

Projeto e Análise de Algoritmos

Árvore geradora mínima

Lehilton Pedrosa

Primeiro Semestre de 2020

Árvore geradora mínima

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Considere a seguinte situação:

- ▶ temos um conjunto de computadores
- ▶ para conectar dois computadores usamos um cabo
- ▶ queremos que todos eles estejam interconectados

Como **minimizar** o comprimento do cabo utilizado?

- ▶ este é um problema de otimização
- ▶ a entrada pode ser modelada como um grafo

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

Solução:

- ▶ subgrafo gerador conexo T de G

Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Árvore geradora mínima

Problema da árvore geradora mínima (AGM)

Entrada:

- ▶ grafo conexo $G = (V, E)$
- ▶ peso $w(u, v) \geq 0$ para cada aresta (u, v)

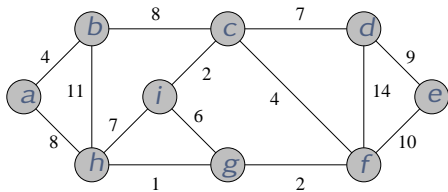
Solução:

- ▶ subgrafo gerador conexo T de G

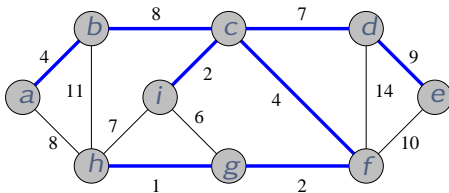
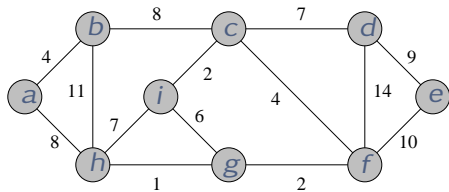
Objetivo:

- ▶ **minimizar** $w(T) = \sum_{(u,v) \in T} w(u, v)$

Exemplo



Exemplo



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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

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**?
- ▶ e se houvesse arestas de peso negativo na entrada?

Veremos dois algoritmos **gulosos**

1. algoritmo de Prim
2. algoritmo de Kruskal

Veremos dois algoritmos **gulosos**

1. algoritmo de Prim
2. algoritmo de Kruskal

Veremos dois algoritmos **gulosos**

1. algoritmo de Prim
2. algoritmo de Kruskal

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Esquema dos algoritmos

Ideia:

- ▶ iremos construir uma árvore incrementalmente
- ▶ denotamos por A o conjunto de arestas da árvore
- ▶ garantimos que A é 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 $A \cup \{(u, v)\}$ também satisfaça
 3. adicionamos (u, v) ao conjunto A

Dizemos que uma tal (u, v) é uma **aresta segura**.

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma **aresta segura**

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma **aresta segura**

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma **aresta segura**

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma aresta segura

Algoritmo genérico

AGM-GENÉRICO(G, w)

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

Esse “algoritmo” está correto:

- ▶ o algoritmo devolve uma árvore geradora
- ▶ essa árvore é subgrafo de alguma AGM
- ▶ então ela também é mínima

Mas o algoritmo está bem definido?

- ▶ se a iteração executa, então A não é árvore geradora
- ▶ então A não contém todas arestas de alguma AGM T
- ▶ assim qualquer aresta de $E[T] \setminus A$ é segura

Os algoritmos reais diferem em como encontrar uma **aresta segura**

Como encontrar arestas seguras

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

- ▶ denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V \setminus S$
- ▶ lembre-se de que um tal conjunto é chamado de **corte**

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

Como encontrar arestas seguras

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

- ▶ denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V \setminus S$
- ▶ lembre-se de que um tal conjunto é chamado de **corte**

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

Como encontrar arestas seguras

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

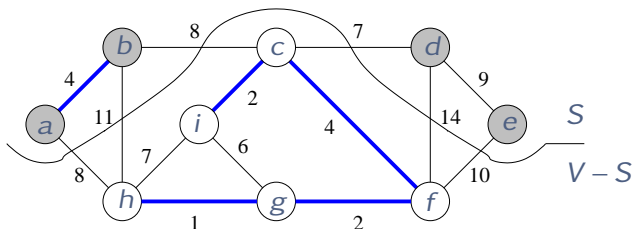
- ▶ denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V \setminus S$
- ▶ lembre-se de que um tal conjunto é chamado de **corte**

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

Como encontrar arestas seguras

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

- ▶ denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V \setminus S$
- ▶ lembre-se de que um tal conjunto é chamado de **corte**

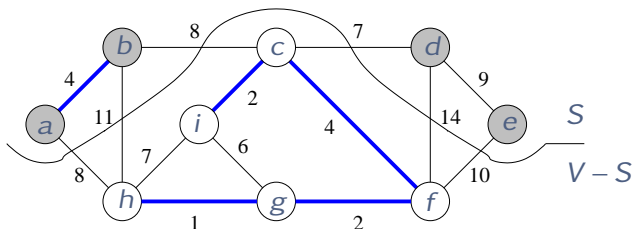


Um corte $\delta(S)$ **respeita** um conjunto A de arestas se não contiver nenhuma aresta de A .

Como encontrar arestas seguras

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

- ▶ denote por $\delta(S)$ o conjunto de arestas de G com um extremo em S e outro em $V \setminus S$
- ▶ lembre-se de que um tal conjunto é chamado de **corte**



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

Arestas leves

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

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 $\delta(S)$ é um corte que respeita A e (u, v) é uma aresta leve desse corte, então (u, v) é uma **aresta segura**.

Arestas leves

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

Teorema

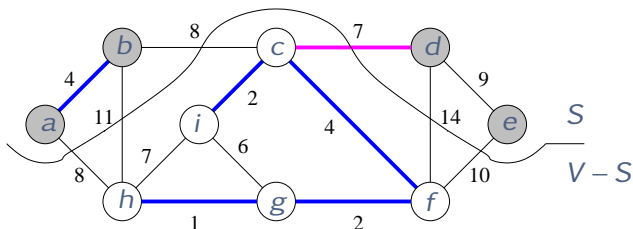
Seja (G, w) um grafo com pesos nas arestas e suponha que A é um subconjunto de arestas de uma AGM de G . Se $\delta(S)$ é um corte que respeita A e (u, v) é uma aresta leve desse corte, então (u, v) é uma **aresta segura**.

Arestas leves

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

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 $\delta(S)$ é um corte que respeita A e (u, v) é uma aresta leve desse corte, então (u, v) é uma **aresta segura**.



Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar

Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar

Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar

Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar

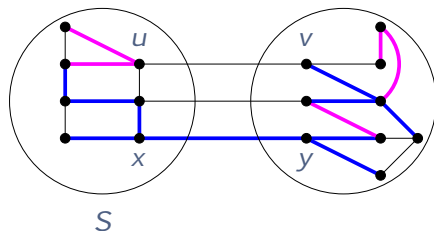
Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar



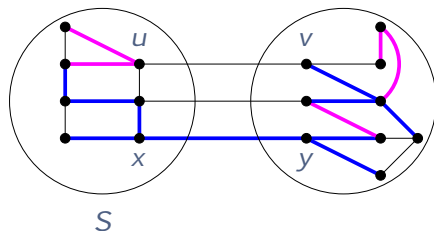
Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar



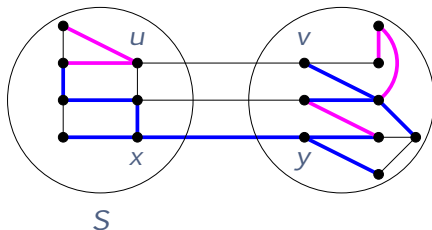
Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar



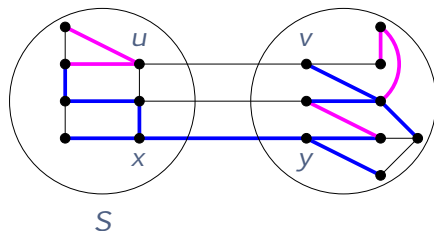
Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema

Seja T uma AGM que contém A

- ▶ tome $\delta(S)$ um corte que respeita A
- ▶ e seja (u, v) uma **aresta leve** deste corte
- ▶ se (u, v) estiver em T , então não há nada a mostrar



Suponha que (u, v) **não** é uma aresta de T

- ▶ construiremos uma AGM T' que contém $A \cup \{(u, v)\}$
- ▶ e daí concluiremos que (u, v) é **segura**

Prova do teorema (cont)

Existe um único caminho P de u a v em T

- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta

Note que (x,y) não pertence a A pois o corte respeita A

Prova do teorema (cont)

