Construção e Análise de Algoritmos

aula 16: Árvores geradoras mínimas

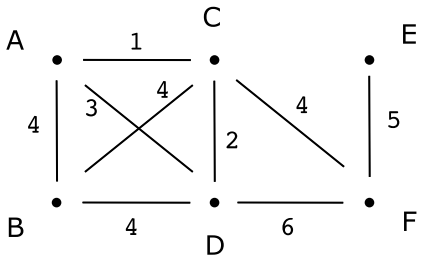
1 Introdução

> *O governo do estado anunciou um novo projeto de infraestrutura que está causando desconfiança na capital.*
>
> *Ele promete melhorar a vida dos habitantes de várias cidades que ninguém consegue encontrar no mapa: Cajuzeiro do Norte, Quicá Dá, Qui Será Meufim, Itamendoin, Guaranámiranga, Pacotin, e por aí vai ...*
>
> *A ideia é asfaltar estradas que conectam essas cidades, de modo que qualquer um possa ir de uma cidade a outra (passando por mais cidades, se necessário), viajando apenas por estradas asfaltadas.*
>
> *Nem todo par de cidades possui uma estrada entre elas.*
>
> *E o custo de asfaltar estradas diferentes também pode ser diferente.*
>
> *O governo afirma, é claro, que espera gastar a menor quantidade de recursos possível nesse projeto.*

Pronto, aqui nós temos mais um problema de otimização.

A melhor maneira de visualizar o problema é através de um grafo (i.e., um diagrama de vértices e arestas).

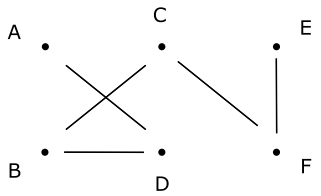
Abaixo nós temos um exemplo.

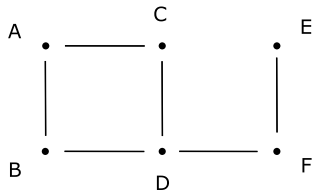


Cada vértice do grafo corresponde a uma cidade do problema, e a presença de uma aresta entre dois vértices indica uma estrada entre as respectivas cidades, que pode ser asfaltada. O número ao lado da aresta corresponde ao custo de asfaltar essa estrada.

Não é difícil ver que uma solução para o problema corresponde a um subconjunto das arestas que conectam todos os vértices.

Abaixo nós temos dois exemplos





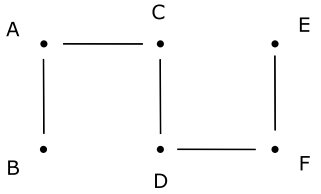
O segundo exemplo já nos permite fazer uma primeira observação importante:

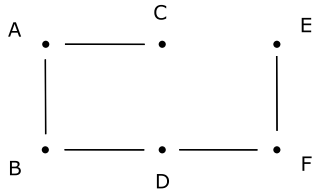
- Como o objetivo é minimizar o custo total, a solução não deve conter um ciclo.

(Porque?)

Para eliminar o ciclo, basta remover qualquer uma de suas arestas.

Por exemplo,





Note que, ao remover uma aresta do ciclo, a segunda solução passa a ter o mesmo número de arestas que a primeira.

Essa observação nos dá uma segunda informação importante sobre o nosso problema:

- Qualquer solução para um problema com n vértices (ou cidades) deve conter exatamente $n - 1$ arestas

(Porque?)

Portanto, qualquer solução para o problema é sempre

- uma coleção de $n - 1$ arestas
- que não forma ciclos
- e que conecta todos os vértices

Não é difícil ver que um problema pode ter diversas soluções válidas.

E o objetivo é encontrar uma solução de custo mínimo.

2 Estratégia gulosa

Uma última observação importante é que nós podemos eliminar a terceira condição acima:

- uma coleção de $n - 1$ arestas
- que não forma ciclos

(X) x xxx xxxxxxx xxxxx xx xxxxxxxx

Quer dizer, toda coleção com $n-1$ arestas que não forma ciclos certamente conecta todos os vértices.

