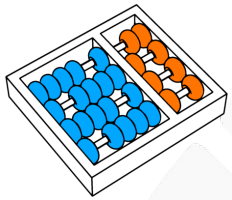
1. MAKE-SET(x): cria um novo conjunto $\{x\}$.
2. UNION(x, y): une os conjuntos que contêm x e y.
   - Se esses conjuntos forem S_x e S_y,



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 $S_x \cup S_y$

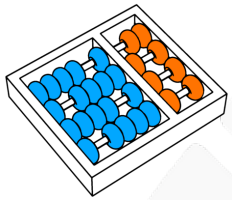


Conjuntos disjuntos

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.



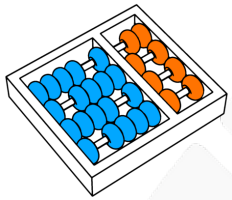
Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

1. MAKE-SET(x): cria um novo conjunto $\{x\}$.

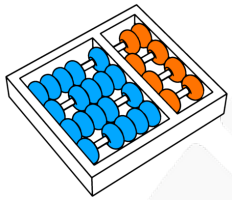
2. UNION(x, y): une os conjuntos que contêm x e y.
   - Se esses conjuntos forem S_x e S_y,
   - então adicionamos o conjunto $S_x \cup S_y$
   - e descartamos S_x e S_y da coleção.

3. FIND-SET(x): devolve o representante do conjunto que contém x.



Exemplo de aplicação

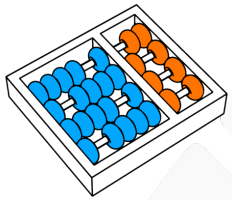
Vamos determinar as componentes conexas de um grafo G



Exemplo de aplicação

Vamos determinar as componentes conexas de um grafo G

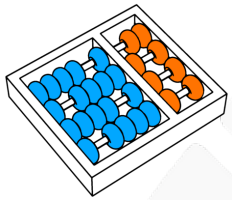
- Primeiro, vamos utilizar a estrutura de dados para conjuntos disjuntos para representar as componentes.



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.



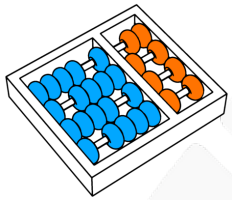
Componentes conexas

Algoritmo 10: CONNECTED-COMPONENTS(G)

```
1 para cada u ∈ V[G]
2     MAKE-SET(u)
3 para cada (u, v) ∈ E[G]
4     se FIND-SET(u)≠FIND-SET(v)
5         UNION(u, v)
```

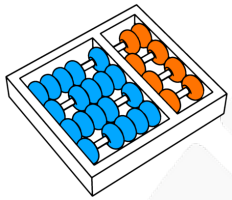
Algoritmo 11: SAME-COMPONENT(u, v)

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



Componentes conexas

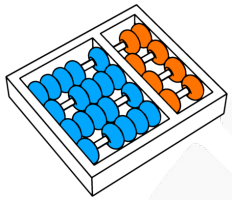
A complexidade depende da implementação:



Componentes conexas

A complexidade depende da implementação:

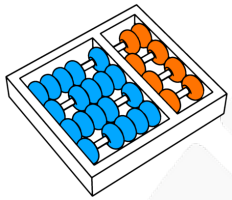
- $|V|$ chamadas a MAKE-SET.



Componentes conexas

A complexidade depende da implementação:

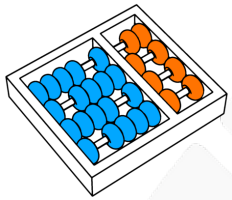
- $|V|$ chamadas a MAKE-SET.
- $2|E|$ chamadas a FIND-SET.



Componentes conexas

A complexidade depende da implementação:

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

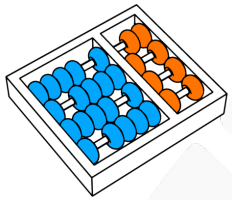


O algoritmo de Kruskal

Agora escrevemos a versão completa do algoritmo de Kruskal:

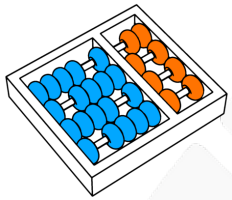
Algoritmo 12: AGM-KRUSKAL(G, w)

```
1 A ← ∅
2 para cada u ∈ V[G]
3     MAKE-SET(u)
4 ordene as arestas em ordem não decrescente de peso
5 para cada (u, v) ∈ E[G] na ordem obtida
6     se FIND-SET(u)≠FIND-SET(v)
7         A ← A ∪ {(u, v)}
8         UNION(u, v)
9 devolva A
```



Complexidade do algoritmo

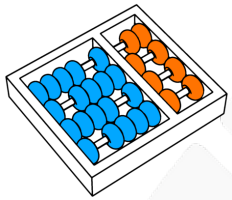
De novo, a complexidade depende da estrutura de dados:



Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados:

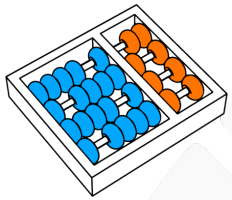
- A ordenação toma tempo $O(E \log E)$.



Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados:

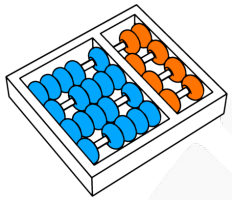
- A ordenação toma tempo $O(E \log E)$.
- $|V|$ chamadas a MAKE-SET.



Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados:

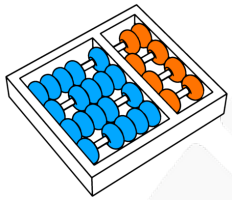
- A ordenação toma tempo $O(E \log E)$.
- $|V|$ chamadas a MAKE-SET.
- $2|E|$ chamadas a FIND-SET.



Complexidade do algoritmo

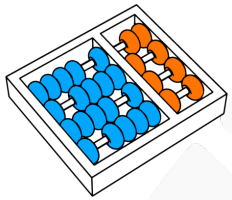
De novo, a complexidade depende da estrutura de dados:

- 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

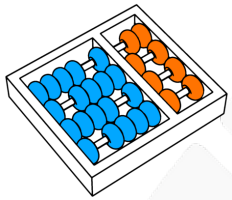
Sequência de chamadas MAKE-SET, UNION e FIND-SET:



Complexidade da operações realizadas

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

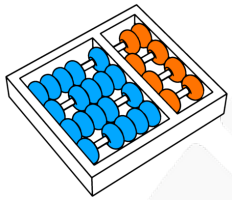
- **n** chamadas a MAKE-SET.



Complexidade da operações realizadas

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

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



Complexidade da operações realizadas

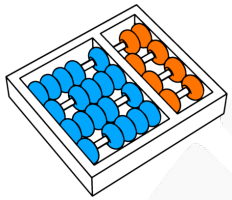
Sequência de chamadas MAKE-SET, UNION e FIND-SET:

- **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



Complexidade da operações realizadas

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

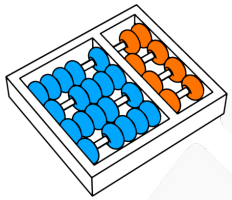
- **n** chamadas a MAKE-SET.
- **m** chamadas no total.

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

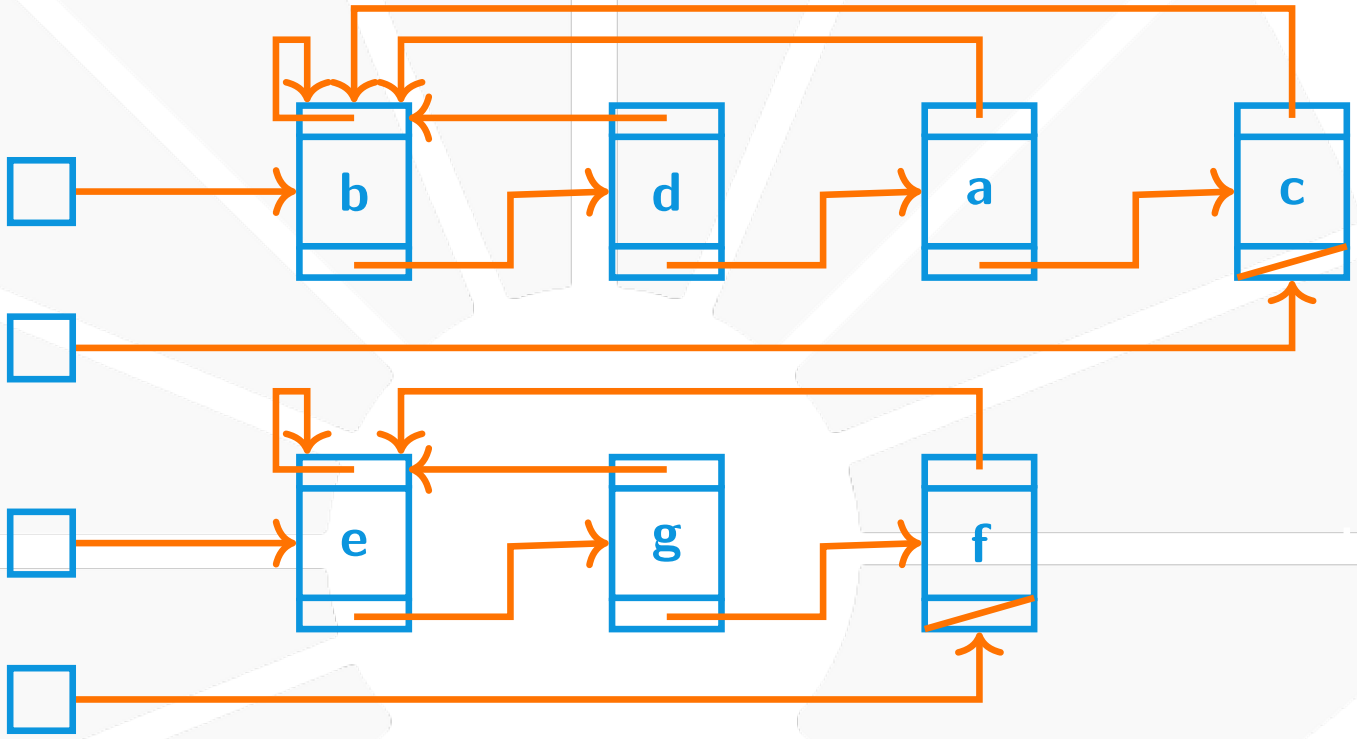
n

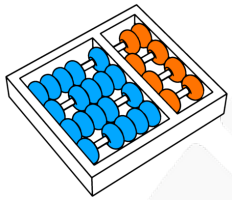
m

Queremos medir a complexidade em termos de **n** e **m**.