Existe um único caminho P de u a v em T

- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta

Note que (x,y) não pertence a A pois o corte respeita A

Prova do teorema (cont)

Existe um único caminho P de u a v em T

- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta

Note que (x,y) não pertence a A pois o corte respeita A

Prova do teorema (cont)

Existe um único caminho P de u a v em T

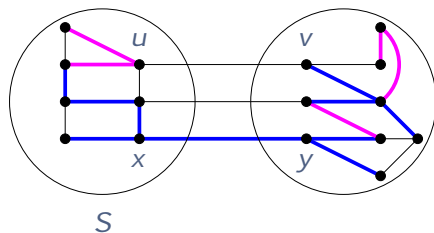
- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta

Note que (x,y) não pertence a A pois o corte respeita A

Prova do teorema (cont)

Existe um único caminho P de u a v em T

- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta

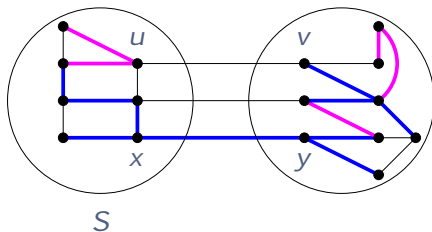


Note que (x,y) não pertence a A pois o corte respeita A

Prova do teorema (cont)

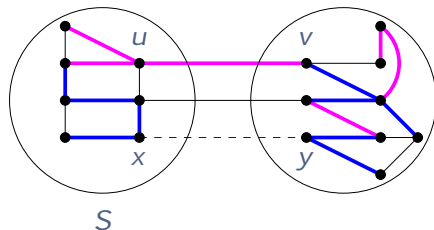
Existe um único caminho P de u a v em T

- ▶ u a v estão em lados opostos do corte $\delta(S)$
- ▶ então alguma aresta de P pertence ao corte
- ▶ seja (x,y) uma tal aresta



Note que (x,y) não pertence a A pois o corte respeita A

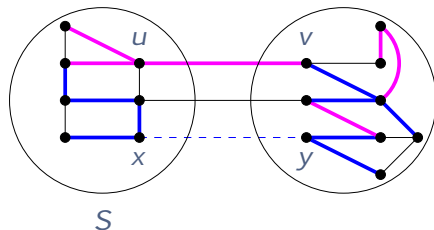
Prova do teorema (cont)



Defina $T' := T - \{(x, y)\} \cup \{(u, v)\}$

- ▶ observe que T' é uma árvore geradora
- ▶ mostraremos que T' também é uma **AGM**

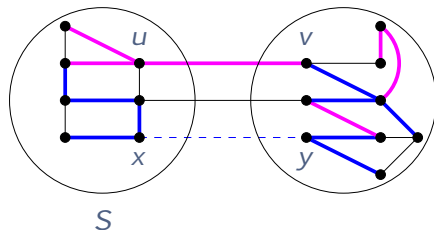
Prova do teorema (cont)



Defina $T' := T - \{(x, y)\} \cup \{(u, v)\}$

- ▶ observe que T' é uma árvore geradora
- ▶ mostraremos que T' também é uma **AGM**

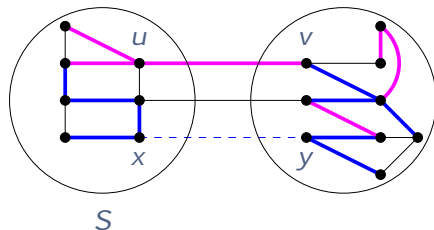
Prova do teorema (cont)



Defina $T' := T - \{(x, y)\} \cup \{(u, v)\}$

- ▶ observe que T' é uma árvore geradora
- ▶ mostraremos que T' também é uma **AGM**

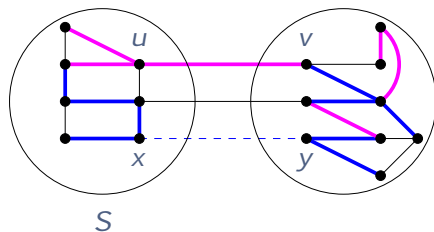
Prova do teorema (cont)



Defina $T' := T - \{(x, y)\} \cup \{(u, v)\}$

- ▶ observe que T' é uma árvore geradora
- ▶ mostraremos que T' também é uma **AGM**

Prova do teorema (cont)



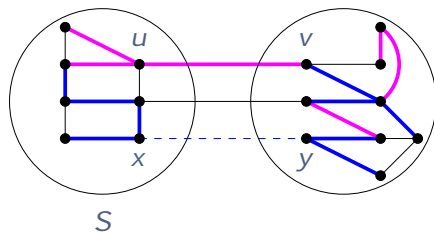
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



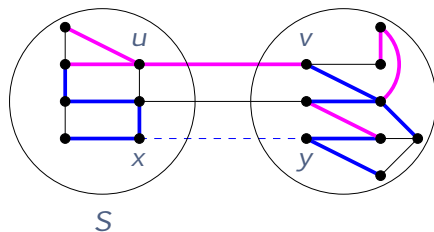
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



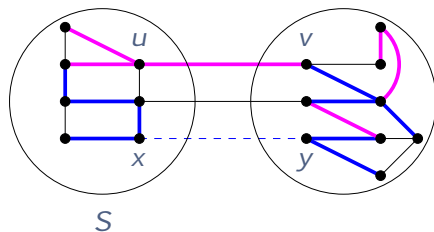
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



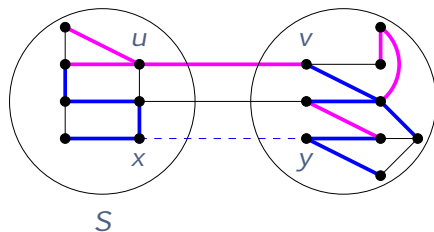
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



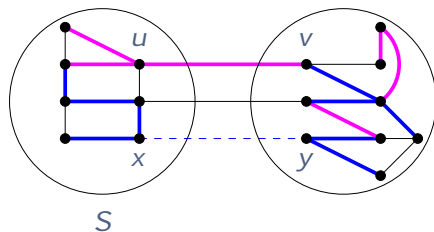
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



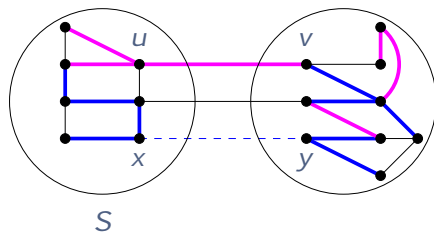
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



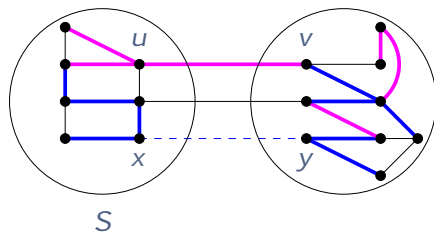
Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

Prova do teorema (cont)



Como (u, v) é uma **aresta leve** do corte $\delta(S)$

- ▶ temos que $w(u, v) \leq w(x, y)$ pois (x, y) pertence ao corte
- ▶ assim, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
- ▶ logo T' também é uma AGM

Além disso, T' contém $A \cup \{(u, v)\}$

- ▶ portanto, (u, v) é uma **aresta segura**
- ▶ e concluímos a prova

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 desse corte, então (u, v) é uma **aresta segura**.

Isso sugere um algoritmo iterativo

- ▶ Prim e Kruskal implementam essa ideia
- ▶ seus algoritmos farão uso desse corolário

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 desse corte, então (u, v) é uma **aresta segura**.

Isso sugere um algoritmo iterativo

- ▶ Prim e Kruskal implementam essa ideia
- ▶ seus algoritmos farão uso desse corolário

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 desse corte, então (u, v) é uma **aresta segura**.

Isso sugere um algoritmo iterativo

- ▶ Prim e Kruskal implementam essa ideia
- ▶ seus algoritmos farão uso desse corolário

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 desse corte, então (u, v) é uma **aresta segura**.