(Porque?)

Essa observação implica que nós só precisamos nos preocupar em construir uma coleção com $n-1$ arestas sem ciclos, que possui custo mínimo.

Vendo as coisas dessa maneira, a estratégia gulosa parece evidente:

- Vá pegando as arestas de menor custo até formar uma coleção de tamanho $n-1$

Quase isso ...

É preciso tomar cuidado para não formar ciclos durante o processo.

Abaixo nós temos uma descrição mais precisa do procedimento guloso

```
Ordenar as arestas de acordo com o seu custo

Para cada aresta nessa lista

   Se a aresta não forma ciclo com as arestas da solução

      Selecione essa aresta para a solução
```

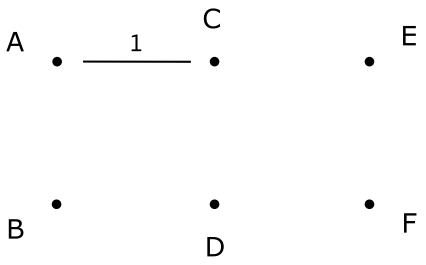
Vejamos como esse procedimento resolve o nosso problema exemplo.

Ordenando as arestas de acordo com o seu custo (e resolvendo os empates pela ordem alfabética), nós obtemos

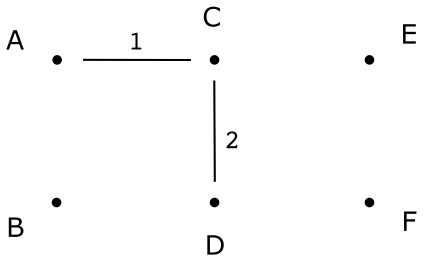
- $A \overset{1}{—} B$
- $C \overset{2}{—} D$
- $A \overset{3}{—} D$
- $A \overset{4}{—} B$
- $B \overset{4}{—} C$
- $B \overset{4}{—} D$
- $C \overset{4}{—} F$
- $E \overset{5}{—} F$
- $D \overset{6}{—} F$

Agora, basta aplicar a regra gulosa.

A primeira aresta não forma ciclo porque ainda não existe nenhuma aresta na solução, logo ela é selecionada

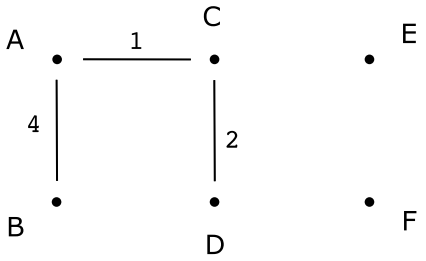


A segunda aresta também não forma ciclo

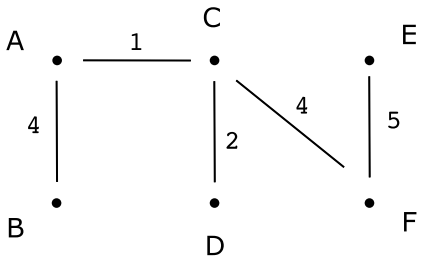


Mas a terceira forma, logo ela é ignorada.

A quarta aresta não forma ciclo, e é selecionada para a solução.



Continuando dessa maneira o procedimento eventualmente produz a seguinte solução



É possível verificar que essa é uma solução ótima.

(Quer tentar?)

Mas, será que o procedimento guloso encontra uma solução ótima em todos os casos?

3 Argumento de otimalidade

Sim.

E, para demonstrar isso, nós vamos construir um argumento utilizando a mesma estratégia da aula passada.

Isto é, nós vamos comparar a solução encontrada pelo nosso algoritmo guloso com uma outra solução qualquer.

Denote por $S = \{s_1, s_2, \ldots, s_{n-1}\}$ a árvore encontrada pelo algoritmo guloso — onde as arestas $s_1, s_2, \ldots$ aparecem na ordem em que elas foram selecionadas pelo algoritmo.