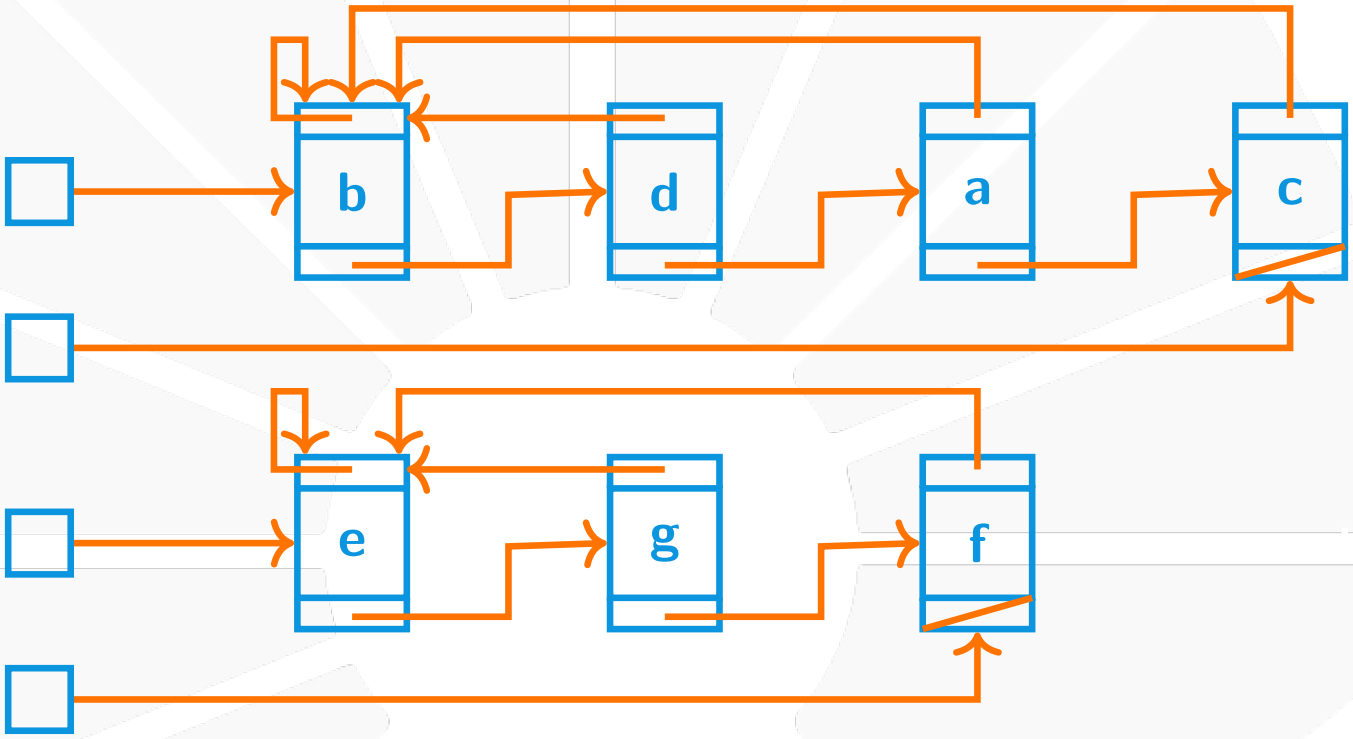


Representação por listas ligadas

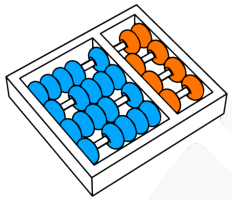




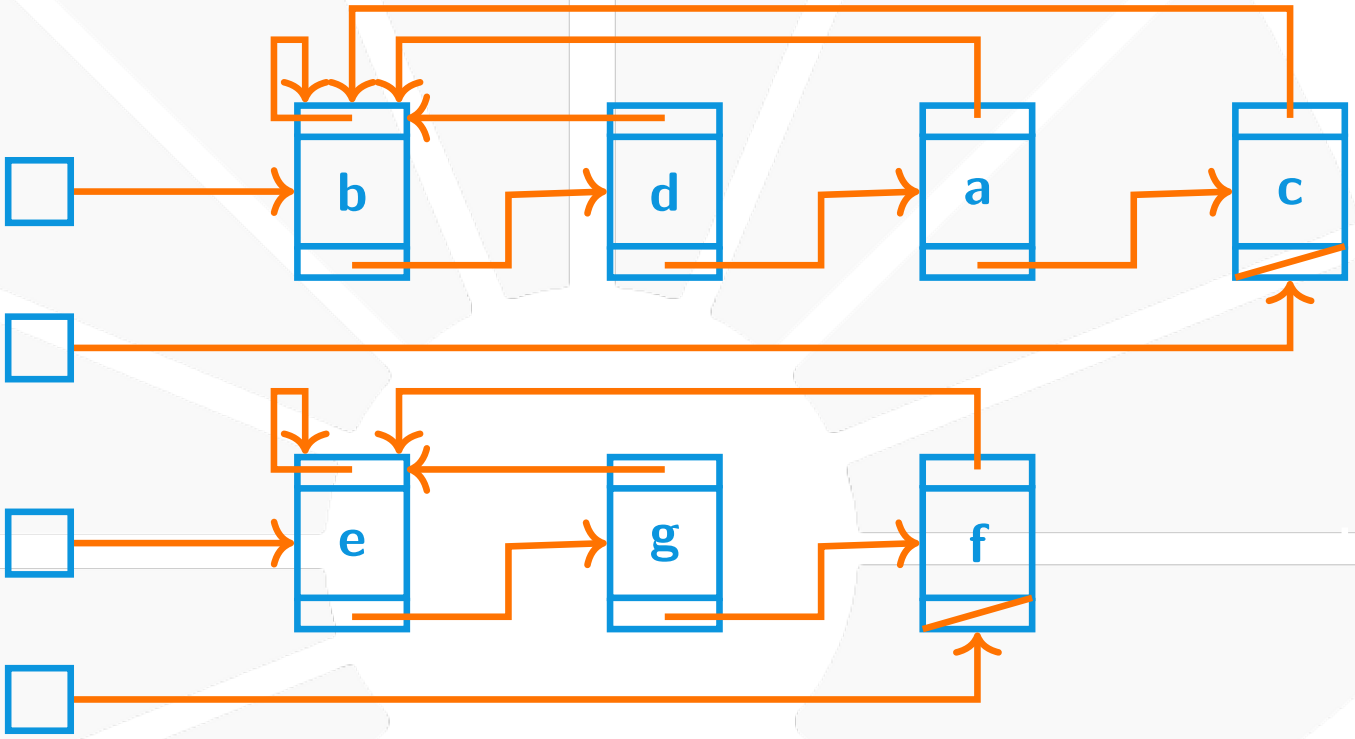
Representação por listas ligadas



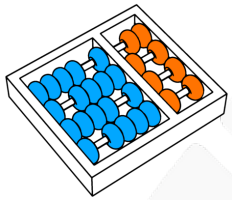
- Cada conjunto tem um representante (início da lista).



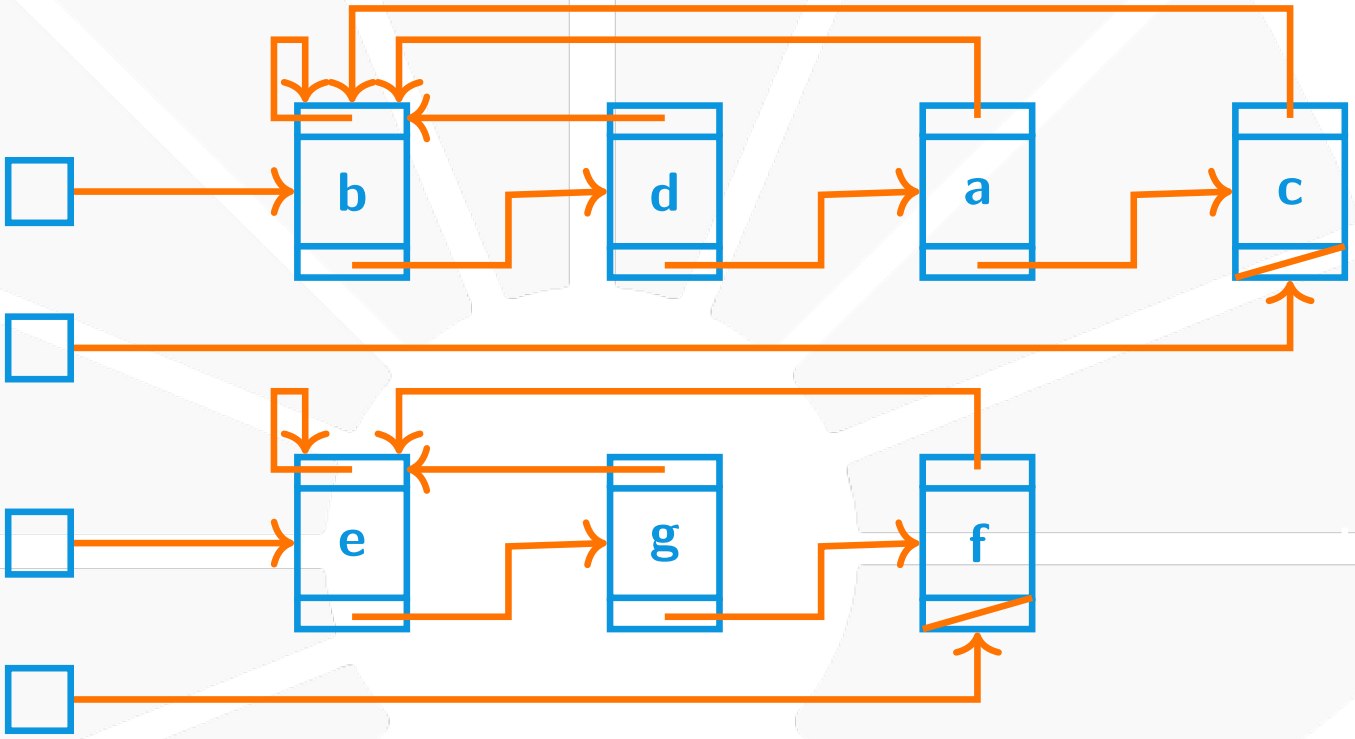
Representação por listas ligadas



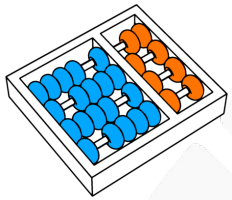
- Cada conjunto tem um representante (início da lista).
- Cada nó tem um campo que aponta para o representante.



Representação por listas ligadas

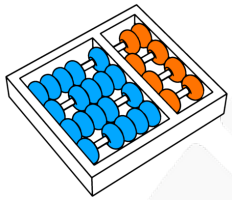


- Cada conjunto tem um representante (início da lista).
- Cada nó tem um campo que aponta para o representante.
- Guarda-se um apontador para o fim de cada lista.



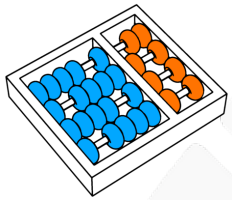
Complexidade usando listas ligadas

- MAKE-SET(x) – $O(1)$.



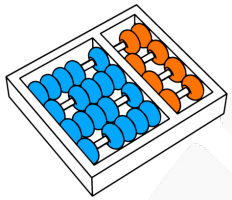
Complexidade usando listas ligadas

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(1)$.



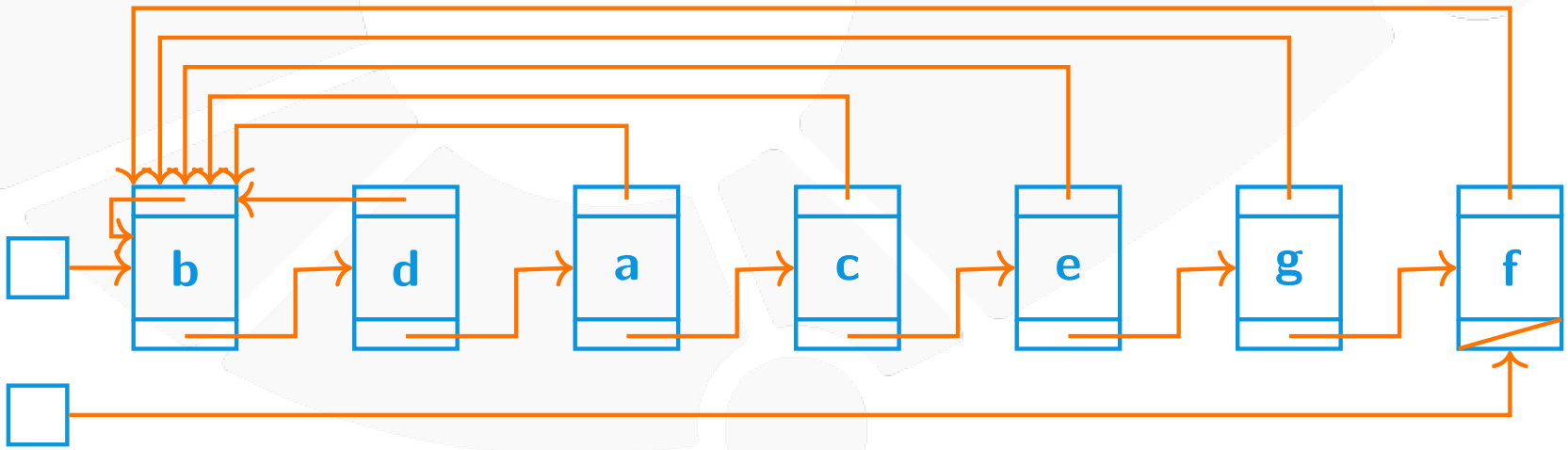
Complexidade usando listas ligadas

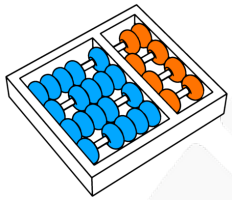
- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(1)$.
- UNION(x, y) – $O(n)$: Temos que concatenar a lista de y no final da lista de x e atualizar os apontadores para o representante.



Complexidade usando listas ligadas

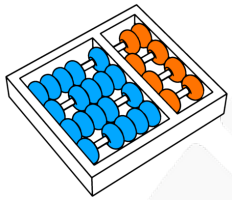
- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(1)$.
- UNION(x, y) – $O(n)$: Temos que concatenar a lista de y no final da lista de x e atualizar os apontadores para o representante.





Um exemplo de pior caso

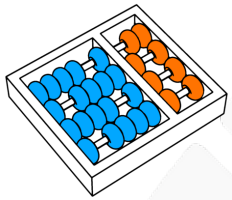
Chamada a operação	Número de atualizações
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$



Um exemplo de pior caso

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

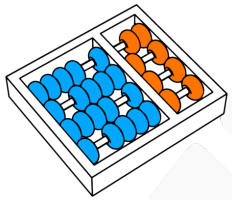
- O número de chamadas a operações é $2n-1$.



Um exemplo de pior caso

Chamada a operação	Número de atualizações
$\textsc{Make-Set}(x_1)$	1
$\textsc{Make-Set}(x_2)$	1
$\vdots$	$\vdots$
$\textsc{Make-Set}(x_n)$	1
$\textsc{Union}(x_2, x_1)$	1
$\textsc{Union}(x_3, x_2)$	2
$\textsc{Union}(x_4, x_3)$	3
$\vdots$	$\vdots$
$\textsc{Union}(x_n, x_{n-1})$	$n-1$

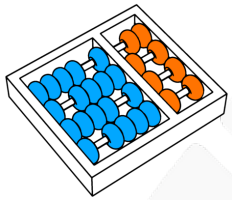
- O número de chamadas a operações é $2n-1$.
- O tempo total é $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$.