Isso sugere um algoritmo iterativo

- ▶ Prim e Kruskal implementam essa ideia
- ▶ seus algoritmos farão uso desse corolário

Algoritmo de Prim

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

O algoritmo de Prim

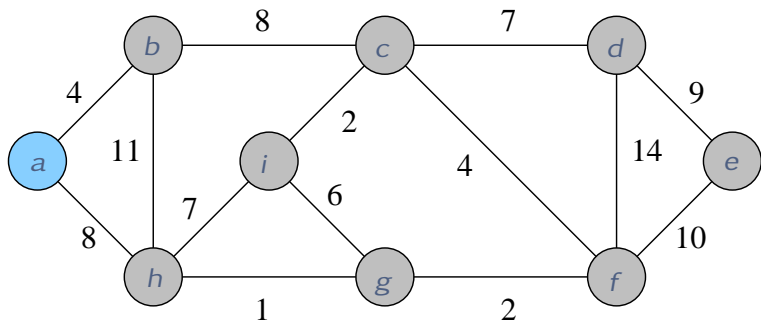
Ideia

- ▶ escolhemos um vértice r arbitrariamente no início
- ▶ o conjunto A são às arestas de uma árvore com raiz r
- ▶ o conjunto S são os vértices dessa árvore
- ▶ em cada iteração, adicionamos uma **aresta leve** de $\delta(S)$

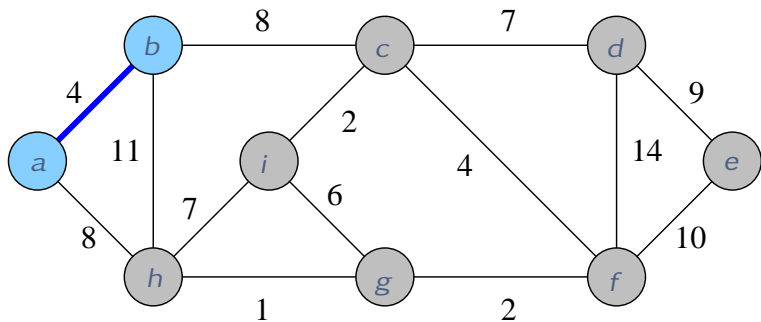
Detalhe de implementação importante

- ▶ como encontrar essa aresta leve **eficientemente**?

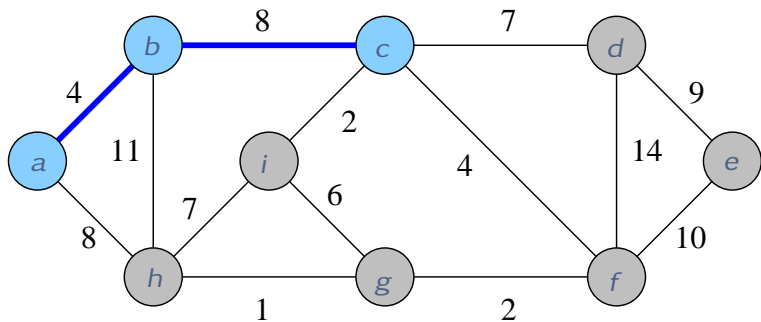
O algoritmo de Prim



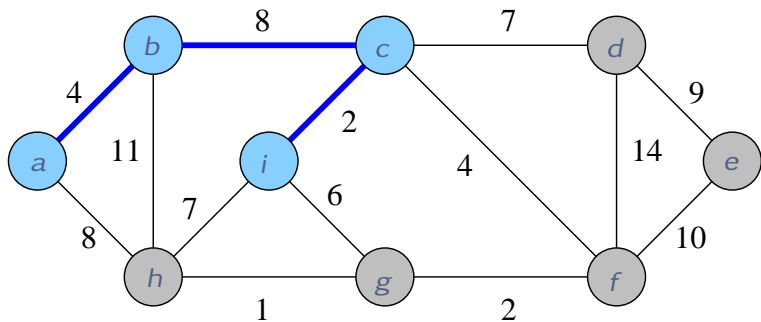
O algoritmo de Prim



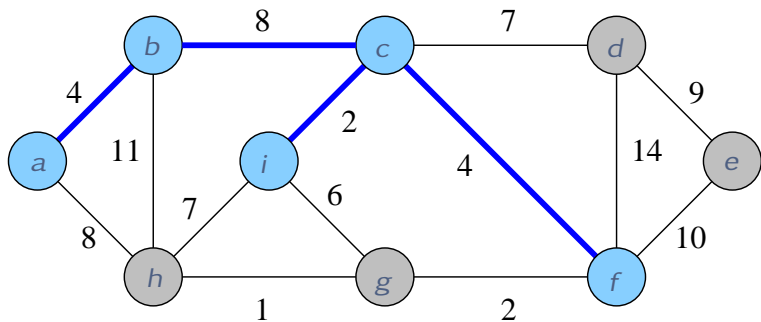
O algoritmo de Prim



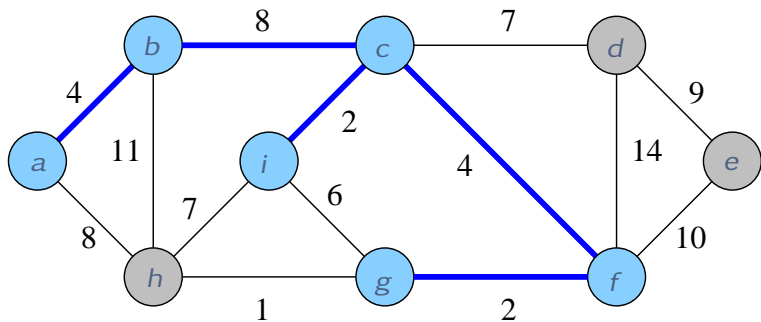
O algoritmo de Prim



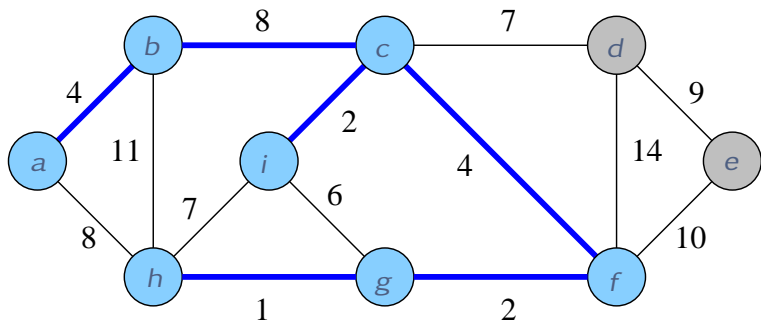
O algoritmo de Prim



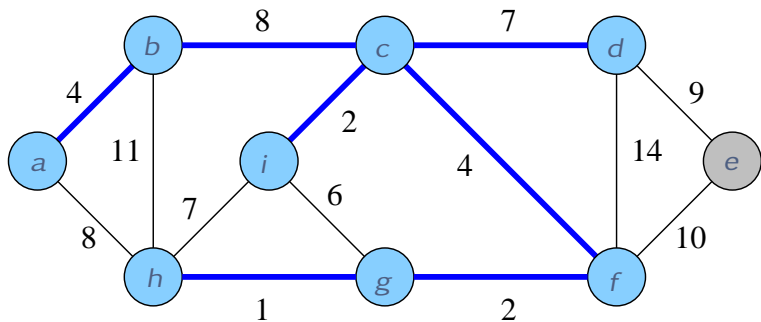
O algoritmo de Prim



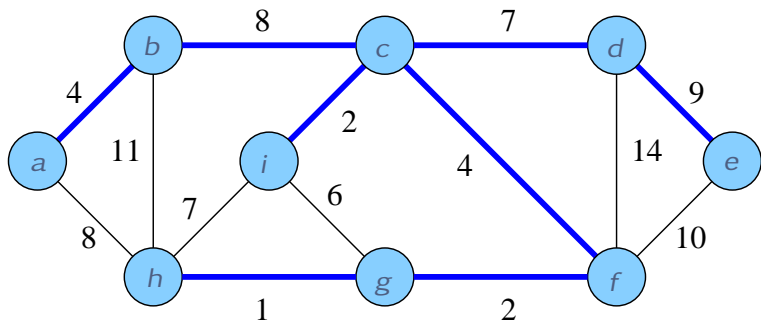
O algoritmo de Prim



O algoritmo de Prim



O algoritmo de Prim



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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

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 $S = V \setminus Q$
- ▶ e as arestas da árvore são $A = \{(u, \pi[u]) : u \in S \setminus \{r\}\}$

O algoritmo de Prim

AGM-PRIM(G, w, r)

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

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Correção do algoritmo

Invariantes

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

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

- ▶ as invariantes implicam que no início da iteração do laço, $(u, \pi[u])$ é uma **aresta segura**
- ▶ portanto, o algoritmo está correto

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade do algoritmo

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

$$O(V) \cdot \text{INSERT} + O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$$

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

Observações:

- ▶ podemos inicializar o min-heap em tempo $O(V)$
- ▶ usamos $V = O(E)$ pois sabemos que G é conexo

Complexidade usando min-heap

Se implementarmos Q como um min-heap, então

- ▶ INSERT consome tempo $O(\lg V)$
- ▶ EXTRACT-MIN consome tempo $O(\lg V)$
- ▶ DECREASE-KEY consome tempo $O(\lg V)$