E seja $T = \{t_1, t_2, \ldots, t_{n-1}\}$ uma outra árvore geradora do grafo que alguém encontrou (sabe-se lá como ...).

Dessa vez, a ordem $t_1, t_2, \ldots$ não tem qualquer significado.

Logo, não faz sentido comparar, por exemplo, as arestas s_1 e t_1.

Portanto, nós vamos raciocinar de maneira ligeiramente diferente do que fizemos na aula passada.

Vejamos.

O nosso objetivo é mostrar que $\mathsf{custo}(S) \leq \mathsf{custo}(T)$.

E a primeira observação é que a aresta s_1 pode fazer parte da árvore T ou não.

Suponha que não.

Nesse caso, nós podemos adicionar a aresta s_1 a T.

Fazendo isso, a árvore T deixa de ser uma árvore, pois ela passa a ter um ciclo.

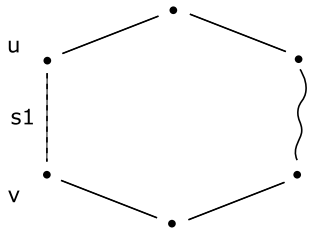
Porque?

A resposta é fácil.

Suponha que s_1 seja a aresta (u, v).

Como T é uma árvore geradora, já havia em T um caminho entre u e v.

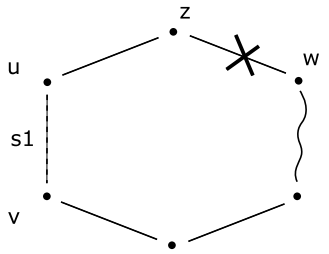
Quando nós adicionamos a aresta $s_1 = (u, v)$ a T, esse caminho se fecha em um ciclo



A próxima observação é que s_1 é uma aresta de custo mínimo do grafo — pois ela foi a primeira aresta selecionada pelo algoritmo guloso.

Isso significa que todas as outras arestas do ciclo possuem custo ao menos $\mathsf{custo}(s_1)$ — talvez mais.

A ideia, então, é remover uma das outras arestas do ciclo, digamos, (z, w)



A observação chave, a seguir, é que após a adição da aresta $s_1 = (u, v)$ e a remoção da aresta (u, w)

$$T' \quad = \quad T \ \cup \ \{s_1\} \ - \ (z, w)$$

nós ainda temos uma árvore geradora do grafo.

(Porque?)

Além disso, como $\mathsf{custo}(s_1) \leq \mathsf{custo}(z, w)$, segue que

$$\mathsf{custo}(T') \quad \leq \quad \mathsf{custo}(T)$$

Nesse ponto, é conveniente fazer uma pausa para esclarecer as coisas.

O nosso objetivo inicial era mostrar que

$$\mathsf{custo}(S) \quad \leq \quad \mathsf{custo}(T) \tag{1}$$

Agora, nós acabamos de transformar T e T', satisfazendo

$$\mathsf{custo}(T') \quad \leq \quad \mathsf{custo}(T) \tag{2}$$

Mais adiante, se nós conseguirmos mostrar que

$$\mathsf{custo}(S) \quad \leq \quad \mathsf{custo}(T') \tag{3}$$

então (2) e (3) vão implicar que (1) vale.

Portanto, a partir de agora nós vamos trabalhar com a árvore T'*.

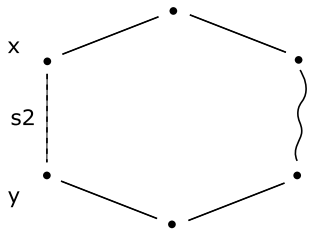
*No caso em que T já tinha a aresta s_1, nós simplesmente passamos a chamar T de T'.

Vamos adiante.

A seguir, nós observamos que a aresta s_2 pode fazer parte de T' ou não.

Suponha que não.

Nesse caso, nós podemos adicionar a aresta $s_2 = (x, y)$ a T', criando um ciclo.