Um exemplo de pior caso

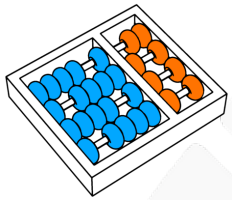
Chamada a operação	Número de atualizações
$\textsc{Make-Set}(x_1)$	1
$\textsc{Make-Set}(x_2)$	1
$\vdots$	$\vdots$
$\textsc{Make-Set}(x_n)$	1
$\textsc{Union}(x_2, x_1)$	1
$\textsc{Union}(x_3, x_2)$	2
$\textsc{Union}(x_4, x_3)$	3
$\vdots$	$\vdots$
$\textsc{Union}(x_n, x_{n-1})$	$n-1$

- 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

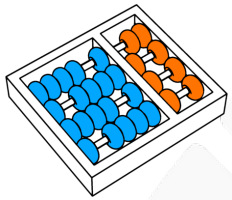
Entendendo o pior caso



Uma heurística simples

Entendendo o pior caso

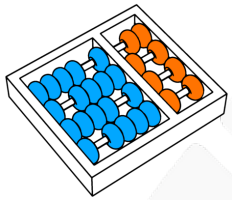
- Cada chamada a UNION gasta em média tempo $\Theta(n)$.



Uma heurística simples

Entendendo o pior caso

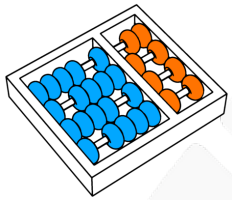
- Cada chamada a UNION gasta em média tempo $\Theta(n)$.
- Isso porque concatenamos a maior lista no final da menor.



Uma heurística simples

Entendendo o pior caso

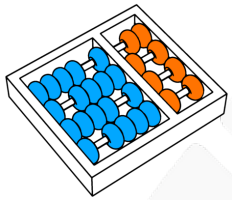
- Cada chamada a UNION gasta em média tempo $\Theta(n)$.
- Isso porque concatenamos a maior lista no final da menor.
- Para evitar isso, podemos concatenar a **MENOR** lista no final.



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***.

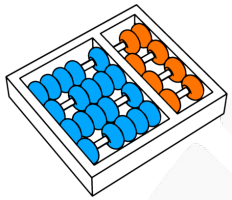


Uma heurística simples

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:



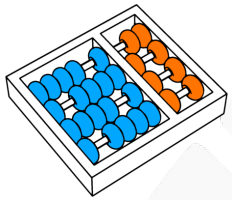
Uma heurística simples

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.



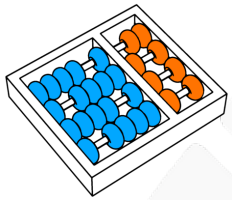
Uma heurística simples

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 $\Theta(n)$.



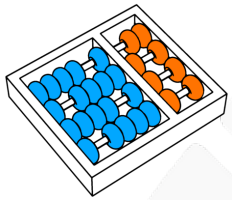
Uma heurística simples

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:

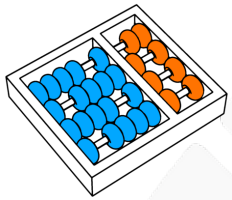
- Basta guardar o tamanho de cada lista.
- Pode ser que uma chamada a UNION leve tempo $\Theta(n)$.
- Mas isso não pode acontecer sempre.



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* $O(m + n \log n)$.

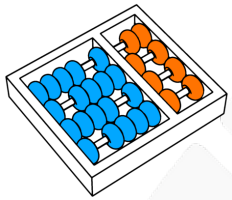


Uma heurística simples

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:



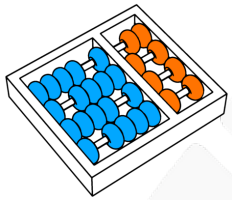
Uma heurística simples

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 $O(m)$.



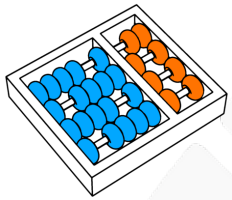
Uma heurística simples

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 atualizamos um nó, a lista que o continha pelo menos dobra.



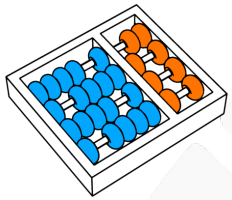
Uma heurística simples

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 atualizamos um nó, a lista que o continha pelo menos dobra.
- Mas uma lista só pode dobrar no máximo $O(\log n)$ vezes.



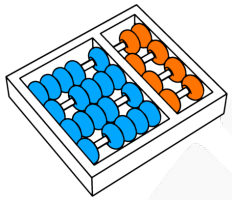
Uma heurística simples

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 atualizamos 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 e atualizado $O(\log n)$ vezes.



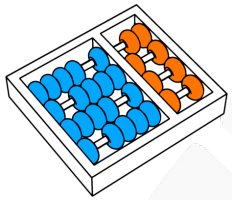
Uma heurística simples

Teorema

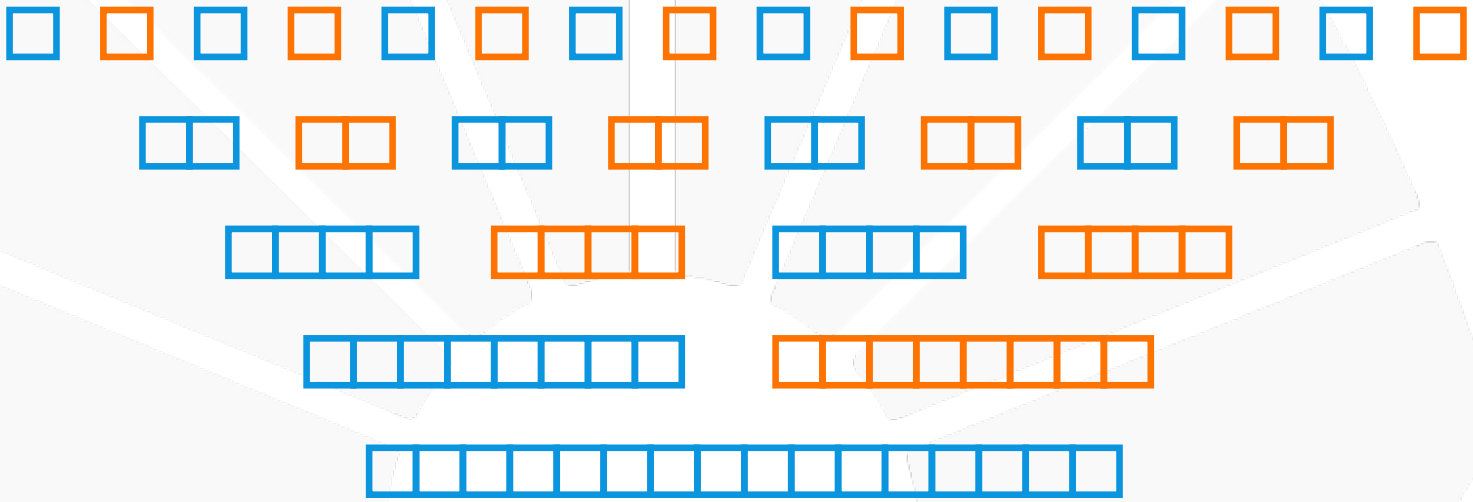
Suponha que executamos uma sequência de m chamadas a MAKE-SET, UNION *e* FIND-SET. *Se utilizarmos a representação por listas ligadas com a weighted-union heuristic, então o* ***TEMPO TOTAL*** *gasto será* $O(m + n \log n)$.

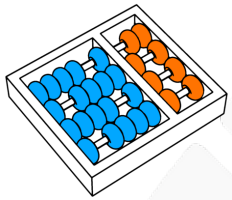
Demonstração:

- O tempo total das chamadas MAKE-SET e FIND-SET é $O(m)$.
- Ao atualizamos 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 e atualizado $O(\log n)$ vezes.
- Portanto, o tempo total com chamadas a UNION é $O(n \log n)$.

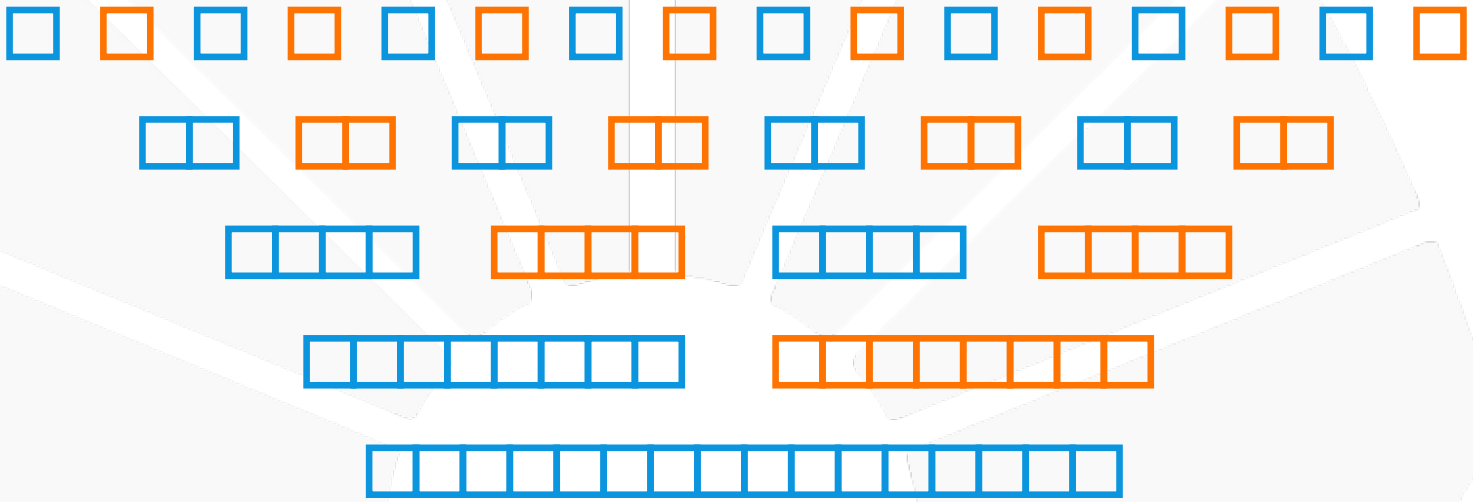


Um exemplo de pior caso

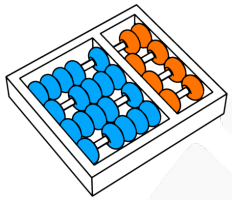




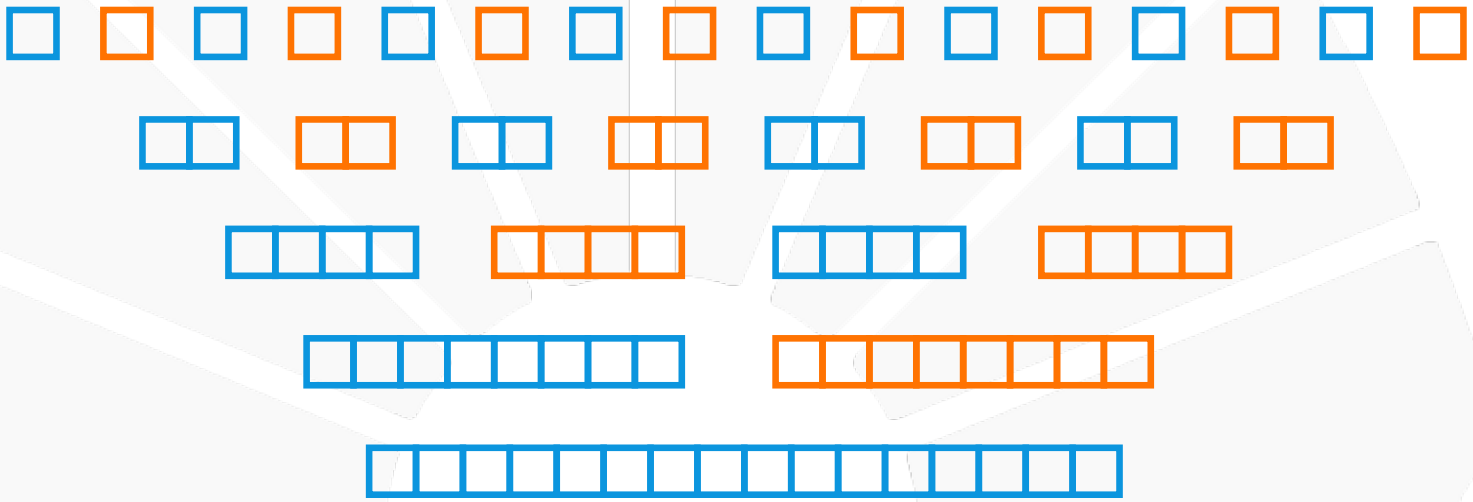
Um exemplo de pior caso



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

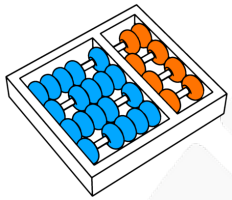


Um exemplo de pior caso

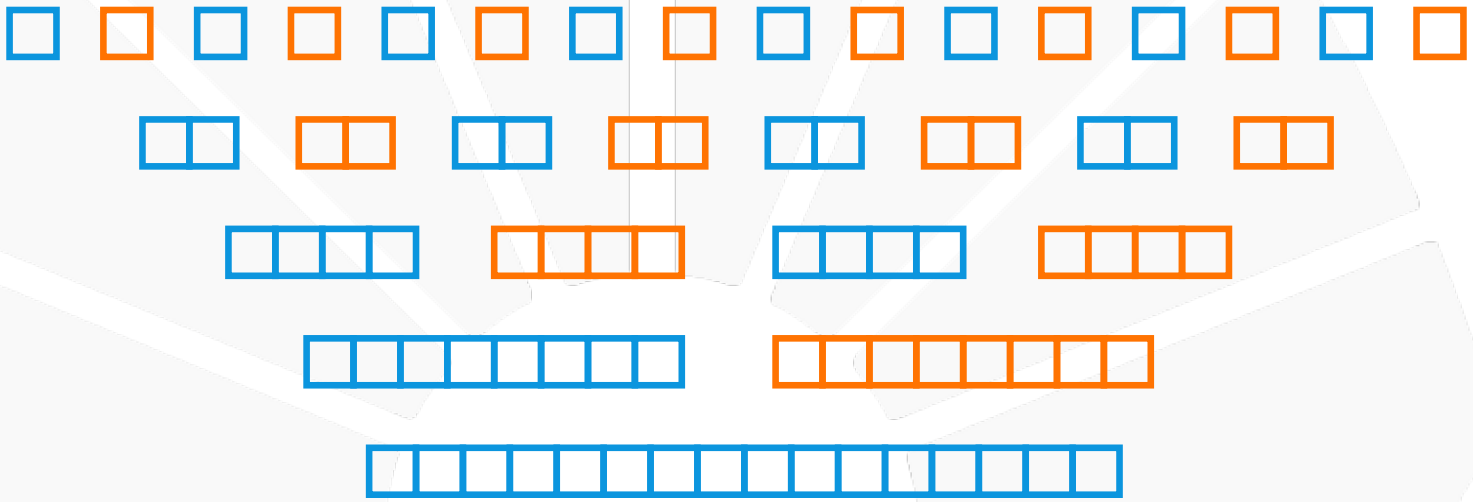


O custo total de UNION nesse exemplo é $\Theta(n \log n)$:

- Há $\Theta(\log n)$ níveis, cada um representando a coleção de conjuntos disjuntos em determinado instante.

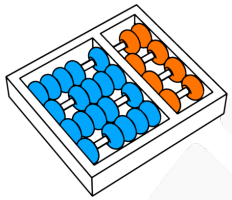


Um exemplo de pior caso

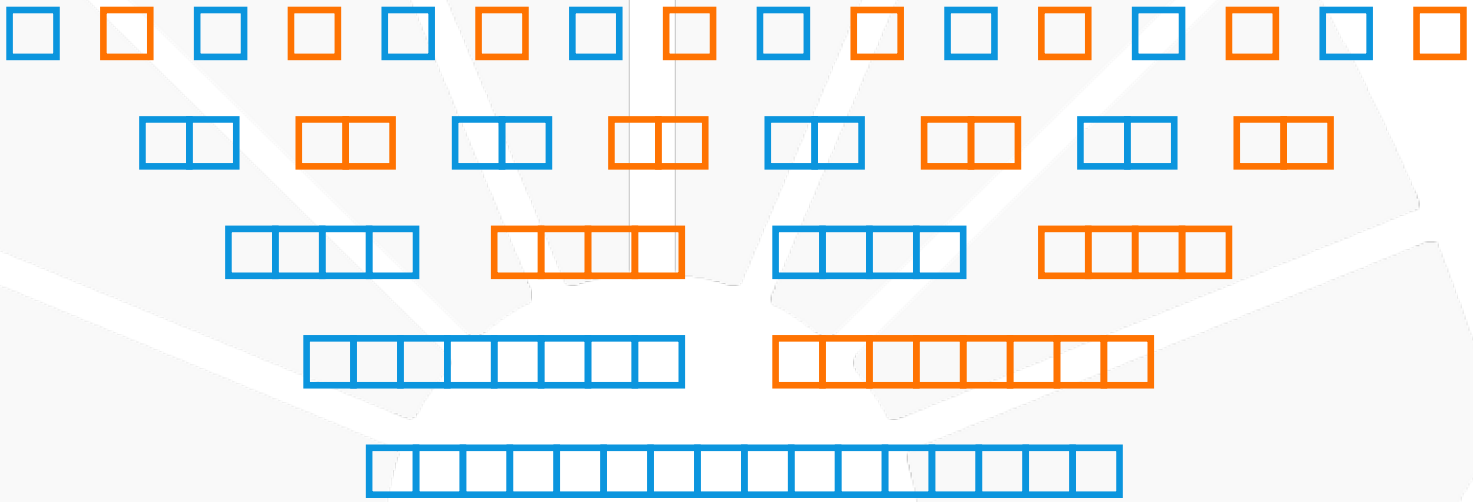


O custo total de UNION nesse exemplo é $\Theta(n \log n)$:

- 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.

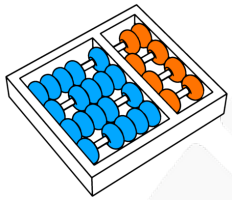


Um exemplo de pior caso



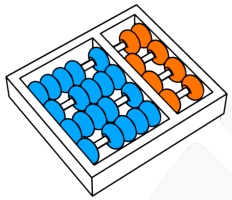
O custo total de UNION nesse exemplo é $\Theta(n \log n)$:

- 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

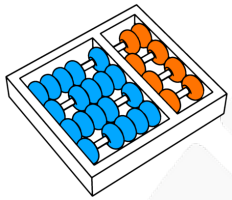
Relembrando, a complexidade de AGM-KRUSKAL é dada por:



Complexidade do algoritmo de Kruskal

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

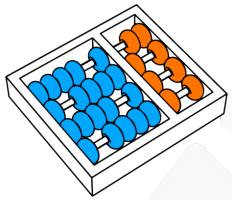
- Ordenação, que toma tempo $O(E \log E)$.



Complexidade do algoritmo de Kruskal

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

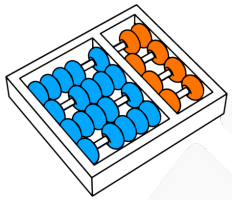
- Ordenação, que toma tempo $O(E \log E)$.
- $|V|$ chamadas a MAKE-SET.



Complexidade do algoritmo de Kruskal

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

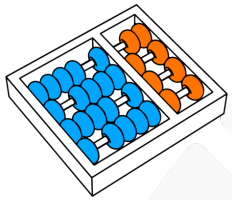
- Ordenação, que toma tempo $O(E \log E)$.
- $|V|$ chamadas a MAKE-SET.
- $2|E|$ chamadas a FIND-SET.



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.

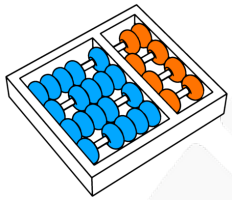


Complexidade do algoritmo de Kruskal

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:



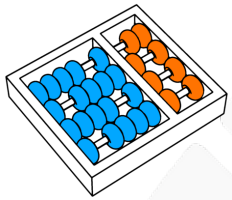
Complexidade do algoritmo de Kruskal

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:

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



Complexidade do algoritmo de Kruskal

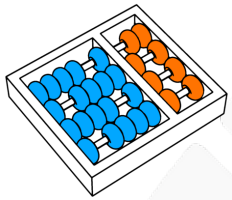
Relembrando, a complexidade de AGM-KRUSKAL é dada por:

- 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.



Complexidade do algoritmo de Kruskal

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

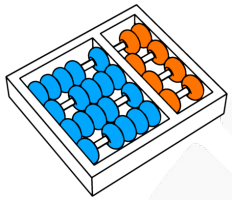
- Ordenação, que toma tempo $O(E \log E)$.
- $|V|$ chamadas a MAKE-SET.
- $2|E|$ chamadas a FIND-SET.
- $|V| - 1$ chamadas a UNION.

O tempo total utilizando a representação por listas ligadas é:

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

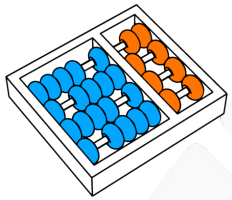
- O tempo é dominado pela ordenação das arestas.
- Se já estiverem ordenadas, então gastamos $O(E + V \log V)$.

Conjuntos disjuntos com florestas de conjuntos



Outra representação

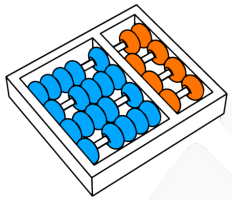
Representar uma coleção por uma floresta:



Outra representação

Representar uma coleção por uma floresta:

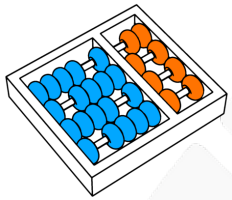
- Cada conjunto corresponde a uma árvore enraizada.



Outra representação

Representar uma coleção por uma floresta:

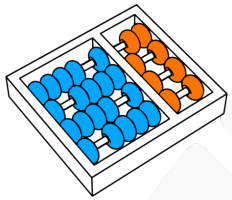
- Cada conjunto corresponde a uma árvore enraizada.
- O representante de um conjunto é a raiz.



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**.

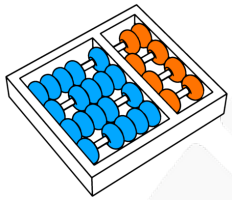


Outra representação

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:



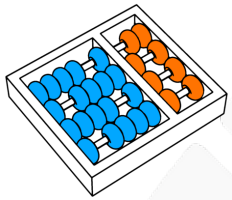
Outra representação

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:



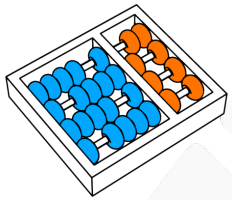
Outra representação

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.



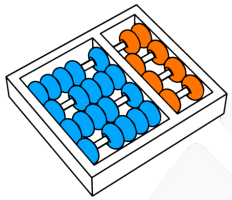
Outra representação

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).



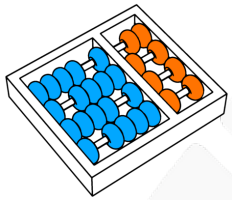
Outra representação

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:



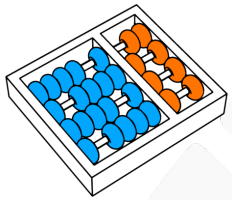
Outra representação

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**.



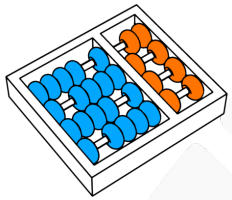
Outra representação

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.



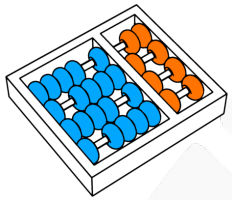
Outra representação

Representar uma coleção por uma floresta:

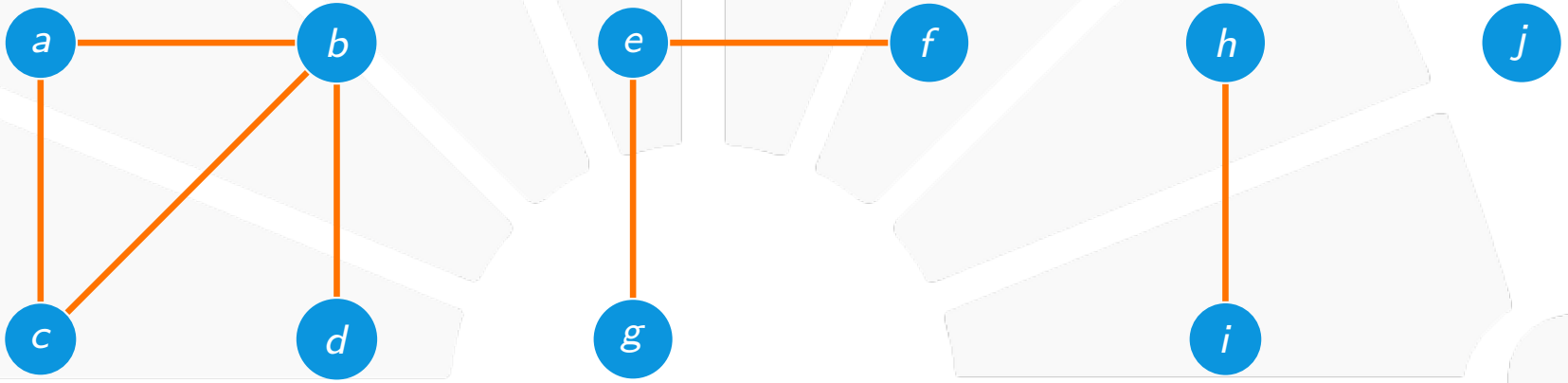
- Cada conjunto corresponde a uma árvore enraizada.
- O representante de um conjunto é a raiz.
- Tal floresta é a chamada **DISJOINT-SET FOREST**.

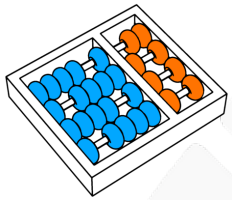
Veremos duas implementações:

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.