Então o tempo total será

$$O(V \lg V + V \lg V + E \lg V) = O(E \lg V)$$

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

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

Refletindo sobre como analisamos uma operação

- ▶ supomos que todas as chamadas levam o mesmo tempo
- ▶ e 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

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o tempo total das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

Análise amortizada

Ideia da análise amortizada

- ▶ suponha que na execução fazemos m chamadas a $p(S)$
- ▶ e que o **tempo total** das operações p é $T(n)$
- ▶ então o custo amortizado de p é $T(n)/m$

Exemplo

- ▶ e.g., 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

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

Revisitando a complexidade de Prim

Um **heap de Fibonacci** é uma estrutura de dados

- ▶ utilizada para guardar um conjunto de $|V|$ elementos
- ▶ implementa as operações de fila de prioridade
 - ▶ EXTRACT-MIN – tempo $O(\lg V)$
 - ▶ 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

- ▶ tempo total melhora para $O(V + E + V \lg V) = O(E + V \lg V)$
- ▶ na prática, a implementação com min-heap é melhor

O algoritmo de Kruskal

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

O algoritmo de Kruskal

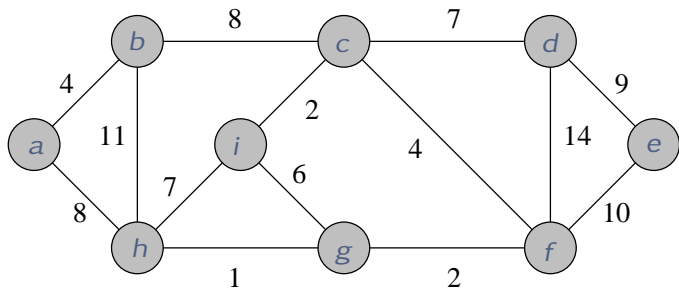
Ideia

- ▶ o subgrafo $G_A = (V, A)$ 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** do $\delta(C)$

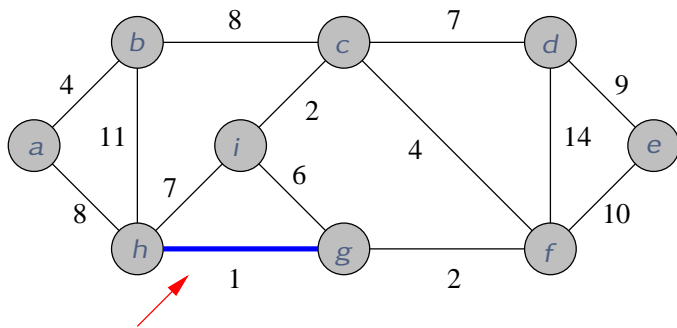
Detalhe de implementação importante

- ▶ como saber se (u, v) liga componentes distintas?