Dessa vez é preciso um pouco mais de cuidado, pois a aresta s_1 pode fazer parte do ciclo, e é possível que $\mathsf{custo}(s_1) < \mathsf{custo}(s_2)$.

Mas, apesar disso, nós podemos afirmar com segurança que o ciclo contém alguma aresta $(g, h) \neq s_1, s_2$.

Porque?

Ora, porque s_1 e s_2 sozinhas não podem formar um ciclo — as duas arestas fazem parte de S, e S não possui ciclos.

Portanto, é possível identificar alguma aresta $(g, h) \neq s_1, s_2$ nesse ciclo e removê-la, para obter uma nova árvore geradora:

$$T'' \quad = \quad T' \ \cup \ \{s_2\} \ - \ (g, h)$$

Além disso, nós podemos concluir que

$$\mathsf{custo}(s_2) \quad \leq \quad \mathsf{custo}(g, h)$$

(Porque?)

E isso implica que

$$\mathsf{custo}(T'') \quad \leq \quad \mathsf{custo}(T')$$

A seguir, nós vamos trabalhar com T''[†].

Note que a coisa está ficando repetitiva ...

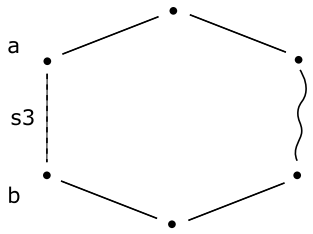
Mas, nós vamos dar mais um passo no argumento para examinar um último detalhe: o caso em que o algoritmo guloso descarta uma aresta.

[†]Mais uma vez, se a aresta s_2 já faz parte de T', nós simplesmente passamos a chamar T' de T''.

Como usual, nós observamos que a aresta s_3 pode fazer parte de T'' ou não.

Suponha que não.

Nesse caso, nós podemos adicionar a aresta $s_3 = (a, b)$ a T'', criando um ciclo



Como no caso anterior, esse ciclo pode conter as arestas s_1 e s_2, que podem ter custo menor que $\mathsf{custo}(s_3)$.

Além disso, em princípio, o ciclo pode conter arestas que foram descartadas pelo algoritmo guloso antes da seleção de s_3 — e que podem ter custo menor que $\mathsf{custo}(s_3)$.

Mas, isso é impossível!

Porque?

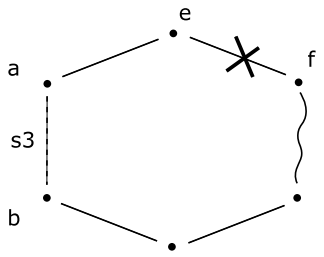
Ora, nós sabemos que a árvore T'' contém as arestas s_1, s_2.

Logo, se T'' contivesse alguma aresta descartada pelo algoritmo guloso (antes da seleção de s_3), então T'' teria um ciclo.

Mas, isso não é o caso.

Com base nessa observação, nós podemos dizer com segurança que o ciclo contém alguma aresta (e, f) diferente de qualquer aresta examinada pelo algoritmo até a seleção de s_3.

Removendo essa aresta do ciclo



nós obtemos mais uma árvore geradora

$$T''' \quad = \quad T'' \ \cup \ \{s_3\} \ - \ (e, f)$$

com a garantia de que

$$\mathsf{custo}(T''') \quad \leq \quad \mathsf{custo}(T'')$$

E, a seguir, nós continuamos trabalhando com T'''.

(Você quer tentar dar mais um passo?)

Prosseguindo dessa maneira, nós vamos construindo uma sequência de árvores $T', T'', T''', \ldots$ que possuem cada vez mais arestas de S.