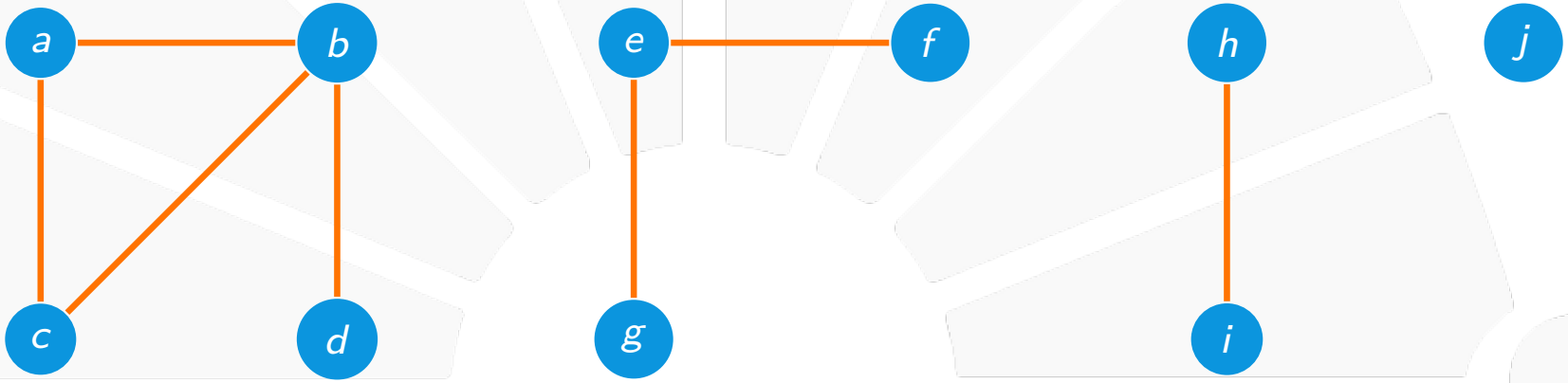


Exemplo de grafo

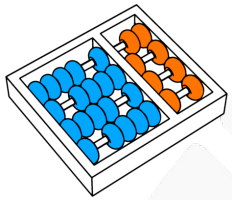




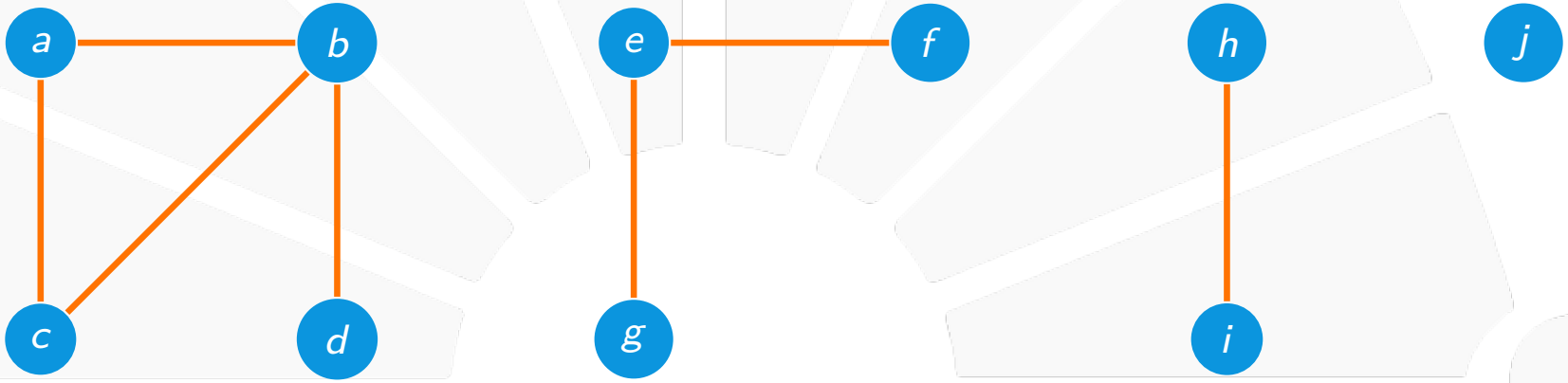
Exemplo de grafo



Veja o grafo:

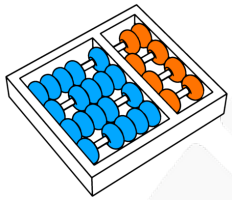


Exemplo de grafo

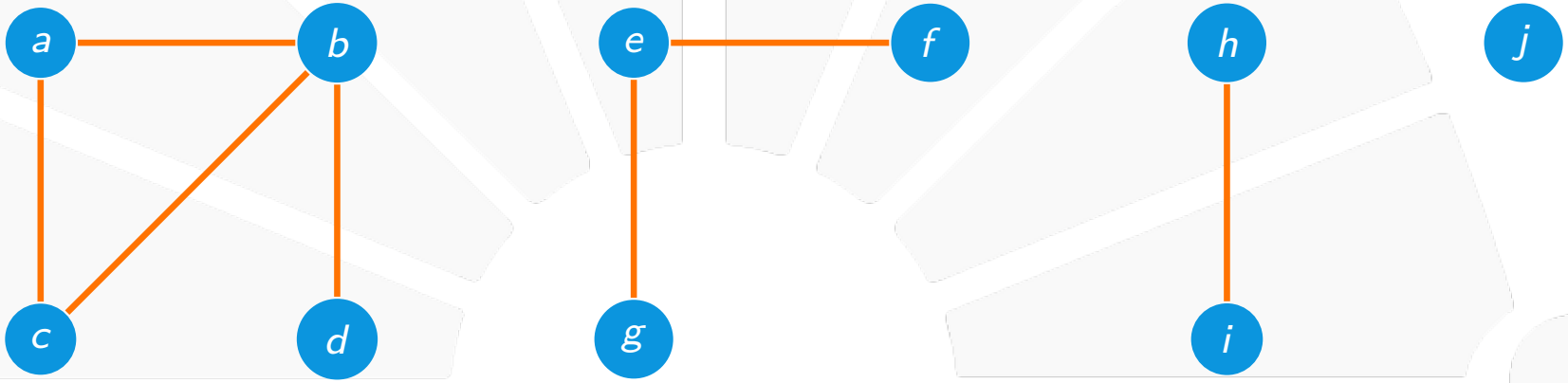


Veja o grafo:

- Considere um conjunto para cada componente.

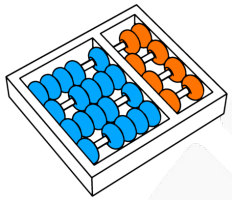


Exemplo de grafo

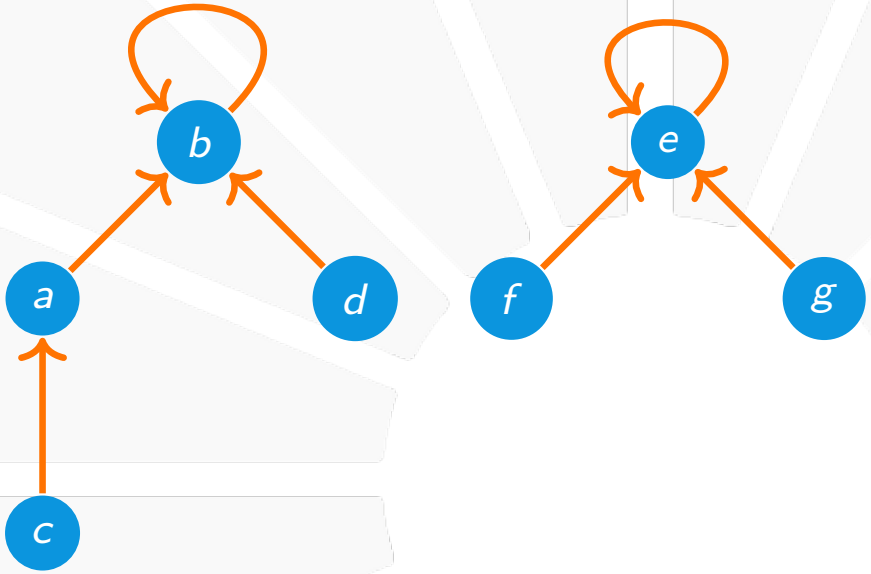


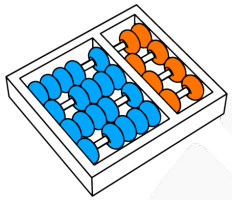
Veja o grafo:

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

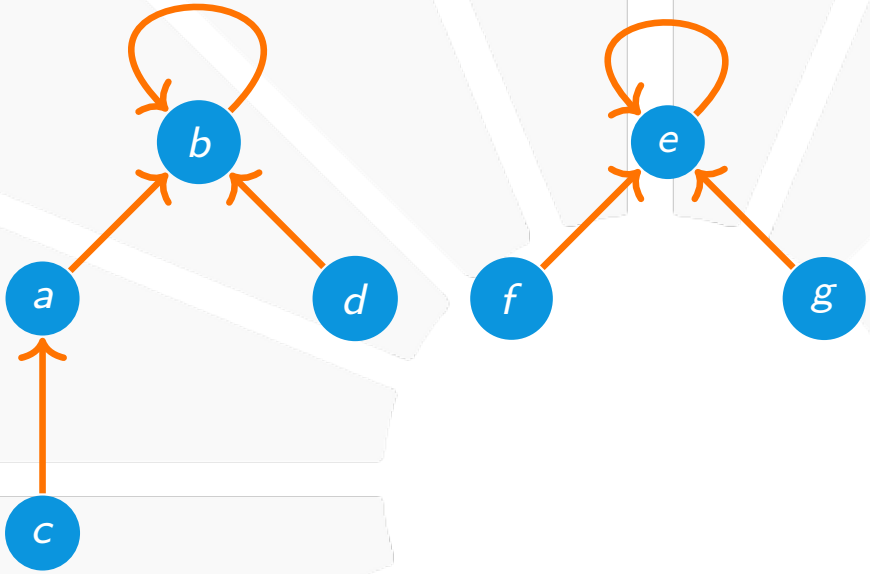


Exemplo de representação

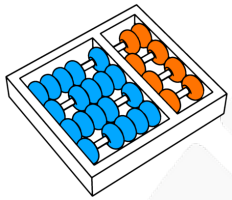




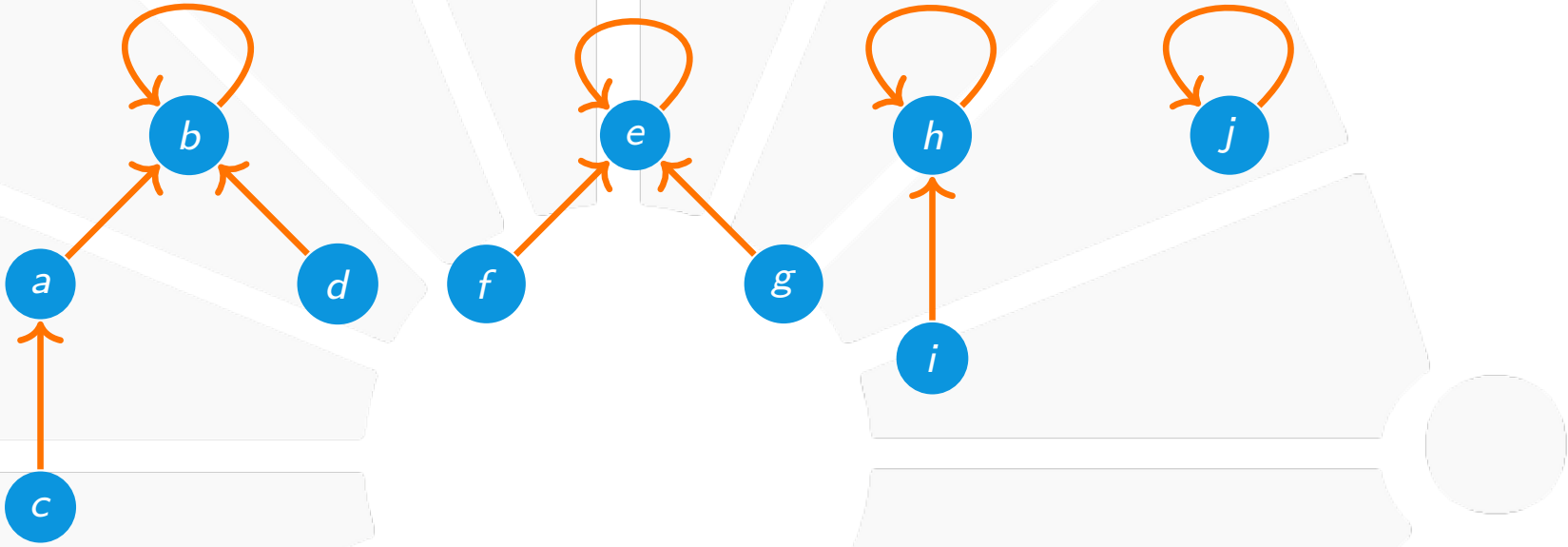
Exemplo de representação



Convenções:

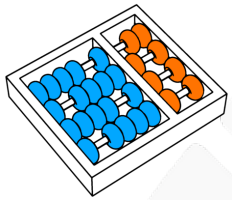


Exemplo de representação

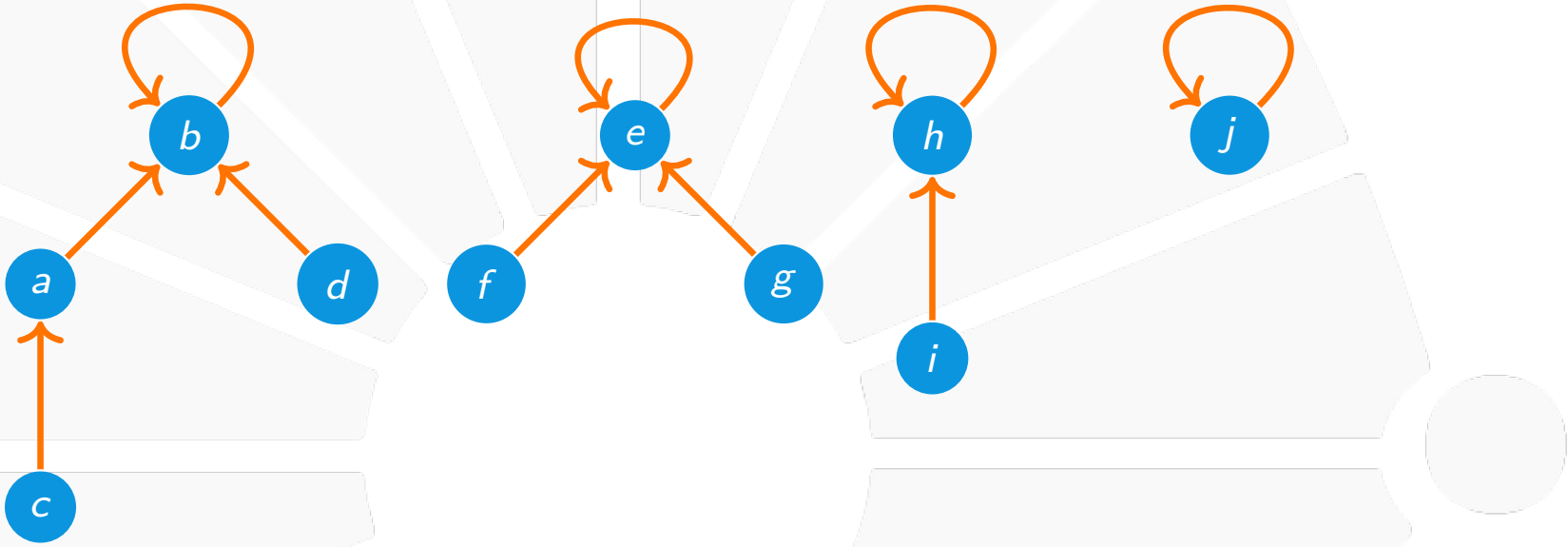


Convenções:

- Cada conjunto é uma árvore enraizada.

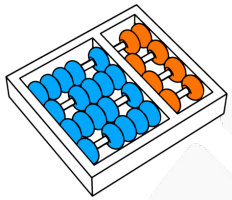


Exemplo de representação

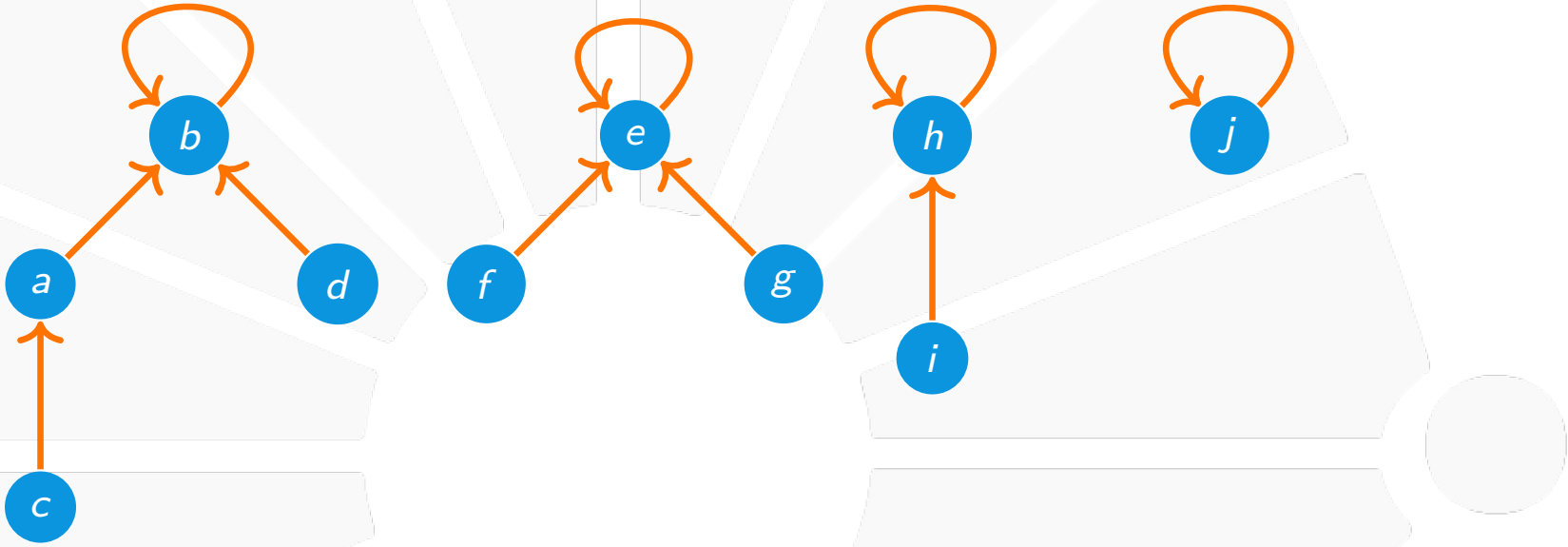


Convenções:

- Cada conjunto é uma árvore enraizada.
- Cada elemento aponta para seu pai.

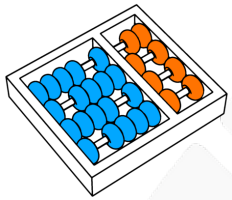


Exemplo de representação

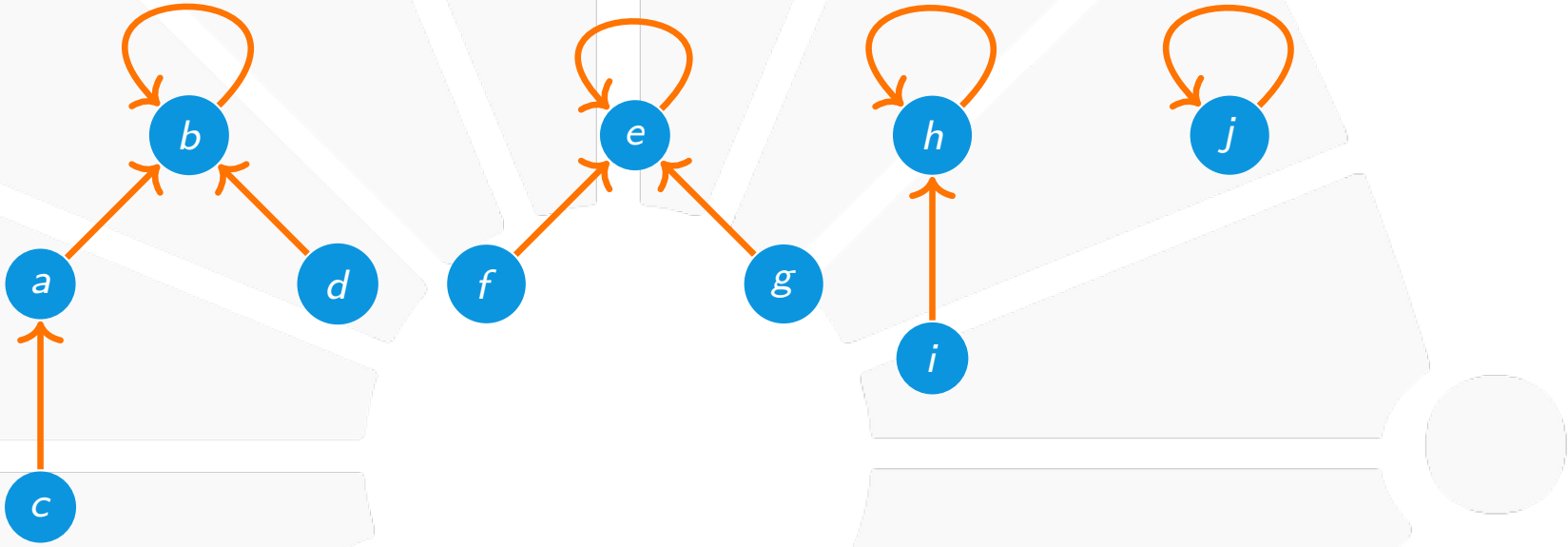


Convenções:

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

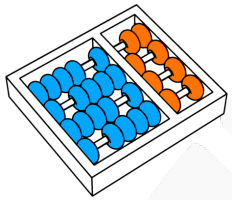


Exemplo de representação

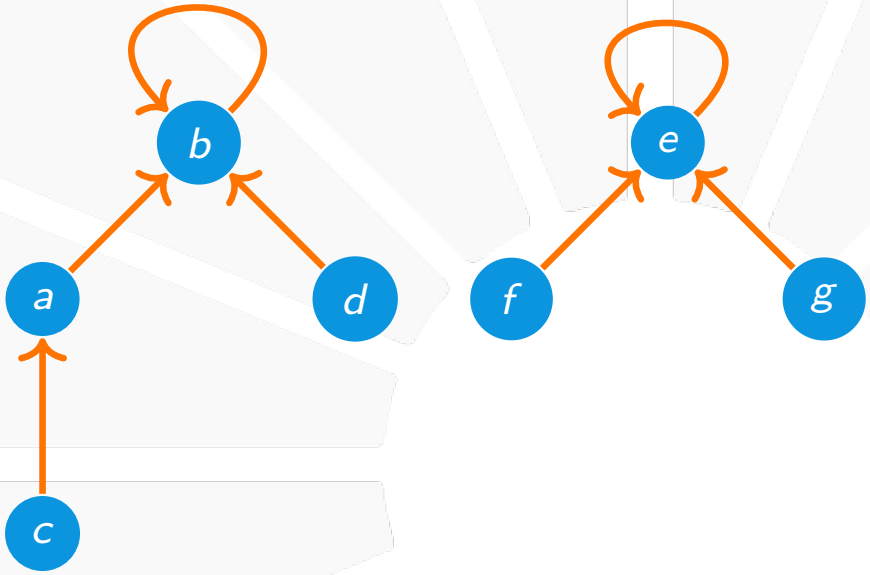


Convenções:

- 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.