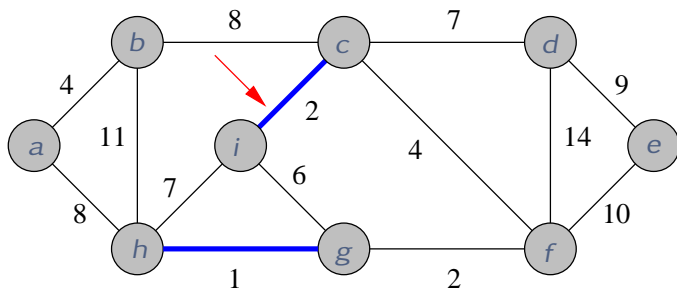
O algoritmo de Kruskal



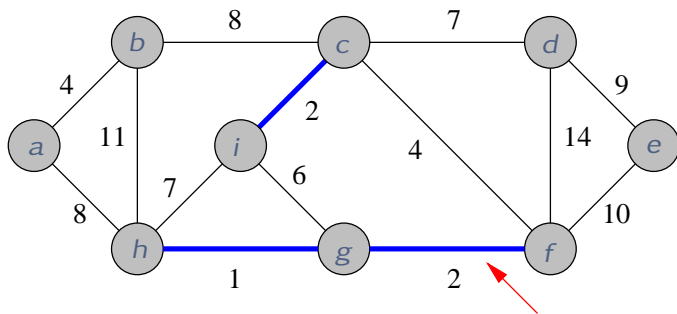
O algoritmo de Kruskal



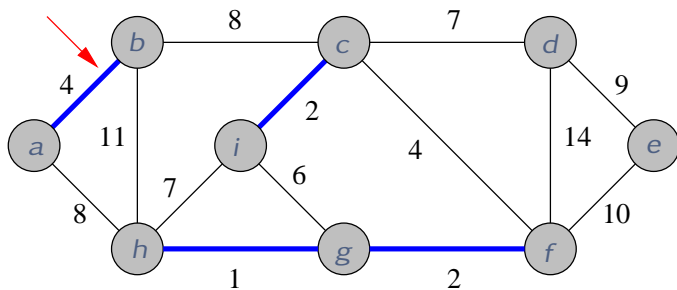
O algoritmo de Kruskal



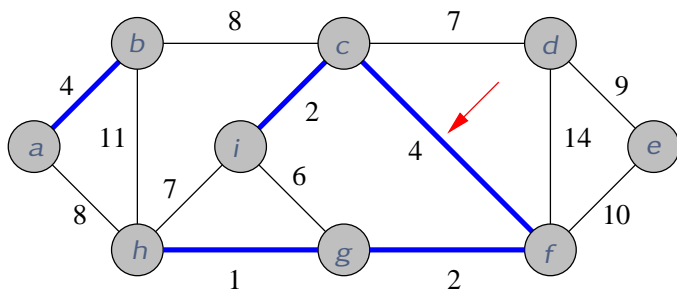
O algoritmo de Kruskal



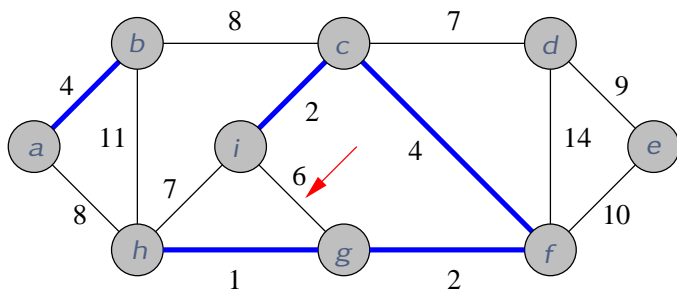
O algoritmo de Kruskal



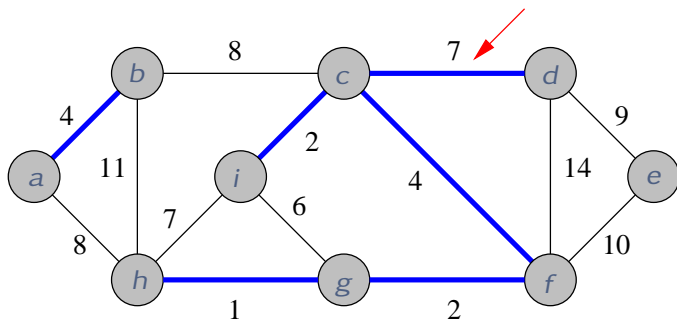
O algoritmo de Kruskal



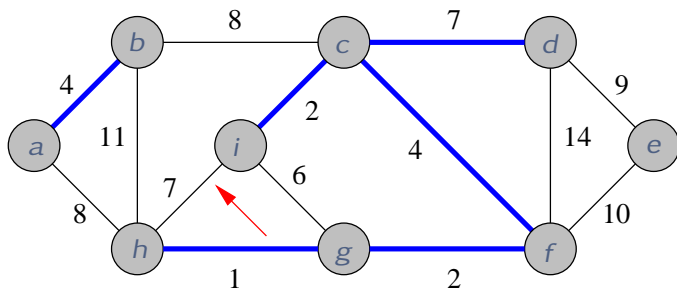
O algoritmo de Kruskal



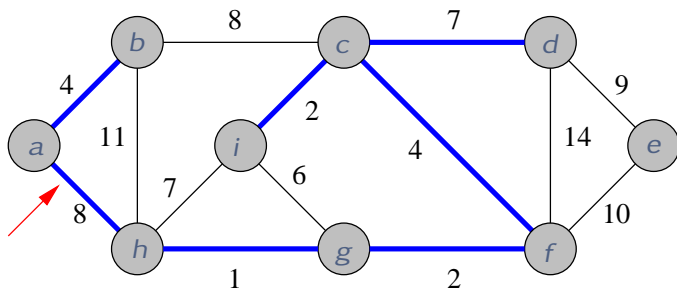
O algoritmo de Kruskal



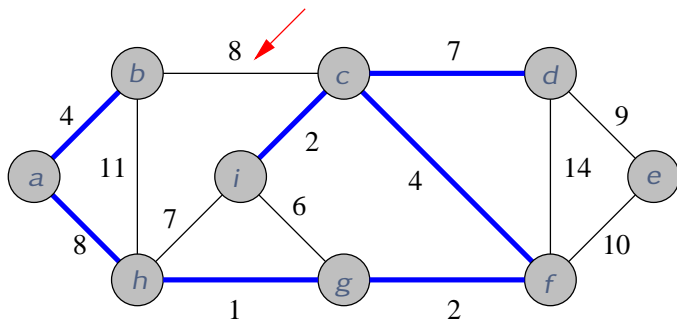
O algoritmo de Kruskal



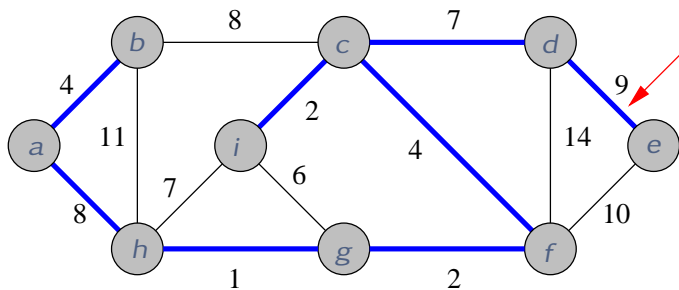
O algoritmo de Kruskal



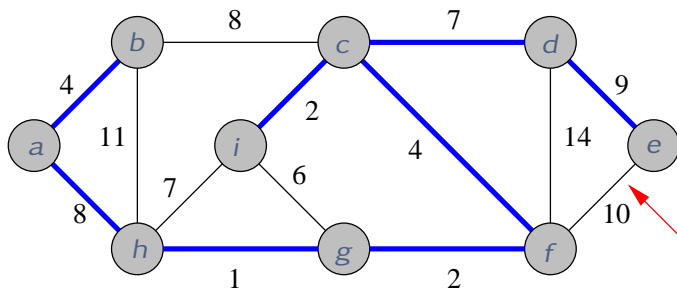
O algoritmo de Kruskal



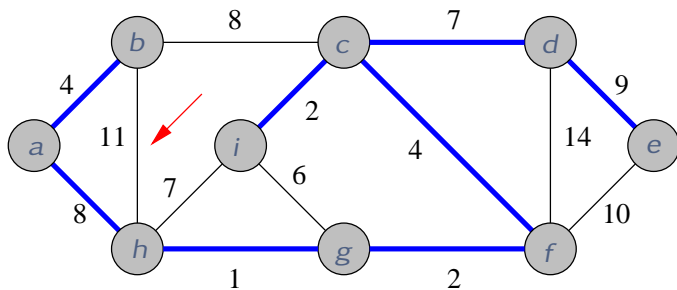
O algoritmo de Kruskal



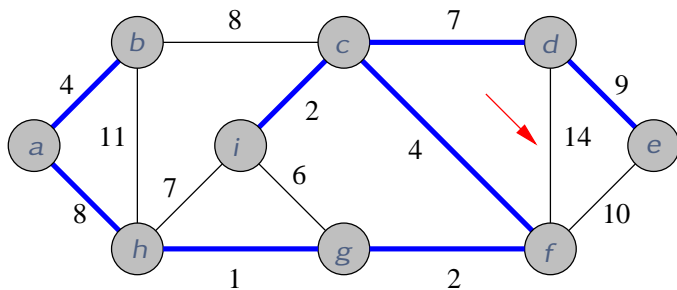
O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal



O algoritmo de Kruskal

AGM-KRUSKAL(G, w)

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

Esta é uma versão preliminar do algoritmo

- ▶ ainda falta detalhar como implementar a linha 4
- ▶ como fazer isso **eficientemente**?

O algoritmo de Kruskal

AGM-KRUSKAL(G, w)

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

Esta é uma versão preliminar do algoritmo

- ▶ ainda falta detalhar como implementar a linha 4
- ▶ como fazer isso **eficientemente**?

O algoritmo de Kruskal

AGM-KRUSKAL(G, w)

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

Esta é uma versão preliminar do algoritmo

- ▶ ainda falta detalhar como implementar a linha 4
- ▶ como fazer isso **eficientemente**?

O algoritmo de Kruskal

AGM-KRUSKAL(G, w)

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

Esta é uma versão preliminar do algoritmo

- ▶ ainda falta detalhar como implementar a linha 4
- ▶ como fazer isso **eficientemente**?

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Estrutura de dados

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Estrutura de dados

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Estrutura de dados

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Estrutura de dados

Como guardar a floresta sendo construída?

- ▶ basta guardar o conjunto A das arestas
- ▶ assim a floresta é $G_A = (V, A)$

Mas precisamos representar as componentes de G_A

- ▶ durante o algoritmo, as componentes de G_A mudam
- ▶ queremos **determinar** qual componente contém vértice u
- ▶ além de **fazer a união** das componentes que contêm u e v

Qual estrutura realiza essas operações eficientemente?

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de conjuntos disjuntos
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um representante
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

Queremos uma estrutura de dados

- ▶ que mantém coleção S_1, S_2, \dots, S_k de **conjuntos disjuntos**
- ▶ permite remover ou adicionar conjuntos à tal coleção
- ▶ e identifica cada conjunto por um **representante**
 - ▶ que é um elemento do próprio conjunto
 - ▶ a escolha do representante é irrelevante
 - ▶ mas o representante de um conjunto não pode mudar

Conjuntos disjuntos

A estrutura de dados deve permitir as seguintes operações:

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

Exemplo de aplicação

Vamos determinar as componentes conexas de um grafo G

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

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

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

CONNECTED-COMPONENTS(G)

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

SAME-COMPONENT(u, v)

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

CONNECTED-COMPONENTS(G)

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

SAME-COMPONENT(u, v)

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

A complexidade depende da implementação

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

A complexidade depende da implementação

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

A complexidade depende da implementação

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

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

O algoritmo de Kruskal

Agora escrevemos a versão completa do algoritmo de Kruskal

AGM-KRUSKAL(G, w)

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

De novo, a complexidade depende da estrutura de dados

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

Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados

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

Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados

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

Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados

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

Complexidade do algoritmo

De novo, a complexidade depende da estrutura de dados

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

Complexidade da operações realizadas

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

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



Queremos medir a complexidade em termos de n e m .

Complexidade da operações realizadas

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

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



Queremos medir a complexidade em termos de n e m .

Complexidade da operações realizadas

Seequência de chamadas MAKE-SET, UNION e FIND-SET

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



Queremos medir a complexidade em termos de n e m .

Complexidade da operações realizadas

Seequência de chamadas MAKE-SET, UNION e FIND-SET

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



Queremos medir a complexidade em termos de n e m .

Complexidade da operações realizadas

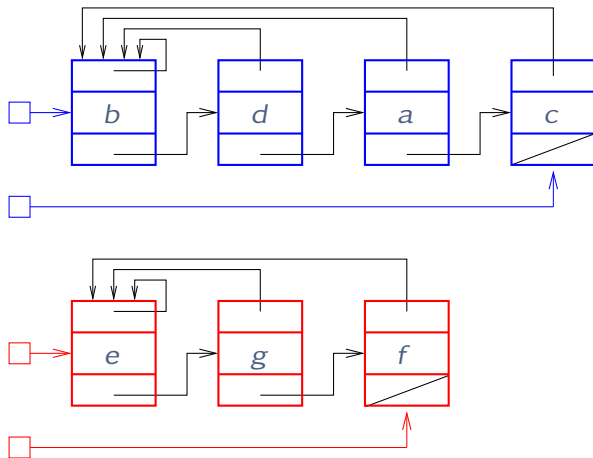
Seequência de chamadas MAKE-SET, UNION e FIND-SET

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



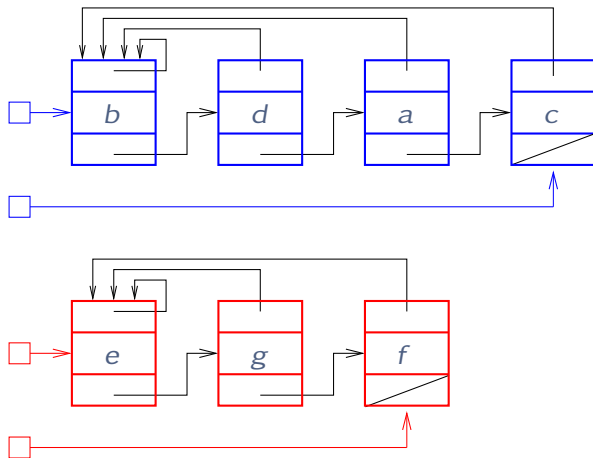
Queremos medir a complexidade em termos de n e m .