No final das contas, nós vamos obter uma árvore $T''^{\cdots}$ que possui todas as arestas de S:

$$T''^{\cdots} \quad = \quad \{s_1, s_2, \ldots, s_{n-1}\}$$

Ora, mas nesse caso $T''^{\cdots}$ é a própria árvore S, de modo que a seguinte igualdade é trivial

$$\mathsf{custo}(S) \quad \leq \quad \mathsf{custo}(T''^{\cdots})$$

Finalmente, com a série de desigualdades produzidas pelo argumento

$$\mathsf{custo}(T''^{\cdots}) \quad \leq \quad \ldots \quad \leq \quad \mathsf{custo}(T'') \quad \leq \quad \mathsf{custo}(T') \quad \leq \quad \mathsf{custo}(T)$$

nós podemos concluir que

$$\mathsf{custo}(S) \quad \leq \quad \mathsf{custo}(T)$$

Esse argumento demonstra que a árvore S encontrada pelo algoritmo guloso tem custo menor ou igual ao custo de uma árvore qualquer T.

E isso significa que S é uma solução ótima.

4 Algoritmo e análise de complexidade

Agora que sabemos que a nossa estratégia gulos é ótima, só nos resta construir o algoritmo e analisar o seu tempo de execução.

Dessa vez, como o procedimento guloso é muito simples, nós podemos implementá-lo na forma de um único laço.

```
 Procedimento  AGM-Gul ( V,A: listas de vértices e arestas )
  {
1.   Ordenar a lista A de acordo com o custo das arestas

2.   S  <--  vazio

3.   Para cada aresta  ai  em  A  (na ordem acima)
      {
4.        Se ( ai não forma ciclo com as arestas de S )

5.             Incluir a aresta ai na solução
      }
6.   Retorna (S)
  }
```

Agora, denote por n, m as quantidades de vértices e arestas do grafo, respectivamente.

É imediato ver que a linha `1` executa em tempo $\Theta(n \log n)$.

E também é imediato ver que o laço das linhas `3-5` dá m voltas.

Caso as linhas no interior do laço executassem em tempo $O(1)$, a análise acabava aqui.

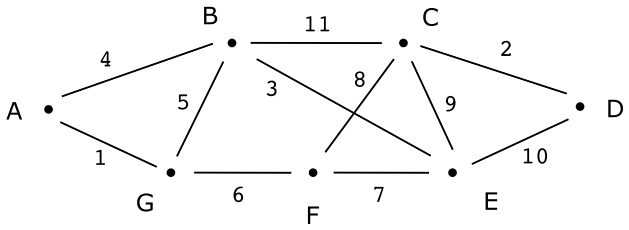
Mas, a tarefa de verificar se um conjunto de arestas forma um ciclo pode levar mais tempo do que isso — dependendo de como a coisa é implementada.

(Pense um pouco sobre isso ...)

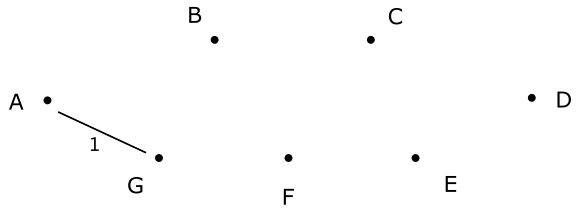
Portanto, a chave para a eficiência do algoritmo está em encontrar uma maneira inteligente de realizar esse teste.

E, para fazer isso, é útil examinar em mais detalhe a maneira como o algoritmo trabalha.

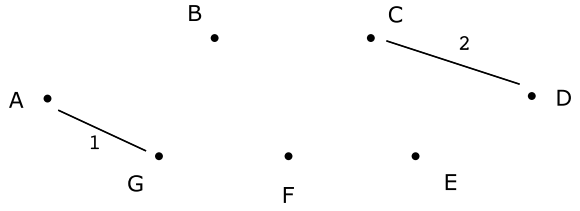
Considere esse outro exemplo de grafo



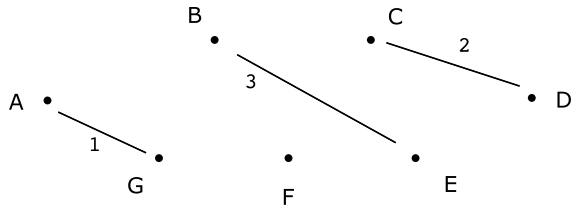
Segundo a lógica do nosso algoritmo, a primeira aresta a ser incluída na solução é A — G



Depois o algoritmo seleciona a aresta C — D



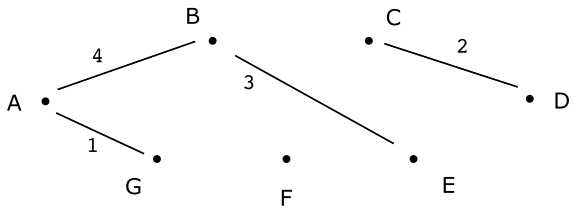
E, a seguir, a aresta B — E é selecionada



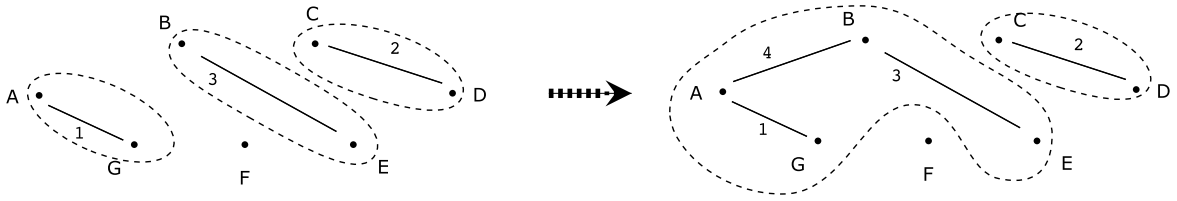
A observação importante aqui é que as arestas que fazem parte da solução (parcial) estão todas espalhadas pelo grafo.

Mas, veja o que acontece em seguida.

A próxima aresta selecionada pelo algoritmo é $A — B$



Aqui aconteceu uma coisa interessante: as arestas da solução que formavam 3 grupos de vértices (ou componentes), agora formam apenas 2



Isso faz sentido, pois a ideia é que no final do processo as arestas formem

Finalmente, dando mais um passo, nós descobrimos uma coisa ainda mais interessante.

A aresta que é considerada pelo algoritmo em seguida é $B — G$.

```
< Figura: detecção de ciclo >
```

Mas, como se pode ver na figura, ela não será incluída na solução pois ela forma um ciclo com as demais.

E agora nós temos uma maneira esperta de detectar ciclos:

- no momento em que uma aresta (u, v) é considerada para fazer parte da solução
- basta verificar se u e v estão no mesmo componente
- se isso for o caso, então a aresta forma um ciclo com as demais

O próximo passo consiste em implementar essa solução no algoritmo.

Especificamente, nós temos 2 problemas:

- encontrar uma maneira de armazenar as informações sobre os componentes
- encontrar uma maneira (eficiente) de atualizar essas informações,
  a medida que o algoritmo executa

A solução para o primeiro problema é relativamente simples.

Nós vamos imaginar que todo componente tem um *chefe*.

E vamos assumir que todo vértice "*conhece*"o chefe do seu componente:

`u.chefe` : indica o chefe do componente em que u se encontra

E agora é fácil: para testar se dois vértices u, v estão no mesmo componente, basta fazer

```
Se ( u.chefe = v.chefe )
```

Se eles estiverem, então a aresta (u, v) forma um ciclo com aquelas que já estão na solução, e deve ser descartada.

Mas, se eles não estiverem, então a aresta (u, v) será incluída na solução, e os componentes de u e v serão combinados em um único componente.

Na prática, isso significa que os campos `.chefe` de um dos componentes devem ser atualizados com o chefe do outro componente.

E aqui surge a oportunidade para uma pequena esperteza ...

Se o componente com chefe A tem apenas 2 vértices e o componente com chefe J tem 10 vértices, então é mais fácil (rápido) atualizar os vértices do componente A do que o contrário.

Isto é, é como se o menor componente fosse absorvido pelo maior.

Para implementar essa solução, nós vamos introduzir mais dois campos que serão utilizados pelos vértices chefe:

`u.tam` : indica o tamanho do componente que tem u como chefe

`u.comp` : lista de vértices do componente que tem u como chefe

E agora é fácil:

- no momento em que a aresta (u, v) é incluída na solução
- nós descobrimos os chefes de u e v

```
a  <--  u.chefe                b  <--  v.chefe
```

- descobrimos qual é o maior componente

```
Se ( a.tam  >  b.tam )
```

- e atualizamos os campos `.chefe` do menor componente

```
Para cada vértice  x  em  b.comp
    x.chefe  <--  a
```

Abaixo nós temos a nova descrição de alto nível do nosso algoritmo guloso.

```
 Procedimento  AGM-Gul ( V,A: listas de vértices e arestas )
  {
1.   Ordenar a lista A de acordo com o custo das arestas

2.   S  <--  vazio

3.   Inicializar componentes:  cada vértice é seu próprio chefe

4.   Para cada aresta  ai=(u,v)  em  A  (na ordem acima)
      {
5.        Se ( u.chefe  é diferente de  v.chefe )

6.           Incluir a aresta  ai  na solução

7.           Atualiza-Componentes (u,v)
      }
8.   Retorna (S)
  }
```

Não é difícil ver que a nova linha 3 executa em tempo $\Theta(n)$.

Mas, nós não temos como determinar o tempo de execução da linha 7.

Quer dizer, esse tempo varia de iteração para iteração — dependendo dos tamanhos dos componentes de u e v.

A solução para essa dificuldade consiste em estimar o tempo total acumulado por todas as chamadas a `Atualiza-Componentes()`.

A primeira vista isso parece difícil, mas outra pequena esperteza resolve o problema.

Note que a "única"coisa que o procedimento `Atualiza-Componentes()` faz é atualizar os chefes dos vértices, por meio de instruções da forma

```
u.chefe  <--  a
```

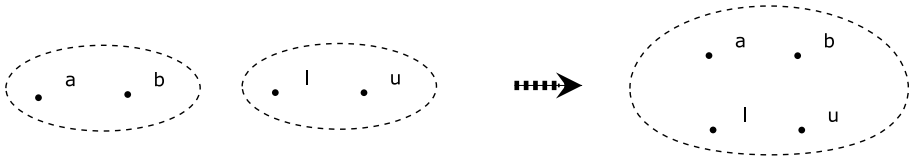
Nesse ponto, nós podemos perguntar:

- *Quantas vezes um vértice pode ter o seu chefe atualizado (no pior caso)?*

Bom, no início, o vértice é o chefe de si mesmo

Mas, em algum momento, ele pode ser absorvido por um outro componente

E, depois, o seu componente pode ser absorvido por um outro componente



A observação chave aqui é que o componente de u só pode ser absorvido por um componente maior ou igual a ele.

E isso significa que, a cada vez que o componente de u é absorvido, ele (ao menos) dobra de tamanho.

Ora, mais o componente de u não pode dobra de tamanho pra sempre ...

Especificamente, como o grafo tem n vértices, o componente de u pode dobrar de tamanho no máximo $\log_2 n$ vezes.

Isso significa que o campo `u.chefe` será atualizado no máximo $\log n$ vezes.

E isso significa que o tempo total acumulado pelo procedimento `Atualiza-Componentes()` é no máximo $O(n \log n)$ — pois nós temos um total de n vértices.

(Não é legal?)

E agora acabou.

- a linha `1` executa em tempo $\Theta(m \log m)$
- a linha `2` executa em tempo $O(1)$
- a linha `2` executa em tempo $O(n)$
- o tempo do laço das linhas `4-7` é dominado pelas chamadas a `Atualiza-Componentes()`, que levam tempo $O(n \log n)$
- e a linha `8` executa em tempo $O(1)$

Portanto, o algoritmo executa em tempo

$$\Theta(m \log m) \;+\; O(1) \;+\; O(n) \;+\; O(n \log n) \quad = \quad \Theta(m \log m)$$

(assumindo que $m \geq n$).