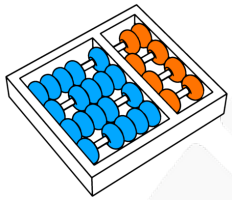


Implementação simples

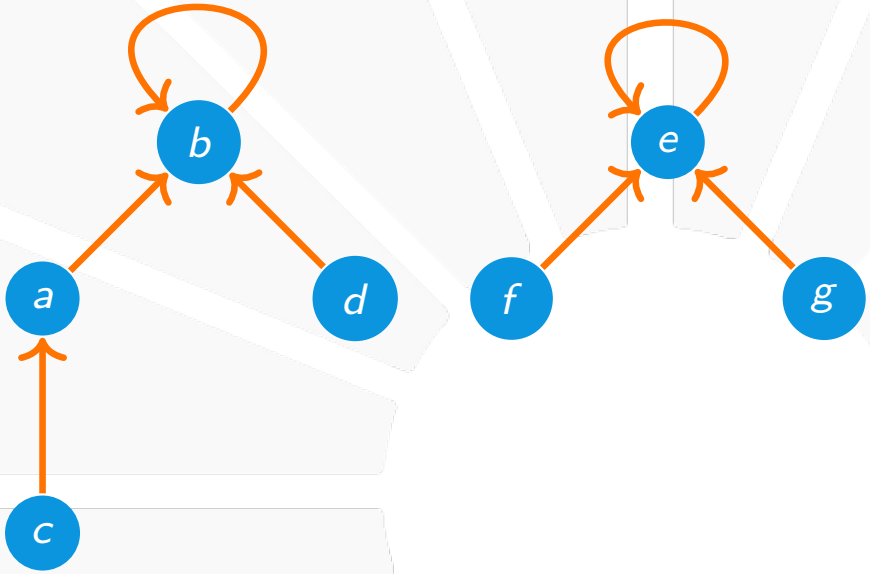


Algoritmo 13: MAKE-SET(x)

```
1 pai[x] ← x
```

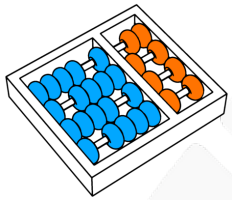


Implementação simples

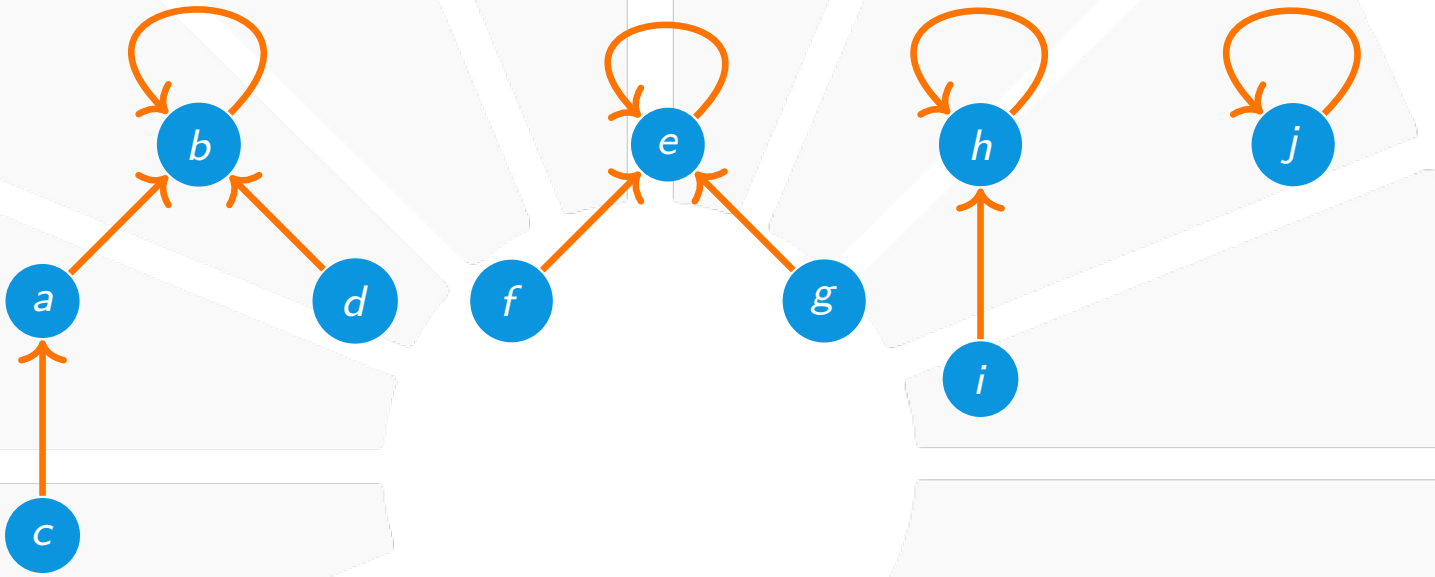


Algoritmo 14: FIND-SET(x)

```
1 se x = pai[x]
2     devolva x
3 senão
4     devolva FIND-SET(pai[x])
```

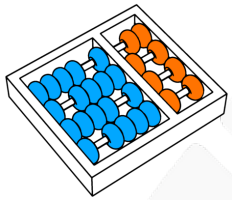


Implementação simples



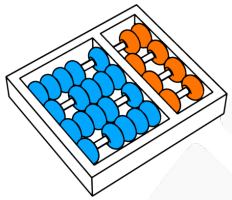
Algoritmo 15: UNION(x, y)

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



Complexidade da implementação simples

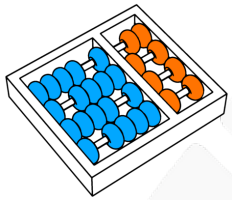
Tempo das operações:



Complexidade da implementação simples

Tempo das operações:

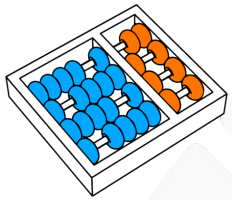
- MAKE-SET(x) – $O(1)$.



Complexidade da implementação simples

Tempo das operações:

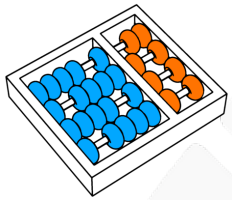
- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.



Complexidade da implementação simples

Tempo das operações:

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- UNION(x, y) – $O(n)$.

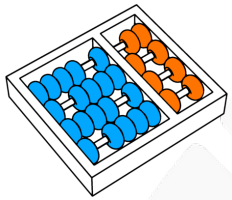


Complexidade da implementação simples

Tempo das operações:

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- UNION(x, y) – $O(n)$.

Não é melhor do que a representação por listas ligadas:



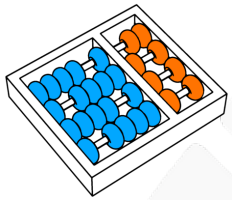
Complexidade da implementação simples

Tempo das operações:

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- 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.



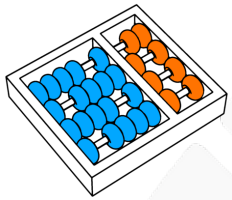
Complexidade da implementação simples

Tempo das operações:

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- 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 $\Theta(n^2)$.



Complexidade da implementação simples

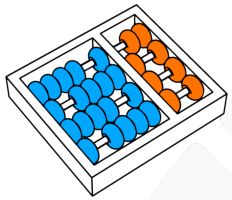
Tempo das operações:

- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- 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:



Complexidade da implementação simples

Tempo das operações:

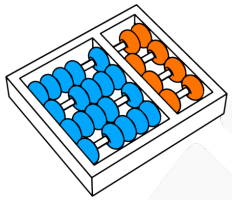
- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- UNION(x, y) – $O(n)$.

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*.



Complexidade da implementação simples

Tempo das operações:

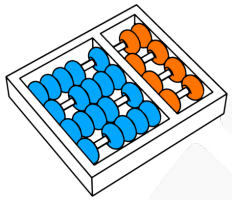
- MAKE-SET(x) – $O(1)$.
- FIND-SET(x) – $O(n)$.
- UNION(x, y) – $O(n)$.

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 $\Theta(n^2)$.

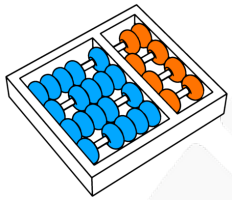
Podemos melhorar isso usando duas heurísticas:

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



Union by rank

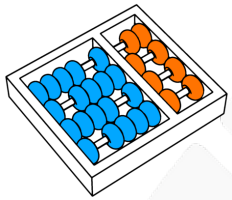
Ideia emprestada da **WEIGHTED-UNION HEURISTIC**:



Union by rank

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

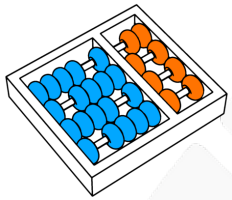
- Cada nó x está associado a um número rank$[x]$:



Union by rank

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

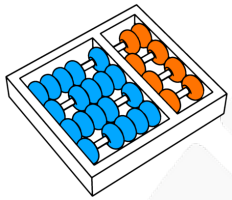
- Cada nó x está associado a um número rank$[x]$:
  - Pode ser a altura de x na árvore,



Union by rank

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

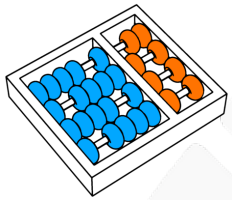
- Cada nó x está associado a um número rank$[x]$:
  - Pode ser a altura de x na árvore,
  - ou pode ser um número **MENOR**.



Union by rank

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

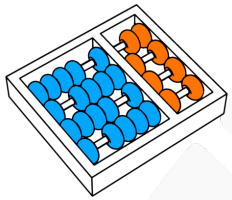
- Cada nó x está associado a um número rank$[x]$:
  - Pode ser a altura de x na árvore,
  - ou pode ser um número **MENOR**.
- A raiz com menor rank aponta para a raiz com maior rank.



Union by rank

Algoritmo 16: MAKE-SET(x)

```
1 pai[x] ← x
2 rank[x] ← 0
```



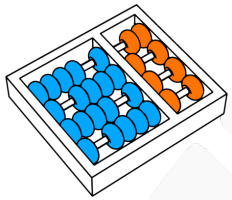
Union by rank

Algoritmo 19: MAKE-SET(x)

```
1 pai[x] ← x
2 rank[x] ← 0
```

Algoritmo 20: UNION(x, y)

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



Union by rank

Algoritmo 22: MAKE-SET(x)

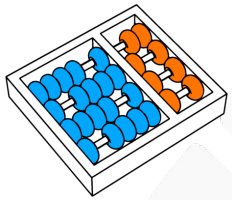
```
1 pai[x] ← x
2 rank[x] ← 0
```

Algoritmo 23: UNION(x, y)

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

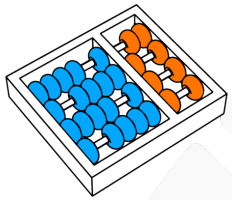
Algoritmo 24: LINK(x, y)

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



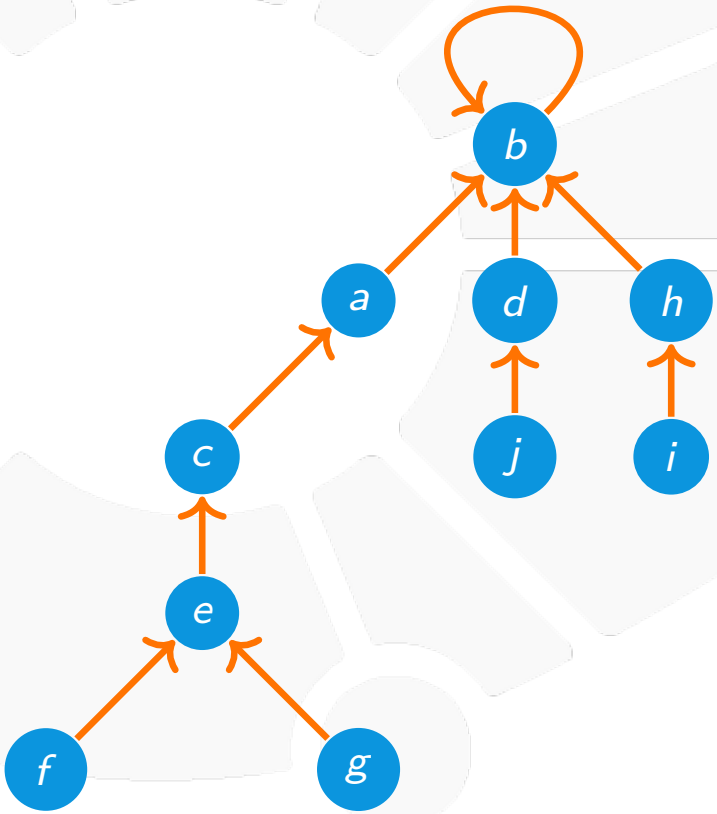
Path compression

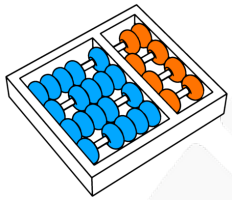
A ideia é muito simples: ao tentar determinar o representante (**RAIZ** da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.



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.

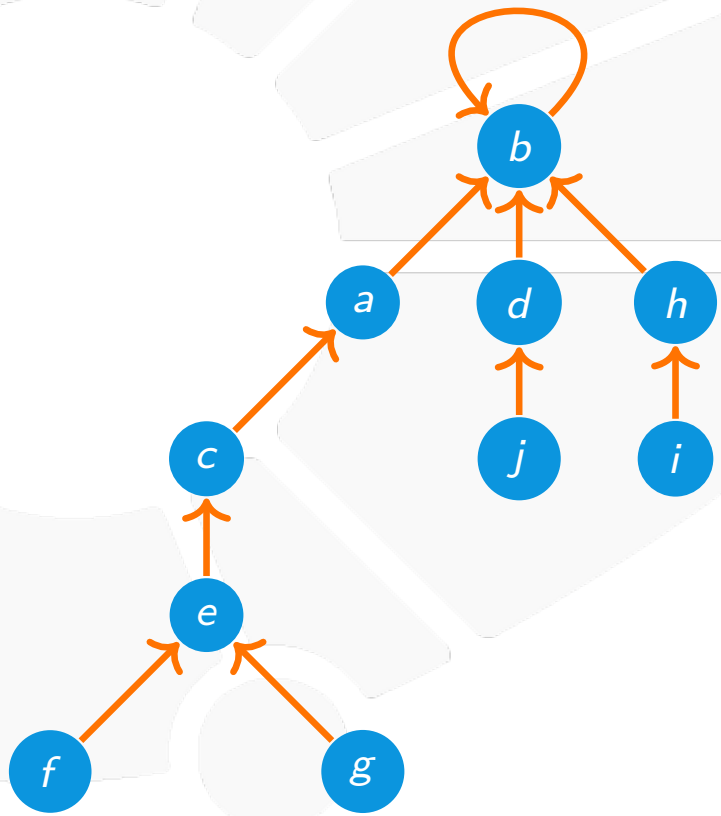


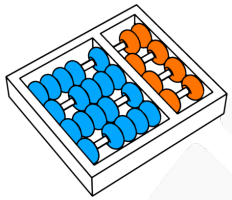


Path compression

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**)