Representação por listas ligadas



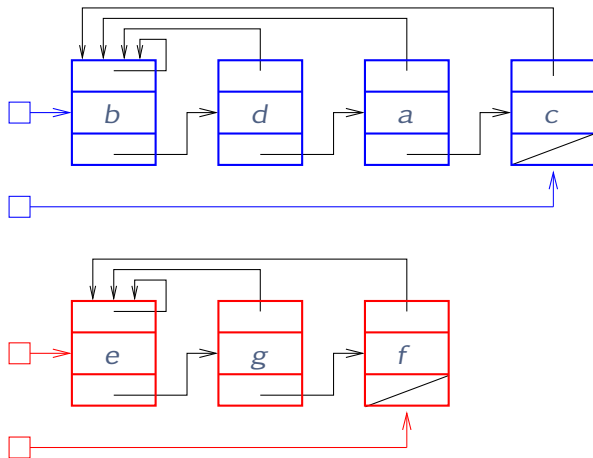
- ▶ Cada conjunto tem um representante (início da lista)
- ▶ Cada nó tem um campo que aponta para o representante
- ▶ Guarda-se um apontador para o fim de cada lista

Representação por listas ligadas



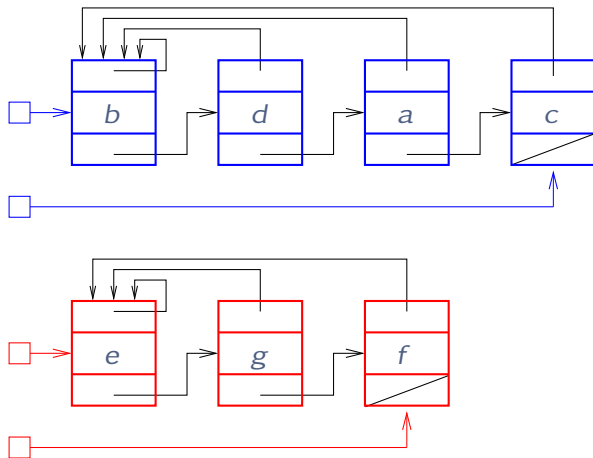
- ▶ Cada conjunto tem um representante (início da lista)
- ▶ Cada nó tem um campo que aponta para o representante
- ▶ Guarda-se um apontador para o fim de cada lista

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

Representação por listas ligadas



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

Complexidade usando listas ligadas

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

Complexidade usando listas ligadas

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

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

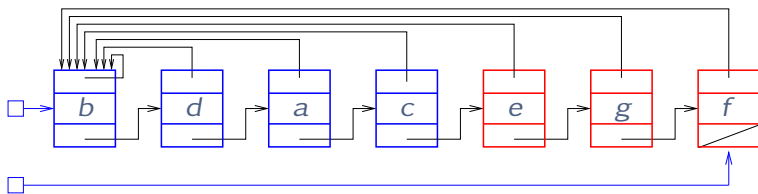
- ▶ $\text{MAKE-SET}(x) - O(1)$
- ▶ $\text{FIND-SET}(x) - O(1)$
- ▶ $\text{UNION}(x, y) - O(n)$
 - ▶ temos que concatenar a lista de y no final da lista de x
 - ▶ e atualizar os apontadores para o representante

Complexidade usando listas ligadas

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

Complexidade usando listas ligadas

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



Um exemplo de pior caso

Chamada a operação	Número de atualizações
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
\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 $2n - 1$

▶ O tempo total é $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$

▶ O custo amortizado por operação é $\frac{\Theta(n^2)}{2n-1} = \Theta(n)$

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 $2n - 1$

- ▶ O tempo total é $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$

- ▶ O custo amortizado por operação é $\frac{\Theta(n^2)}{2n-1} = \Theta(n)$

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 $2n - 1$

- ▶ O tempo total é $n + \sum_{i=1}^{n-1} i = \Theta(n^2)$

- ▶ O custo amortizado por operação é $\frac{\Theta(n^2)}{2n-1} = \Theta(n)$

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

Uma heurística simples

Entendendo o pior caso

- ▶ cada chamada a union gasta em média tempo $\Theta(n)$
- ▶ isso porque concatenamos a maior lista no final da menor
- ▶ para evitar isso, podemos concatenar **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

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 **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

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 **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

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 **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

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 **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

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 **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

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 **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

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 **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

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

Demonstração

- ▶ o tempo total das chamadas MAKE-SET e FIND-SET é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a UNION é $O(n \lg n)$

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 $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas MAKE-SET e FIND-SET é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a UNION é $O(n \lg n)$

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 $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas **MAKE-SET** e **FIND-SET** é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a **UNION** é $O(n \lg n)$

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` heurística, então o **tempo total** gasto será $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas `MAKE-SET` e `FIND-SET` é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a `UNION` é $O(n \lg n)$

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 $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas `MAKE-SET` e `FIND-SET` é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a `UNION` é $O(n \lg n)$

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` heurística, então o **tempo total** gasto será $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas `MAKE-SET` e `FIND-SET` é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a `UNION` é $O(n \lg n)$

Uma heurística simples

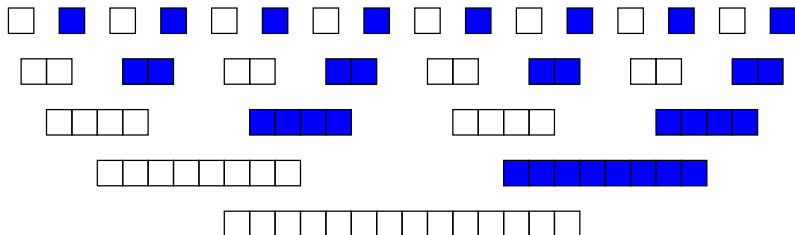
Teorema

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` heurística, então o **tempo total** gasto será $O(m + n \lg n)$.

Demonstração

- ▶ o tempo total das chamadas `MAKE-SET` e `FIND-SET` é $O(m)$
- ▶ ao atualizarmos um nó, a lista que o continha dobra
- ▶ mas uma lista só pode dobrar no máximo $O(\lg n)$ vezes
- ▶ assim, cada nó só é atualizado $O(\lg n)$ vezes
- ▶ portanto, o tempo total com chamadas a `UNION` é $O(n \lg n)$

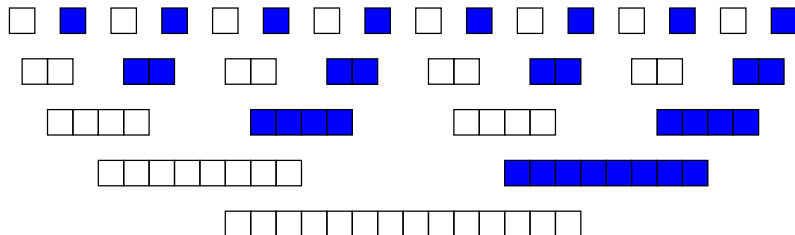
Um exemplo de pior caso



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

- ▶ cada nível da figura representa a coleção de conjuntos disjuntos em determinado momento
- ▶ entre um nível e o próximo, as lista em azul são concatenadas às listas da esquerda
- ▶ assim, em cada nível, $n/2$ apontadores são atualizados
- ▶ mas há $\Theta(\lg n)$ níveis

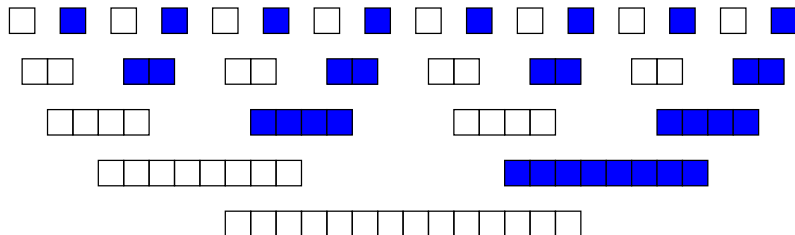
Um exemplo de pior caso



O custo total de **UNION** nesse exemplo é $\Theta(n \lg n)$

- ▶ cada nível da figura representa a coleção de conjuntos disjuntos em determinado momento
- ▶ entre um nível e o próximo, as lista em azul são concatenadas às listas da esquerda
- ▶ assim, em cada nível, $n/2$ apontadores são atualizados
- ▶ mas há $\Theta(\lg n)$ níveis

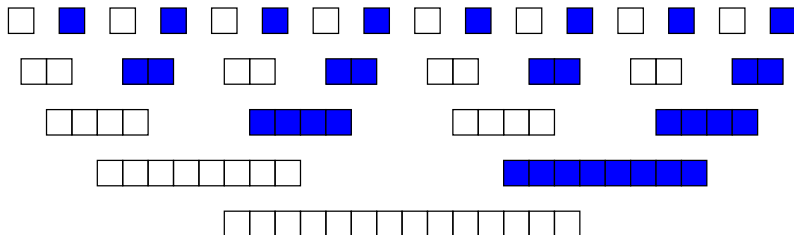
Um exemplo de pior caso



O custo total de **UNION** nesse exemplo é $\Theta(n \lg n)$

- ▶ cada nível da figura representa a coleção de conjuntos disjuntos em determinado momento
- ▶ entre um nível e o próximo, as lista em azul são concatenadas às listas da esquerda
- ▶ assim, em cada nível, $n/2$ apontadores são atualizados
- ▶ mas há $\Theta(\lg n)$ níveis

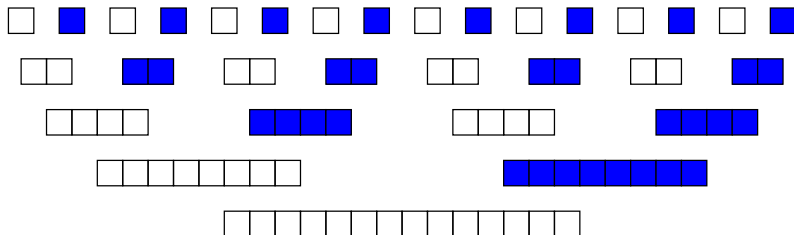
Um exemplo de pior caso



O custo total de **UNION** nesse exemplo é $\Theta(n \lg n)$

- ▶ cada nível da figura representa a coleção de conjuntos disjuntos em determinado momento
- ▶ entre um nível e o próximo, as lista em azul são concatenadas às listas da esquerda
- ▶ assim, em cada nível, $n/2$ apontadores são atualizados
- ▶ mas há $\Theta(\lg n)$ níveis

Um exemplo de pior caso



O custo total de **UNION** nesse exemplo é $\Theta(n \lg n)$

- ▶ cada nível da figura representa a coleção de conjuntos disjuntos em determinado momento
- ▶ entre um nível e o próximo, as lista em azul são concatenadas às listas da esquerda
- ▶ assim, em cada nível, $n/2$ apontadores são atualizados
- ▶ mas há $\Theta(\lg n)$ níveis

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Complexidade do algoritmo de Kruskal

Relembrando, a complexidade de AGM-KRUSKAL é dada por

- ▶ ordenação que toma tempo $O(E \lg E)$
- ▶ $|V|$ 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

$$O(E \lg E) + O(V + E + V \lg V) = O(E \lg E) = O(E \lg V)$$

- ▶ o tempo é dominado pela ordenação das arestas
- ▶ se já estiverem ordenadas, então gastamos $O(E + V \lg V)$

Conjuntos disjuntos com florestas de conjuntos

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas *union by rank* e *path compression*
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas *union by rank* e *path compression*
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante UNION
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante FIND-SET
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

Representação por florestas de conjuntos

Podemos utilizar outra forma de representação

- ▶ uma coleção é representada por uma floresta
- ▶ cada conjunto corresponde a uma árvore enraizada
- ▶ o representante de um conjunto é a raiz
- ▶ é a chamada **disjoint-set forest**

Veremos duas implementações

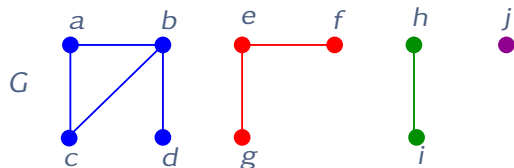
1. uma implementação simples

- ▶ só altera a floresta durante **UNION**
- ▶ o tempo não melhor que listas (assintoticamente)

2. uma implementação mais elaborada

- ▶ utiliza heurísticas **union by rank** e **path compression**
- ▶ também alteramos a floresta durante **FIND-SET**
- ▶ é a melhor implementação de conhecida

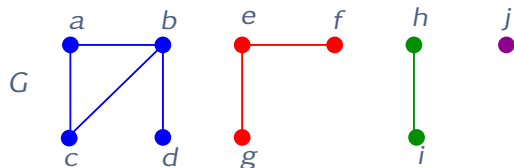
Exemplo de grafo



Veja o grafo

- ▶ considere um conjunto para cada componente
- ▶ como representar os conjuntos com disjoint-set forest?

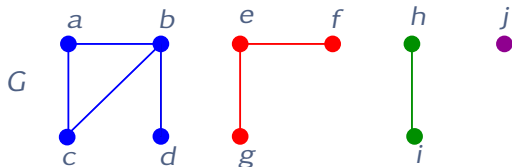
Exemplo de grafo



Veja o grafo

- ▶ considere um conjunto para cada componente
- ▶ como representar os conjuntos com disjoint-set forest?

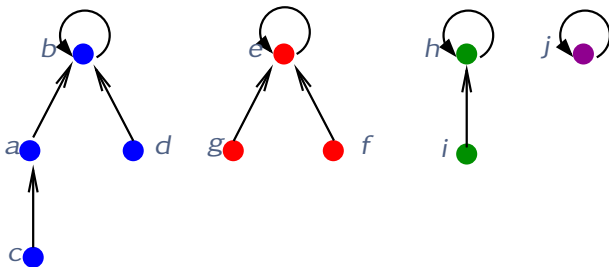
Exemplo de grafo



Veja o grafo

- ▶ considere um conjunto para cada componente
- ▶ como representar os conjuntos com disjoint-set forest?

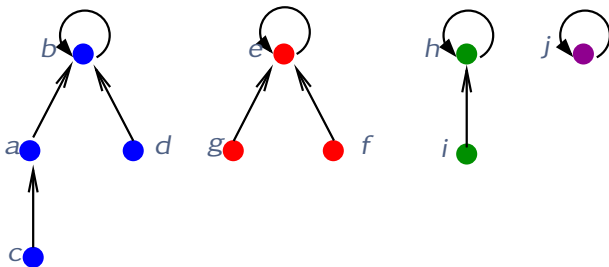
Exemplo de representação



Convenções

- ▶ cada conjunto é uma árvore enraizada
- ▶ cada elemento aponta para seu pai
- ▶ a raiz aponta para si mesma
- ▶ ela é o representante do conjunto

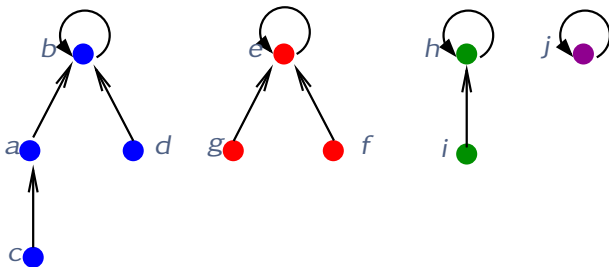
Exemplo de representação



Convenções

- ▶ cada conjunto é uma árvore enraizada
- ▶ cada elemento aponta para seu pai
- ▶ a raiz aponta para si mesma
- ▶ ela é o representante do conjunto

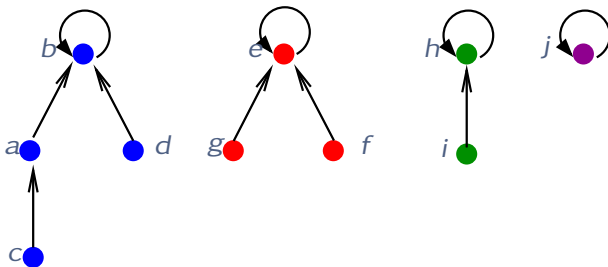
Exemplo de representação



Convenções

- ▶ cada conjunto é uma árvore enraizada
- ▶ cada elemento aponta para seu pai
- ▶ a raiz aponta para si mesma
- ▶ ela é o representante do conjunto

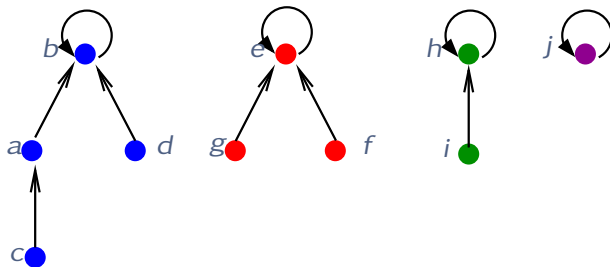
Exemplo de representação



Convenções

- ▶ cada conjunto é uma árvore enraizada
- ▶ cada elemento aponta para seu pai
- ▶ a **raiz** aponta para si mesma
- ▶ ela é o representante do conjunto

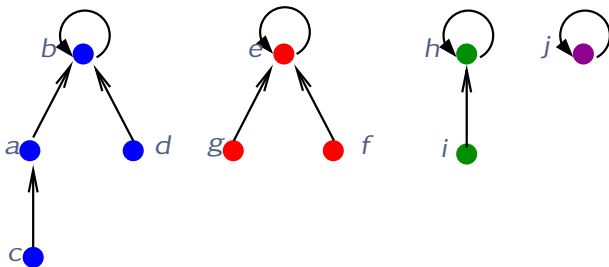
Exemplo de representação



Convenções

- ▶ cada conjunto é uma árvore enraizada
- ▶ cada elemento aponta para seu pai
- ▶ a **raiz** aponta para si mesma
- ▶ ela é o representante do conjunto

Implementação simples



MAKE-SET(x)

1 $pai[x] \leftarrow x$

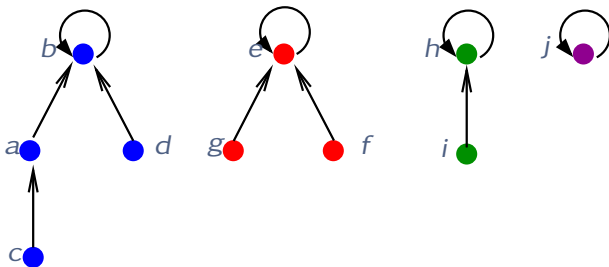
FIND-SET(x)

1 se $x = pai[x]$

2 então devolva x

3 senão devolva **FIND-SET**($pai[x]$)

Implementação simples



MAKE-SET(x)

1 $pai[x] \leftarrow x$

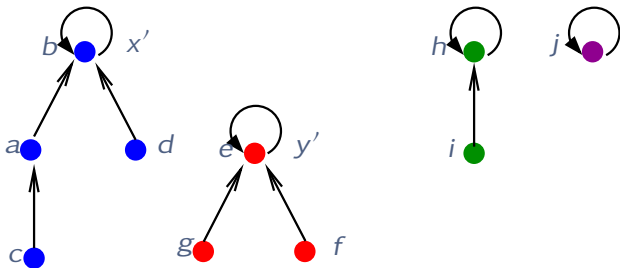
FIND-SET(x)

1 se $x = pai[x]$

2 então devolva x

3 senão devolva **FIND-SET**($pai[x]$)

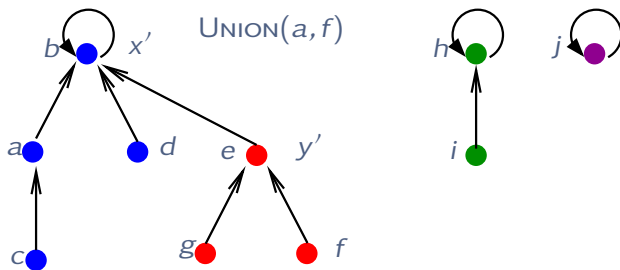
Implementação simples (cont)



UNION(x, y)

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

Implementação simples (cont)



UNION(x, y)

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

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

- ▶ considere uma sequência de $n - 1$ chamadas a **UNION**
- ▶ que resulta em uma cadeia linear com n nós
- ▶ daí, n chamadas a **FIND-SET** pode levar tempo total $\Theta(n^2)$

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

- ▶ considere uma sequência de $n - 1$ chamadas a **UNION**
- ▶ que resulta em uma cadeia linear com n nós
- ▶ daí, n chamadas a **FIND-SET** pode levar tempo total $\Theta(n^2)$

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Complexidade da implementação simples

Tempo das operações

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

Não é melhor do que a representação por listas ligadas

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

Podemos melhorar isso usando duas heurísticas

- ▶ union by rank
- ▶ path compression

Union by rank

Ideia emprestada da **weighted-union heuristic**

- ▶ cada nó x está associado a um número $rank[x]$
 - ▶ 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

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

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

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

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

MAKE-SET(x)

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

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

UNION(x, y)

1 $\text{LINK}(\text{FIND-SET}(x), \text{FIND-SET}(y))$

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

1 se $\text{rank}[x] > \text{rank}[y]$ então

2 $\text{pai}[y] \leftarrow x$

3 senão

4 $\text{pai}[x] \leftarrow y$

5 se $\text{rank}[x] = \text{rank}[y]$ então

6 $\text{rank}[y] \leftarrow \text{rank}[y] + 1$

Union by rank

MAKE-SET(x)

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

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

UNION(x, y)

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

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

1 se $\text{rank}[x] > \text{rank}[y]$ então

2 $\text{pai}[y] \leftarrow x$

3 senão

4 $\text{pai}[x] \leftarrow y$

5 se $\text{rank}[x] = \text{rank}[y]$ então

6 $\text{rank}[y] \leftarrow \text{rank}[y] + 1$

Union by rank

MAKE-SET(x)

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

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

UNION(x, y)

1 $\text{LINK}(\text{FIND-SET}(x), \text{FIND-SET}(y))$

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

1 se $\text{rank}[x] > \text{rank}[y]$ então

2 $\text{pai}[y] \leftarrow x$

3 senão

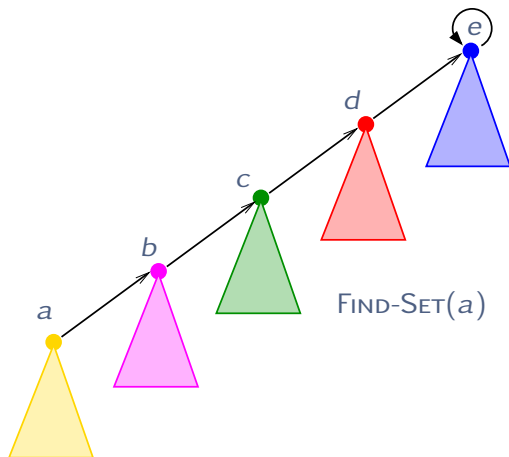
4 $\text{pai}[x] \leftarrow y$

5 se $\text{rank}[x] = \text{rank}[y]$ então

6 $\text{rank}[y] \leftarrow \text{rank}[y] + 1$

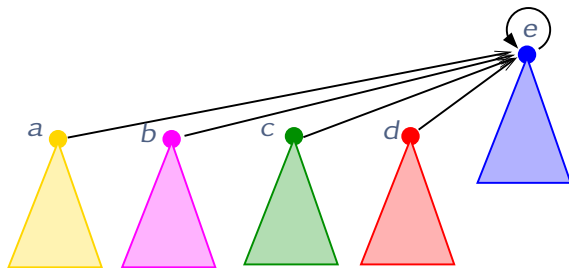
Path compression

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



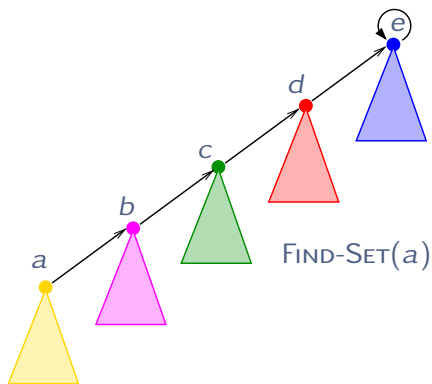
Path compression

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(*a*)

Path compression



FIND-SET(x)

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

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**

- ▶ suponha que realizamos m chamadas no total
- ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**

- ▶ suponha que realizamos m chamadas no total
- ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**

- ▶ suponha que realizamos m chamadas no total
- ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

Análise das heurísticas

Analisando separadamente

1. se utilizarmos somente **union by rank**
 - ▶ suponha que realizamos m chamadas no total
 - ▶ o tempo total será $O(m \lg n)$ (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 $O(m\alpha(n))$
- ▶ $\alpha(n)$ é uma função que cresce **muito muito** lentamente

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

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
- ▶ mas num ritmo realmente devagar
- ▶ uma demonstração está em CLRS

Estimando o tempo amortizado

Quão pequeno é o custo de cada operação?

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

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

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

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

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)

AGM-KRUSKAL(G, w)

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

Complexidade:

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

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

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

Complexidade:

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

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

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

Complexidade:

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

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

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

Complexidade:

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

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

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

Complexidade:

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

O algoritmo de Kruskal (de novo)

AGM-KRUSKAL(G, w)

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

Complexidade:

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