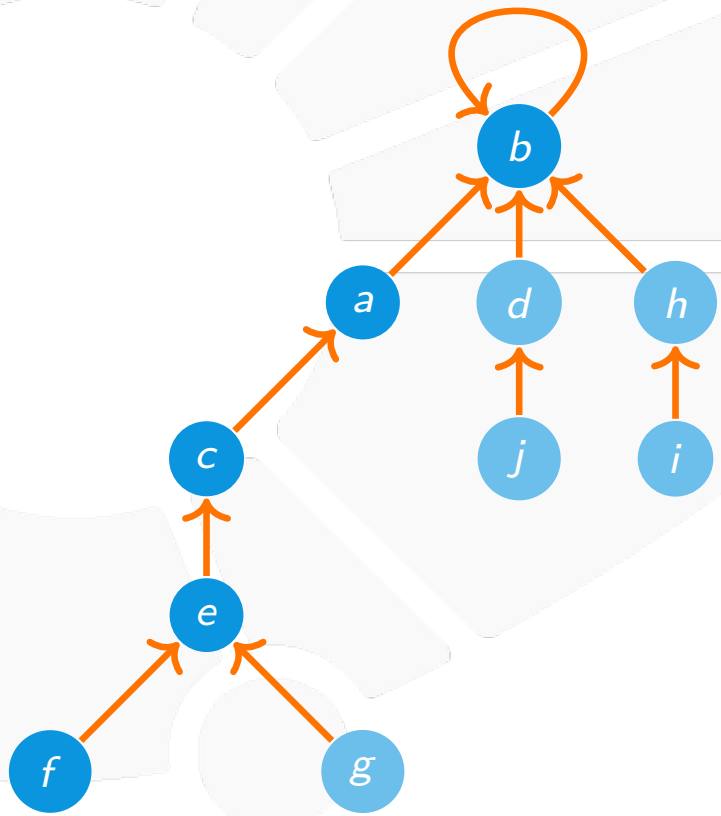


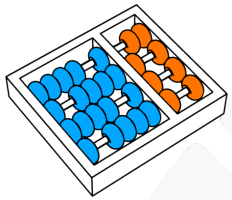


Path compression

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**)

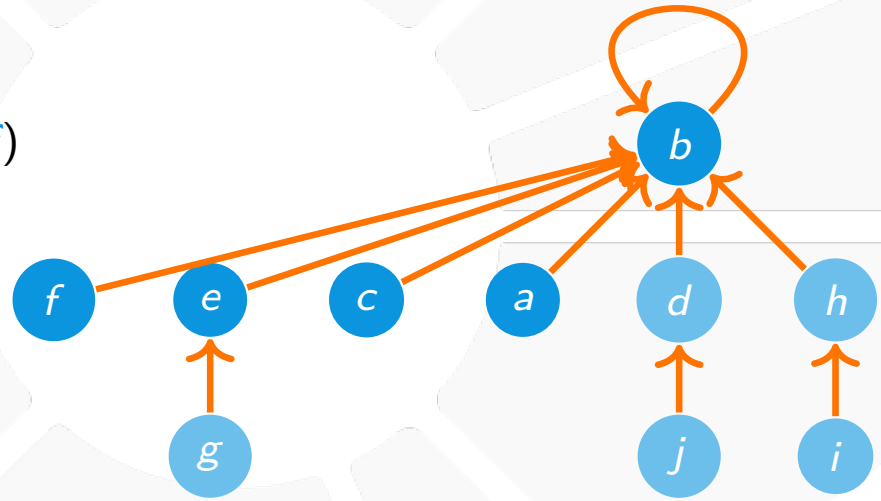


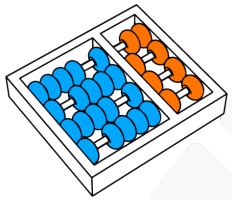


Path compression

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.

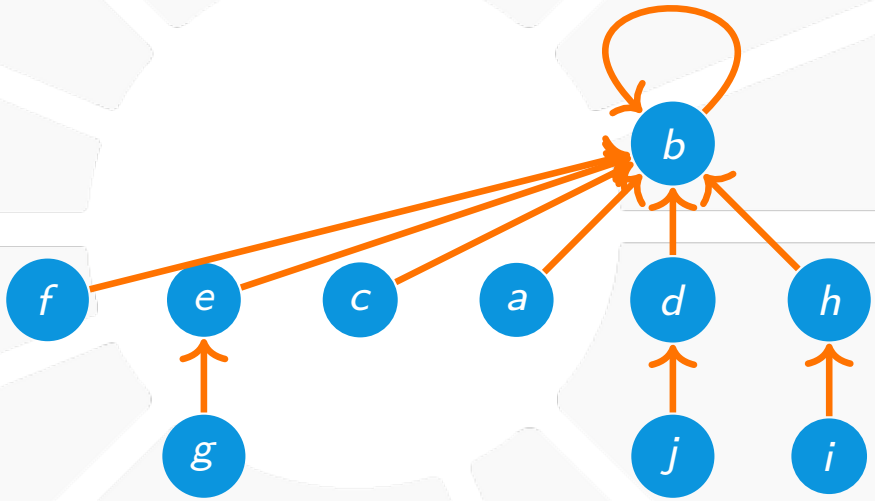
FIND-SET(**f**)

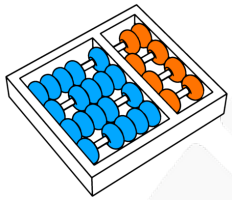




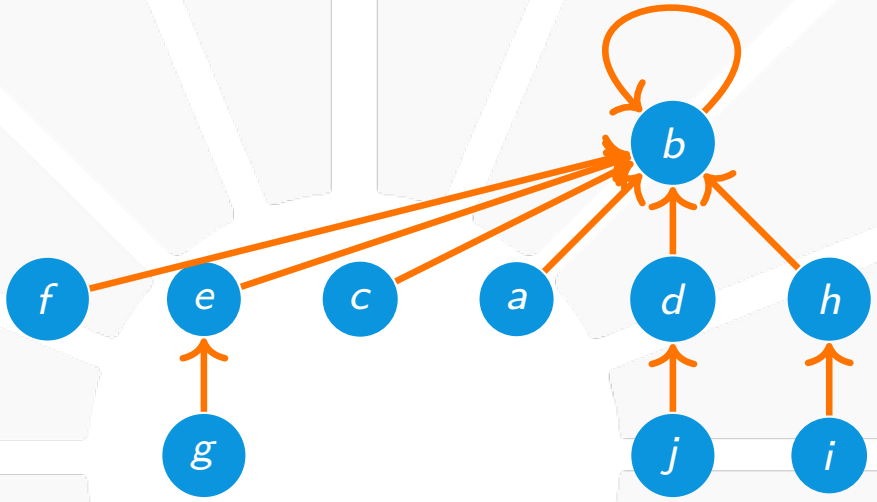
Path compression

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.



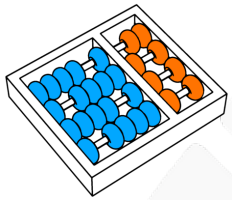


Path compression



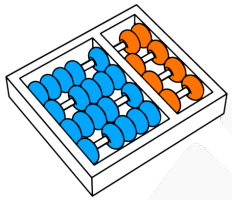
Algoritmo 25: FIND-SET(x)

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



Análise das heurísticas

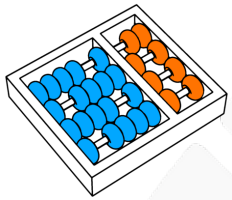
Analisando separadamente:



Análise das heurísticas

Analisando separadamente:

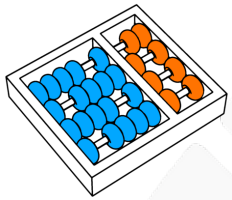
1. Se utilizarmos somente **UNION BY RANK**:



Análise das heurísticas

Analisando separadamente:

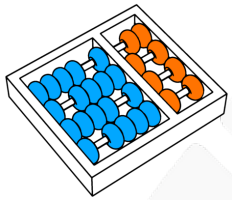
1. Se utilizarmos somente **UNION BY RANK**:
   - Suponha que realizamos m chamadas no total.



Análise das heurísticas

Analisando separadamente:

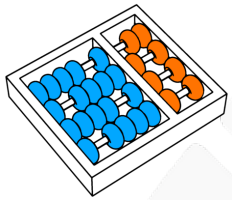
1. Se utilizarmos somente **UNION BY RANK**:
   - Suponha que realizamos m chamadas no total.
   - O tempo total será $O(m \log n)$. Por quê?



Análise das heurísticas

Analisando separadamente:

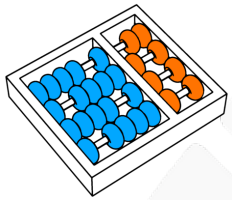
1. Se utilizarmos somente **UNION BY RANK**:
   - Suponha que realizamos m chamadas no total.
   - O tempo total será $O(m \log n)$. Por quê?
2. Se utilizarmos apenas **PATH COMPRESSION**:



Análise das heurísticas

Analisando separadamente:

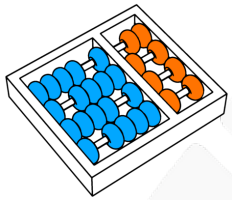
1. Se utilizarmos somente **UNION BY RANK**:
   - Suponha que realizamos m chamadas no total.
   - O tempo total será $O(m \log n)$. Por quê?
2. Se utilizarmos apenas **PATH COMPRESSION**:
   - Suponha que realizamos f chamadas a FIND-SET.



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 $O(n + f \cdot (1 + \log_{2+f/n} n))$.

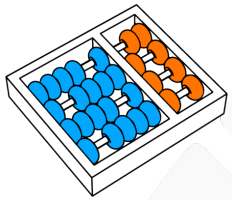


Análise das heurísticas

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:



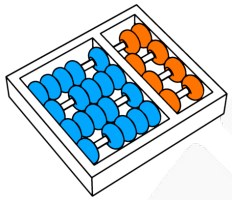
Análise das heurísticas

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 $O(m\alpha(n))$.



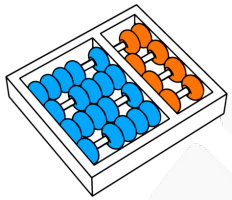
Análise das heurísticas

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:

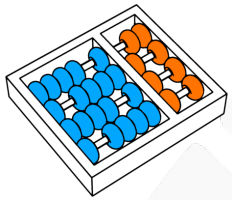
- Mostra-se que o tempo total é $O(m\alpha(n))$.
- Onde, $\alpha(n)$ é uma função que cresce **MUITO MUITO** lentamente.



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.*

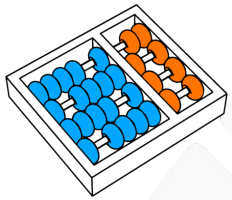


Complexidade com as duas heurísticas

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:



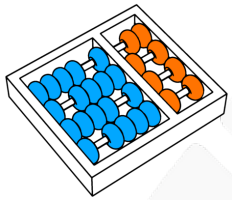
Complexidade com as duas heurísticas

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 $\alpha(n)$.



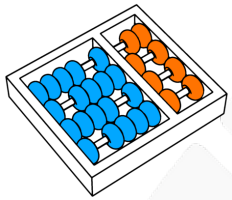
Complexidade com as duas heurísticas

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.



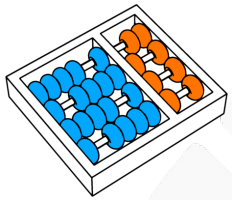
Complexidade com as duas heurísticas

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.



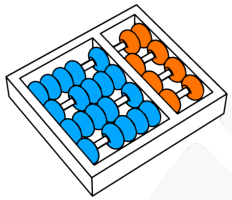
Complexidade com as duas heurísticas

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:

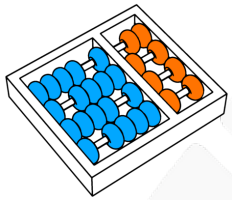
- Ele implica que o custo amortizado por chamada é $\alpha(n)$.
- O valor de $\alpha(n)$ cresce arbitrariamente com n.
- Contudo, o crescimento é num ritmo realmente devagar.
- Uma demonstração está em CLRS.



Estimando o tempo amortizado

Quão pequeno é o custo de cada operação?

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

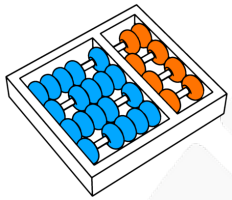


Estimando o tempo amortizado

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!

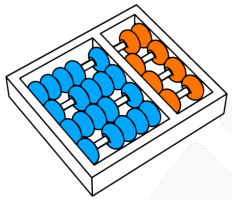


Estimando o tempo amortizado

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.

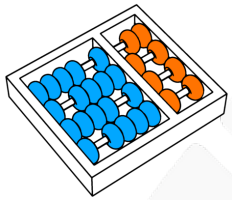


Estimando o tempo amortizado

Quão pequeno é o custo de cada operação?

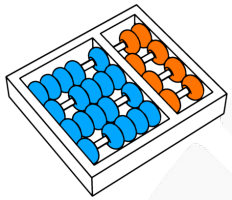
$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq 16^{512} \end{cases}$$

- 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)

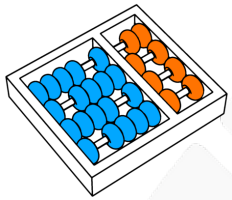
Complexidade:



O algoritmo de Kruskal (de novo)

Complexidade:

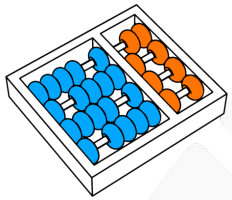
- Ordenação das arestas: **tempo $O(E \log E)$**.



O algoritmo de Kruskal (de novo)

Complexidade:

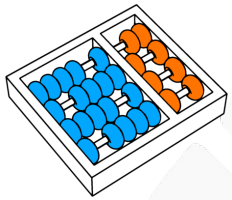
- Ordenação das arestas: **tempo $O(E \log E)$**.
- $O(E)$ chamadas a MAKE-SET, FIND-SET e UNION: **tempo $O(E\alpha(V))$**.



O 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.



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!

Árvore geradora mínima

MC558 - Projeto e Análise de Algoritmos II

5

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP