

Universidade Federal de São Carlos - UFSCar

Departamento de Computação - DC

CEP 13565-905, Rod. Washington Luiz, s/n, São Carlos, SP

Algoritmos Gulosos - Parte 3

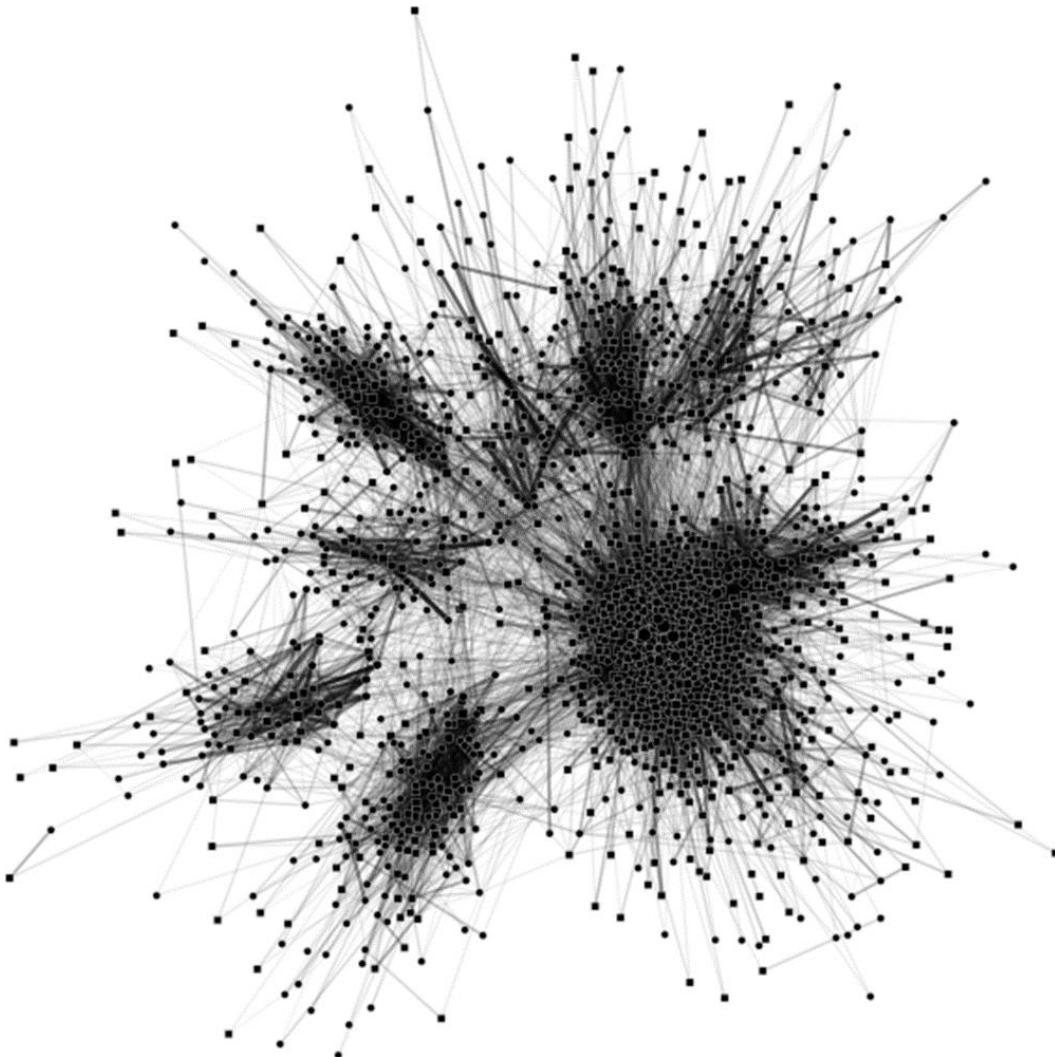
Prof. Dr. Alan Demétrius Baria Valejo

CCO-00.2.01 - Projeto e Análise de Algoritmos (*Design And Analysis Of Algorithms*)

1001525 - Projeto e Análise de Algoritmos - Turma A

- Grafos
 - Definições
 - Árvore Geradora Mínima
 - Prim
 - Kruskal
 - Busca
-



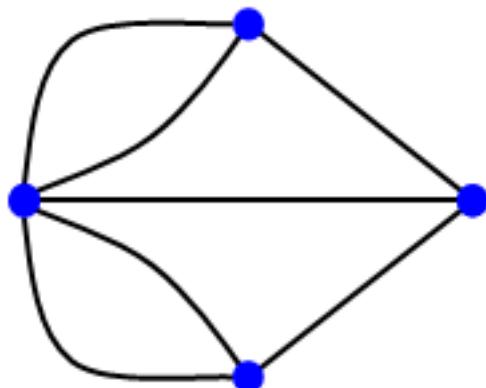
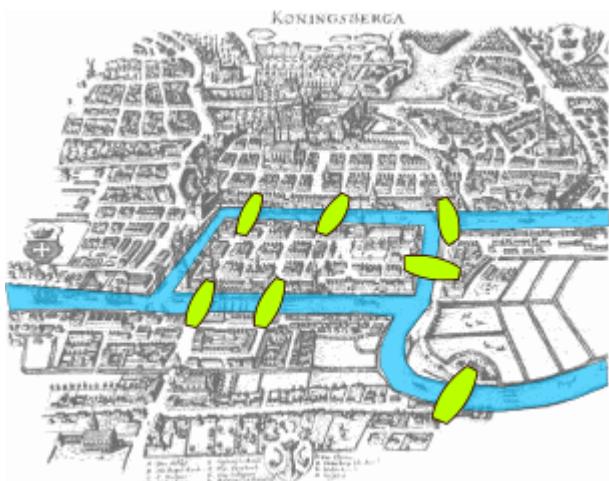


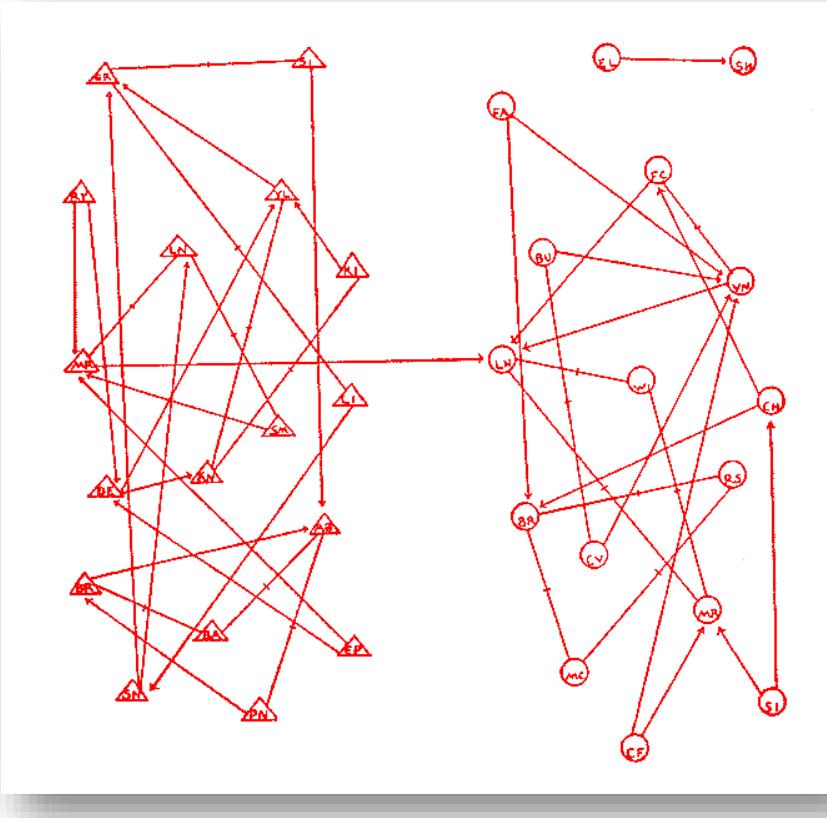
- Grafos são estruturas relacionais utilizadas para representar sistemas ou partes deles, nos quais os objetos do sistema são representados por vértices e as relações entre eles são representadas por arestas.



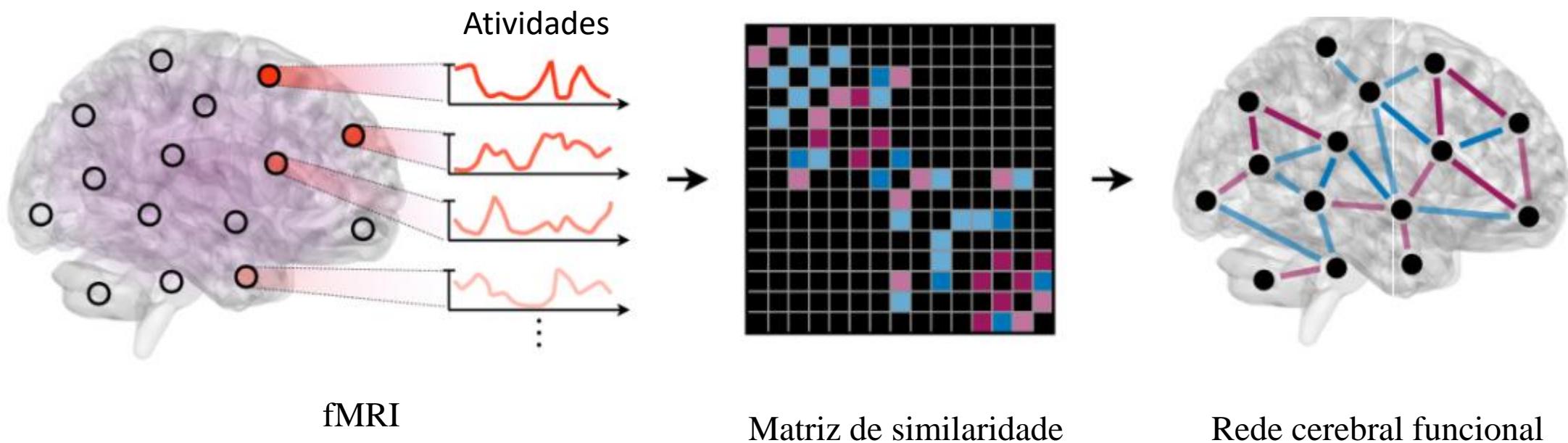
- Grafos são relevantes tanto na matemática quanto na computação, pois conseguem modelar uma grande variedade de cenários, como:
 - Redes físicas (elétrica, comunicações, transportes),
 - redes conceituais (Web, sociais, lógicas, biológicas),
 - estruturas como listas encadeadas e árvores,
 - relações de dependência ou interação (grafo de filmes e atores),
 - mapas, etc.

- Teoria dos Grafos
 - Leonhard Euler (1736)
 - Sete pontes de Königsberg
 - É possível atravessar todas as pontes passando-se apenas uma vez por cada uma delas?
 - “geometria de posição”



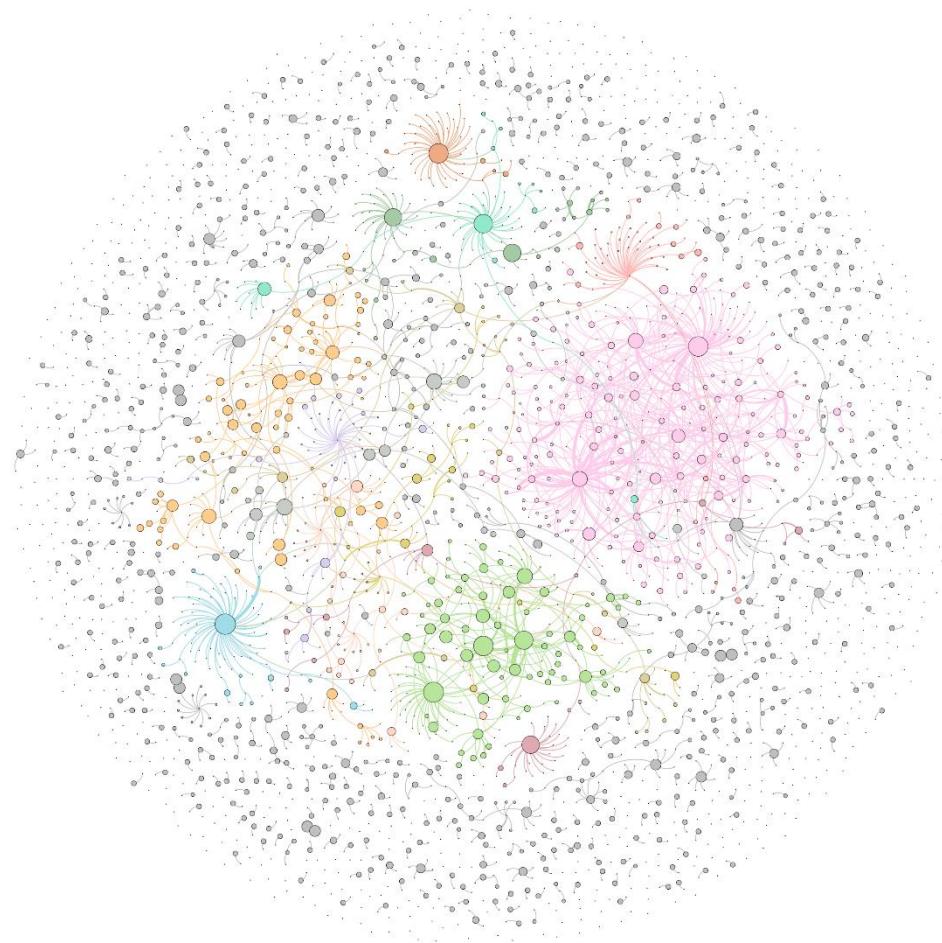


Jacob Moreno e o sociograma (1930)



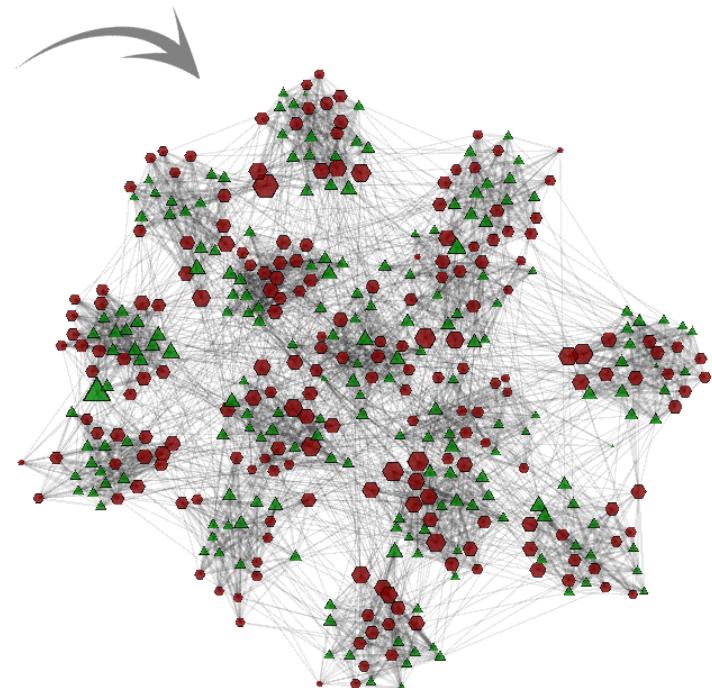
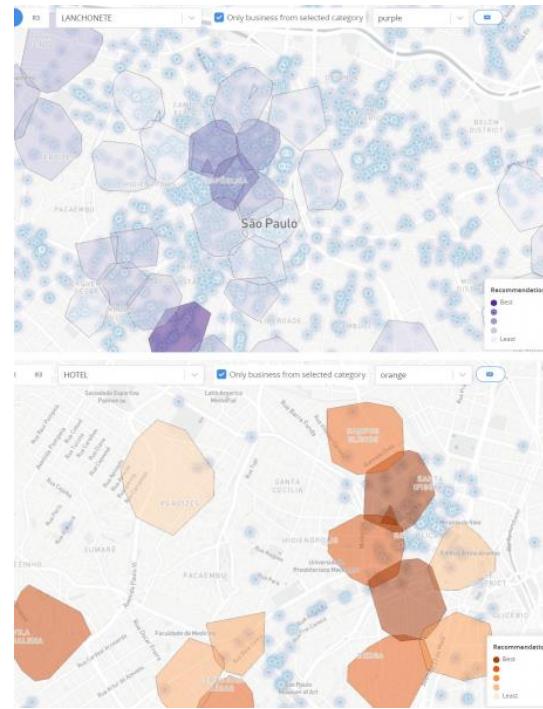
Lynn, Christopher W., and Danielle S. Bassett. “The physics of brain network structure, function and control.”

- Grafo de textos do Twitter
- Propagação de *fake news*



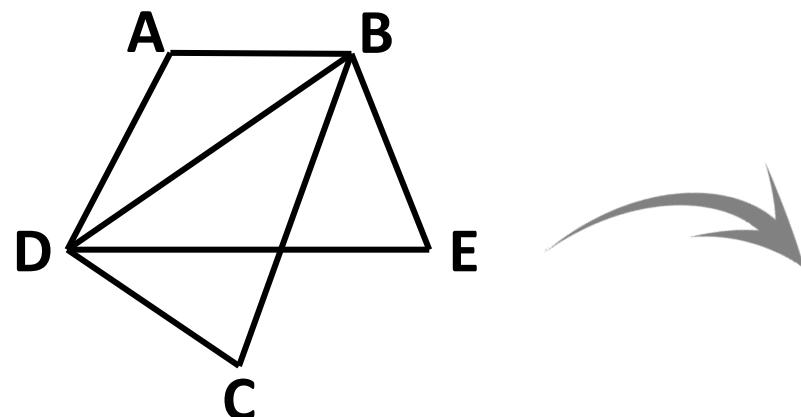
Twitter conversation graph on October 14, 2020. Location was Thailand
<https://medium.com/analytics-vidhya/building-twitter-conversation-graph-5830bee5a7eb>

- Grafos georreferenciados
 - Predição de locais de negócios



Ferreira, V., Valejo, A., Valdivia, P., & Valverde-Rebaza, J. (2019). Exploiting Geographical Data to Improve Recommender Systems for Business Opportunities in Urban Areas. In 2019 8th Brazilian Conference on Intelligent Systems (BRACIS) (pp. 568-573). IEEE.

- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- $\mathcal{V} = \{A, B, C, D, E\}$
- $\mathcal{E} = \{(A, B), (A, D), (B, E), (C, D), (D, E)\}$
- Grafo não direcionado, portanto, $(u, v) = (v, u)$
- Por simplicidade: o número de vértices e arestas são definidos por $n = |\mathcal{V}|$ e $m = |\mathcal{E}|$, respectivamente



Grafo não direcionado e não ponderado

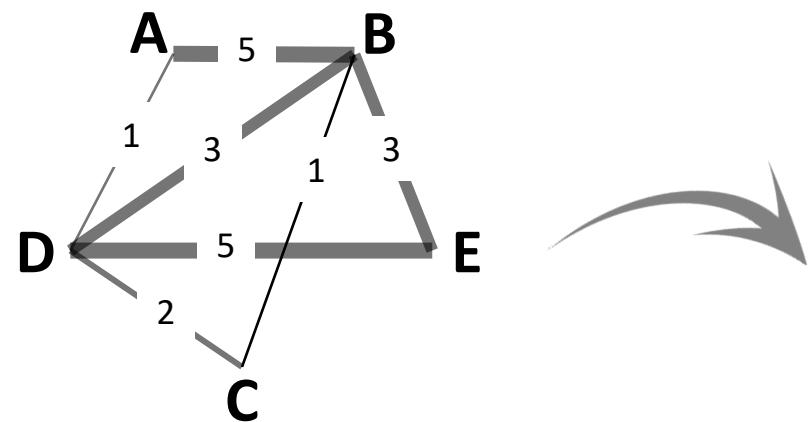
$\mathcal{A}_{n \times n}$ indica matriz de adjacência
 $\mathcal{W}_{n \times n}$ indica matriz ponderada

$$\mathcal{A}_{n \times n} =$$

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	1
C	0	1	0	1	0
D	1	1	1	0	1
E	0	1	0	1	0

Matriz de adjacência

- $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{w})$ tal que $\mathbf{w}(u, v): \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$, é um **grafo ponderado**, sendo \mathbf{w} chamado de **peso** ou **força de associação**
- $\mathcal{V} = \{A, B, C, D, E\}$
- $\mathcal{E} = \{(A, B), (A, D), (B, E), (C, D), (D, E)\}$
- **Grafo não direcionado**, portanto, $(u, v) = (v, u)$



Grafo direcionado e não ponderado

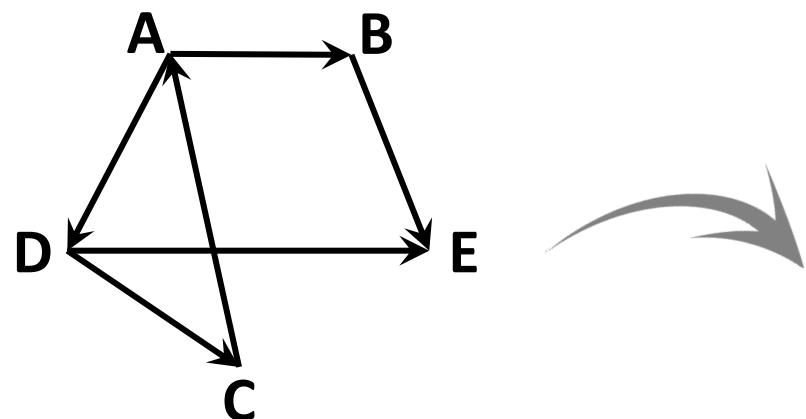
$\mathcal{A}_{n \times n}$ indica matriz de adjacência
 $\mathbf{W}_{n \times n}$ indica matriz ponderada

$$\mathbf{W}_{n \times n} =$$

	A	B	C	D	E
A	0	5	0	1	0
B	5	0	1	3	3
C	0	1	0	1	0
D	1	3	2	0	5
E	0	1	0	2	0

Matriz de adjacência

- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- $\mathcal{V} = \{A, B, C, D, E\}$
- $\mathcal{E} = \{(A, B), (A, D), (B, E), (C, D), (D, E)\}$
- **Grafo direcionado**, portanto, $(u, v) \neq (v, u)$



Grafo não direcionado e não ponderado

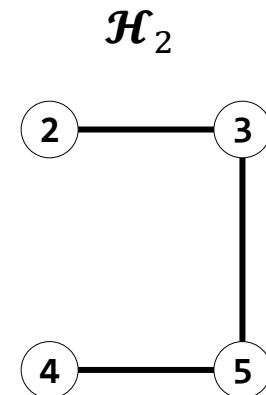
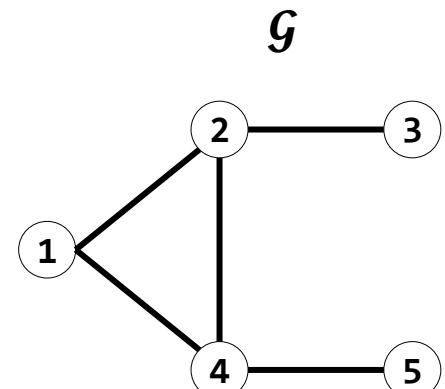
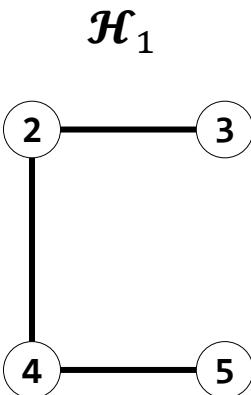
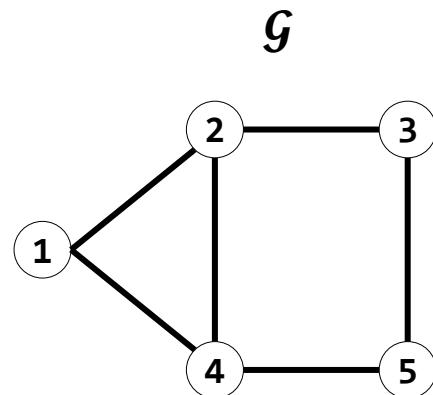
$\mathcal{A}_{n \times n}$ indica matriz de adjacência
 $\mathcal{W}_{n \times n}$ indica matriz ponderada

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	0	1
C	1	0	0	0	0
D	0	0	1	0	1
E	0	0	0	0	0

Matriz de adjacência

Subgrafo

- Um grafo \mathcal{H} é um subgrafo de um grafo \mathcal{G} se $\mathcal{V}(\mathcal{H}) \subseteq \mathcal{V}(\mathcal{G})$ e $\mathcal{E}(\mathcal{H}) \subseteq \mathcal{E}(\mathcal{G})$; escrevemos $\mathcal{H} \subseteq \mathcal{G}$. Neste caso, também dizemos que \mathcal{H} está contido em \mathcal{G} , ou que \mathcal{G} contém \mathcal{H} , ou que \mathcal{G} é um supergrafo de \mathcal{H} .

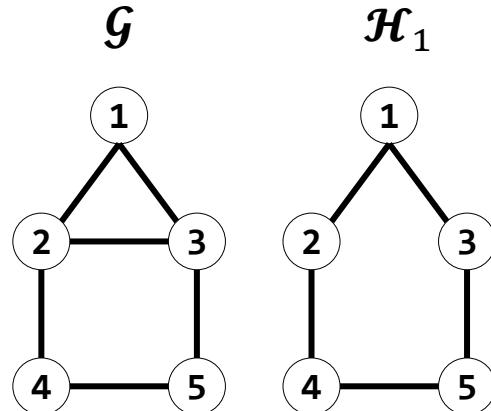


\mathcal{H}_1 é subgrafo de \mathcal{G}

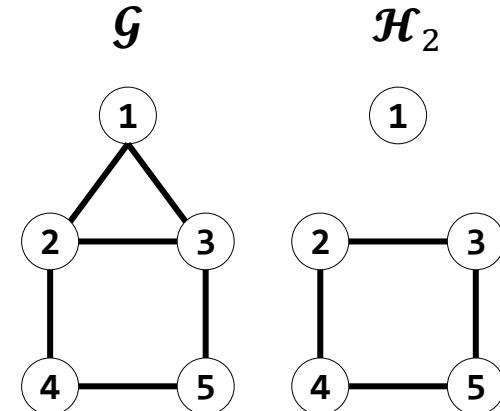
\mathcal{H}_2 não é subgrafo de \mathcal{G}

Subgrafo gerador

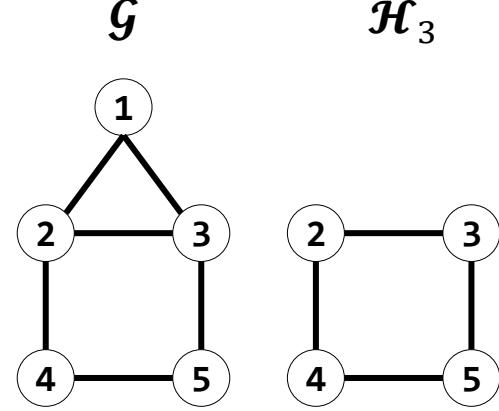
- Dizemos que \mathcal{H} é um subgrafo gerador (*spanning subgraph*) de \mathcal{G} se $\mathcal{H} \subseteq \mathcal{G}$ e $\mathcal{V}(\mathcal{H}) = \mathcal{V}(\mathcal{G})$



\mathcal{H}_1 é subgrafo gerador de \mathcal{G}

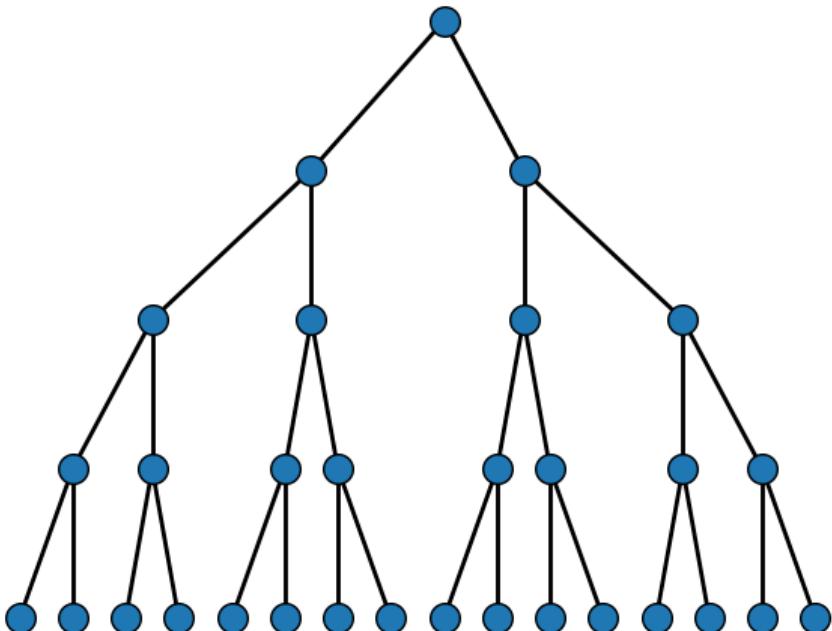


\mathcal{H}_2 é subgrafo gerador de \mathcal{G}

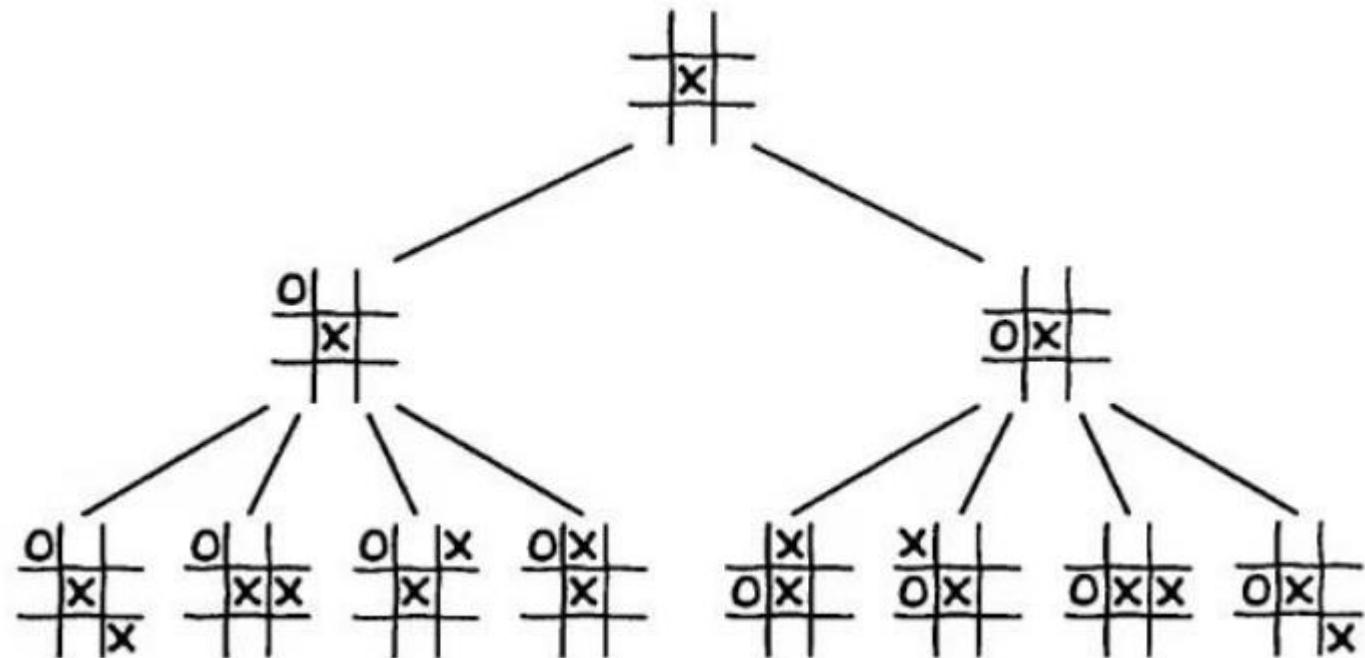


\mathcal{H}_3 não é subgrafo gerador de \mathcal{G}

- Árvore
 - Um grafo não orientado
 - Conexo (existe caminho entre quaisquer dois de seus vértices)
 - Acíclico (não possui ciclos) e sem arestas múltiplas
 - No mínimo $n - 1$ arestas, caso em que o grafo é uma árvore geradora

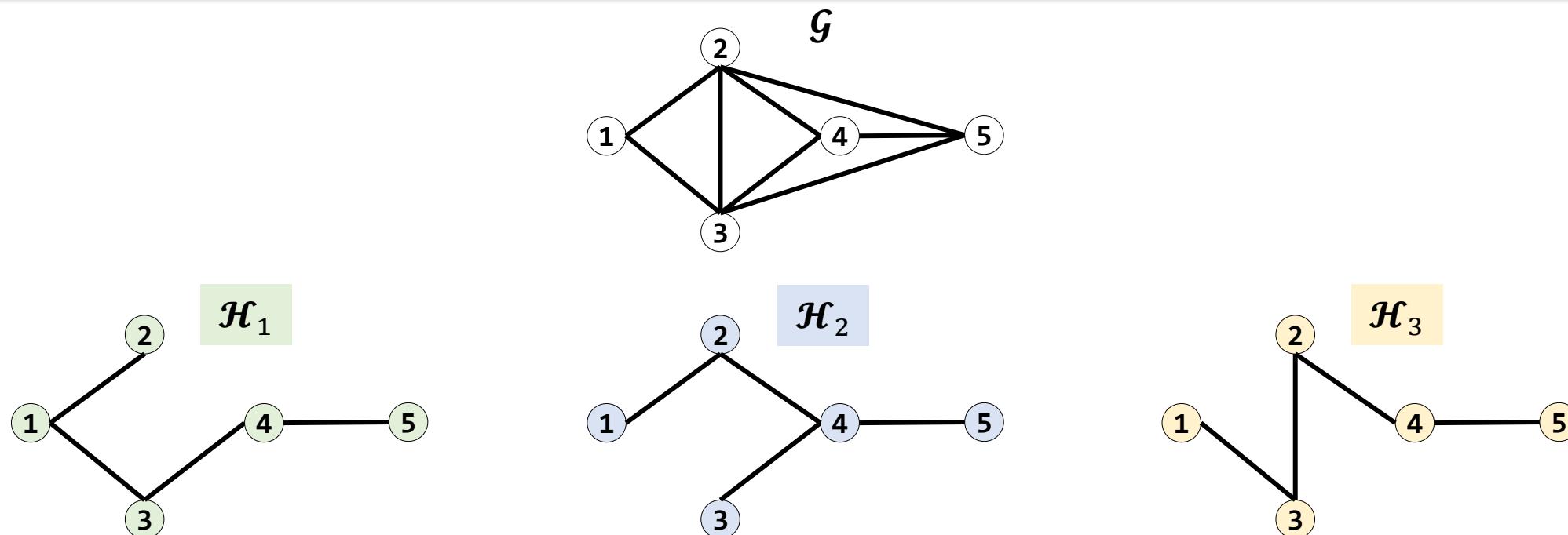


- Jogo da velha
 - Vértices: os estados do jogo.
 - Areias: existe uma aresta entre um estado do jogo e um estado que pode ser obtido através deste.



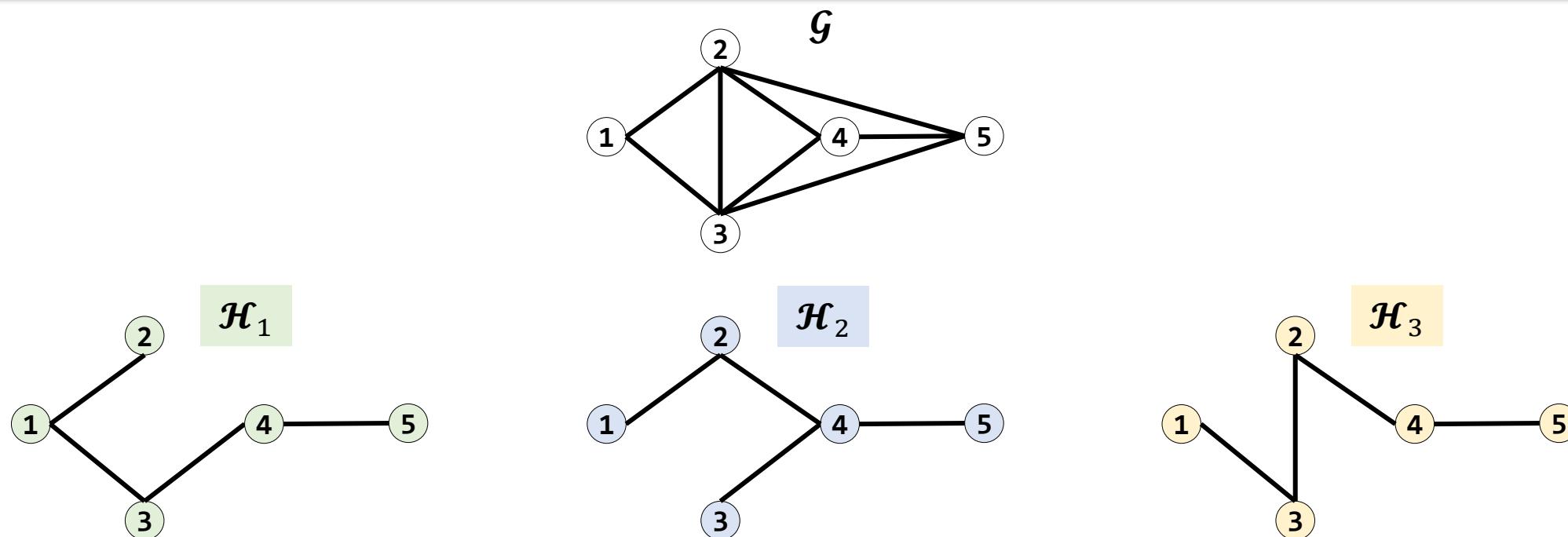
Árvore geradora

- Árvore geradora de um grafo conexo \mathcal{G} é um subgrafo gerador de \mathcal{G} que é uma árvore
 - Árvore geradora \mathcal{H} de \mathcal{G} é um subgrafo de \mathcal{G} constituído de um conjunto de arestas que interligam todos os vértices
 - Ou seja, existe um caminho entre cada par de vértices
 - Toda árvore geradora é um grafo conexo e acíclico
 - Um grafo pode ter várias árvores geradoras
 - Todo grafo conexo possui árvore geradora



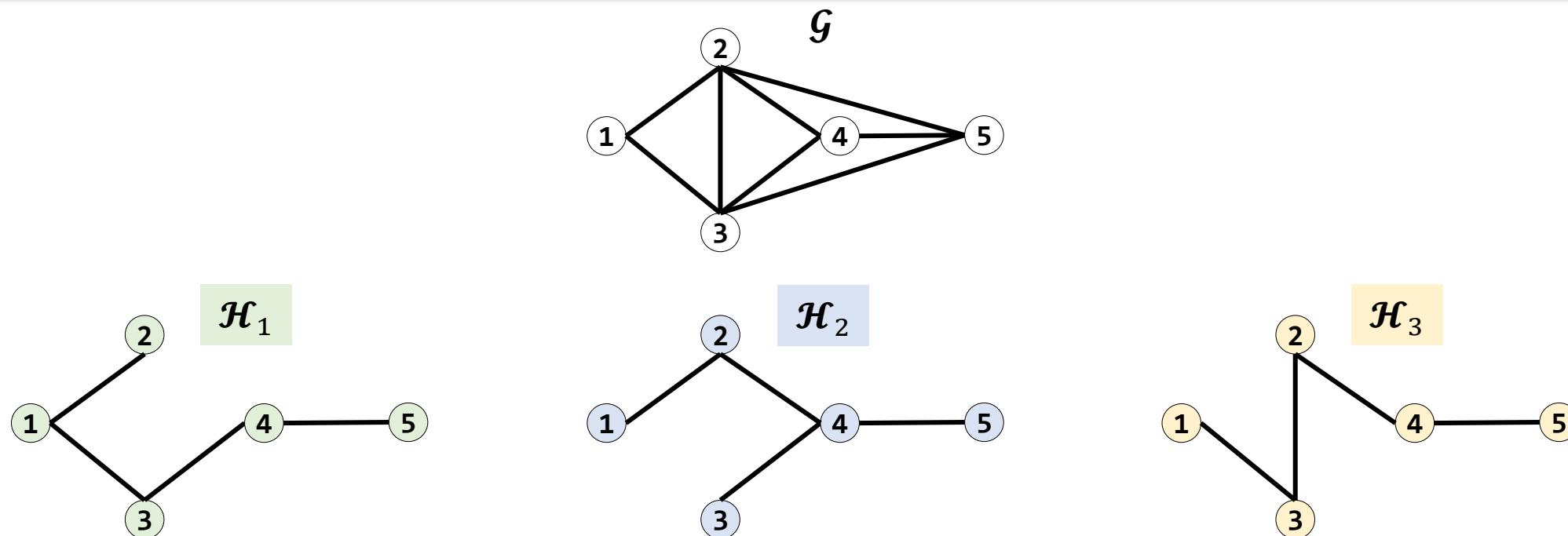
Árvore geradora

- Árvore geradora de um grafo conexo \mathcal{G} é um subgrafo gerador de \mathcal{G} que é uma árvore
 - Árvore geradora \mathcal{H} de \mathcal{G} é um subgrafo de \mathcal{G} constituído de um conjunto de arestas que interligam todos os vértices
 - Ou seja, existe um caminho entre cada par de vértices
 - Toda árvore geradora é um grafo conexo e acíclico
 - Um grafo pode ter várias árvores geradoras
 - Todo grafo conexo possui árvore geradora



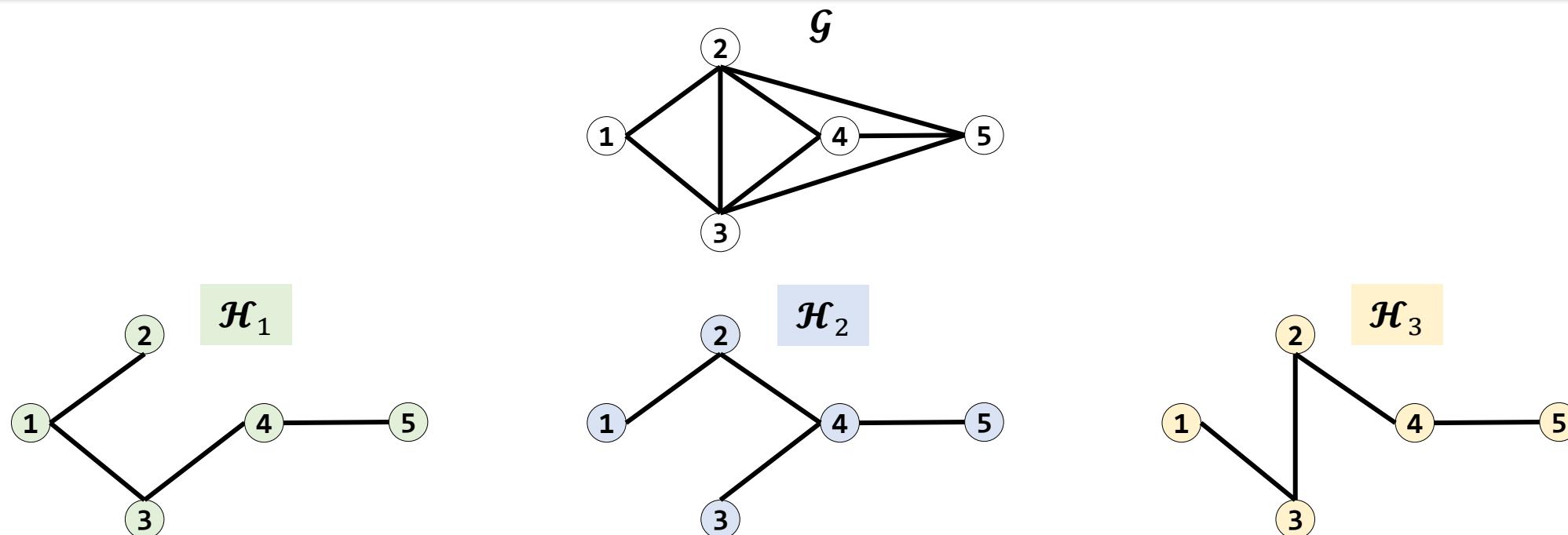
Árvore geradora

- Árvore geradora de um grafo conexo \mathcal{G} é um subgrafo gerador de \mathcal{G} que é uma árvore
 - Árvore geradora \mathcal{H} de \mathcal{G} é um subgrafo de \mathcal{G} constituído de um conjunto de arestas que interligam todos os vértices
 - Ou seja, existe um caminho entre cada par de vértices
 - **Toda árvore geradora é um grafo conexo e acíclico**
 - Um grafo pode ter várias árvores geradoras
 - Todo grafo conexo possui árvore geradora



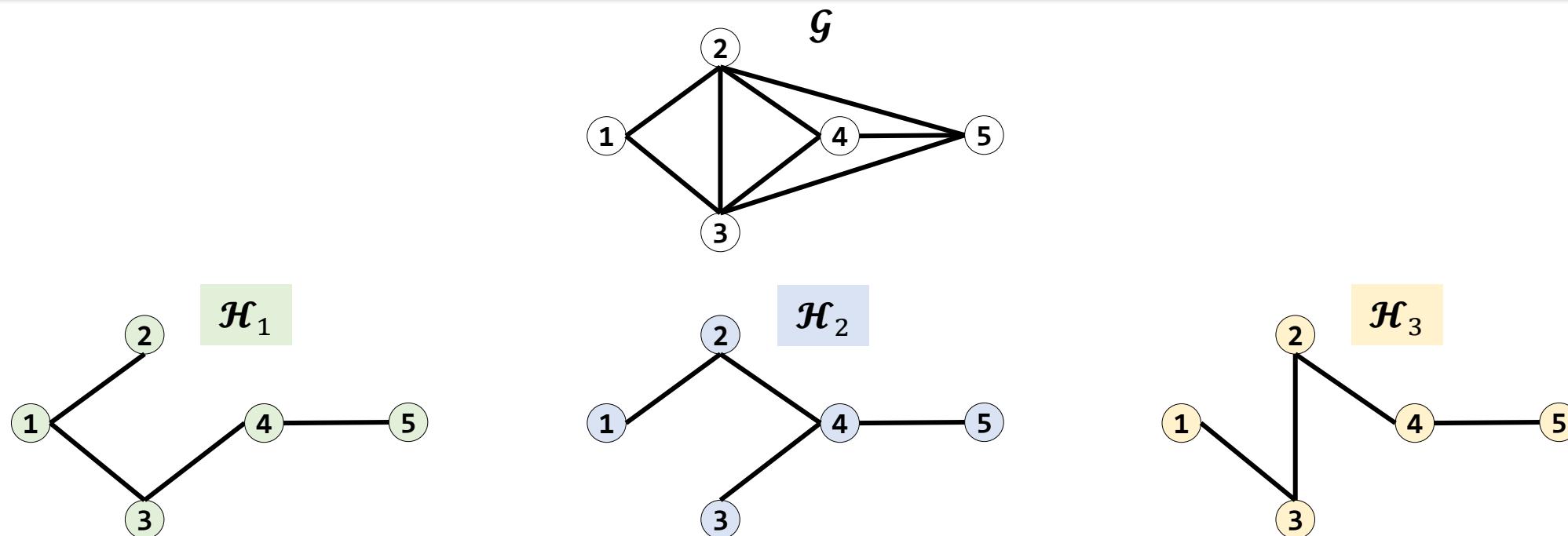
Árvore geradora

- Árvore geradora de um grafo conexo \mathcal{G} é um subgrafo gerador de \mathcal{G} que é uma árvore
 - Árvore geradora \mathcal{H} de \mathcal{G} é um subgrafo de \mathcal{G} constituído de um conjunto de arestas que interligam todos os vértices
 - Ou seja, existe um caminho entre cada par de vértices
 - Toda árvore geradora é um grafo conexo e acíclico
 - Um grafo pode ter várias árvores geradoras
 - Todo grafo conexo possui árvore geradora

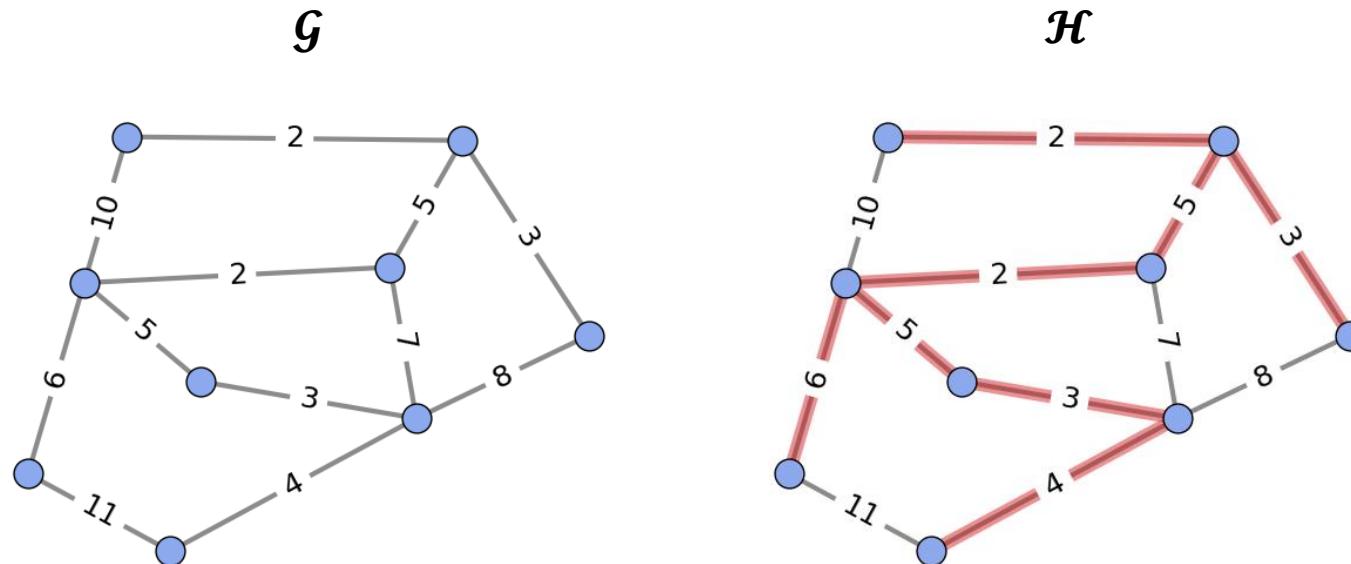


Árvore geradora

- Árvore geradora de um grafo conexo \mathcal{G} é um subgrafo gerador de \mathcal{G} que é uma árvore
 - Árvore geradora \mathcal{H} de \mathcal{G} é um subgrafo de \mathcal{G} constituído de um conjunto de arestas que interligam todos os vértices
 - Ou seja, existe um caminho entre cada par de vértices
 - Toda árvore geradora é um grafo conexo e acíclico
 - Um grafo pode ter várias árvores geradoras
 - **Todo grafo conexo possui árvore geradora**

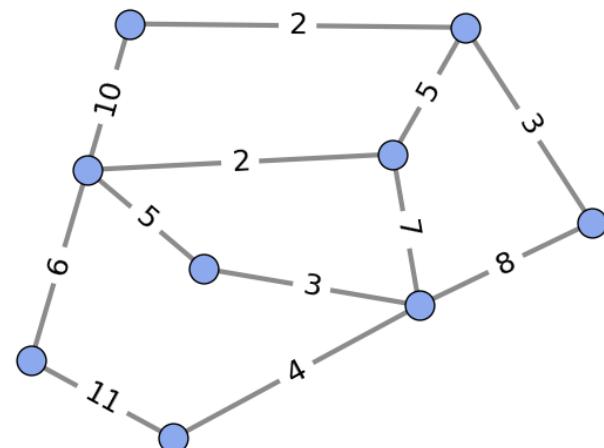


- É uma árvore geradora de um grafo ponderado cuja soma dos pesos das arestas é mínima
- Árvores geradoras mínimas também são conhecidas pela abreviatura MST de *minimum spanning tree*
- Um único grafo pode ter diferentes árvores geradoras mínimas
- Se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST

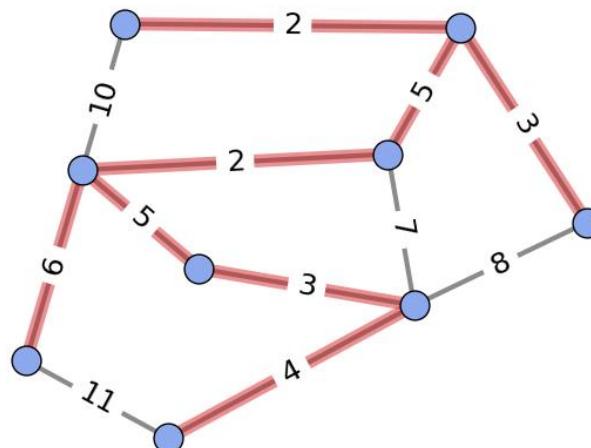


- É uma árvore geradora de um grafo ponderado cuja soma dos pesos das arestas é mínima
- Árvores geradoras mínimas também são conhecidas pela abreviatura MST de *minimum spanning tree*
- Um único grafo pode ter diferentes árvores geradoras mínimas
- Se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST

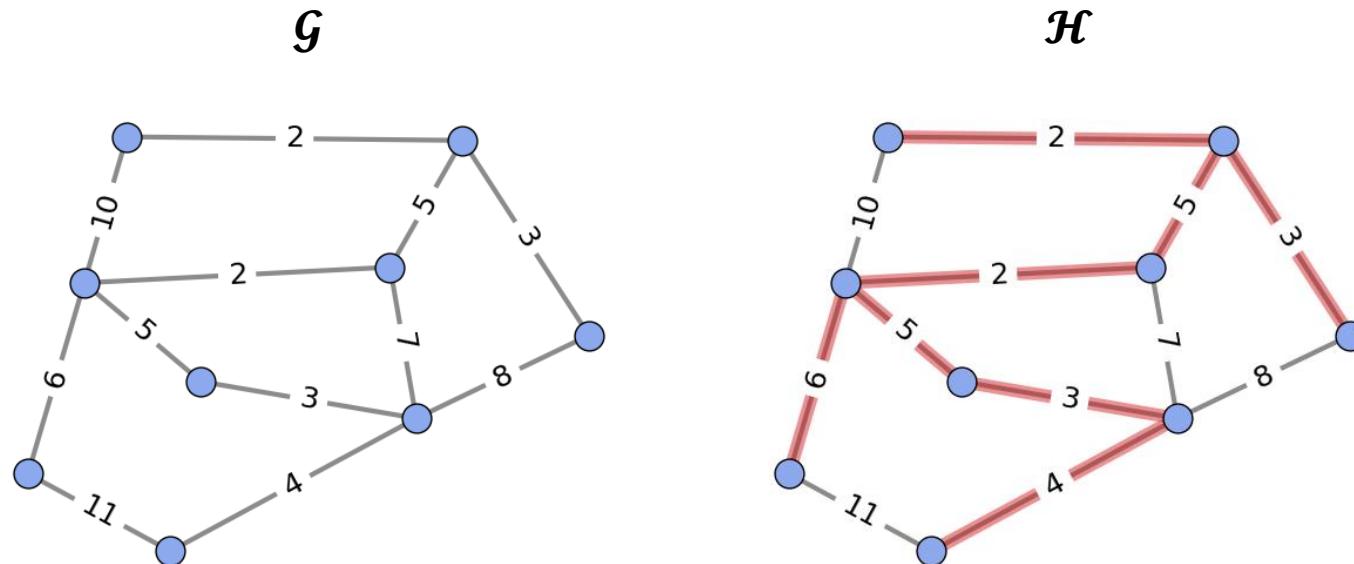
G



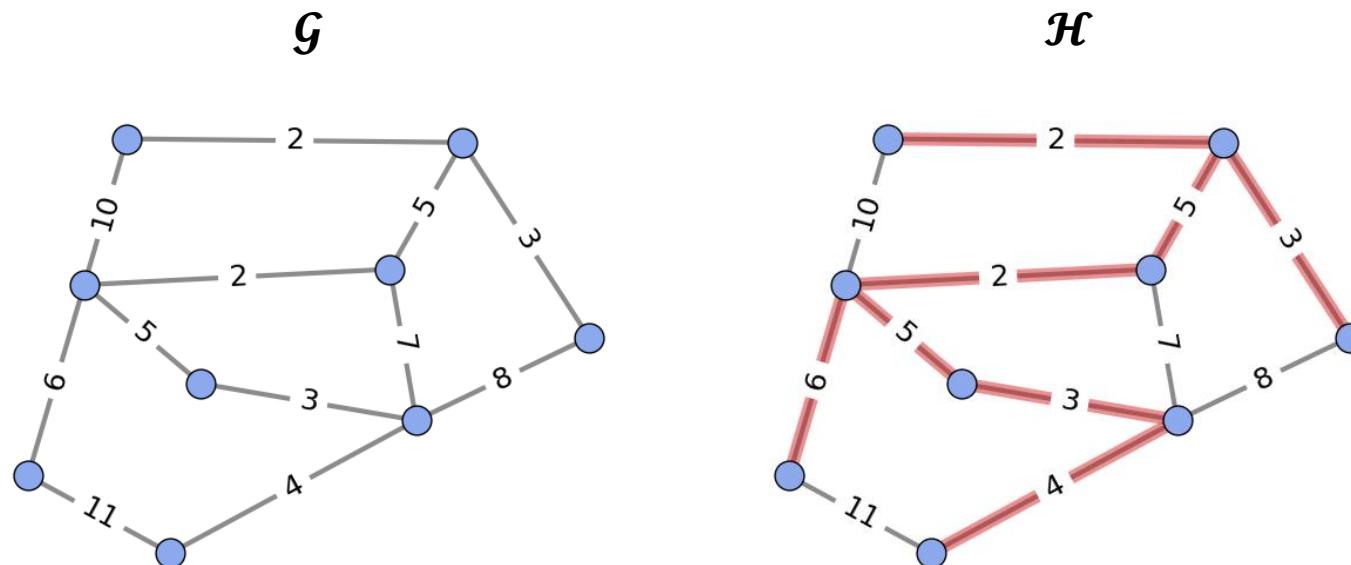
H



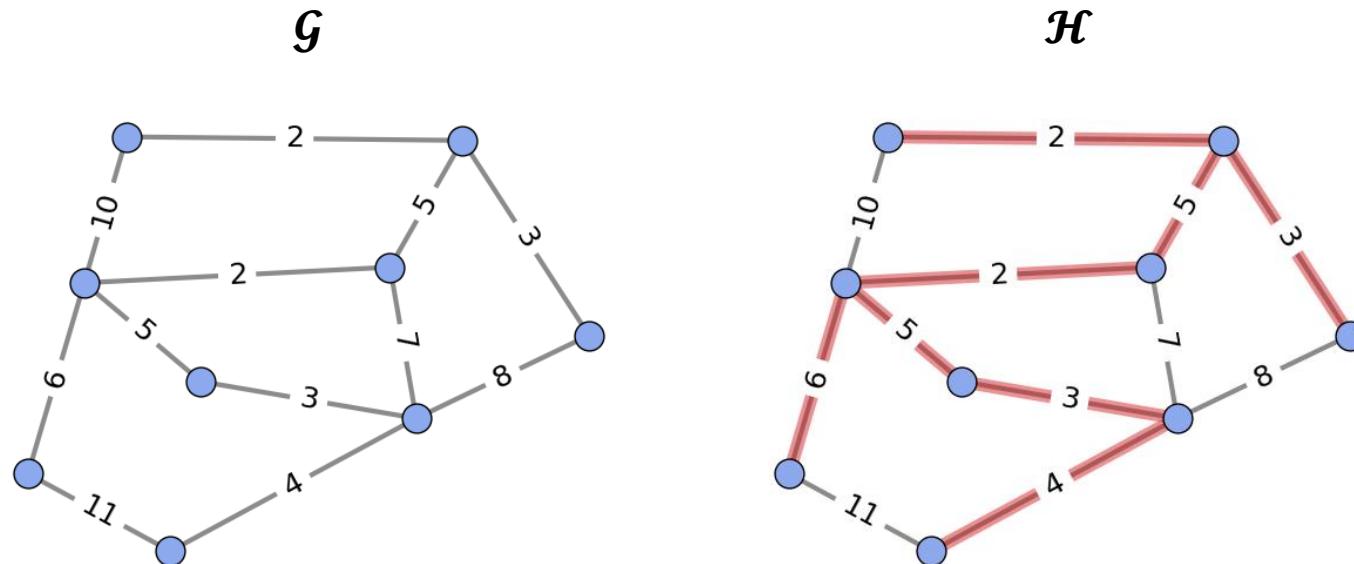
- É uma árvore geradora de um grafo ponderado cuja soma dos pesos das arestas é mínima
- Árvores geradoras mínimas também são conhecidas pela abreviatura **MST** de *minimum spanning tree*
- Um único grafo pode ter diferentes árvores geradoras mínimas
- Se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST



- É uma árvore geradora de um grafo ponderado cuja soma dos pesos das arestas é mínima
- Árvores geradoras mínimas também são conhecidas pela abreviatura MST de *minimum spanning tree*
- Um único grafo pode ter diferentes árvores geradoras mínimas
- Se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST



- É uma árvore geradora de um grafo ponderado cuja soma dos pesos das arestas é mínima
- Árvores geradoras mínimas também são conhecidas pela abreviatura MST de *minimum spanning tree*
- Um único grafo pode ter diferentes árvores geradoras mínimas
- Se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST



- Conectar todos os pontos com custo mínimo
- Cobrir a maior área com menos matéria prima possível
- Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)
 - Reduzir cabos, fibra ótica ...
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- Redes de transmissão de energia
 - Reduzir matéria prima e número de postes de energia

- **Coneectar todos os pontos com custo mínimo**
- Cobrir a maior área com menos matéria prima possível
- Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)
 - Reduzir cabos, fibra ótica ...
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- Redes de transmissão de energia
 - Reduzir matéria prima e número de postes de energia

- Conectar todos os pontos com custo mínimo
- Cobrir a maior área com menos matéria prima possível
- Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)
 - Reduzir cabos, fibra ótica ...
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- Redes de transmissão de energia
 - Reduzir matéria prima e número de postes de energia

- Conectar todos os pontos com custo mínimo
- Cobrir a maior área com menos matéria prima possível
- **Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)**
 - **Reducir cabos, fibra ótica ...**
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- Redes de transmissão de energia
 - Reduzir matéria prima e número de postes de energia

- Conectar todos os pontos com custo mínimo
- Cobrir a maior área com menos matéria prima possível
- Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)
 - Reduzir cabos, fibra ótica ...
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- Redes de transmissão de energia
 - Reduzir matéria prima e número de postes de energia

- Conectar todos os pontos com custo mínimo
- Cobrir a maior área com menos matéria prima possível
- Projetos de redes de telecomunicação (fibra ótica, telefonia, televisão)
 - Reduzir cabos, fibra ótica ...
- Projeto de rodovias
 - Reduzir km em asfalto, reduzir manutenção, ligar o maior número de cidades.
- **Redes de transmissão de energia**
 - Reduzir matéria prima e número de postes de energia

- Prim
 - Kruskal
-

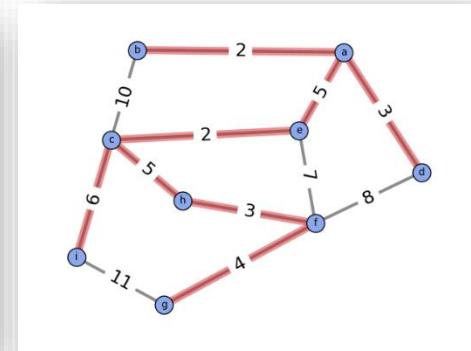
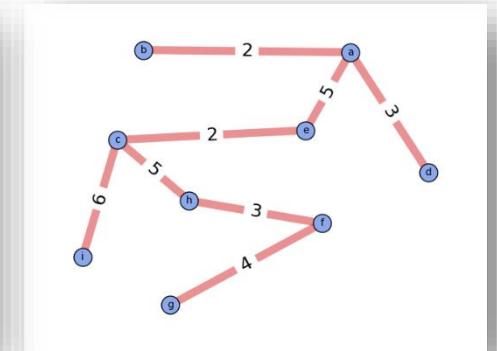
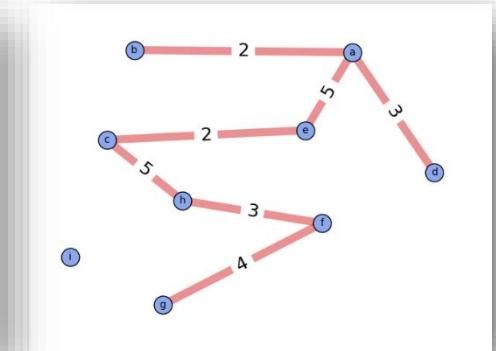
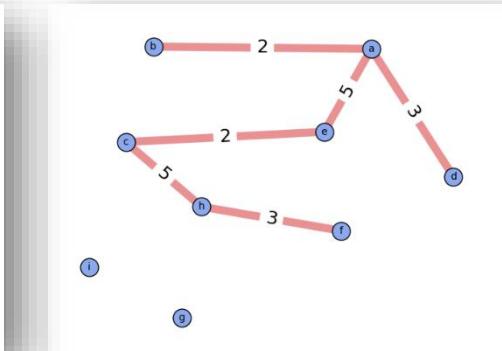
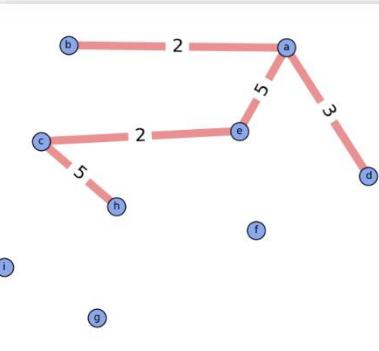
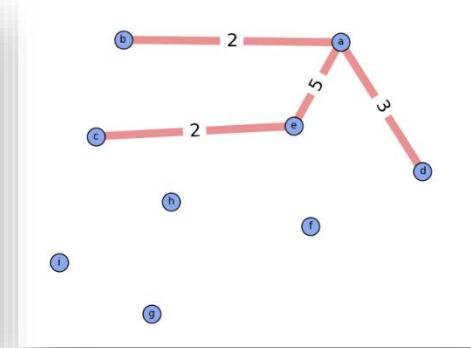
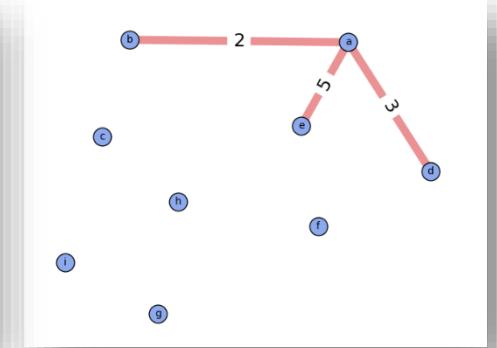
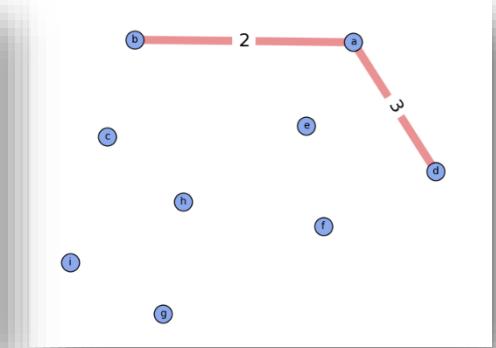
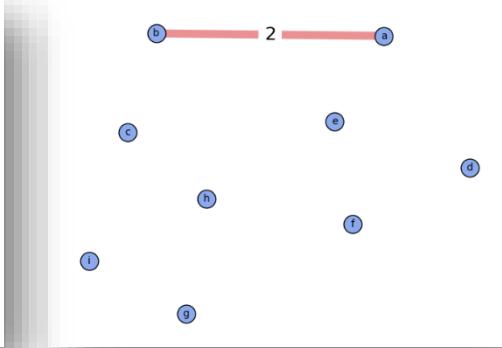
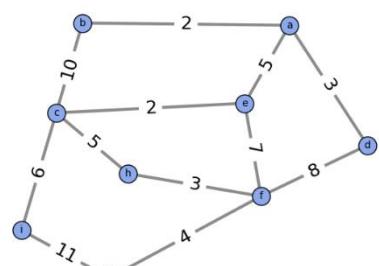
- Robert Clay Prim

- Formou-se em matemática e ciência da computação
- Em 1941 recebeu seu BS em Engenharia Elétrica da Universidade de Princeton
- Em 1949 recebeu seu Ph.D. em Matemática da Universidade de Princeton
- De 1958-1961 atuou como diretor de pesquisa da matemática Bell Laboratories período que criou o algoritmo de Prim

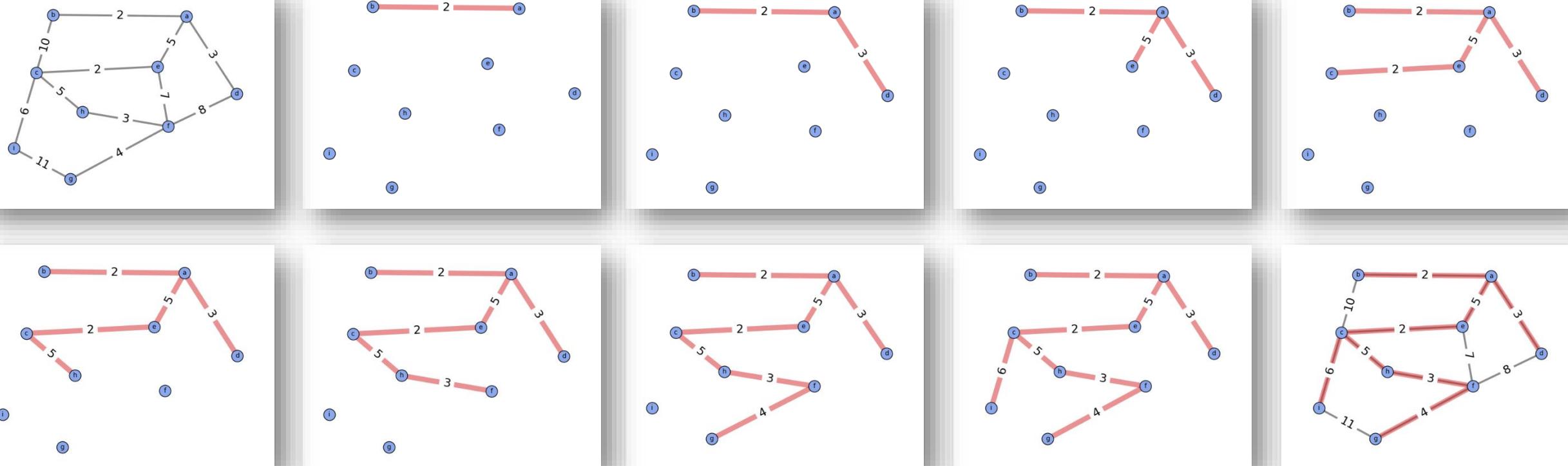
A algoritmo de Prim, foi originalmente descoberto em 1930 pelo matemático Vojtěch Jarník e mais tarde de forma independente por Prim em 1957. Mais tarde, foi redescoberto por Edsger Dijkstra em 1959. Às vezes é referido como o algoritmo DJP ou o algoritmo Jarník.



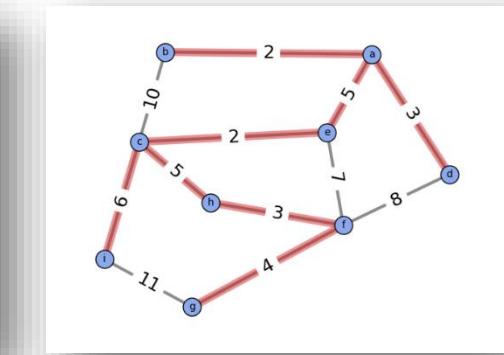
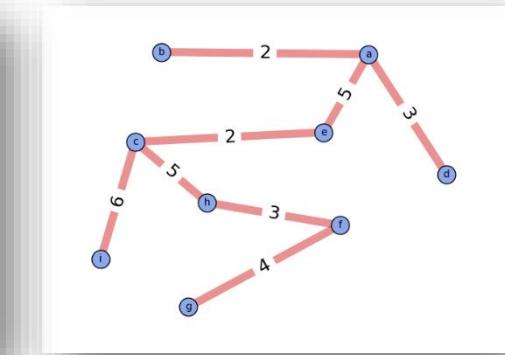
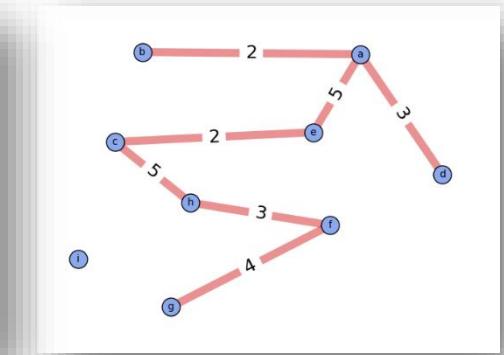
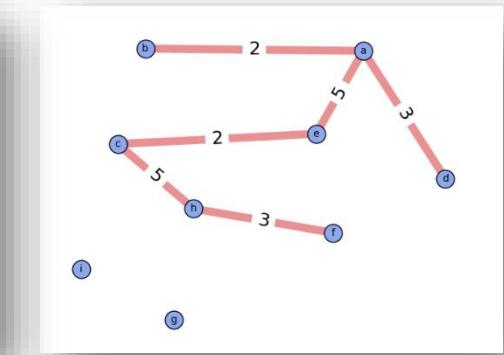
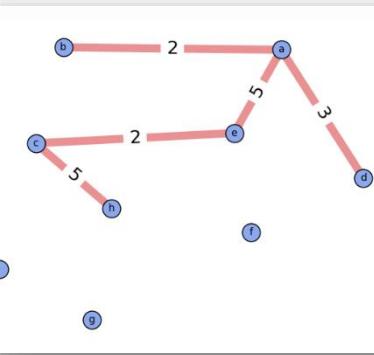
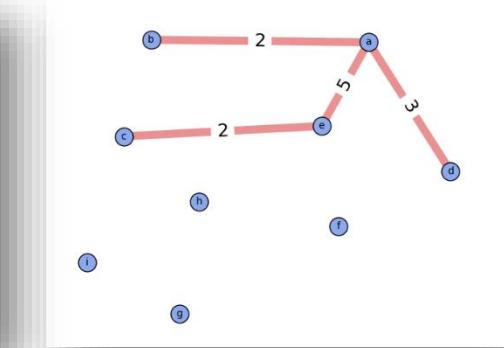
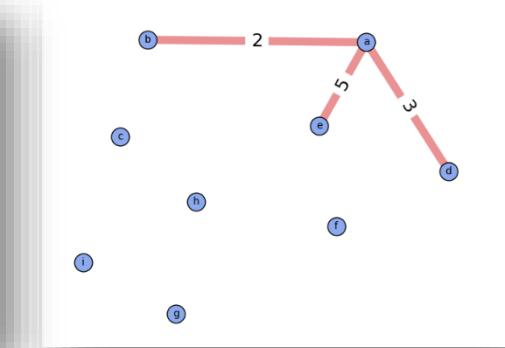
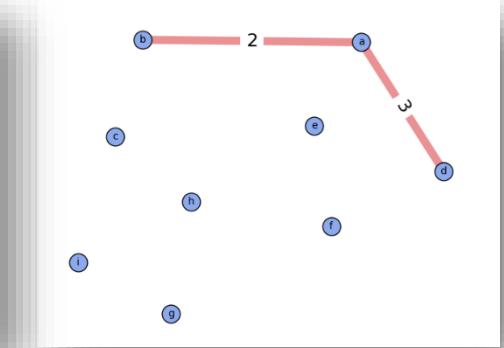
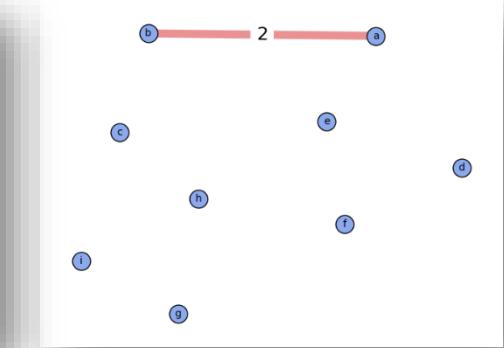
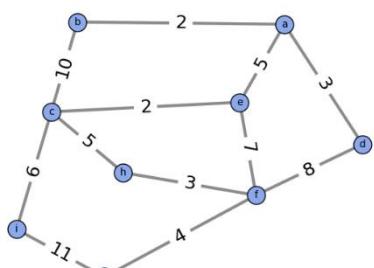
- A solução parte do princípio que existem apenas os vértices
- As arestas são apenas “arestas potenciais”
- O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora
- A árvore cresce de modo a satisfazer o critério de minimalidade baseado em cortes (ou expansão de arestas)



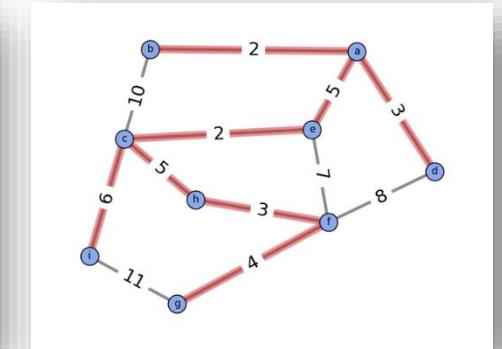
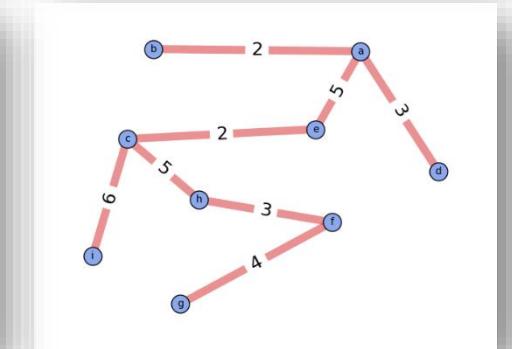
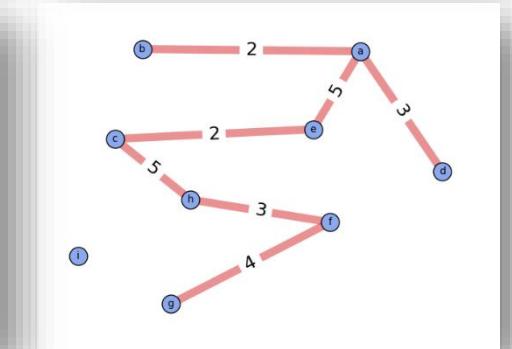
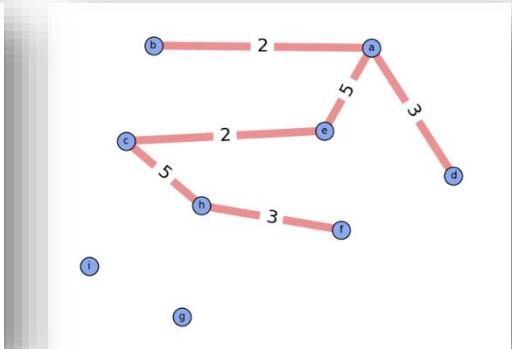
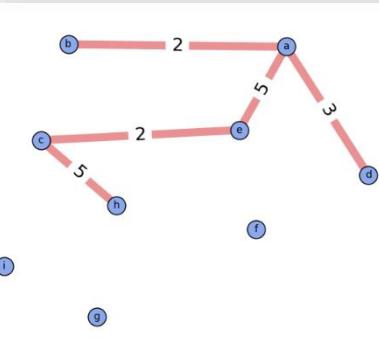
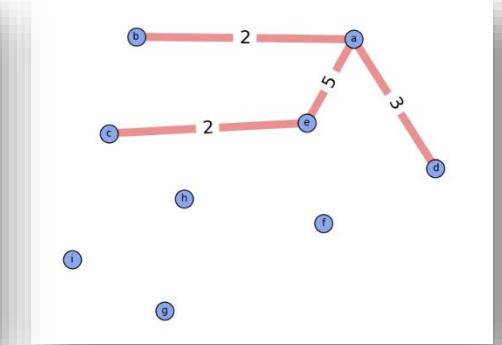
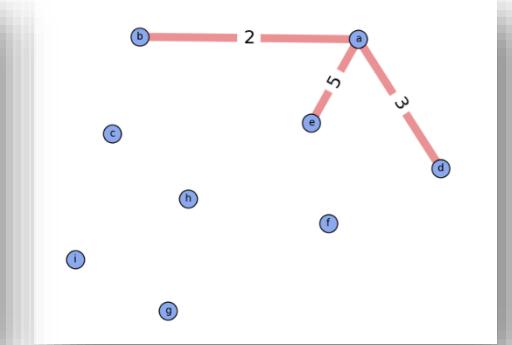
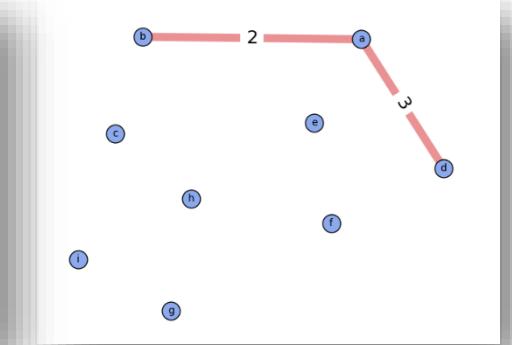
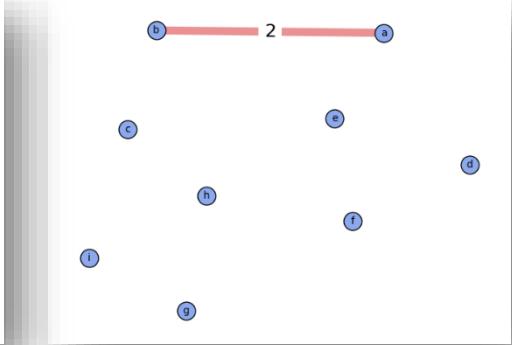
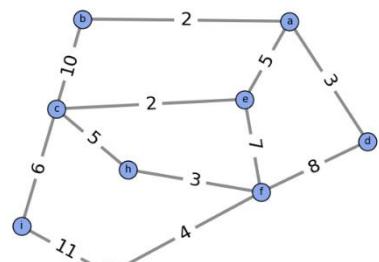
- A solução parte do princípio que existem apenas os vértices
- As arestas são apenas “arestas potenciais”
- O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora
- A árvore cresce de modo a satisfazer o critério de minimalidade baseado em cortes (ou expansão de arestas)



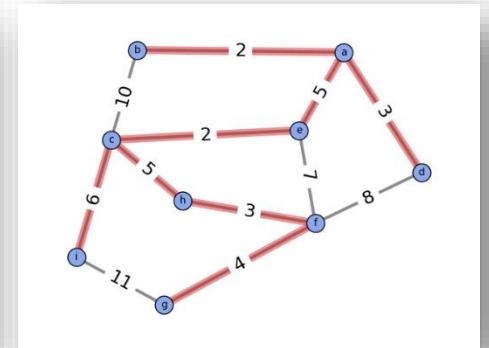
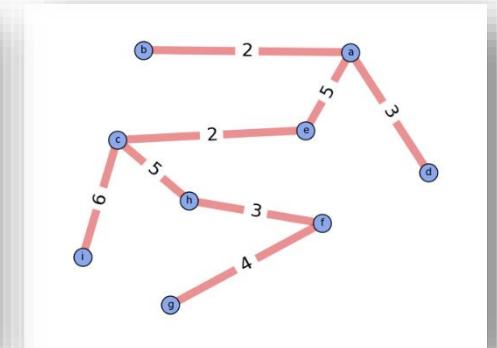
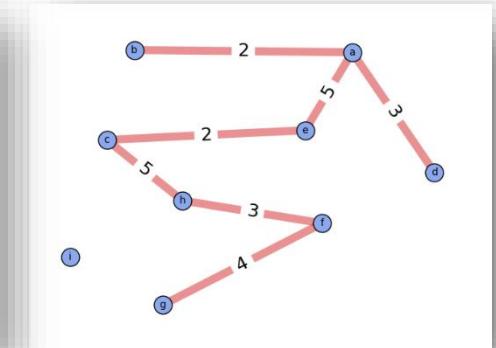
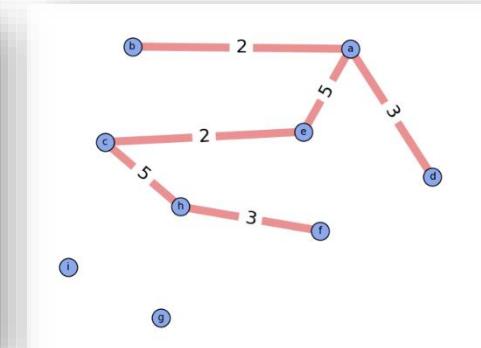
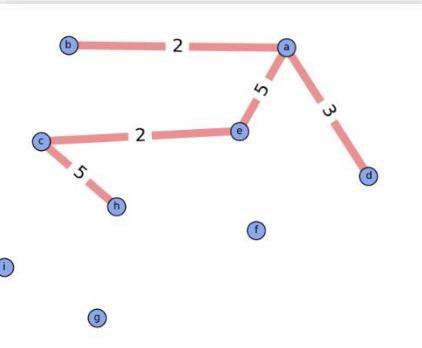
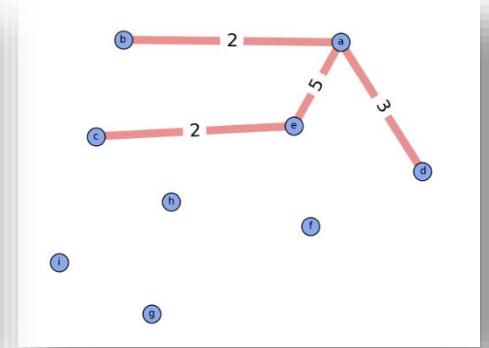
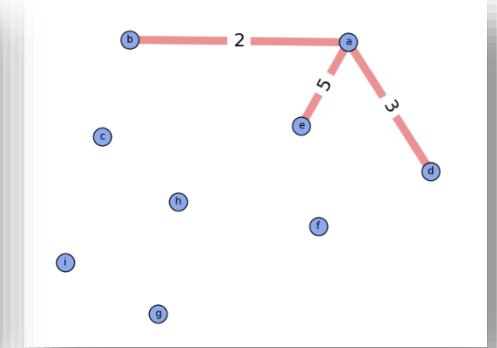
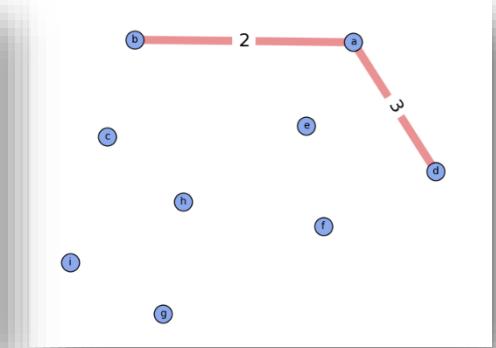
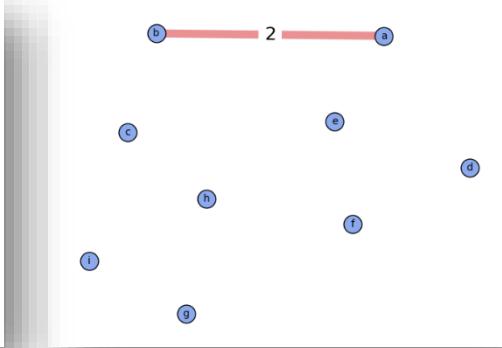
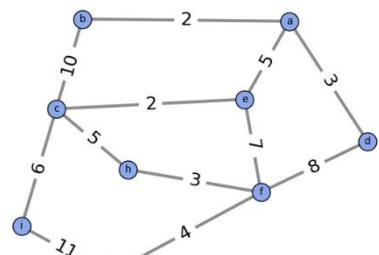
- A solução parte do princípio que existem apenas os vértices
- As arestas são apenas “arestas potenciais”
- O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora
- A árvore cresce de modo a satisfazer o critério de minimalidade baseado em cortes (ou expansão de arestas)



- A solução parte do princípio que existem apenas os vértices
- As arestas são apenas “arestas potenciais”
- **O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora**
- A árvore cresce de modo a satisfazer o critério de minimalidade baseado em cortes (ou expansão de arestas)



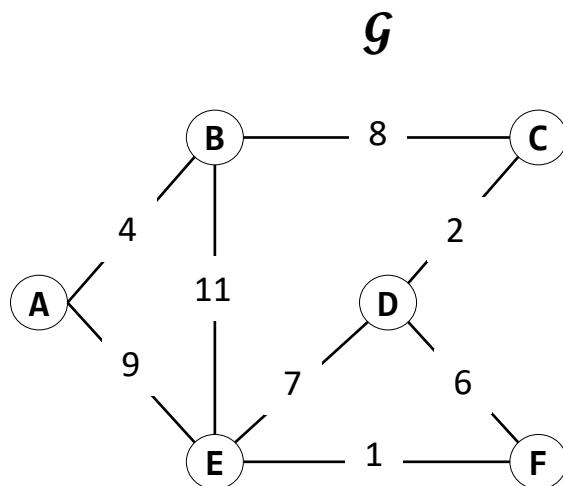
- A solução parte do princípio que existem apenas os vértices
- As arestas são apenas “arestas potenciais”
- O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora
- **A árvore cresce de modo a satisfazer o critério de minimalidade baseado em cortes (ou expansão de arestas)**



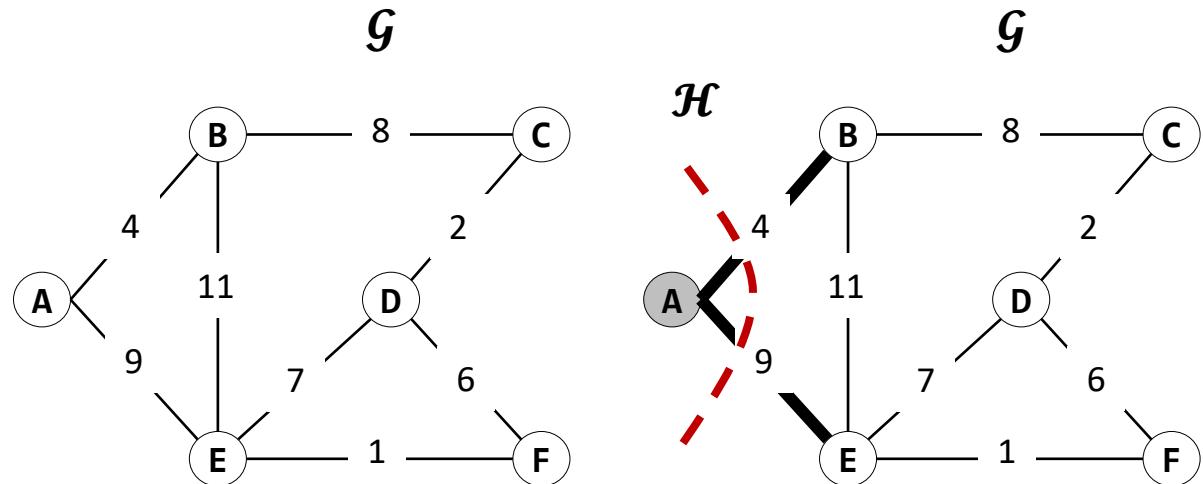
Selecione um vértice s para iniciar o subgrafo
Enquanto há vértices que não estão na **árvore** **Faça**
 Selecione uma **aresta segura**
 Insira a **aresta segura** e seu vértice no subgrafo

- Suponha que \mathcal{H} é uma árvore (não necessariamente geradora) de um grafo não-dirigido conexo \mathcal{G} com custos nas arestas
- Um corte de \mathcal{H} é o conjunto de todas as arestas de \mathcal{G} que têm uma extremidade em \mathcal{H} e outra fora.
- Portanto, a franja ou fronteira de \mathcal{H} são as potenciais arestas candidatas para entrar na árvore \mathcal{H} .

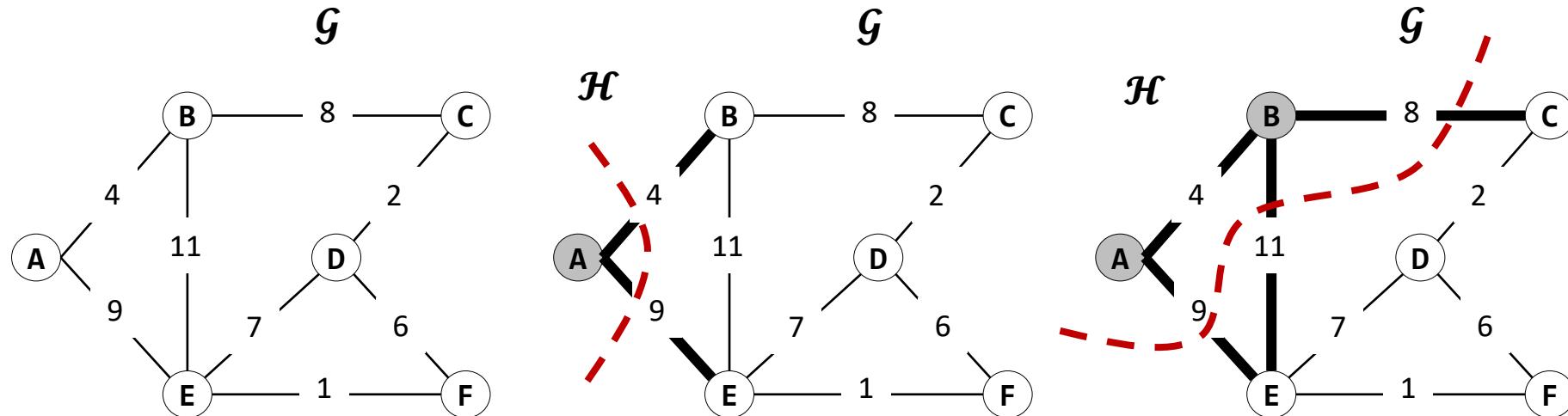
- Suponha que \mathcal{H} é uma árvore (não necessariamente geradora) de um grafo não-dirigido conexo \mathcal{G} com custos nas arestas
- Um corte de \mathcal{H} é o conjunto de todas as arestas de \mathcal{G} que têm uma extremidade em \mathcal{H} e outra fora.
- Portanto, a franja ou fronteira de \mathcal{H} são as potenciais arestas candidatas para entrar na árvore \mathcal{H} .



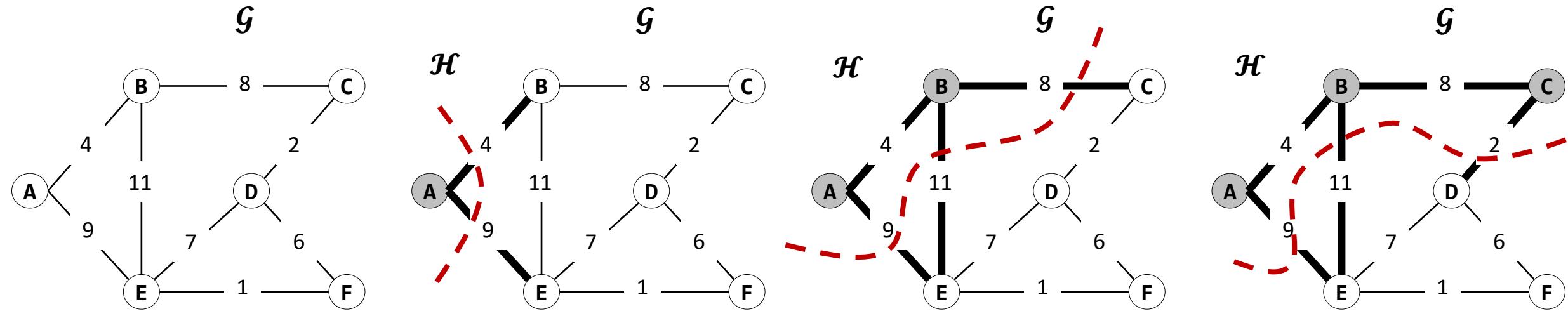
- Suponha que \mathcal{H} é uma árvore (não necessariamente geradora) de um grafo não-dirigido conexo \mathcal{G} com custos nas arestas
- Um corte de \mathcal{H} é o conjunto de todas as arestas de \mathcal{G} que têm uma extremidade em \mathcal{H} e outra fora.
- Portanto, a franja ou fronteira de \mathcal{H} são as potenciais arestas candidatas para entrar na árvore \mathcal{H} .



- Suponha que \mathcal{H} é uma árvore (não necessariamente geradora) de um grafo não-dirigido conexo \mathcal{G} com custos nas arestas
- Um corte de \mathcal{H} é o conjunto de todas as arestas de \mathcal{G} que têm uma extremidade em \mathcal{H} e outra fora.
- Portanto, a franja ou fronteira de \mathcal{H} são as potenciais arestas candidatas para entrar na árvore \mathcal{H} .



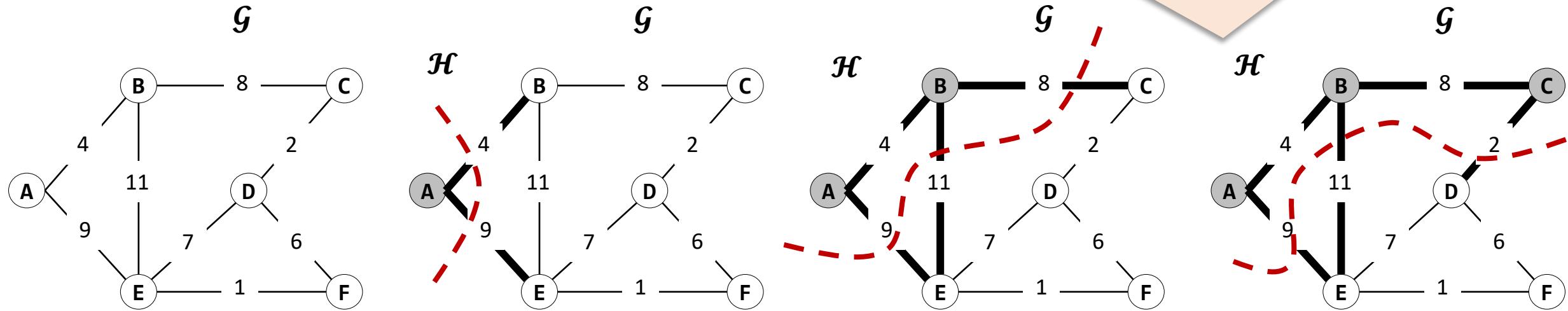
- Suponha que \mathcal{H} é uma árvore (não necessariamente geradora) de um grafo não-dirigido conexo \mathcal{G} com custos nas arestas
- Um corte de \mathcal{H} é o conjunto de todas as arestas de \mathcal{G} que têm uma extremidade em \mathcal{H} e outra fora.
- Portanto, a franja ou fronteira de \mathcal{H} são as potenciais arestas candidatas para entrar na árvore \mathcal{H} .



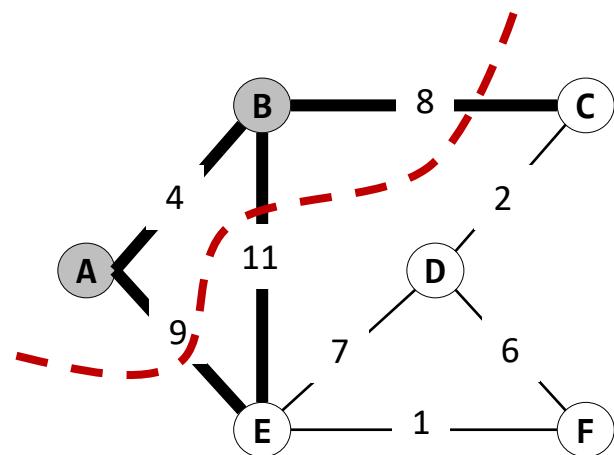
- Suponha que \mathcal{H} é uma árvore (não necessariamente)
- Um corte de \mathcal{H} é o conjunto de todas as arestas
- Portanto, a franja ou fronteira de \mathcal{H} são as po

O que é uma aresta segura?

- É uma aresta que não pertence a \mathcal{H} , porém, faz fronteira com \mathcal{H} . Além disso sua inserção não pode incluir ciclos em \mathcal{H} .



- Uma importante questão é: computacionalmente, como selecionar essa aresta de fronteira?
- Q: fila de prioridade ou *Heap mínimo* contendo os vértices de \mathcal{G} e que fazem fronteira com \mathcal{H}
 - Qual deveria ser o vértice de maior prioridade?
 - Aquele que possui a aresta mais leva que faça fronteira com \mathcal{H}
 - Ou seja, eu terei um *Heap mínimo* no qual a “key” é o “id” do vértice e o “custo” é o “peso de sua aresta” de fronteira com \mathcal{H}

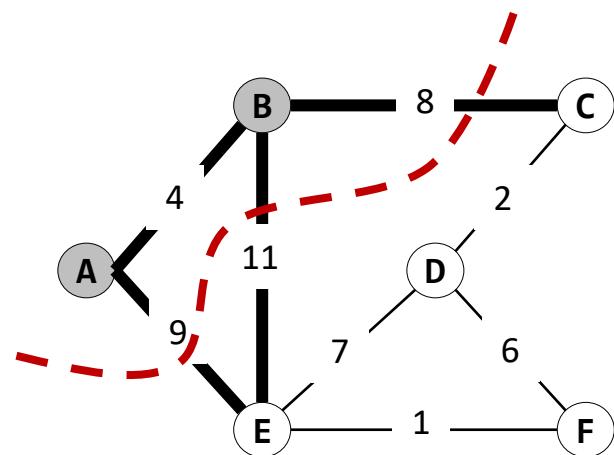


Custo para adicionar o vértice à árvore \mathcal{H} . Nesse caso, o custo representa o “peso” da aresta de menor valor que conecta o vértice à árvore \mathcal{H}

$$Q = \{(C, 8), (E, 9), (E, 11)\}$$

“id” do vértice

- Uma importante questão é: computacionalmente, como selecionar essa aresta de fronteira?
- Q: fila de prioridade ou *Heap mínimo* contendo os vértices de \mathcal{G} e que fazem fronteira com \mathcal{H}
 - Qual deveria ser o vértice de maior prioridade?
 - Aquele que possui a aresta mais leva que faça fronteira com \mathcal{H}
 - Ou seja, eu terei um *Heap mínimo* no qual a “key” é o “id” do vértice e o “custo” é o “peso de sua aresta” de fronteira com \mathcal{H}

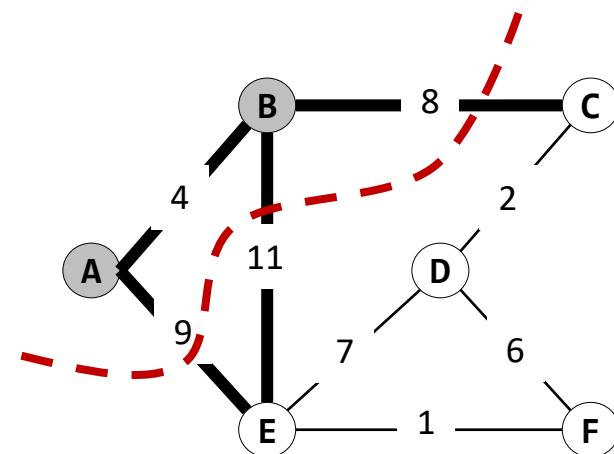


Custo para adicionar o vértice à árvore \mathcal{H} . Nesse caso, o custo representa o “peso” da aresta de menor valor que conecta o vértice à árvore \mathcal{H}

$$Q = \{(C, 8), (E, 9), (E, 11)\}$$

“id” do vértice

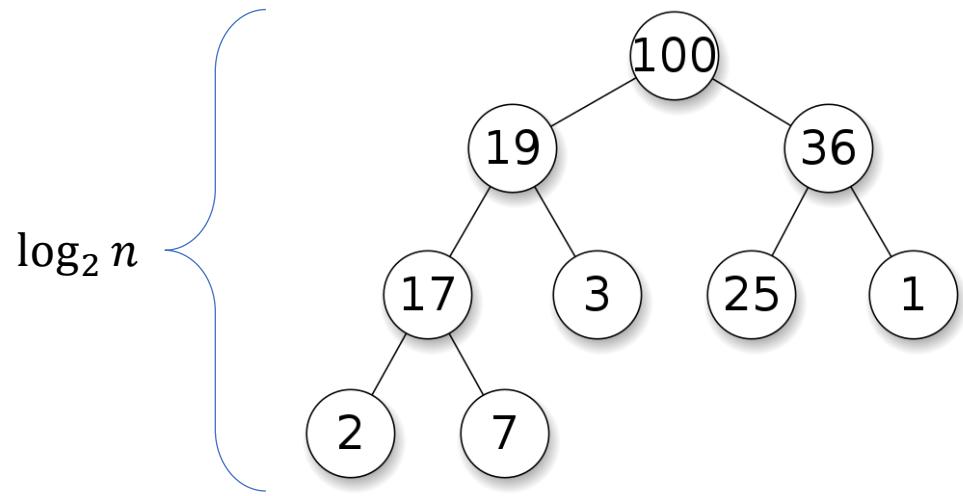
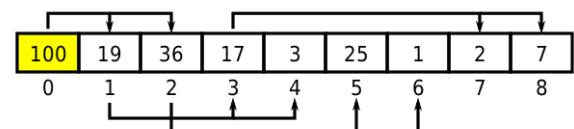
- Uma importante questão é: computacionalmente, como selecionar essa aresta de fronteira?
- Q: fila de prioridade ou *Heap mínimo* contendo os vértices de \mathcal{G} e que fazem fronteira com \mathcal{H}
 - Qual deveria ser o vértice de maior prioridade?
 - Aquele que possui a aresta mais leva que faça fronteira com \mathcal{H}
 - Ou seja, eu terei um *Heap mínimo* no qual a “key” é o “id” do vértice e o “custo” é o “peso de sua aresta” de fronteira com \mathcal{H}



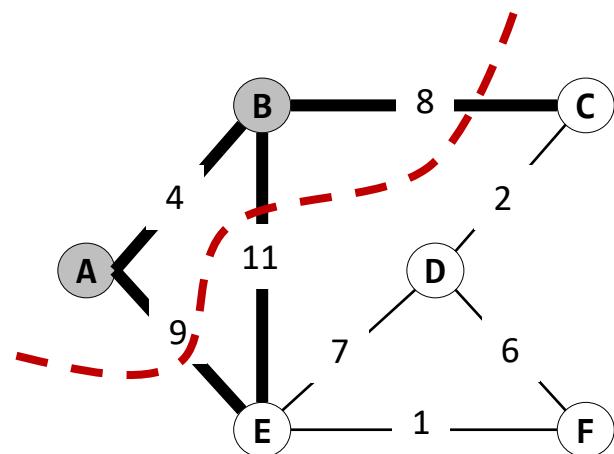
Custo para adicionar o vértice à árvore \mathcal{H} . Nesse caso, o custo representa o “peso” da aresta de menor valor que conecta o vértice à árvore \mathcal{H}

$$Q = \{(C, 8), (E, 9), (E, 11)\}$$

“id” do vértice

Tree representation**Array representation**

- Uma importante questão é: computacionalmente, como selecionar essa aresta de fronteira?
- Q: fila de prioridade ou *Heap mínimo* contendo os vértices de \mathcal{G} e que fazem fronteira com \mathcal{H}
 - Qual deveria ser o vértice de maior prioridade?
 - Aquele que possui a aresta mais leva que faça fronteira com \mathcal{H}
 - Ou seja, eu terei um *Heap mínimo* no qual a “key” é o “id” do vértice e o “custo” é o “peso de sua aresta” de fronteira com \mathcal{H}



Custo para adicionar o vértice à árvore \mathcal{H} . Nesse caso, o custo representa o “peso” da aresta de menor valor que conecta o vértice à árvore \mathcal{H}

$$Q = \{(C, 8), (E, 9), (E, 11)\}$$

“id” do vértice

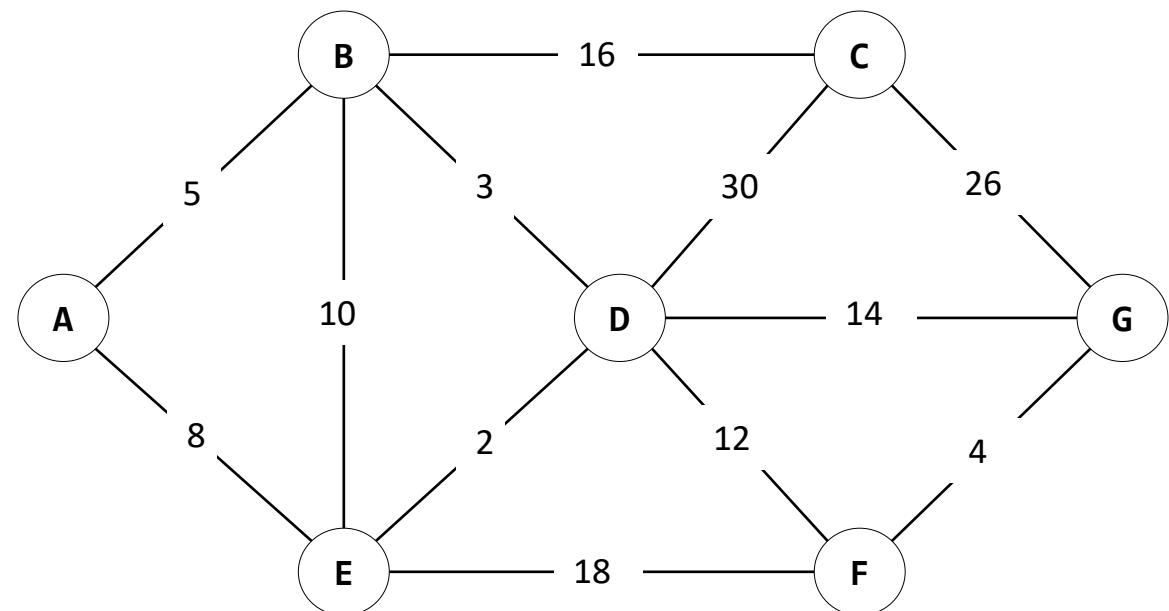
- Algumas estruturas de dados são necessárias
 - π : vetor de predecessores
 - Se u não tem predecessor ou ainda não foi descoberto então $\pi[u] = \text{NULL}$ ou 0.
 - \mathcal{H} : Árvore geradora mínima
 - Q : Heap mínimo contendo os vértices e seus custos para entrar na árvore

Lista de vértices

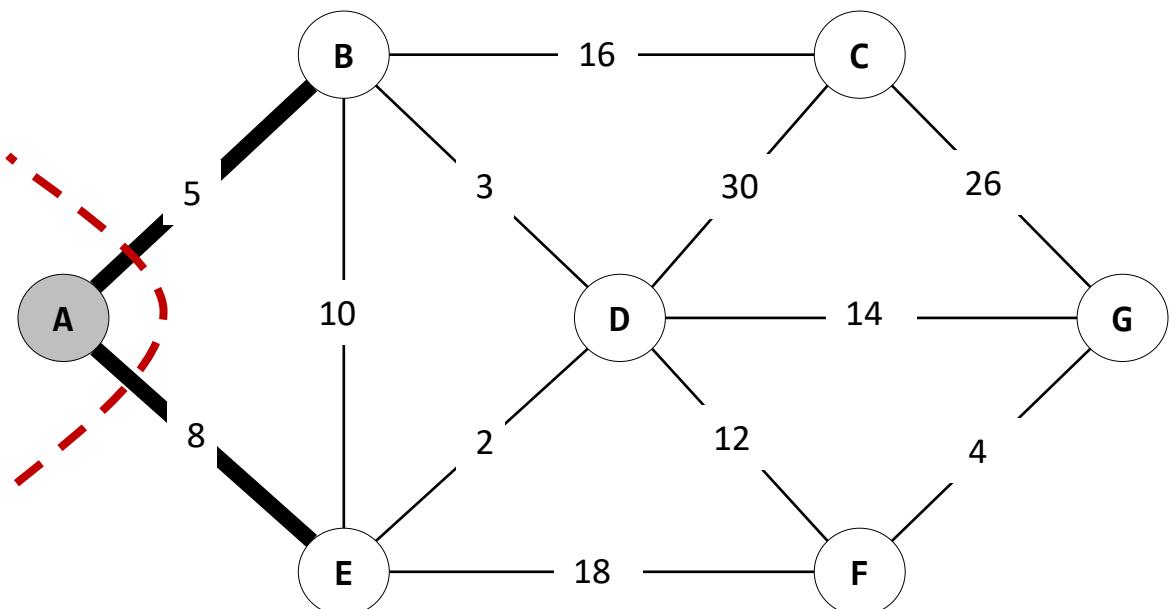
	A	B	C	D	E	F	G
π	NULL						
\mathcal{H}	{}						
Q	{}						

Estruturas de dados

	A	B	C	D	E	F	G
π	NULL						
\mathcal{H}	{}						
Q	{}						

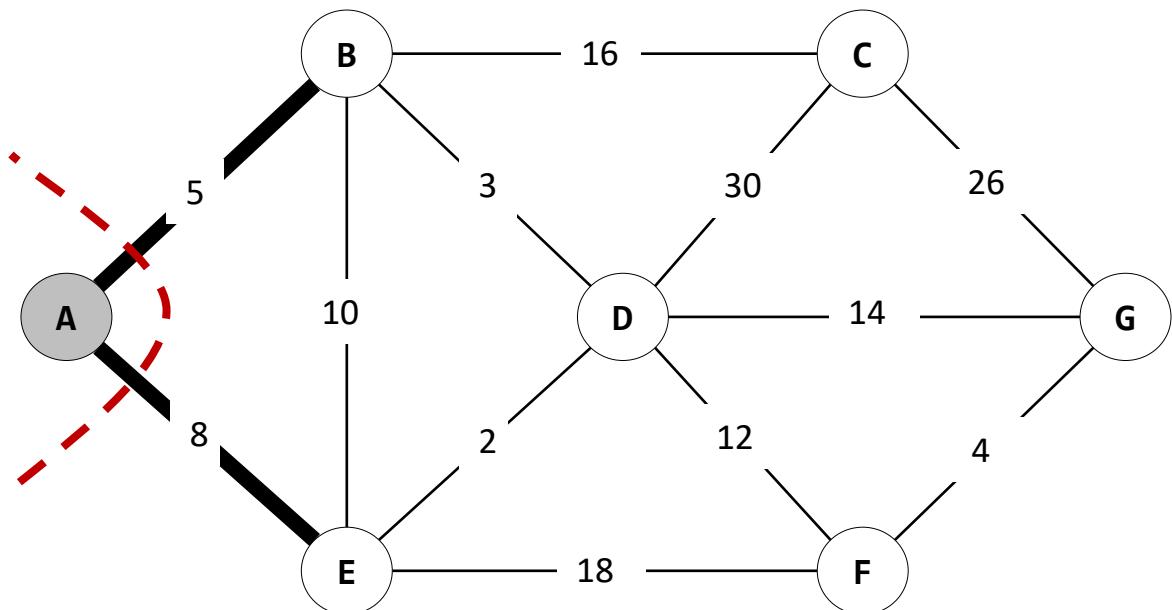


	A	B	C	D	E	F	G
π	NULL	A	NULL	NULL	A	NULL	NULL
\mathcal{H}	{A}						
Q	{(B, 5), (E, 8)}						



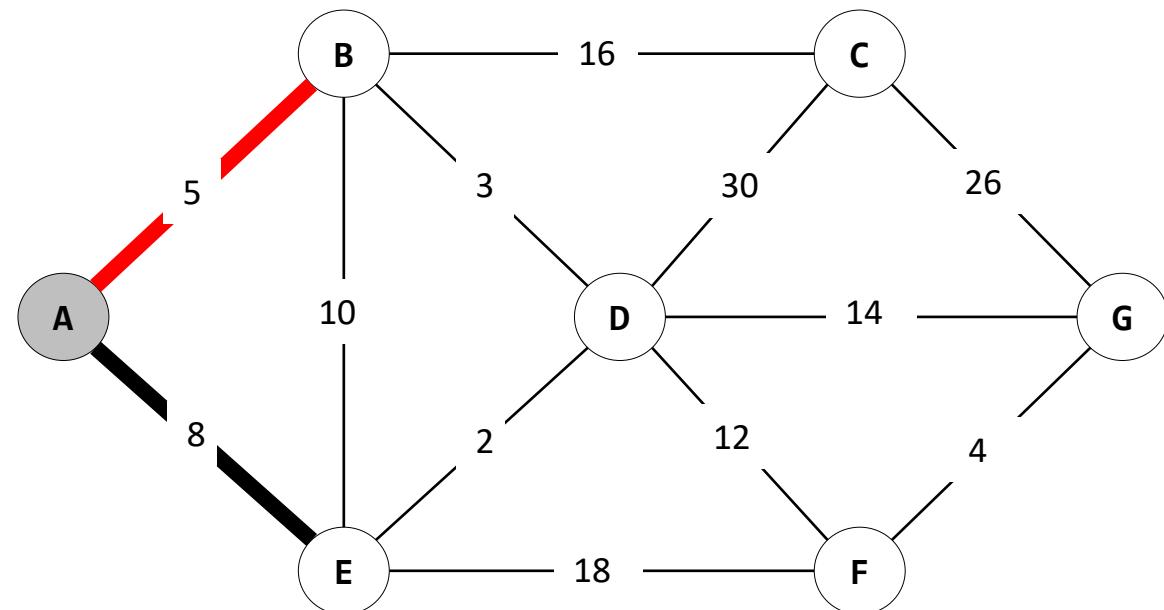
	A	B	C	D	E	F	G
π	NULL	A	NULL	NULL	A	NULL	NULL
\mathcal{H}	{A}						
Q	{(B, 5), (E, 8)}						

Qual escolha gulosa?

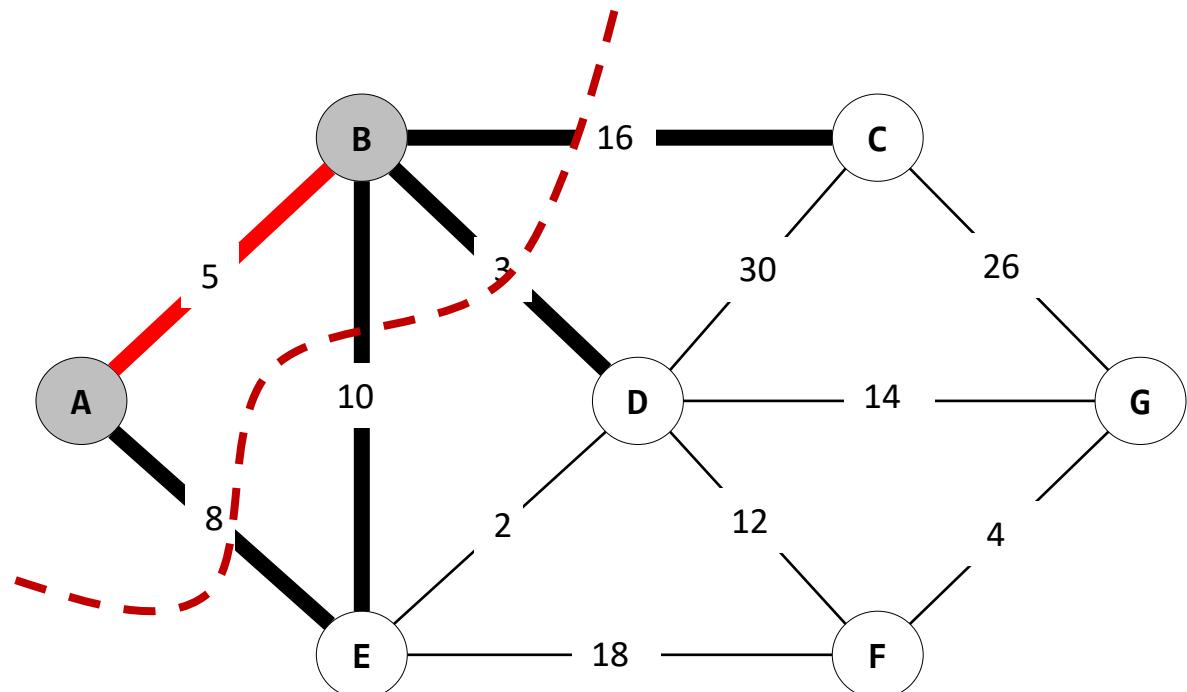


	A	B	C	D	E	F	G
π	NULL	A	NULL	NULL	A	NULL	NULL
\mathcal{H}	{A}						
Q	{(B, 5), (E, 8)}						

Qual escolha gulosa?

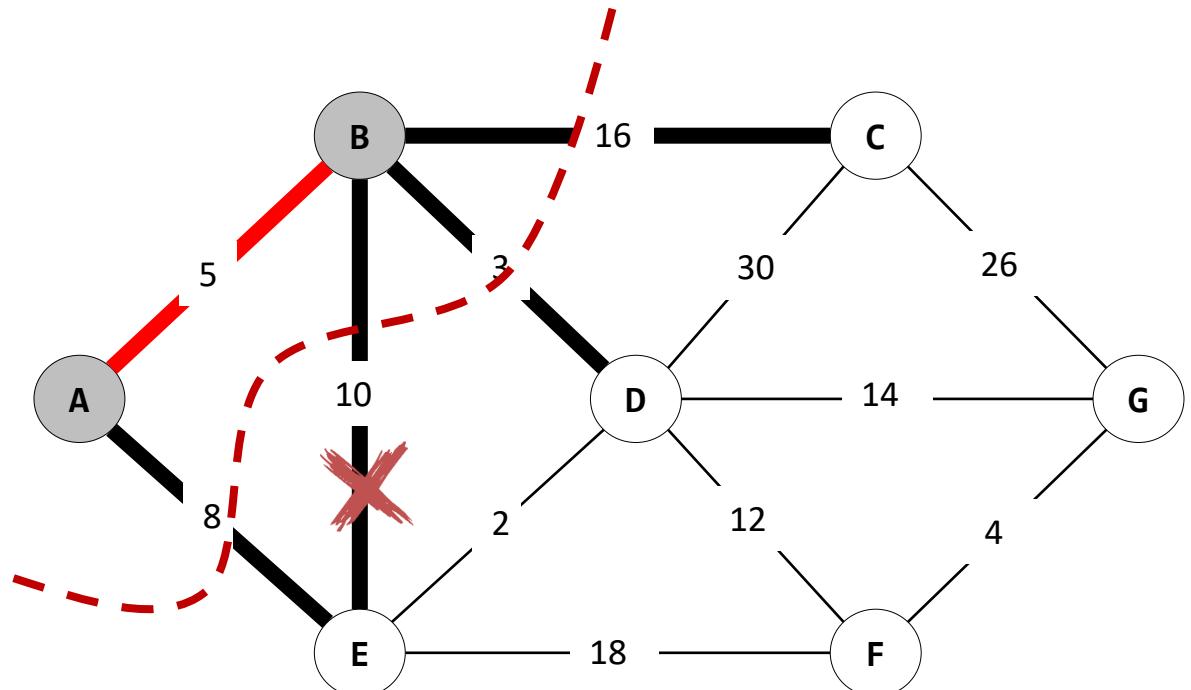


	A	B	C	D	E	F	G
π	NULL	A	B	B	A	NULL	NULL
\mathcal{H}	{A, B}						
Q	{(D, 3), (E, 8), (E,10), (C, 16)}						

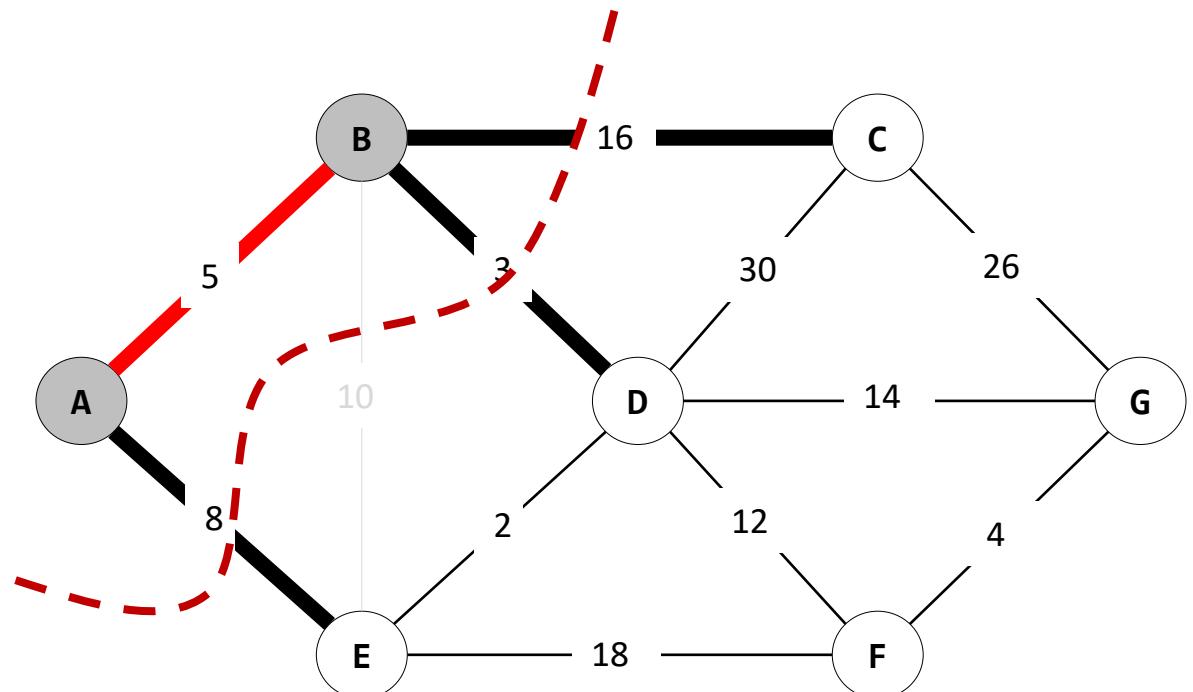


	A	B	C	D	E	F	G
π	NULL	A	B	B	A	NULL	NULL
\mathcal{H}	{A, B}						
Q	{(D, 3), (E, 8), (E, 10), (C, 16)}						

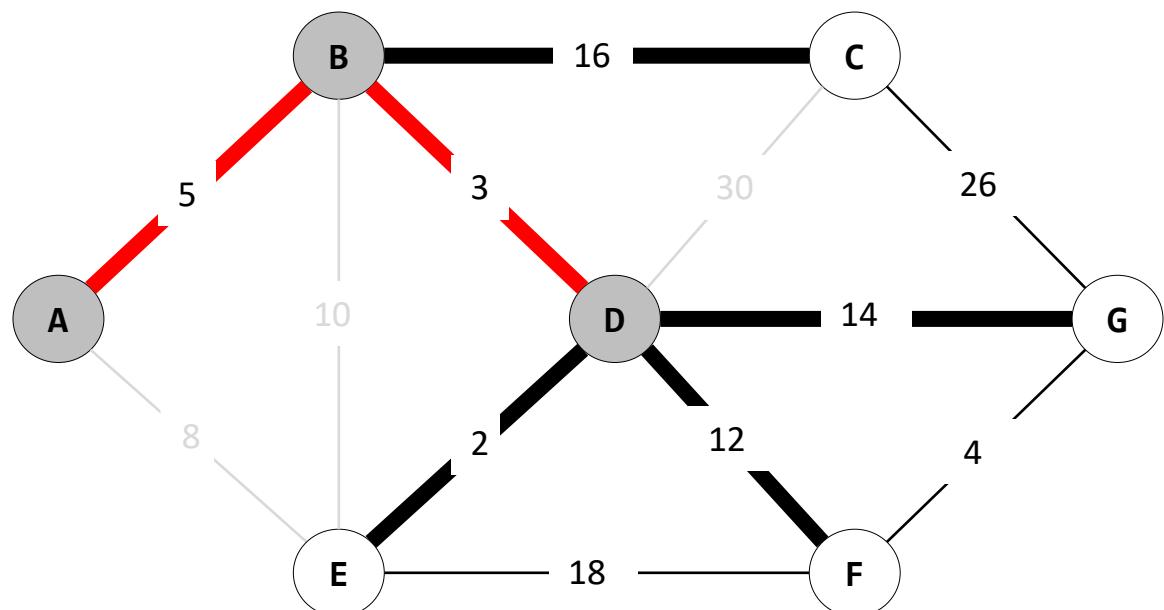
Notem que o vértice E é candidato a entrar na árvore \mathcal{H} através de duas arestas, de peso 8 e 10. Como 8 é menor que 10, localmente, a aresta de peso 10 é descartada, uma vez que a aresta de peso 8 é mais promissora.



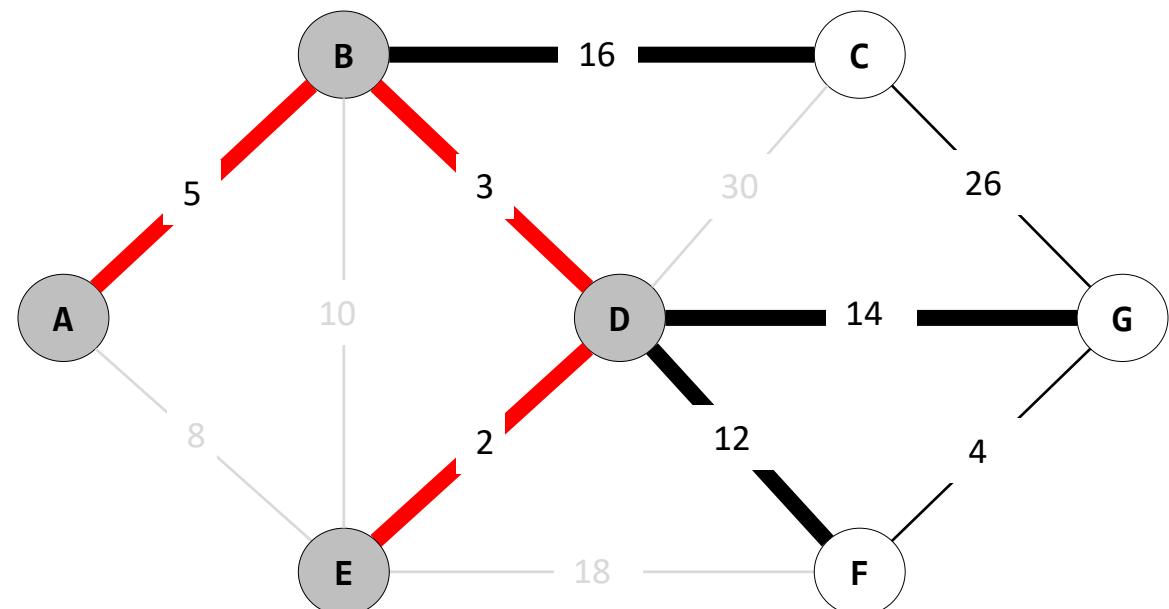
	A	B	C	D	E	F	G
π	NULL	A	B	B	A	NULL	NULL
\mathcal{H}	{A, B}						
Q	{(D, 3), (E, 8), (C, 16)}						



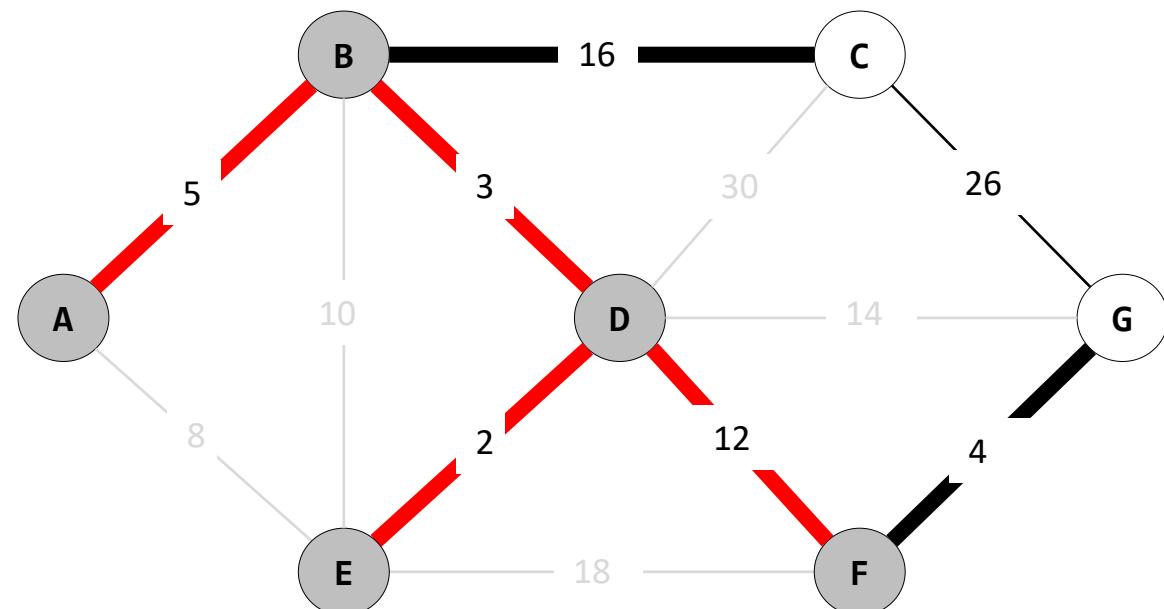
	A	B	C	D	E	F	G
π	NULL	A	B	B	D	D	D
\mathcal{H}	{A, B, D}						
Q	{(E, 2), (F, 12), (G, 14), (C, 16)}						



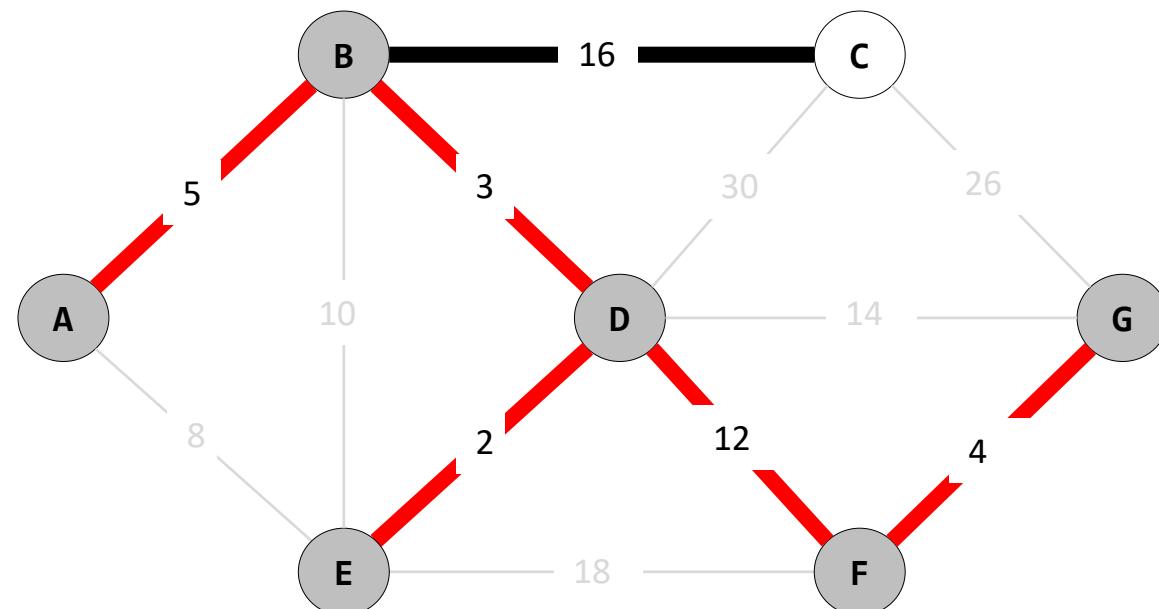
	A	B	C	D	E	F	G
π	NULL	A	B	B	D	D	D
\mathcal{H}	{A, B, D, E}						
Q	{(F, 12), (G, 14), (C, 16)}						



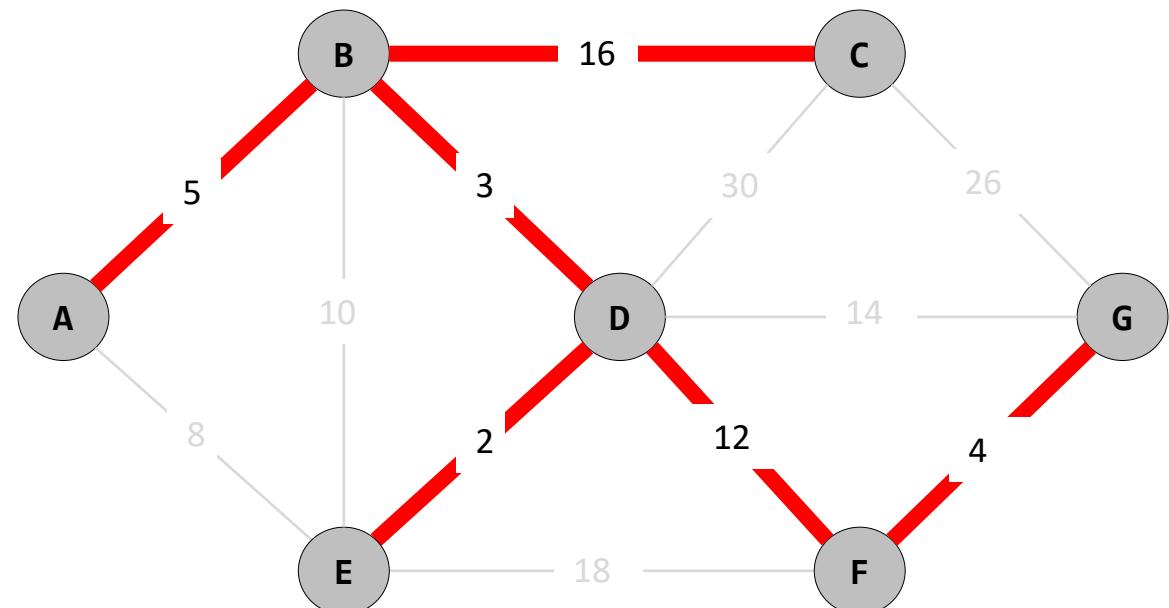
	A	B	C	D	E	F	G
π	NULL	A	B	B	D	D	F
\mathcal{H}	{A, B, D, E, F}						
Q	{(G, 4), (C, 16)}						



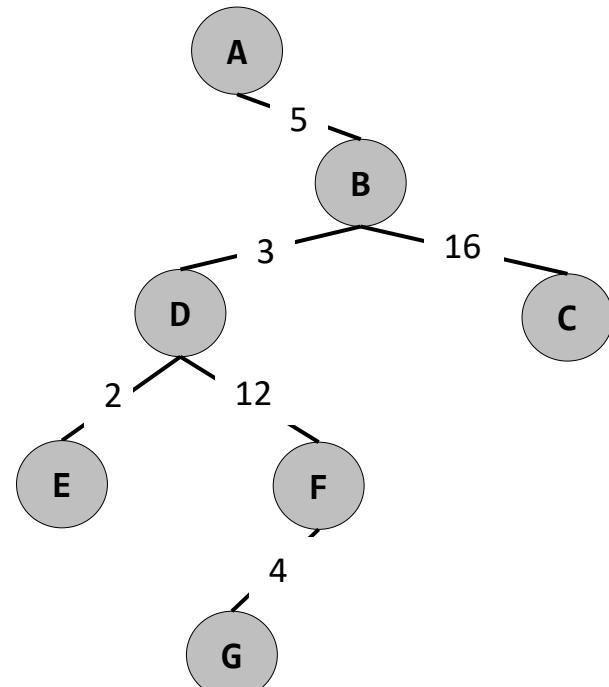
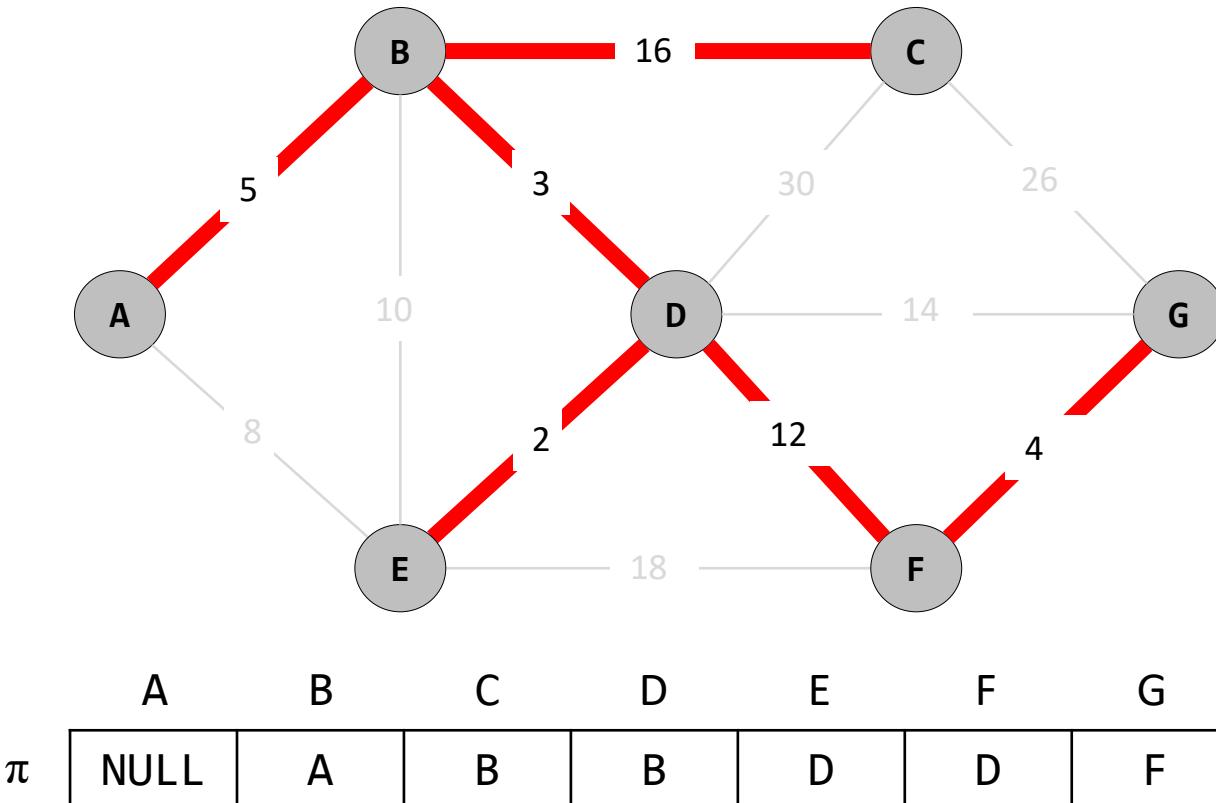
	A	B	C	D	E	F	G
π	NULL	A	B	B	D	D	F
\mathcal{H}	{A, B, D, E, F, G}						
Q	{(C, 16)}						



	A	B	C	D	E	F	G
π	NULL	A	B	B	D	D	F
\mathcal{H}	{A, B, D, E, F, G, C}						
Q	{}						

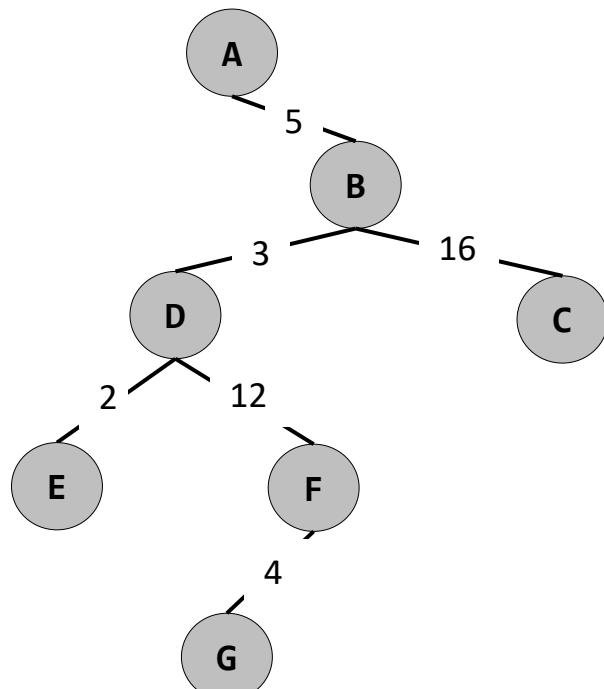
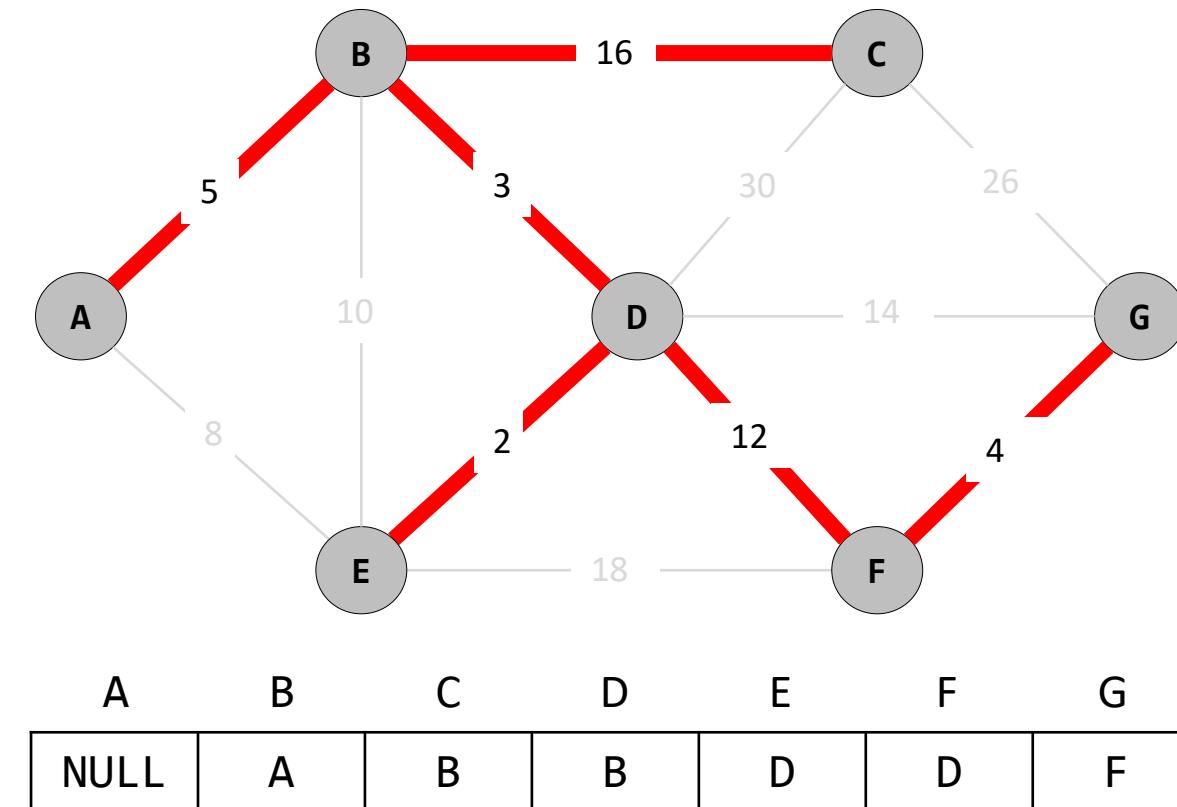


- O Prim retorna uma árvore geradora mínima do grafo, dado um vértice inicial
- O vértice inicial é considerado a raiz da árvore
- Nesse exemplo, a raiz da árvore geradora é A



- O Prim retorna uma árvore geradora mínima do grafo, dado um vértice inicial
- O vértice inicial é considerado a raiz da árvore
- Nesse exemplo, a raiz da árvore geradora é A

Se mudar o vértice raiz ou de origem é possível que a MST mude, caso exista mais de uma MST



```
PrintPath( $\pi$ ,  $s$ ,  $u$ ):  
    if  $u == s$ :  
        print( $s$ )  
    elif  $\pi == \text{NULL}$ :  
        print("não existe nenhum caminho de"  $s$  "para"  $u$ )  
    else:  
        PrintPath( $\pi$ ,  $s$ ,  $\pi[u]$ )  
        print( $u$ )
```

|

```
PRIM( $G$ ,  $s$ )
    for  $u$  in  $V$ :
         $\pi[u] = \text{NULL}$ 
         $Q.\text{adicionar}((u, \infty))$ 
     $Q.\text{atualizar}(s, 0)$ 
    Criar árvore  $H$  vazia
    while  $Q \neq \emptyset$ :
         $(u, custo) = Q.\text{extrair\_min}()$ 
         $H.\text{add}(u)$ 
        for  $v$  in  $N(u)$ :
            custo =  $Q.\text{custo}(v)$ 
            if ( $v$  not in  $H$ ) and ( $w(u, v) < custo$ ):
                 $Q.\text{atualizar}(v, w(u, v))$ 
                 $\pi[v] = u$ 
```

```

PRIM( $G, s$ )
    for  $u$  in  $V$ :
         $\pi[u] = \text{NULL}$ 
        Q.adicionar( $(u, \infty)$ )
    Q.atualizar( $s, 0$ )
    Criar árvore  $H$  vazia
    while  $Q \neq \emptyset$ :
         $(u, custo) = Q.\text{extrair\_min}()$ 
         $H.add(u)$ 
        for  $v$  in  $N(u)$ :
            custo = Q.custo( $v$ )
            if ( $v$  not in  $H$ ) and ( $w(u, v) < custo$ ):
                Q.atualizar( $v, w(u, v)$ )
                 $\pi[v] = u$ 

```

Esse vértice poderia estar com uma aresta na fronteira em iterações passadas. A ideia desse teste é verificar se essa nova aresta é mais segura e substituí-la.

```

PRIM( $G, s$ )
    for  $u$  in  $V$ :
         $\pi[u] = \text{NULL}$ 
         $Q.\text{adicionar}((u, \infty))$ 
     $Q.\text{atualizar}(s, 0)$ 
    Criar árvore  $H$  vazia
    while  $Q \neq \emptyset$ :
         $(u, custo) = Q.\text{extrair\_min}()$ 
         $H.\text{add}(u)$ 
        for  $v$  in  $N(u)$ :
            custo =  $Q.\text{custo}(v)$ 
            if ( $v$  not in  $H$ ) and ( $w(u, v) < custo$ ):
                 $Q.\text{atualizar}(v, w(u, v))$ 
                 $\pi[v] = u$ 

```

Se eu adicionei u em H , os seus adjacentes farão fronteira com H , portanto, seus custos devem ser atualizados na *Heap* mínimo

```

PRIM( $\mathcal{G}$ ,  $s$ )
    for  $u$  in  $\mathcal{V}$ :
         $\pi[u] = \text{NULL}$ 
        Q.adicionar( $(u, \infty)$ ) }  $n \log_2 n$ 
    Q.atualizar( $s, 0$ )
    Criar árvore  $\mathcal{H}$  vazia
    while Q !=  $\emptyset$ :
         $(u, \text{custo}) = Q.\text{extrair\_min}()$ 
         $\mathcal{H}.\text{add}(u)$ 
        for  $v$  in  $\mathcal{N}(u)$ :
            custo = Q.custo( $v$ )
            if ( $v$  not in  $\mathcal{H}$ ) and ( $w(u, v) < \text{custo}$ ):
                Q.atualizar( $v, w(u, v)$ )
                 $\pi[v] = u$ 

```

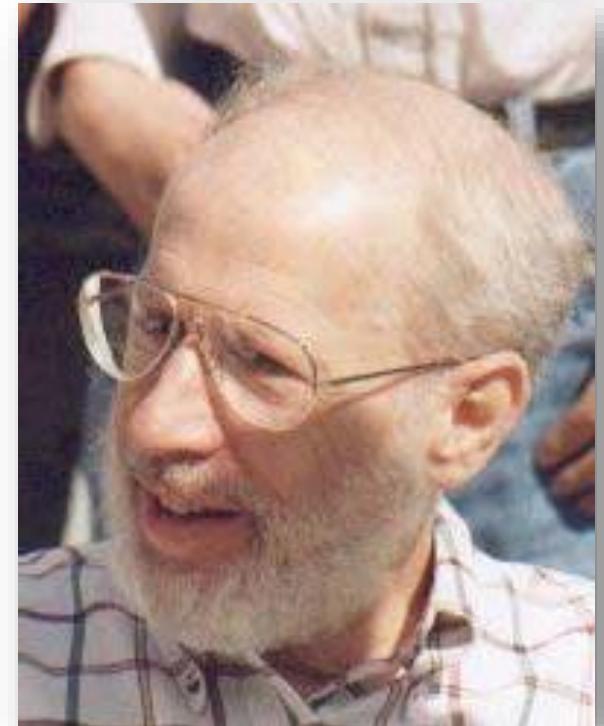
Eficiência: $O([n + m] \log_2 n) = O(m \log_2 n)$,
pois nesse caso o número de arestas supera o
número de vértices.

O laço é executado m vezes e o tempo de
atualização é $\log_2 n$, portanto, $O(m \log_2 n)$

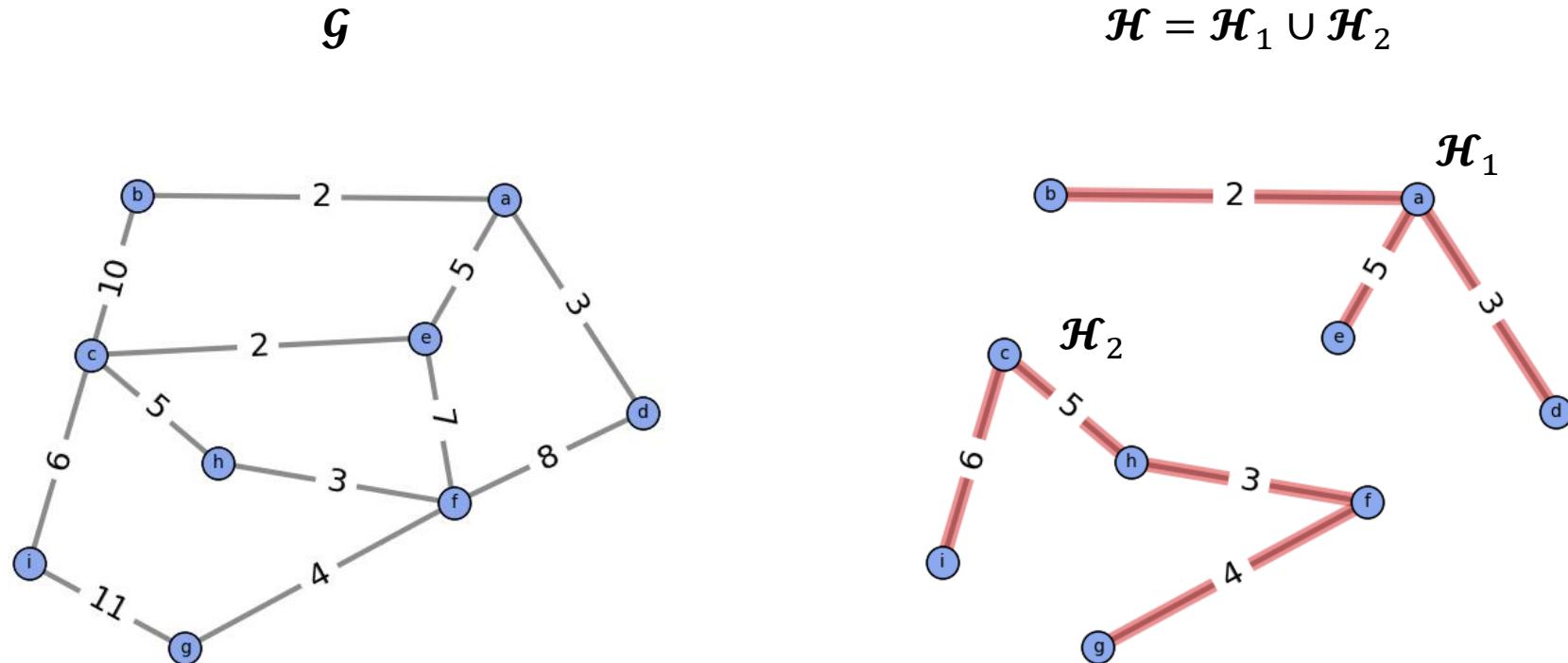
Em 1956 Kruskal (1928–2010) publicou (“On the shortest spanning subtree of a graph and the traveling salesman problem”, Proc. Amer. Math. Soc. 7 (1956), 48-50)

- Joseph Bernard Kruskal Jr. (Nova Iorque, 29 de janeiro de 1928 - Princeton, 19 de setembro de 2010) foi um matemático, estatístico, informático e psicometrista estadunidense.
- Estudou na Universidade de Chicago e na Universidade de Princeton, onde obteve o Ph.D. em 1954, orientado por Roger Lyndon e Paul Erdős.

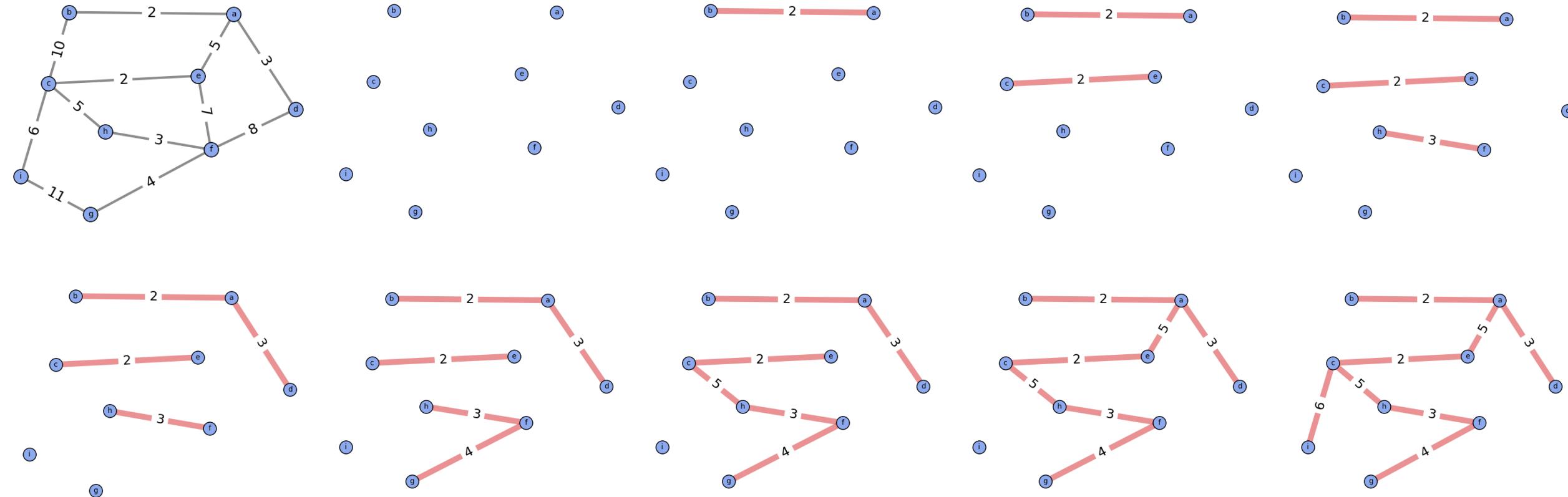
Importante na área de
Teoria dos Grafos



- Uma floresta \mathcal{H} de um grafo \mathcal{G} é definida como uma união disjunta de árvores.



- O algoritmo de Kruskal faz crescer uma floresta geradora até que ela se torne conexa.



$$\mathcal{H} = \emptyset$$

Enquanto \mathcal{H} não é uma árvore geradora (ou árvore conexa):

Encontre uma aresta (u, v) segura para \mathcal{H}

$$\mathcal{H} = \mathcal{H} \cup \{(u, v)\}$$

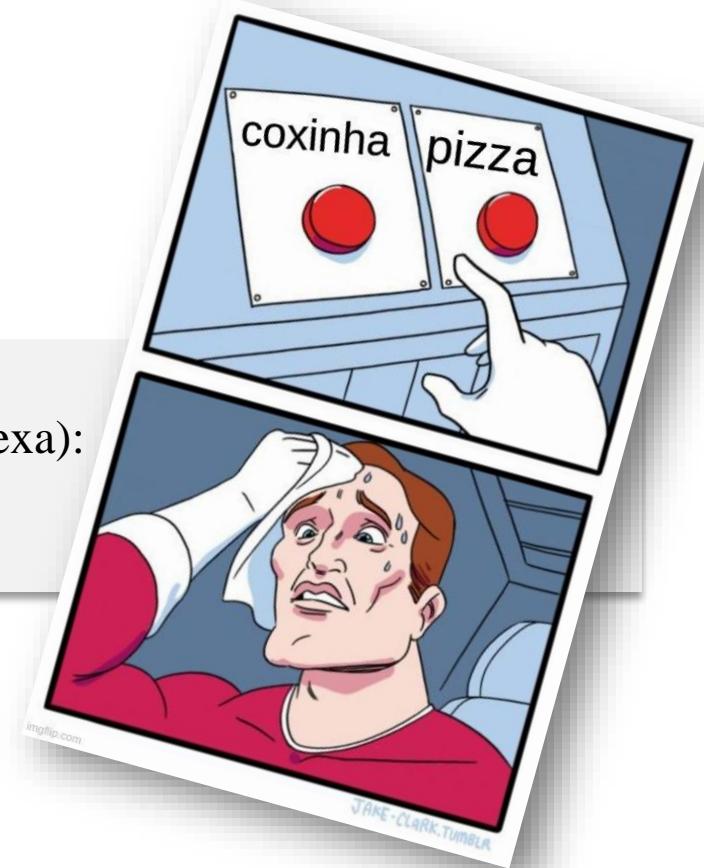
- Cuidado!!!
- Qual é a decisão gulosa?

$$\mathcal{H} = \emptyset$$

Enquanto \mathcal{H} não é uma árvore geradora (ou árvore conexa):

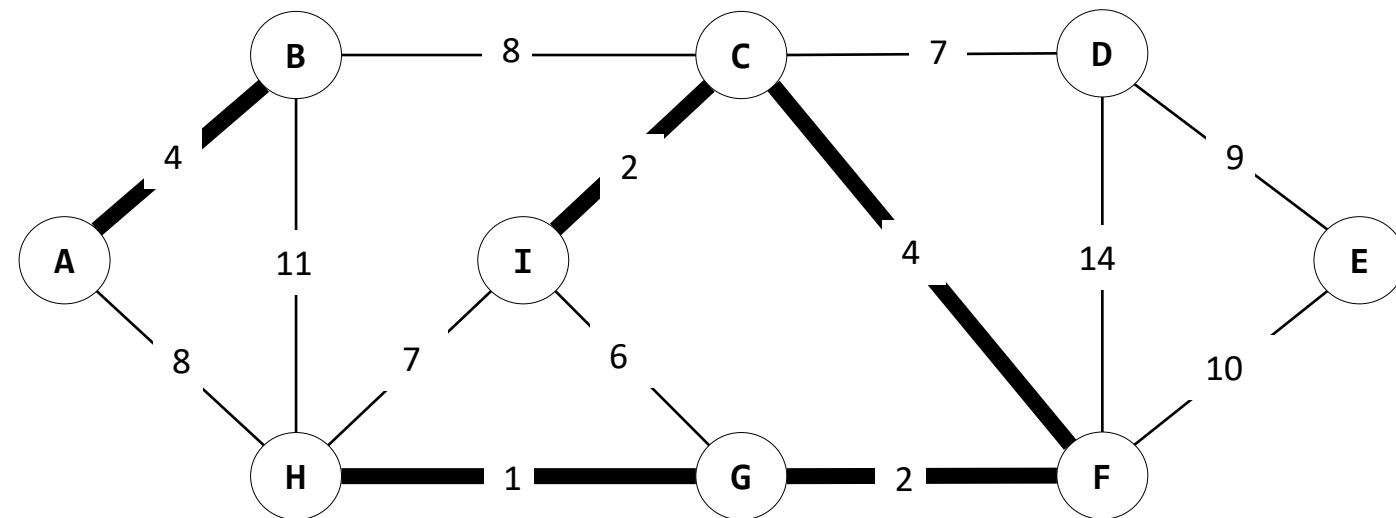
Encontre uma aresta (u, v) segura para \mathcal{H}

$$\mathcal{H} = \mathcal{H} \cup \{(u, v)\}$$

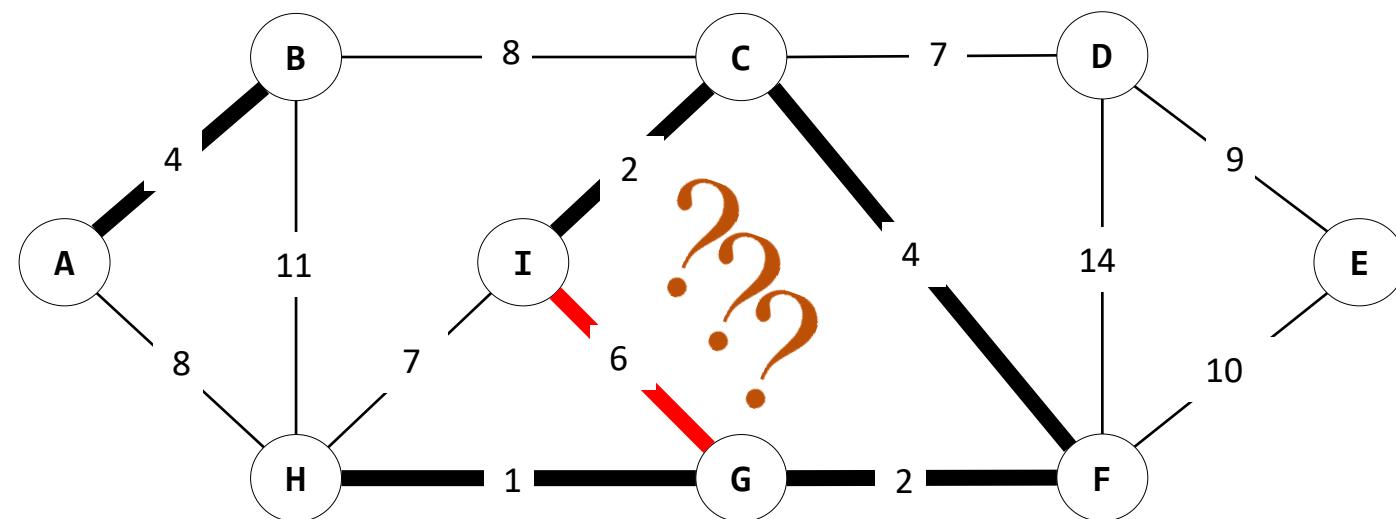


- O Kruskal pega a **menor aresta** com a **restrição de não fechar um ciclo**
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.

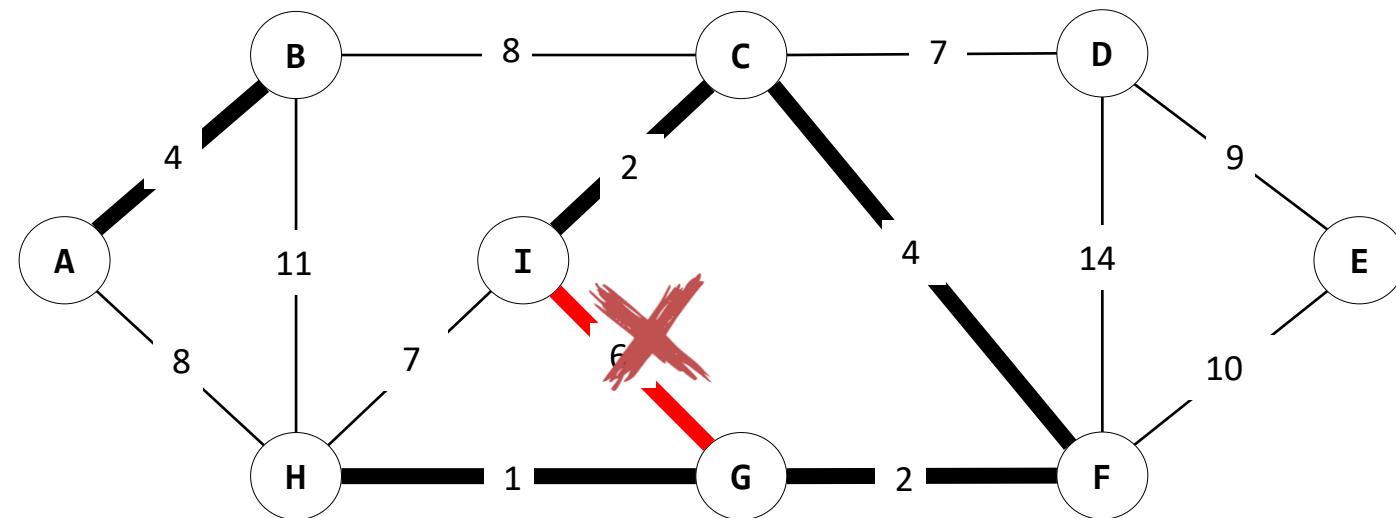
- O Kruskal pega a menor aresta com a restrição de não fechar um ciclo
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.



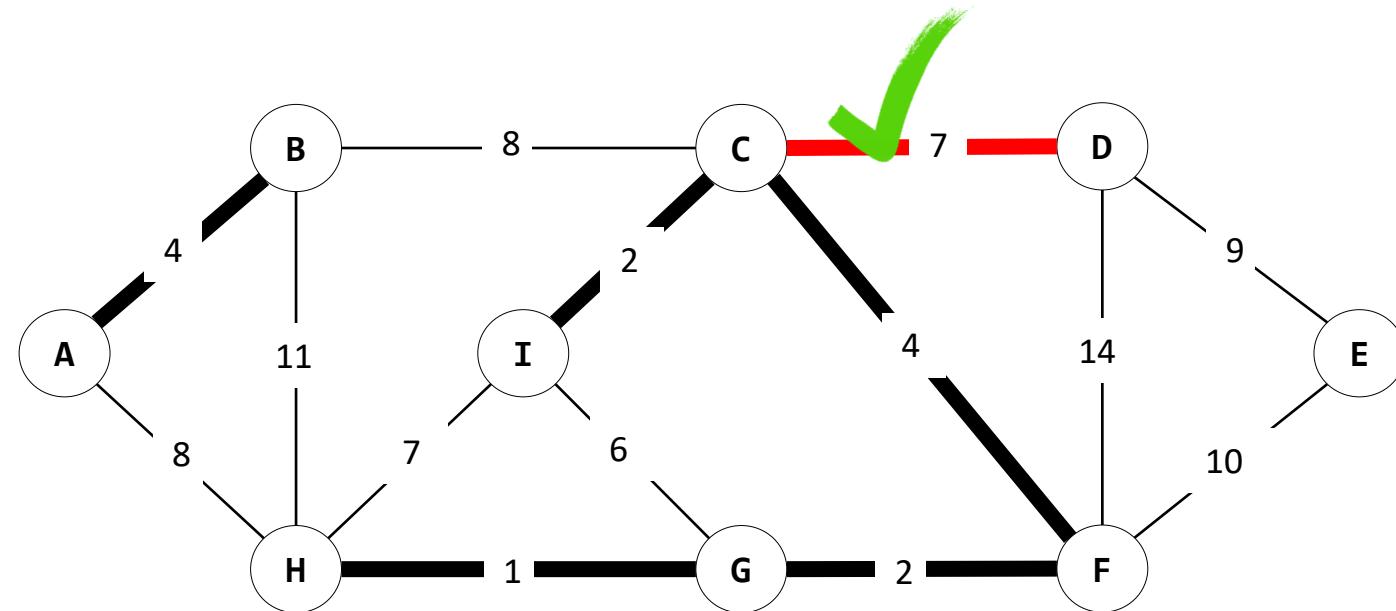
- O Kruskal pega a menor aresta com a restrição de não fechar um ciclo
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.



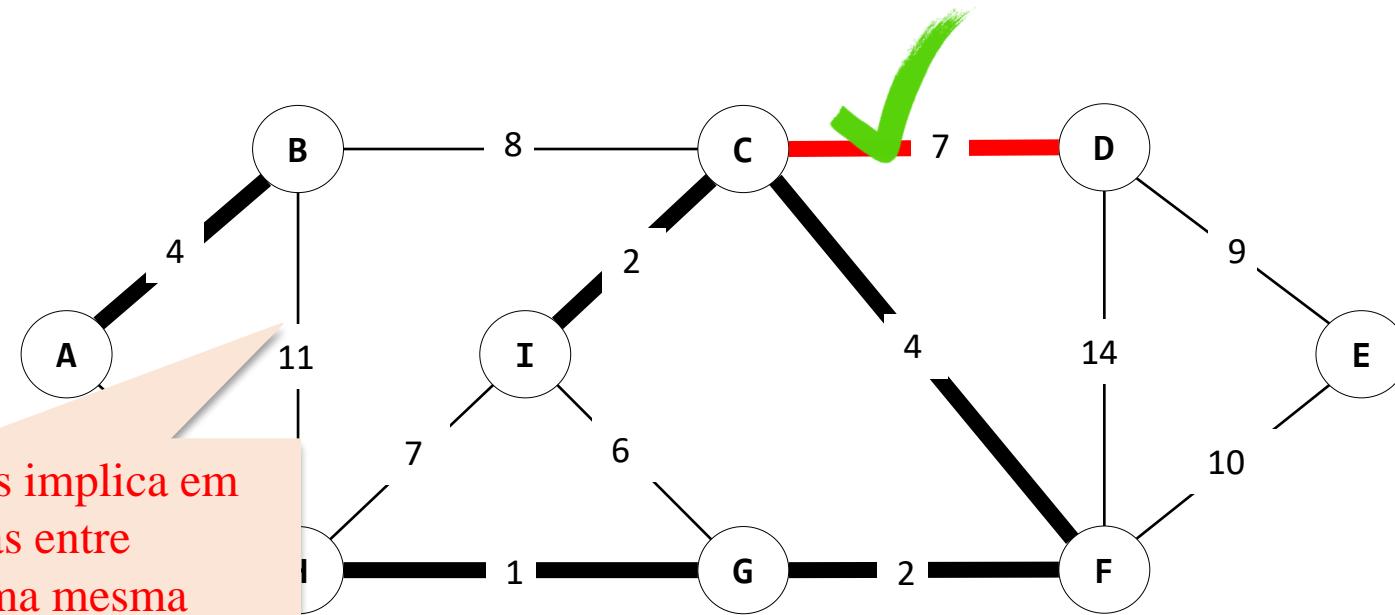
- O Kruskal pega a menor aresta com a restrição de não fechar um ciclo
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.



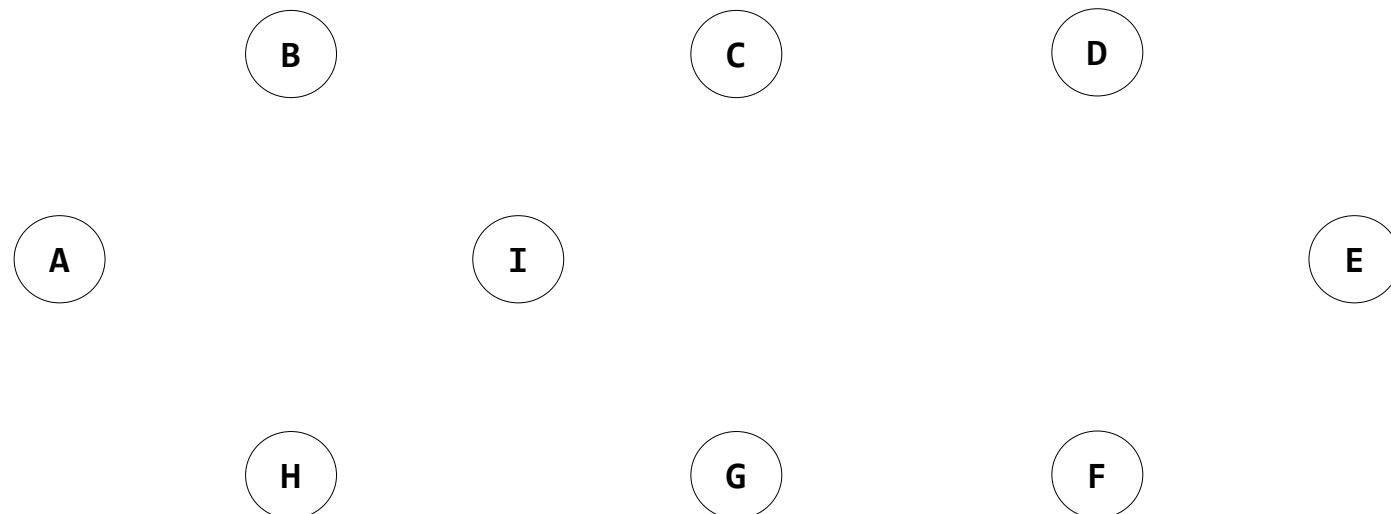
- O Kruskal pega a menor aresta com a restrição de não fechar um ciclo
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.



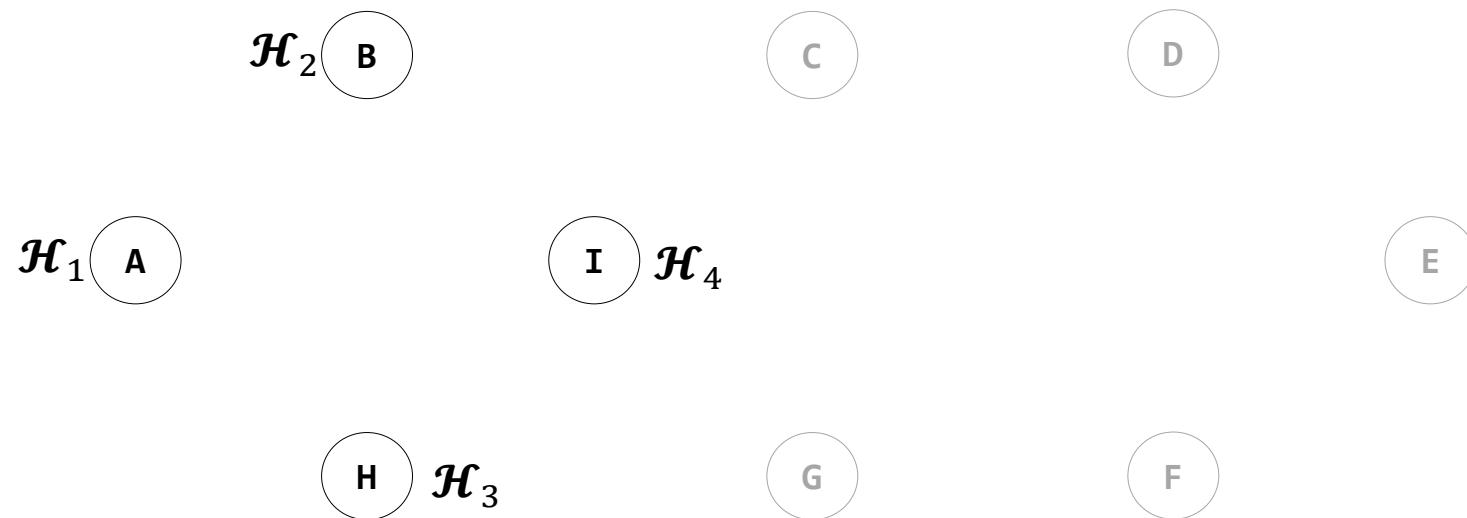
- O Kruskal pega a menor aresta com a restrição de não fechar um ciclo
- No algoritmo de Kruskal, primeiro pegamos as arestas de menor valor tomando cuidado pra não fazer ciclos pois assim não formaremos uma árvore.
- Sendo assim, pegamos as arestas de menor peso até que todos os vértices estejam ligados.



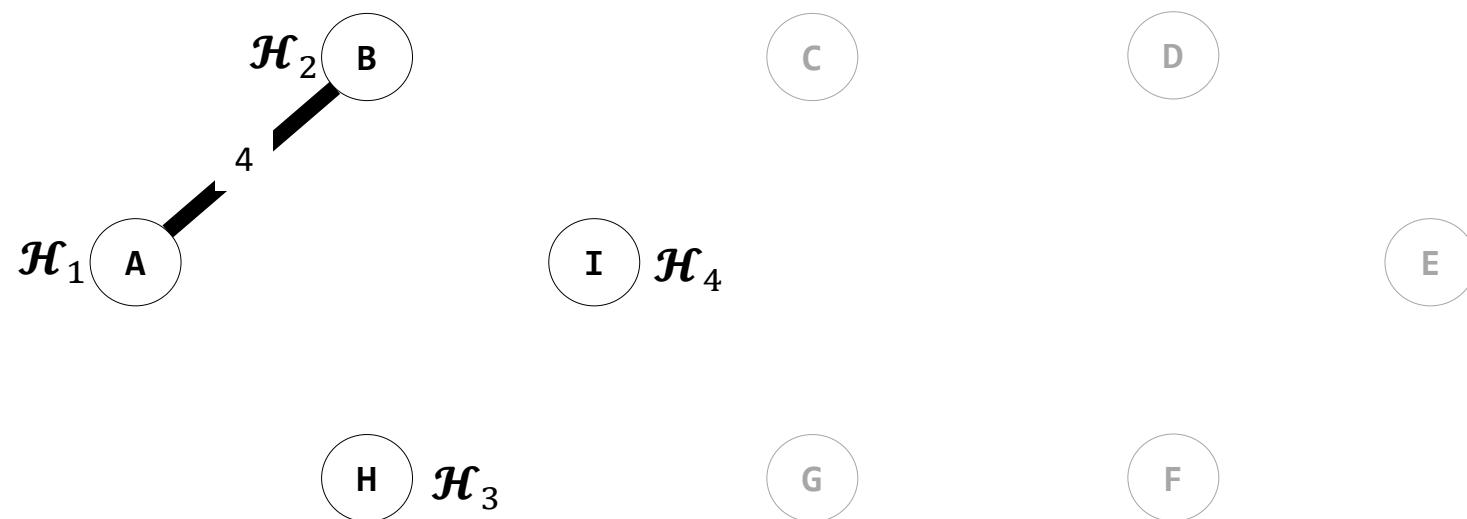
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de **menor peso** que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



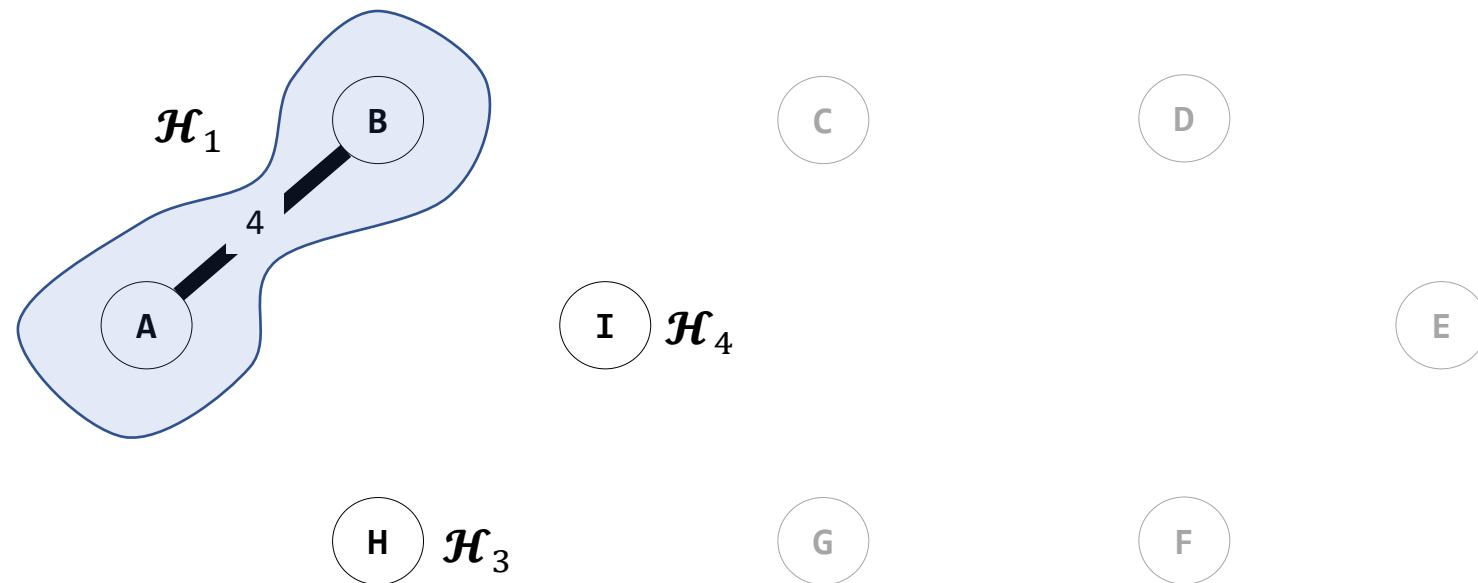
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de **menor peso** que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



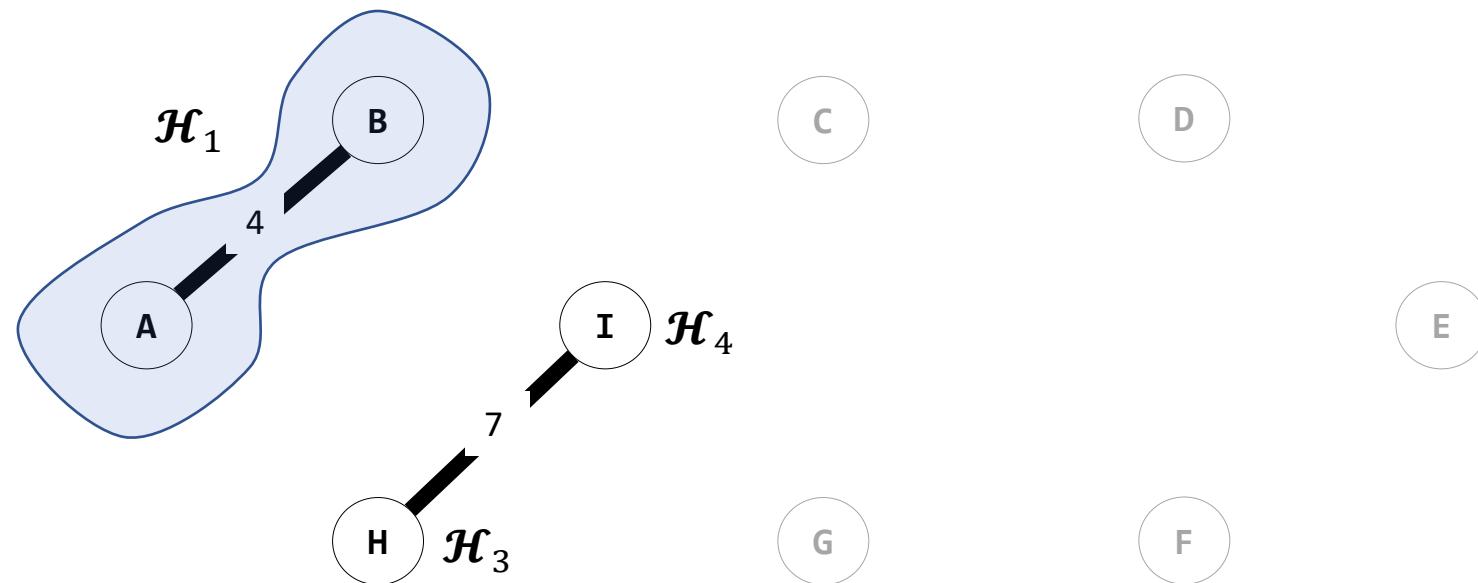
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



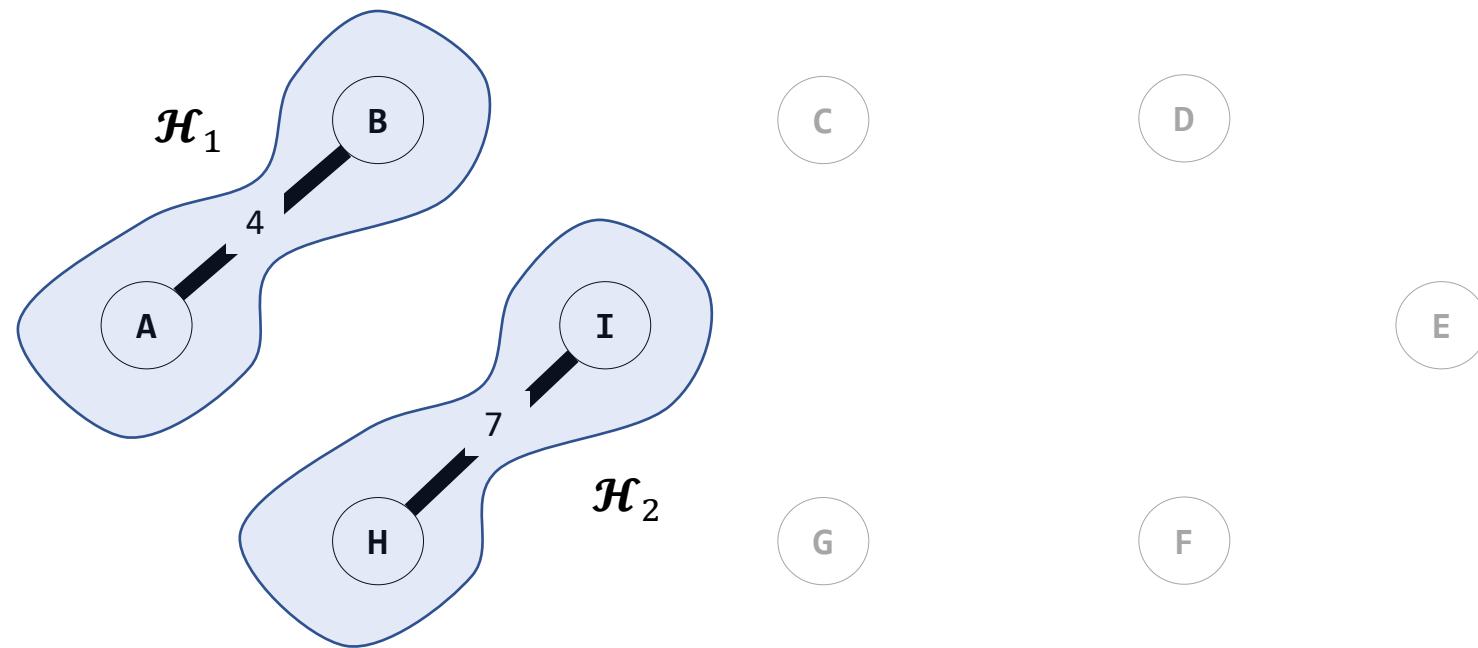
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



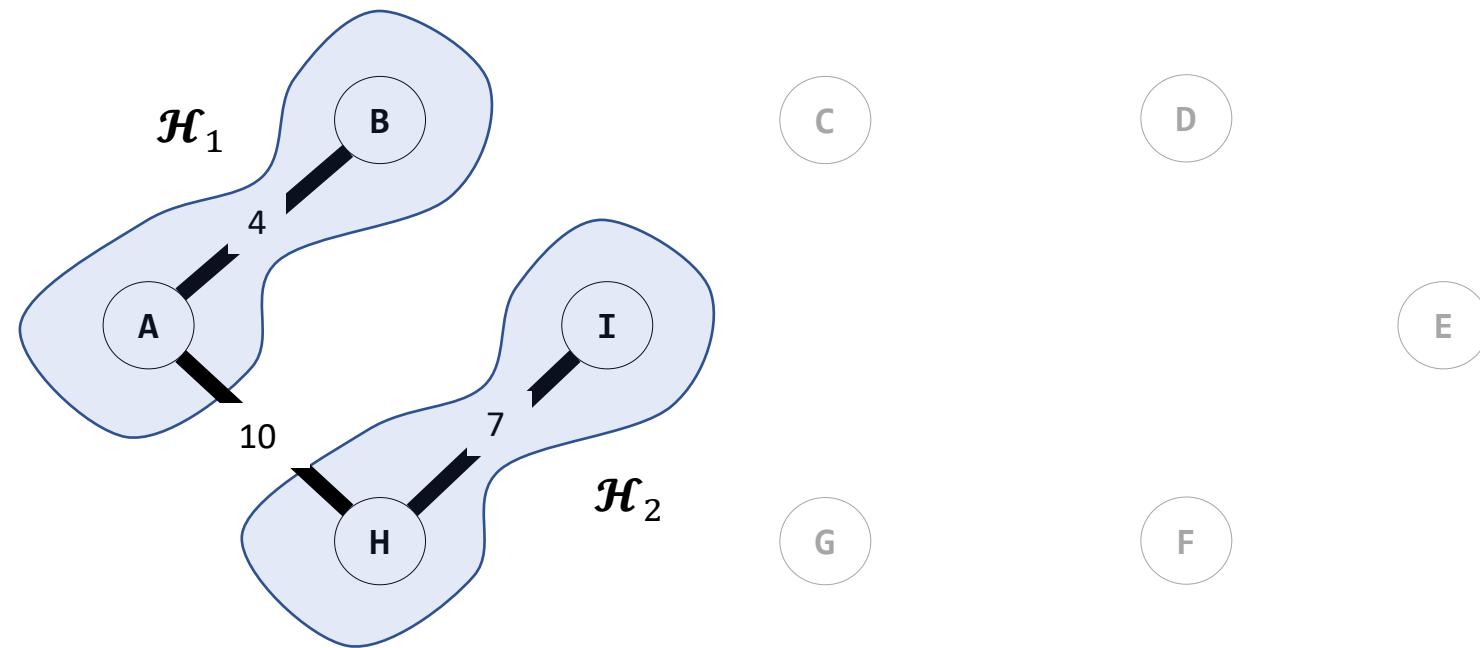
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



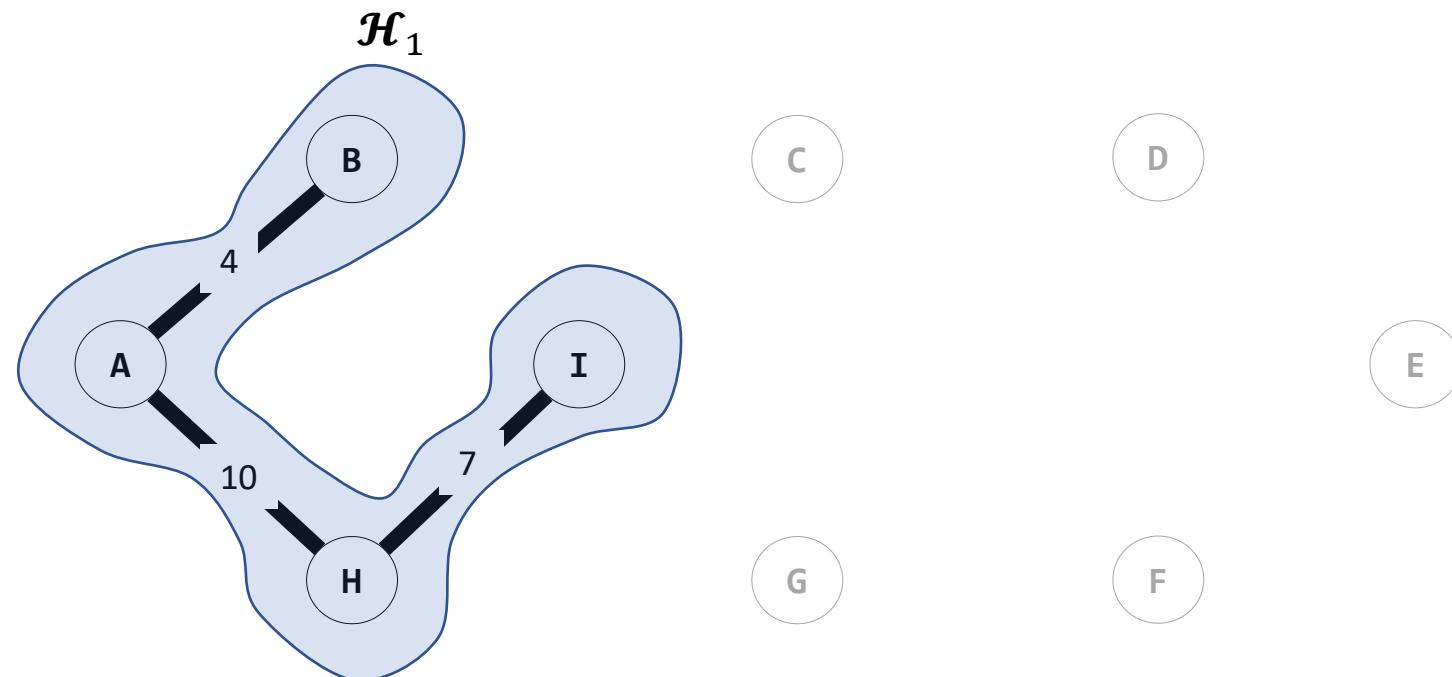
- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j



- Em cada iteração, o algoritmo escolhe uma aresta (u, v) de menor peso que liga vértices de árvores (componentes) distintos \mathcal{H}_i e \mathcal{H}_j

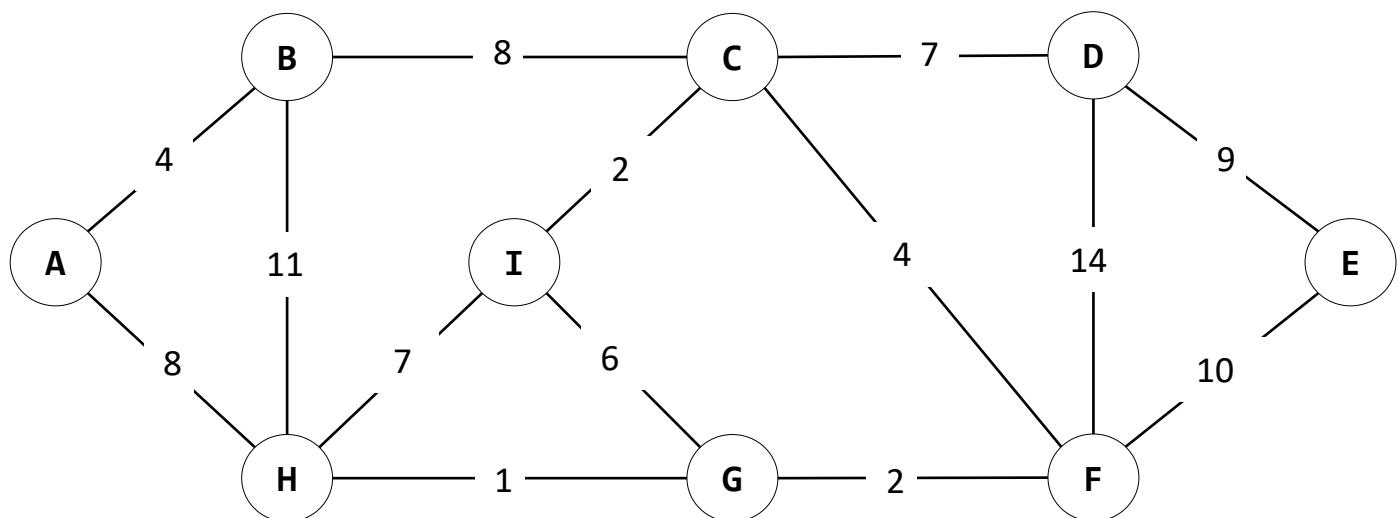


\mathcal{H}

{}

Q

$\{(H,G,1), (C,I,2), (G,F,2), (A,B,4), (C,F,4), (I,G,6), (C,D,7), (I,H,7), (A,H,8), (B,C,8), (D,E,9), (E,F,10), (F,D,14)\}$



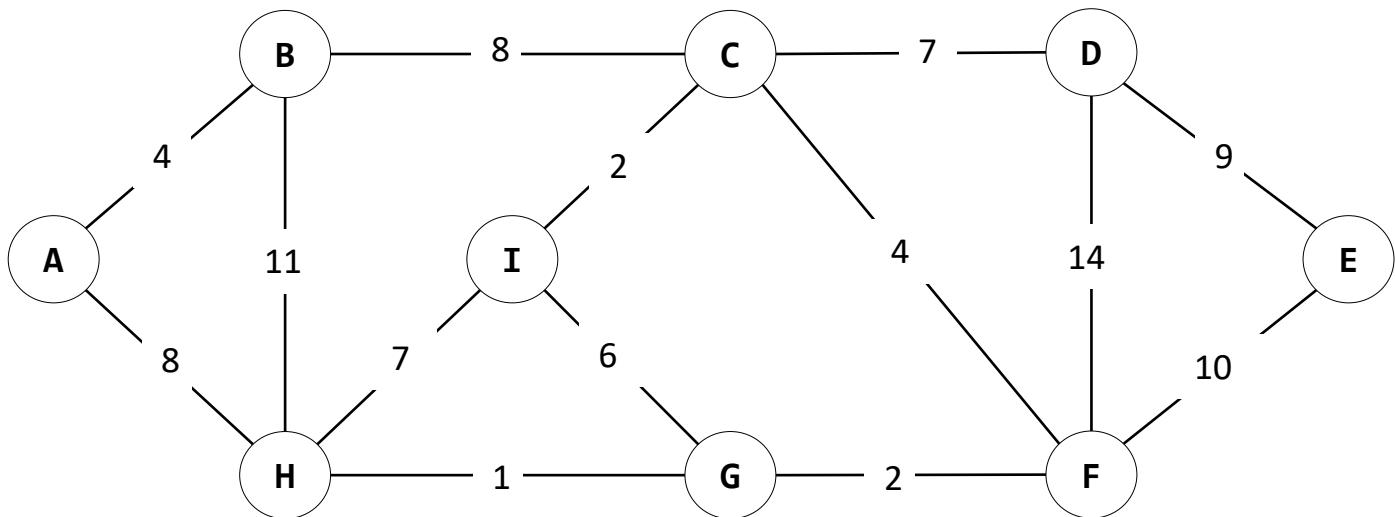
\mathcal{H}

{}

Q

$\{(H,G,1), (C,I,2), (G,F,2), (A,B,4), (C,F,4), (I,G,6), (C,D,7), (I,H,7), (A,H,8), (B,C,8), (D,E,9), (E,F,10), (B,H,11), (F,D,14)\}$

Qual escolha gulosa?

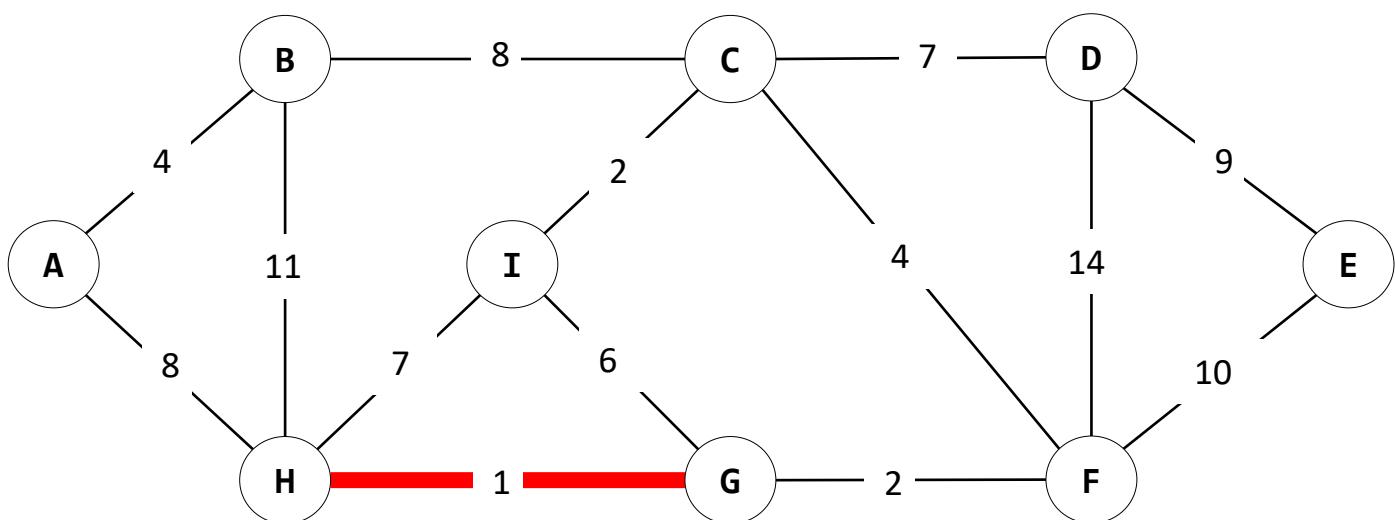


\mathcal{H}

$\{(H, G)\}$

Q

$\{(C, I, 2), (G, F, 2), (A, B, 4), (C, F, 4), (I, G, 6), (C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

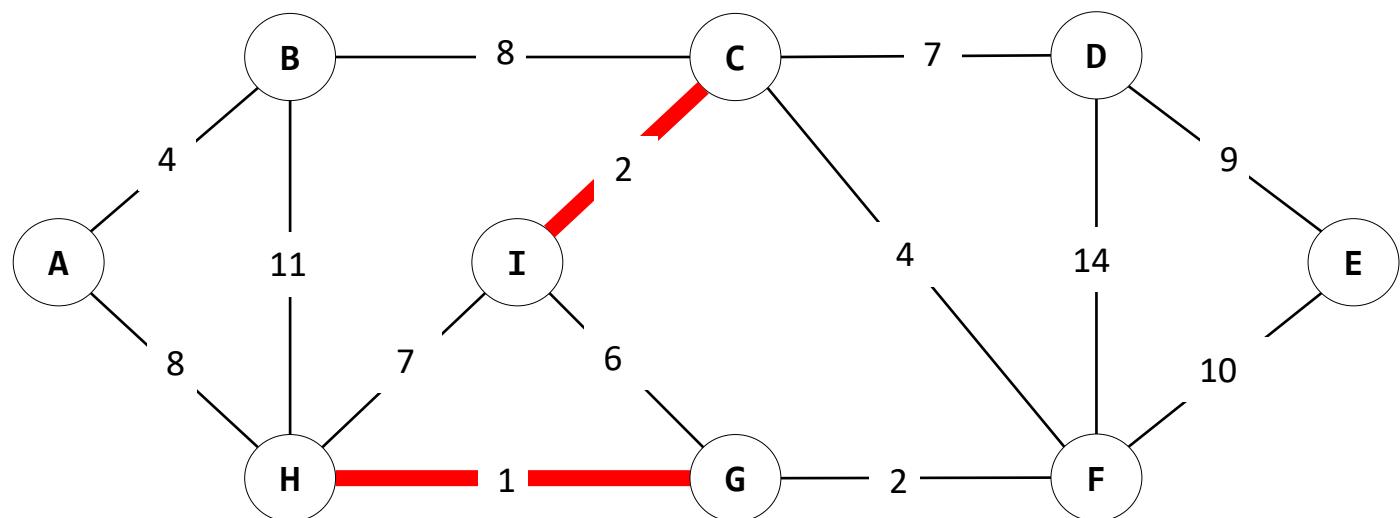


\mathcal{H}

$\{(H,G), (C,I)\}$

Q

$\{(G,F,2), (A,B,4), (C,F,4), (I,G,6), (C,D,7), (I,H,7), (A,H,8), (B,C,8), (D,E,9), (E,F,10), (B,H,11), (F,D,14)\}$

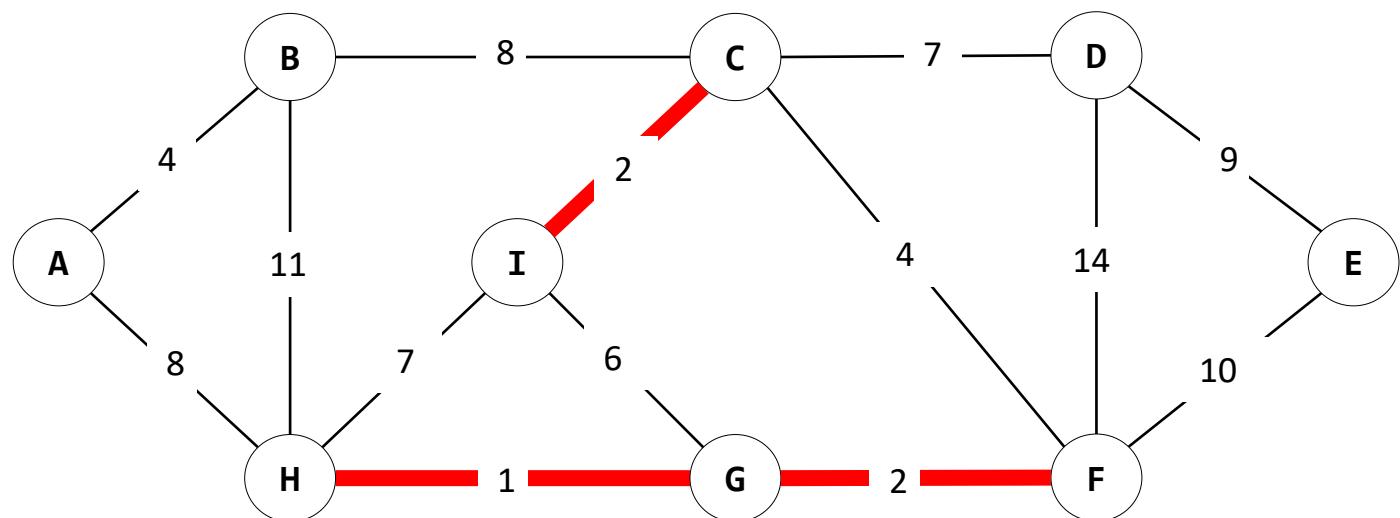


\mathcal{H}

$\{(H, G, F), (C, I)\}$

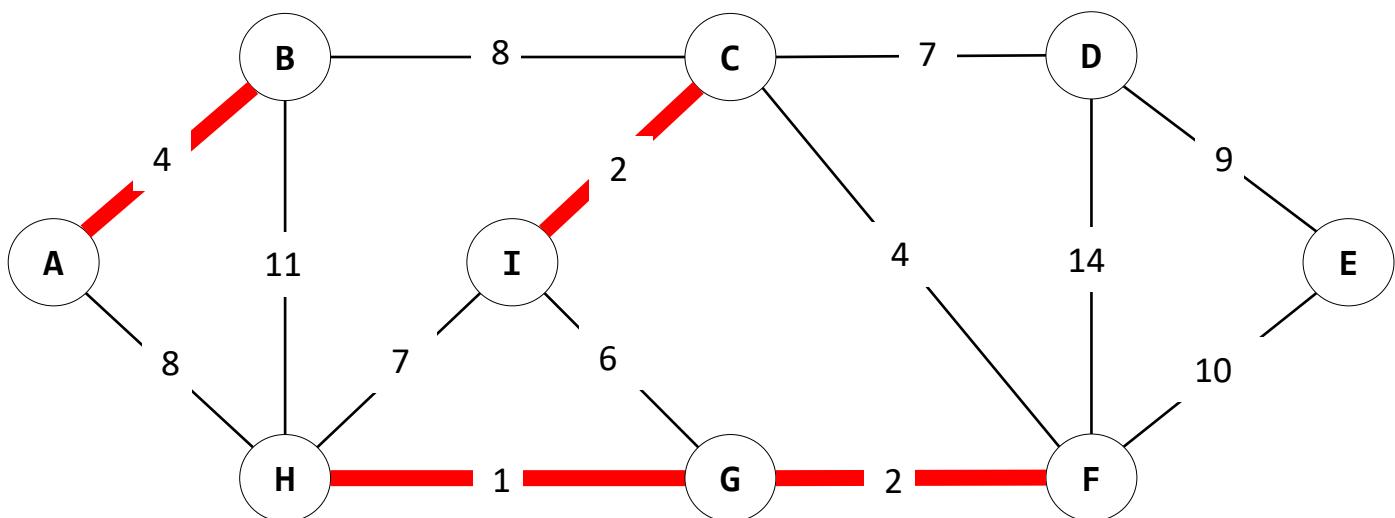
Q

$\{(A, B, 4), (C, F, 4), (I, G, 6), (C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$



\mathcal{H} $\{(H, G, F), (C, I), (B, A)\}$

Q $\{(C, F, 4), (I, G, 6), (C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

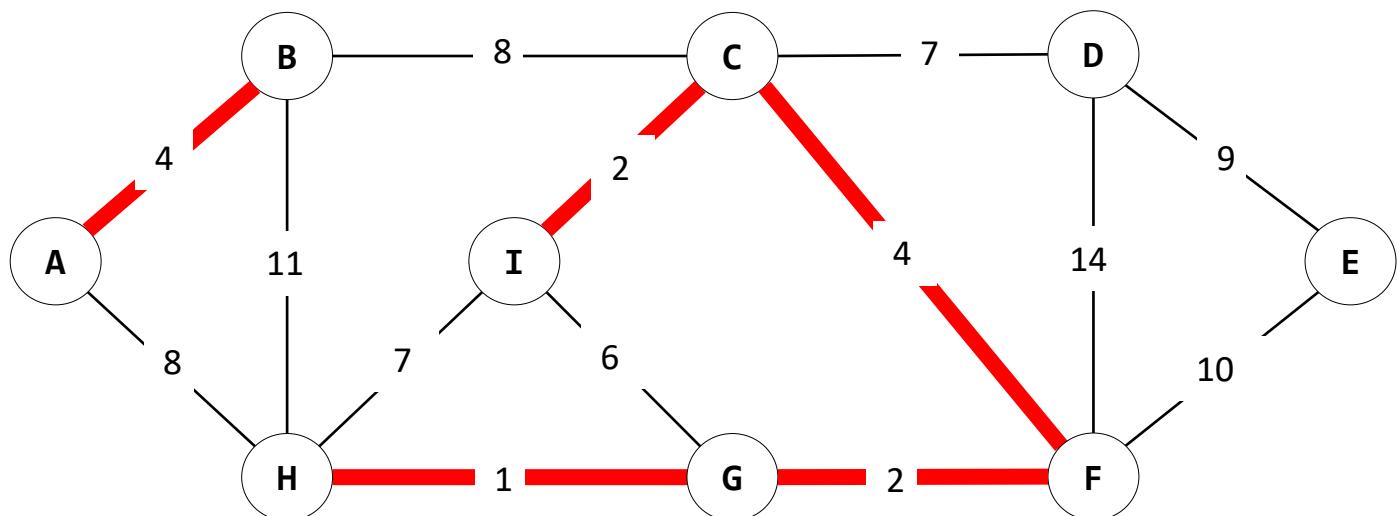


\mathcal{H}

$\{(H, G, F, C, I), (B, A)\}$

Q

$\{(I, G, 6), (C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

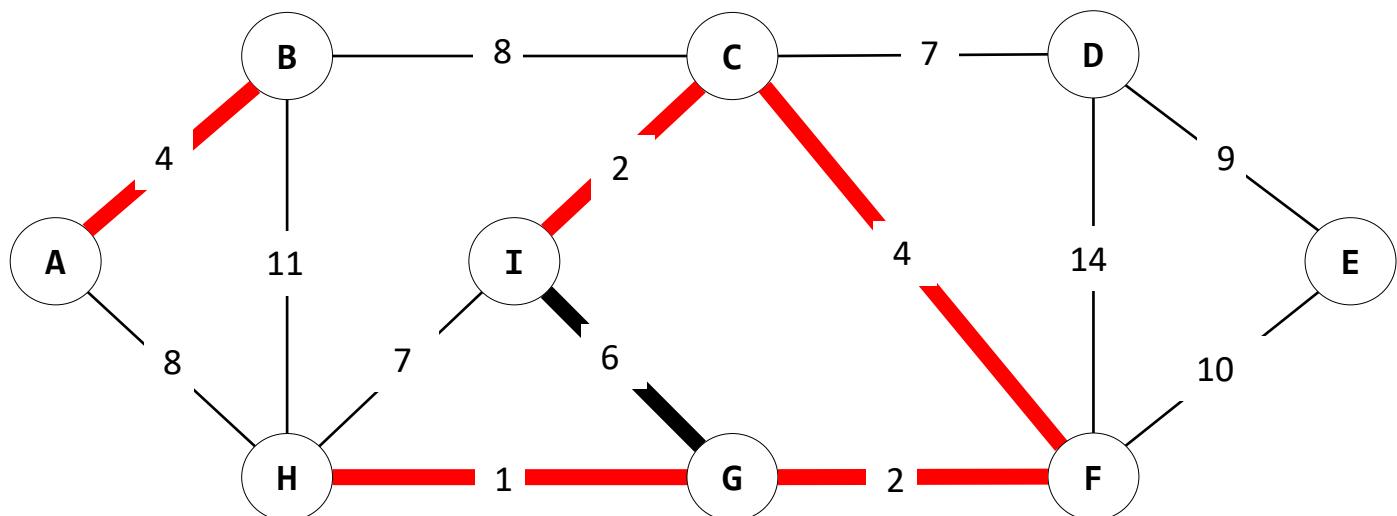


\mathcal{H}

$\{(H, G, F, C, I), (B, A)\}$

Q

$\{(I, G, 6), (C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

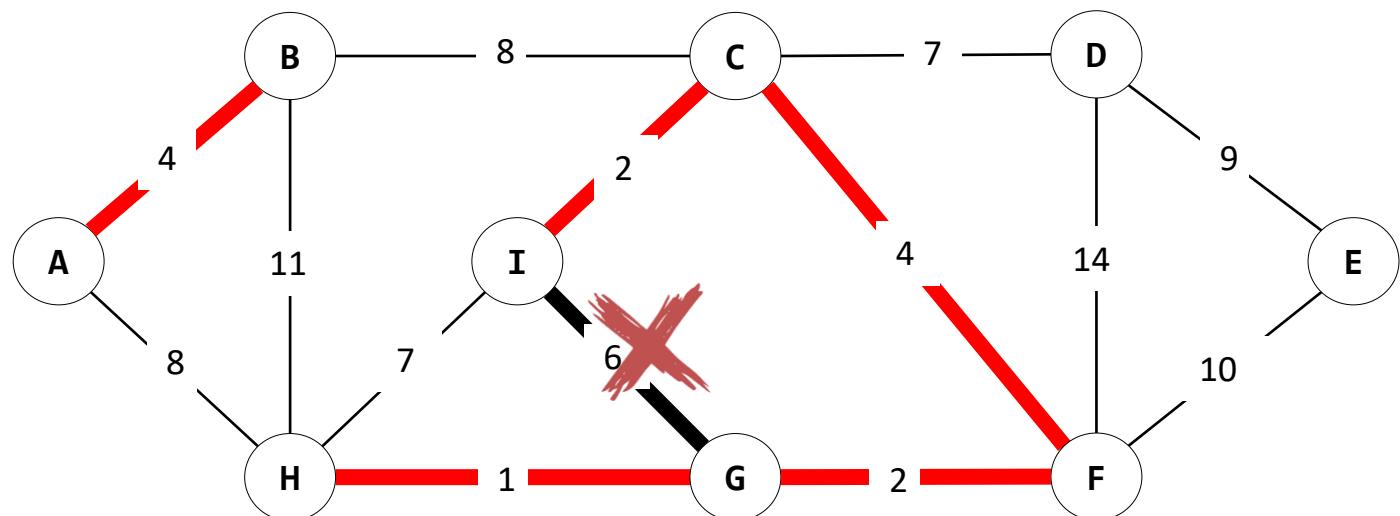


\mathcal{H}

$\{(H, G, F, C, I), (B, A)\}$

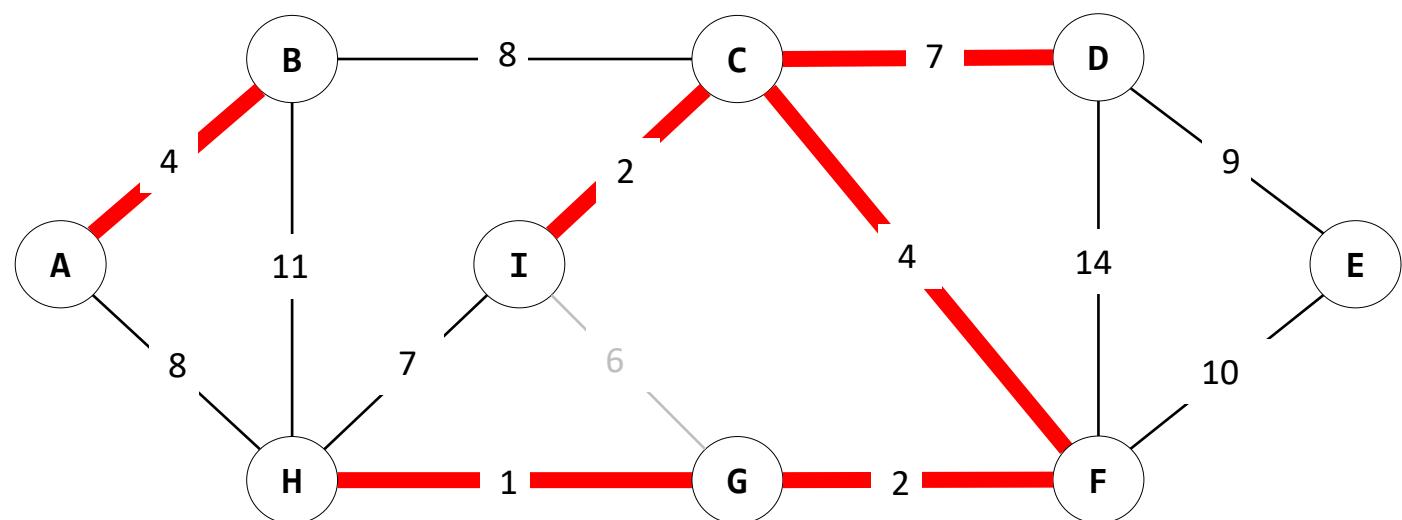
Q

$\{(C, D, 7), (I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$



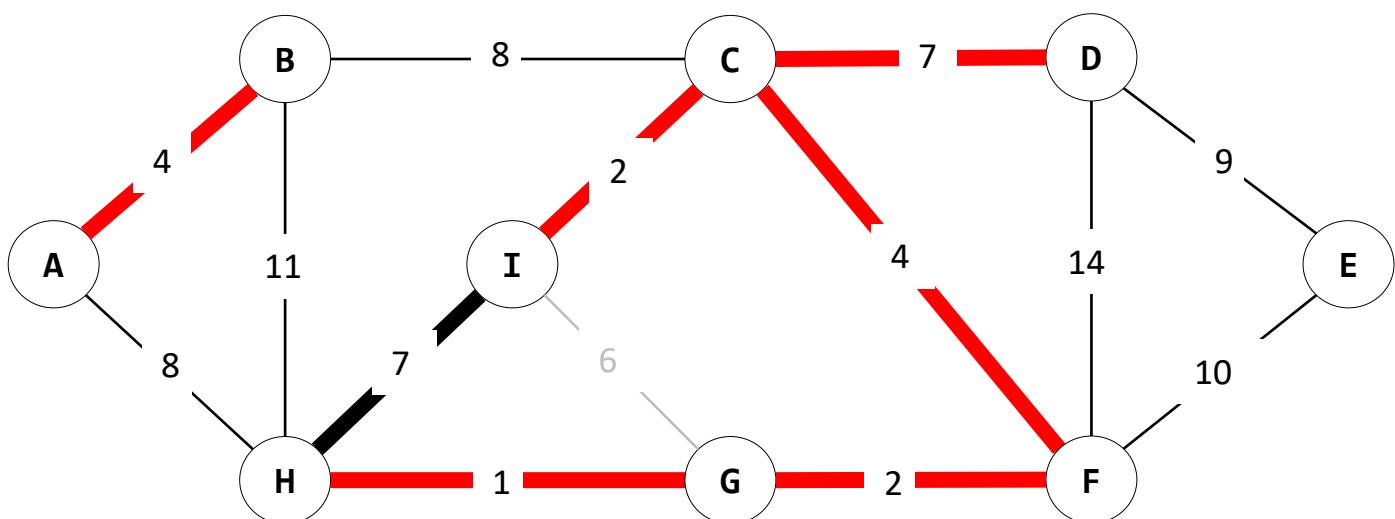
\mathcal{H} $\{(H, G, F, C, I, D), (B, A)\}$

Q $\{(I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$



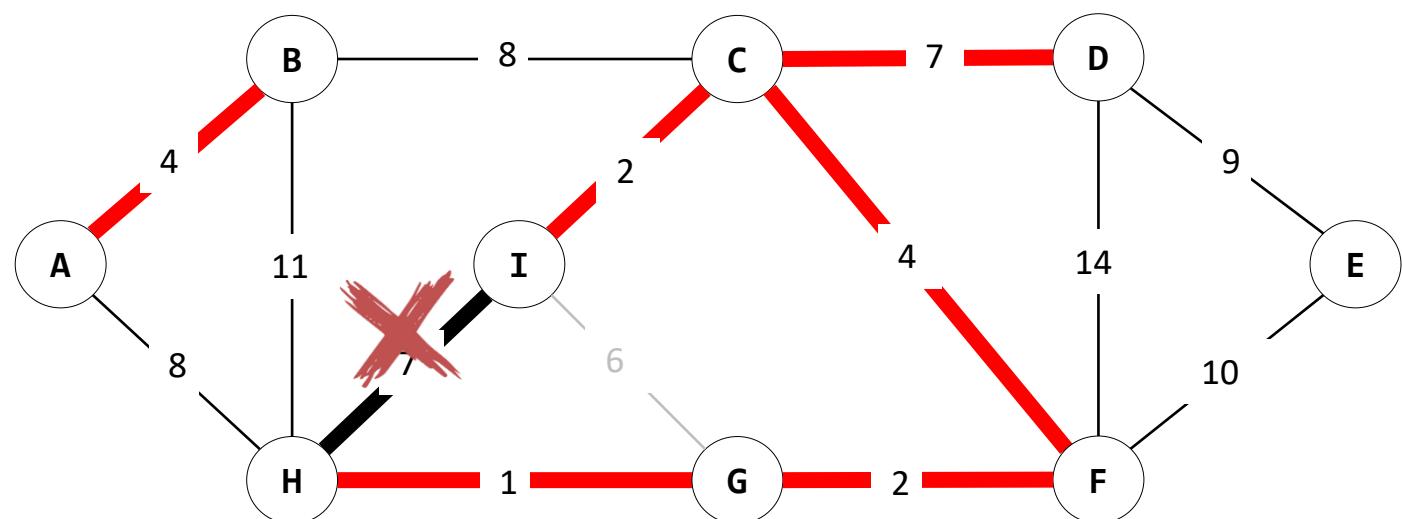
\mathcal{H} $\{(H, G, F, C, I, D), (B, A)\}$

Q $\{(I, H, 7), (A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$



\mathcal{H} $\{(H, G, F, C, I, D), (B, A)\}$

Q $\{(A, H, 8), (B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

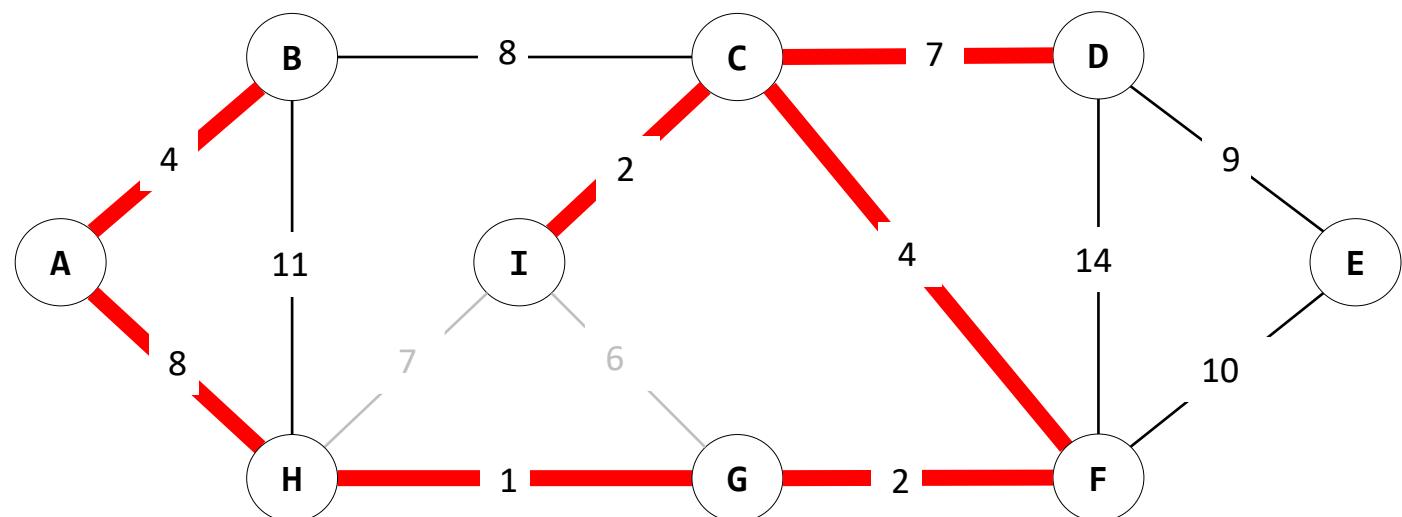


\mathcal{H}

$\{(H, G, F, C, I, D, B, A)\}$

Q

$\{(B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

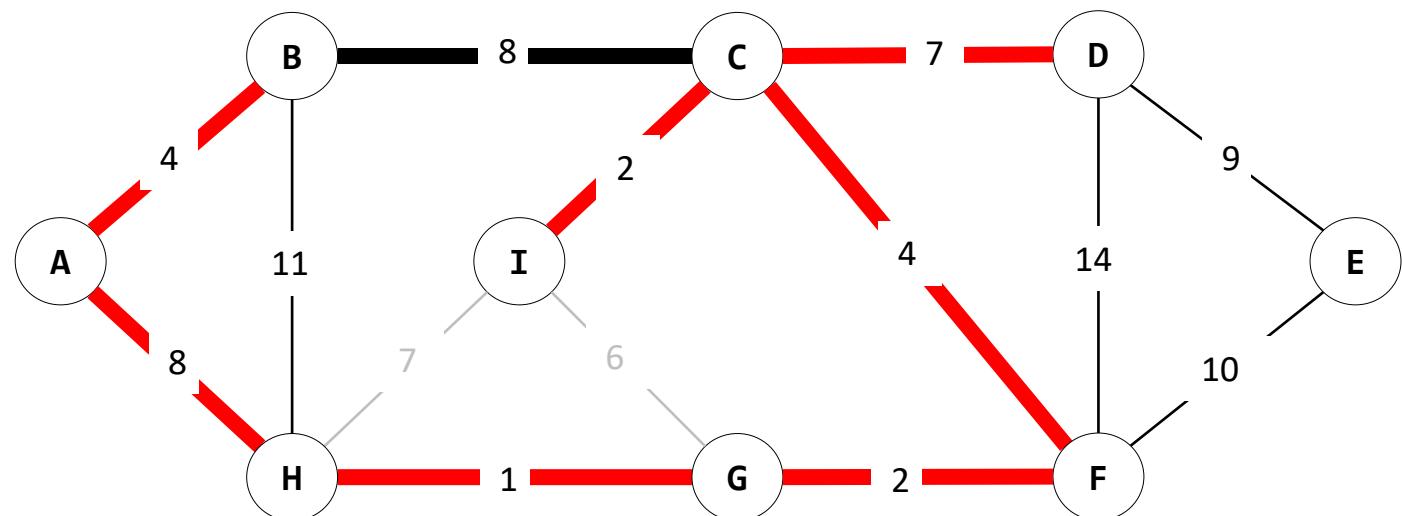


\mathcal{H}

$\{(H, G, F, C, I, D, B, A)\}$

Q

$\{(B, C, 8), (D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$

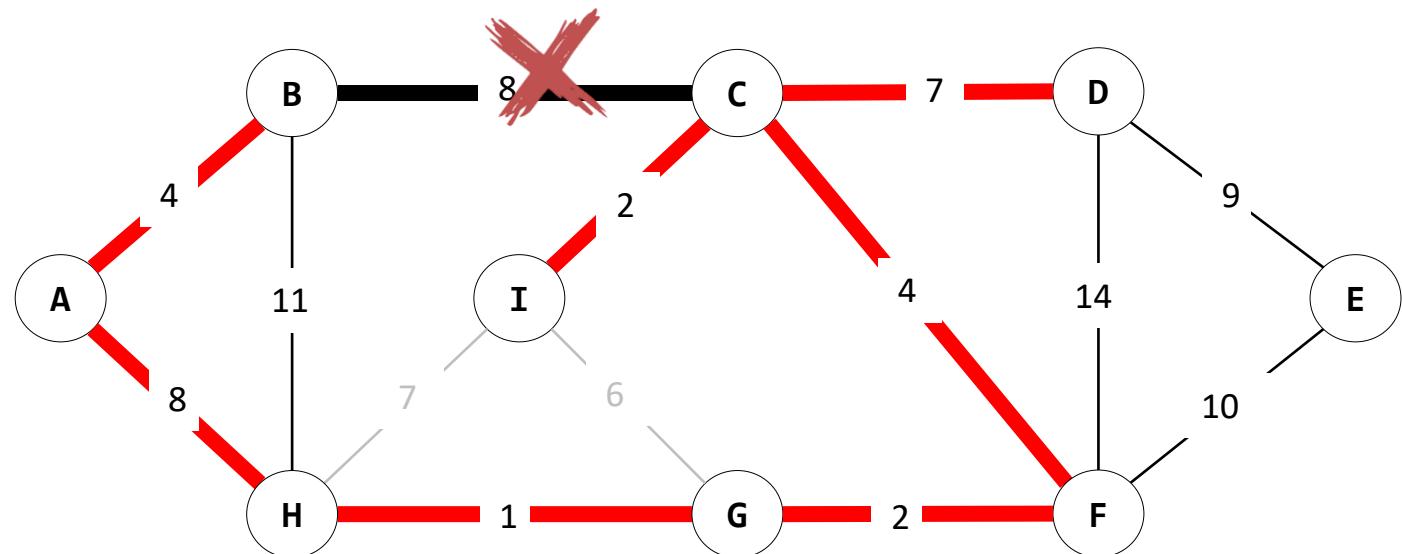


\mathcal{H}

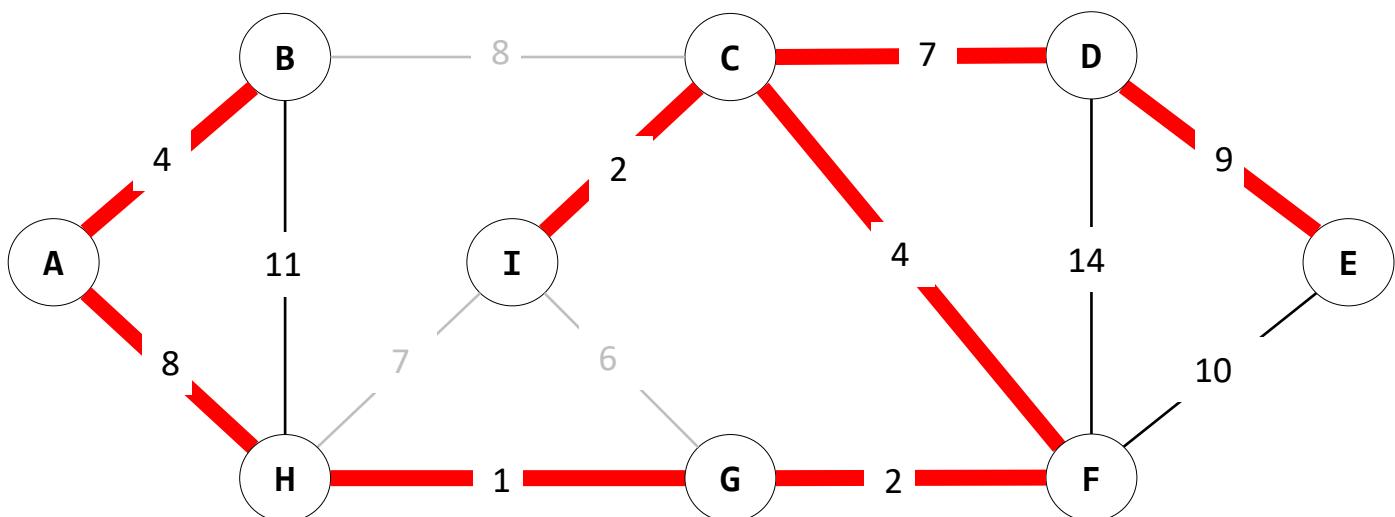
$\{(H, G, F, C, I, D, B, A)\}$

Q

$\{(D, E, 9), (E, F, 10), (B, H, 11), (F, D, 14)\}$



\mathcal{H}	$\{(H, G, F, C, I, D, B, A, E)\}$
Q	$\{(E, F, 10), (B, H, 11), (F, D, 14)\}$

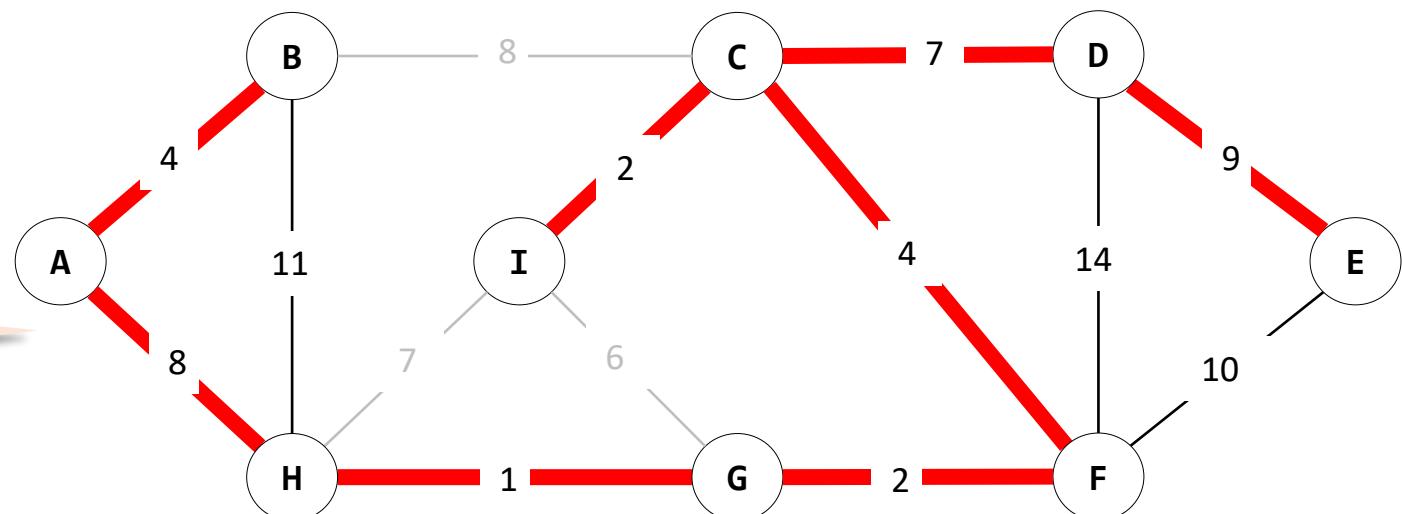


\mathcal{H}
Q

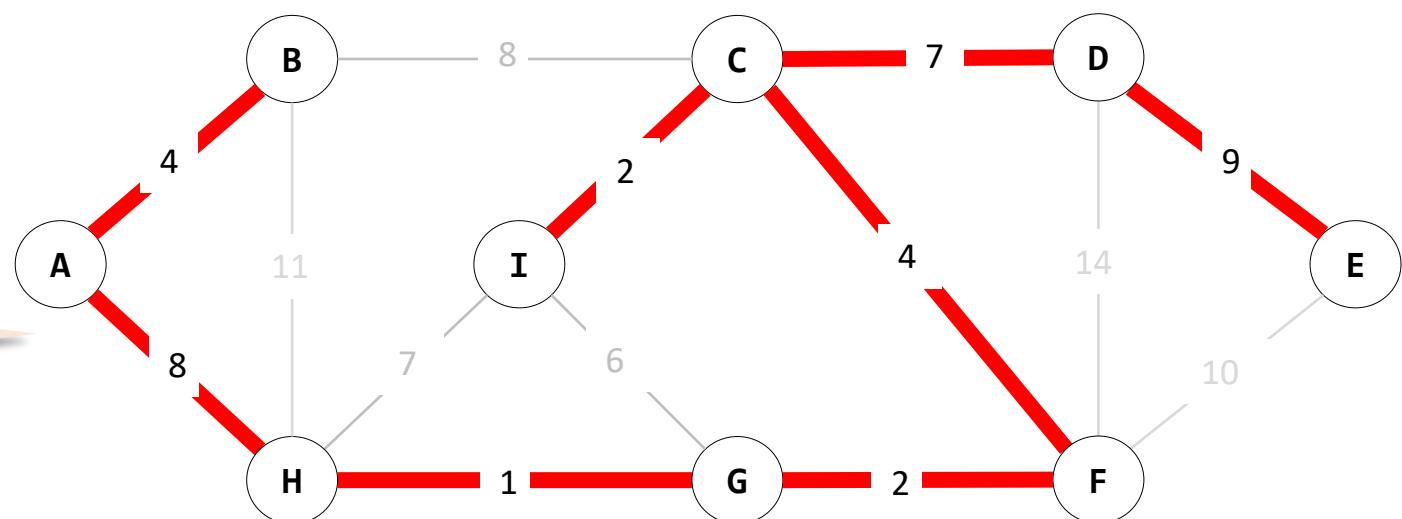
$\{(H, G, F, C, I, D, B, A, E)\}$

$\{(E, F, 10), (B, H, 11), (F, D, 14)\}$

Todos os vértices estão em \mathcal{H} ?
Removo as arestas remanescentes.



\mathcal{H}	$\{(H, G, F, C, I, D, B, A, E)\}$
Q	$\{\}$



Todos os vértices estão em \mathcal{H} ?
Removo as arestas remanescentes.

Kruskal(\mathcal{G})

$\mathcal{H} = \emptyset$

Ordene as arestas de \mathcal{G} em **ordem não-decrescente** de peso

Para cada aresta $(u, v) \in \mathcal{E}$ nessa ordem **Faça**

Se u e v estão em componentes distintos **Então**

$\mathcal{H} = \mathcal{H} \cup \{(u, v)\}$

devolva \mathcal{H}

Problema: Como verificar eficientemente se u e v estão no mesmo componente da floresta?

Kruskal(\mathcal{G})

$\mathcal{H} = \emptyset$

Ordena as arestas de \mathcal{G} em ordem não-decrescente de peso

Se a aresta $(u, v) \in \mathcal{E}$ nessa ordem Faça

Se u e v estão em componentes distintos Então 

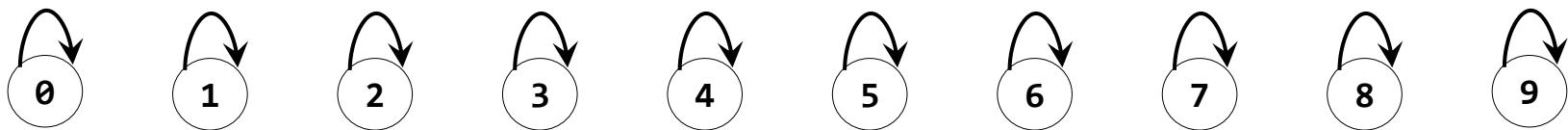
$\mathcal{H} = \mathcal{H} \cup \{(u, v)\}$



- Nome dado a uma estrutura de dados que realiza duas operações
 - **Union**
 - **Find**
- É um tipo abstrato de dados de uma variação específica de conjuntos disjuntos
- **MakeSet(G)**
 - Cria um conjunto unitário com o elemento u
- **FindSet(u):**
 - Devolve o identificador do bloco da partição que contém u
- **Union(u, v):**
 - Substitui os blocos da partição que contêm u e v pela união deles

Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore



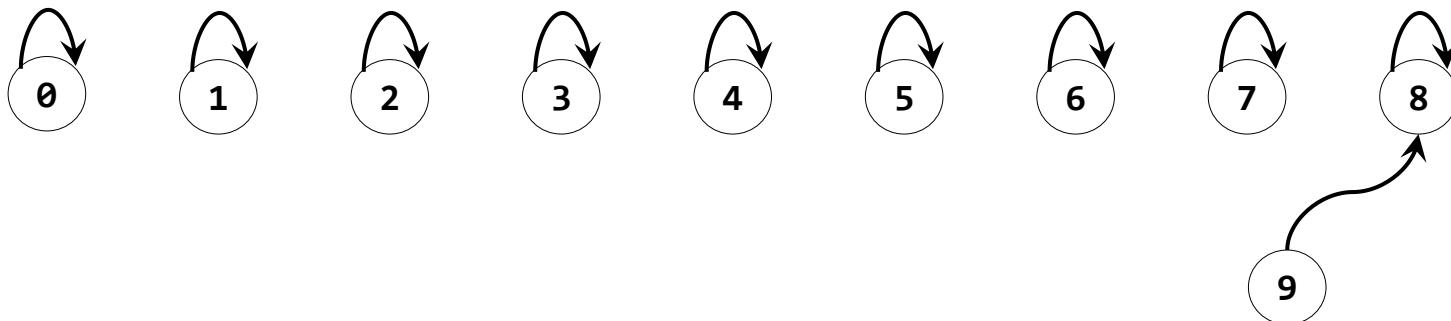
π	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

Inicialmente, cada vértice é um conjunto disjunto e sua raiz é si próprio

Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)



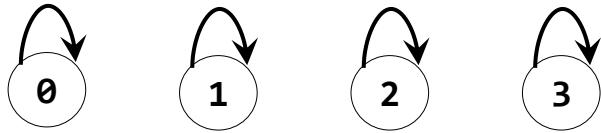
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	8

Union(u, v)

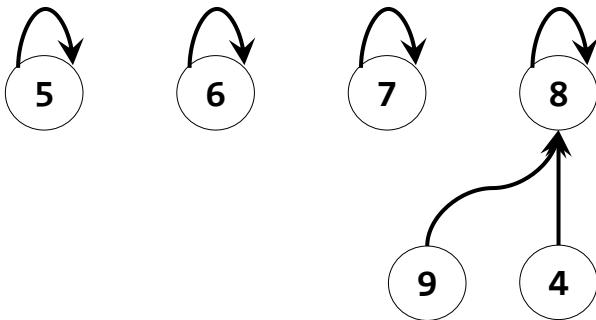
- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)

Union(4, 8)



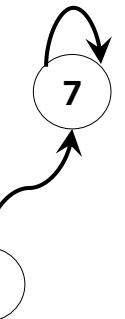
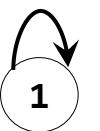
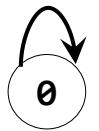
0	1	2	3	4	5	6	7	8	9
0	1	2	3	8	5	6	7	8	8



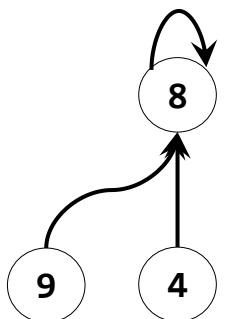
Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)



Union(4, 8)



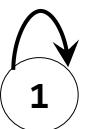
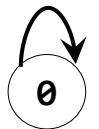
Union(5, 7)

π	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	8	7	6	7	8	8

Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)

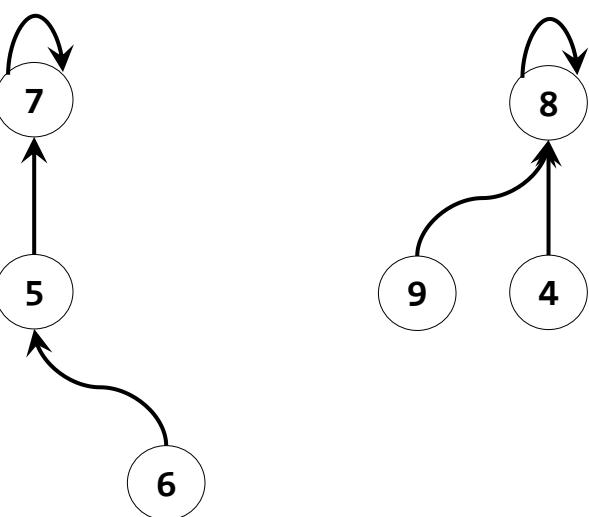


Union(4, 8)

0	1	2	3	4	5	6	7	8	9
0	1	2	3	8	7	5	7	8	8

Union(5, 7)

Union(6, 5)



Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

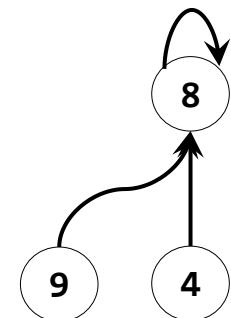
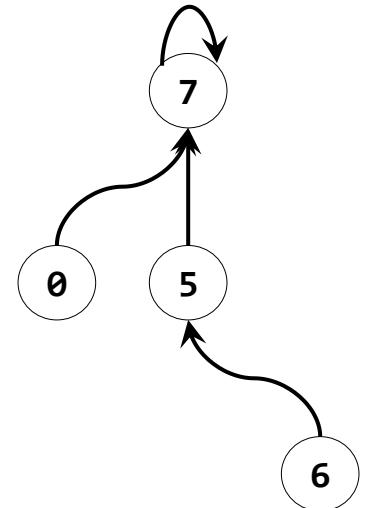
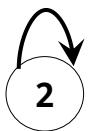
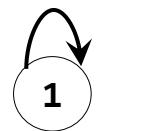
Union(9, 8)

Union(4, 8)

Union(5, 7)

Union(6, 5)

Union(0, 7)



0	1	2	3	4	5	6	7	8	9
7	1	2	3	8	7	5	7	8	8

Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)

Union(4, 8)

Union(5, 7)

Union(6, 5)

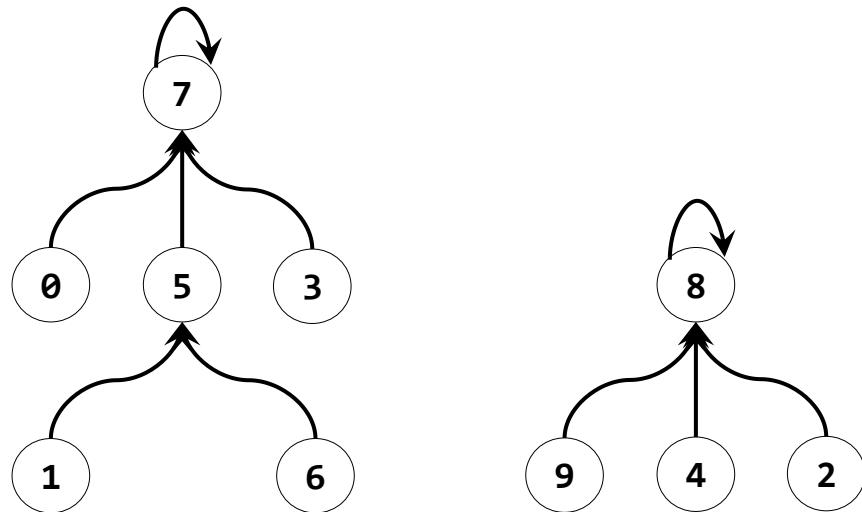
Union(0, 7)

Union(2, 8)

Union(1, 5)

Union(3, 7)

	0	1	2	3	4	5	6	7	8	9
π	7	5	8	7	8	7	5	7	8	8



Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)

Union(4, 8)

Union(5, 7)

Union(6, 5)

Union(0, 7)

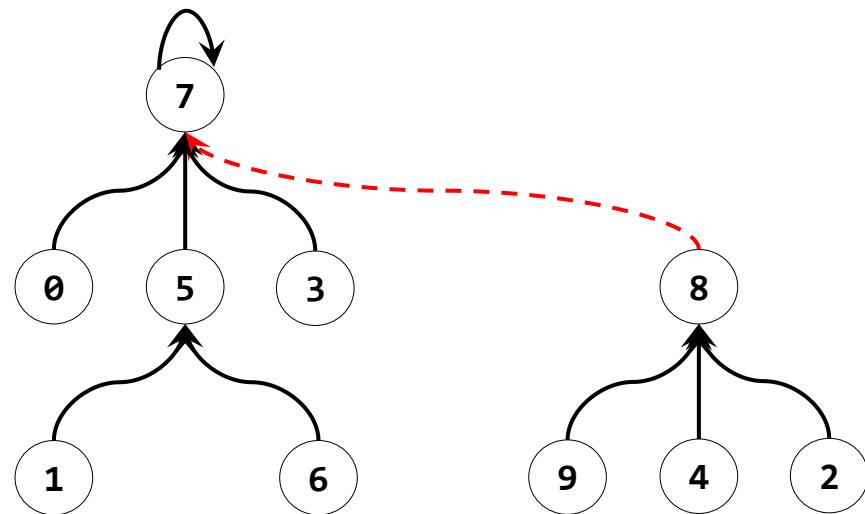
Union(2, 8)

Union(1, 5)

Union(3, 7)

Union(8, 7)

	0	1	2	3	4	5	6	7	8	9
π	7	5	8	7	8	7	5	7	7	8



Union(u, v)

- Cria uma estrutura de árvore chamada de *Parent-Link Representation*, porém, chamaremos apenas de árvore

Union(9, 8)

Union(4, 8)

Union(5, 7)

Union(6, 5)

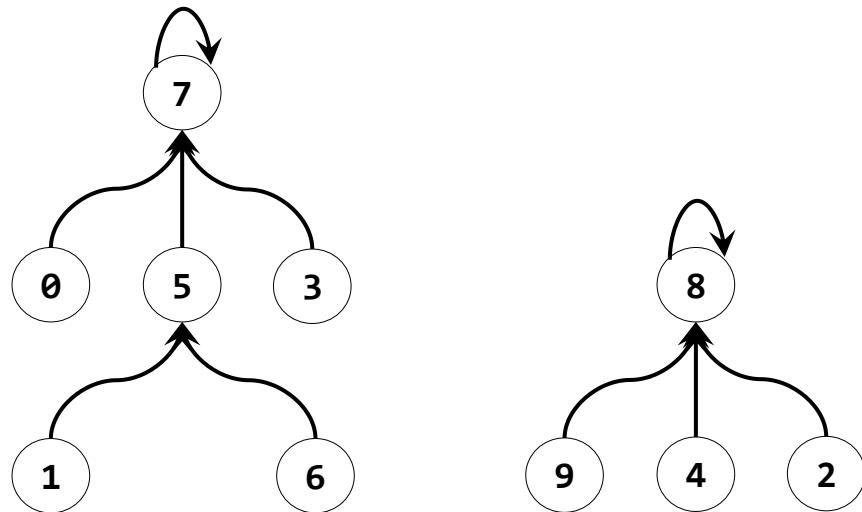
Union(0, 7)

Union(2, 8)

Union(1, 5)

Union(3, 7)

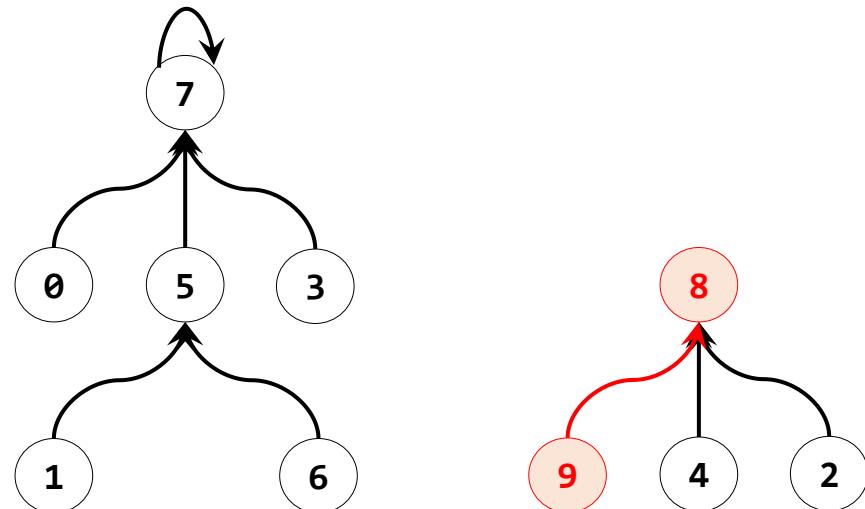
	0	1	2	3	4	5	6	7	8	9
π	7	5	8	7	8	7	5	7	8	8



Find(u)

- Retorna a raiz ou representante da árvore ou conjunto que u pertence

$$\text{Find}(9) = 8$$



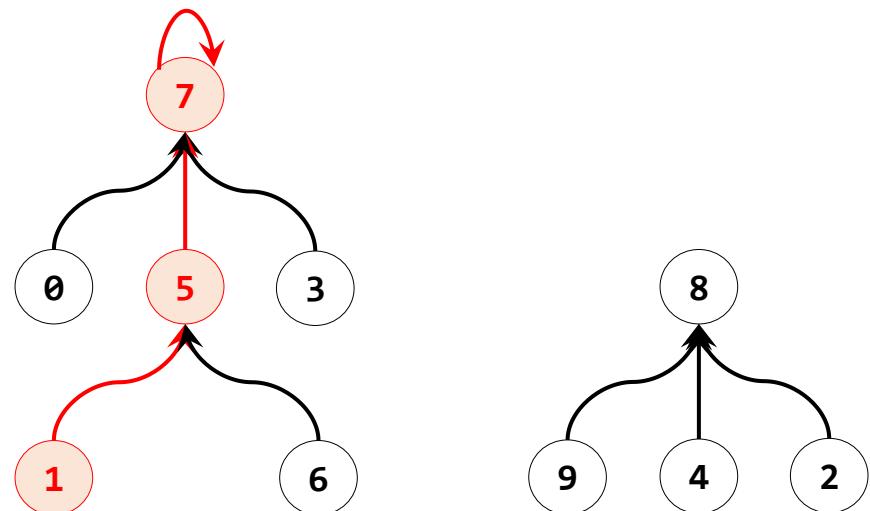
π	0	1	2	3	4	5	6	7	8	9
	7	5	8	7	8	7	5	7	8	8

Find(u)

- Retorna a raiz ou representante da árvore ou conjunto que u pertence

$$\text{Find}(9) = 8$$

$$\text{Find}(1) = 7$$



π	0	1	2	3	4	5	6	7	8	9
	7	5	8	7	8	7	5	7	8	8

Find(u)

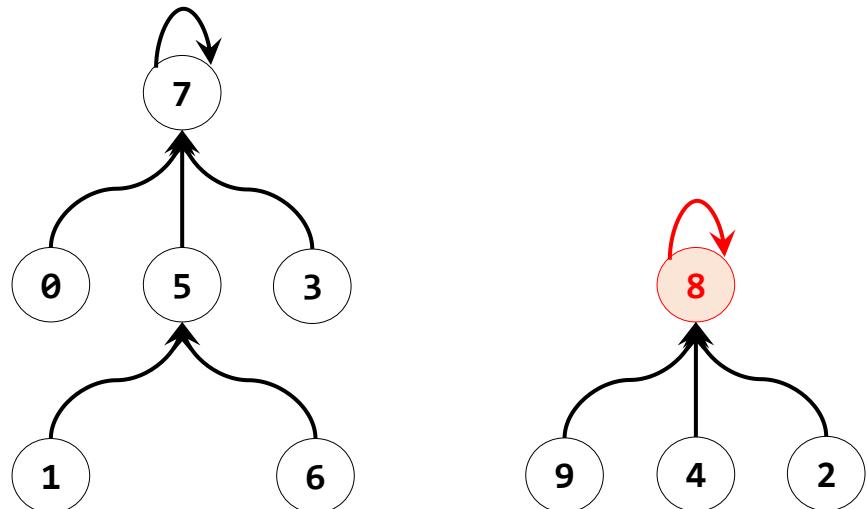
- Retorna a raiz ou representante da árvore ou conjunto que u pertence

Find(9) = 8

Find(1) = 7

Find(8) = 8

π	0	1	2	3	4	5	6	7	8	9
	7	5	8	7	8	7	5	7	8	8



- Implementação *Naïve* ou Ingênua

```
def MakeSet( $\mathcal{G}$ ):  
    for  $u$  in  $\mathcal{V}$ :  
         $\pi[u] = u$   
  
def Find( $u$ ):  
    while  $\pi[u] \neq u$ :  
         $u = \pi[u]$   
    return( $u$ )  
  
def Union( $u, v$ ):  
     $\pi[u] = v$ 
```

- Implementação *Naïve* ou Ingênua

```
def MakeSet(G):
    for u in V:
        π[u] = u
```

```
def Find(u):
    while π[u] != u:
        u = π[u]
    return(u)
```

```
def Union(u, v):
    π[u] = v
```

```
def Find(u):
    if π[u] == u:
        return(u)
    return(Find(π[u]))
```

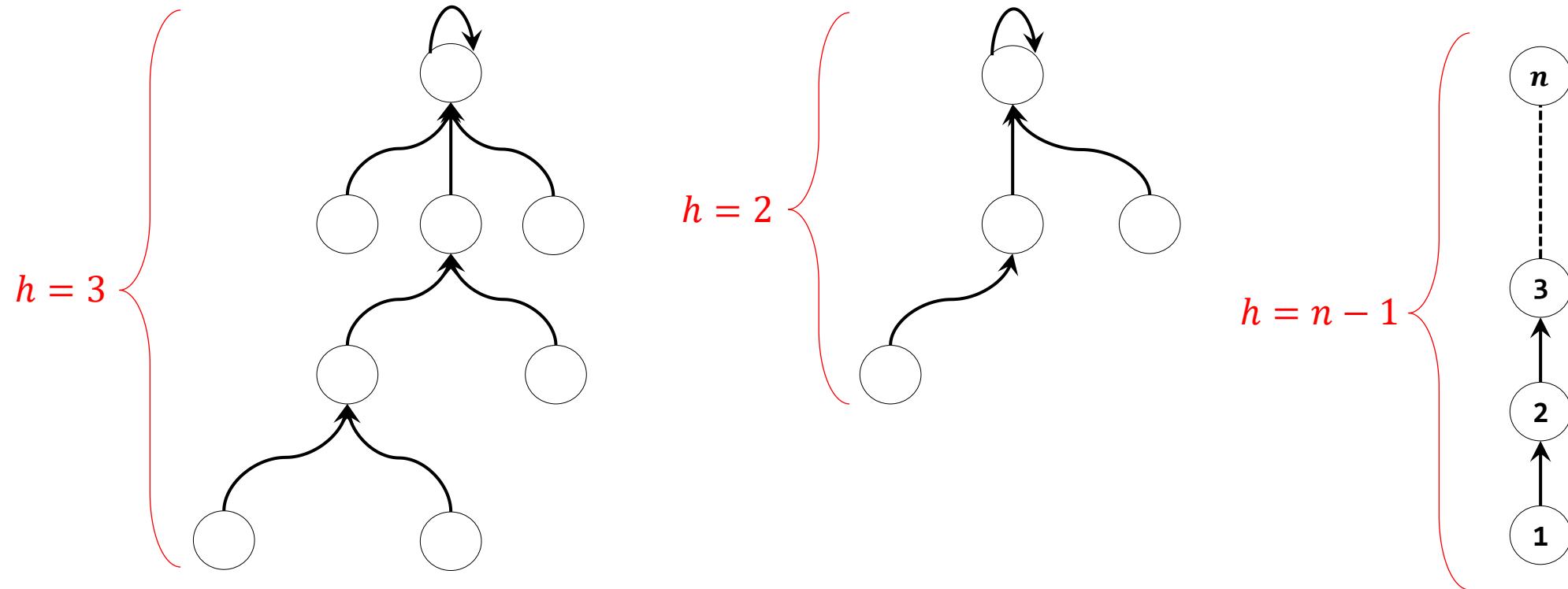
- Implementação *Naïve* ou Ingênua

```
def MakeSet( $\mathcal{G}$ ):  
    for  $u$  in  $\mathcal{V}$ :  
         $\pi[u] = u$   
  
def Find( $u$ ):  
    while  $\pi[u] \neq u$ :  
         $u = \pi[u]$   
    return( $u$ )  
  
def Union( $u, v$ ):  
     $\pi[u] = v$ 
```

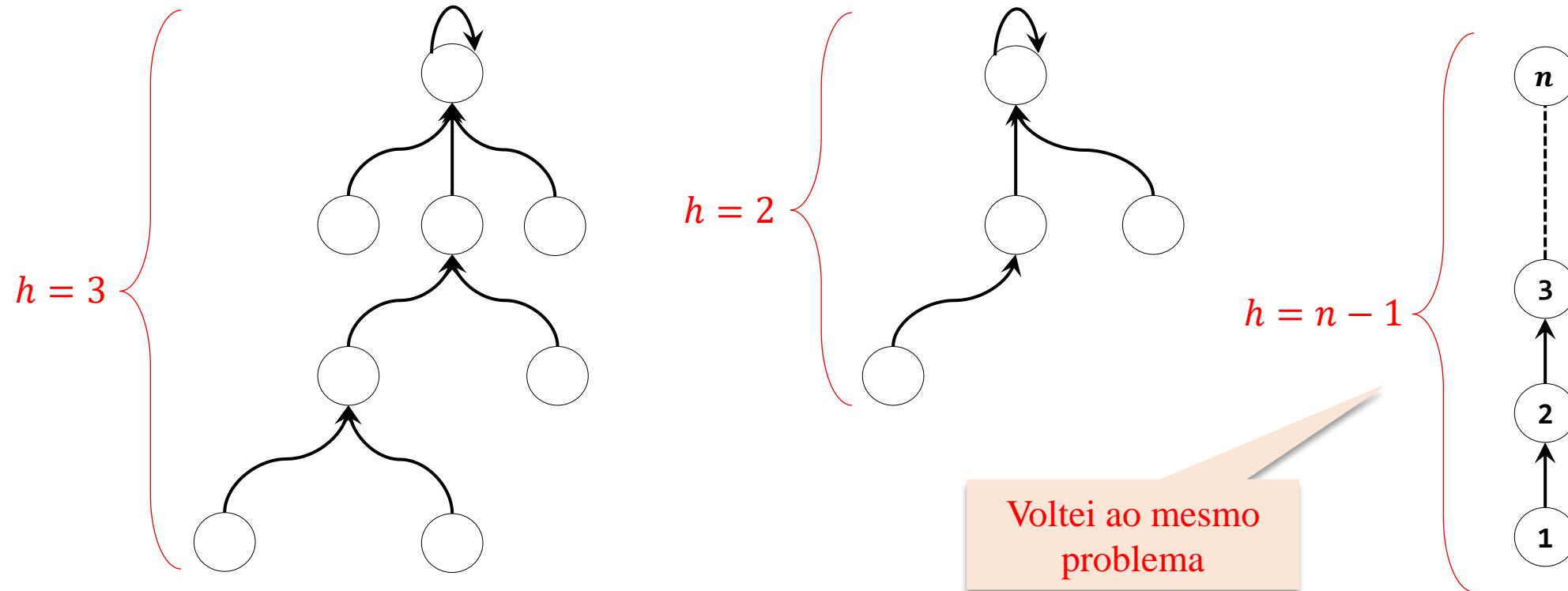


Problemas?

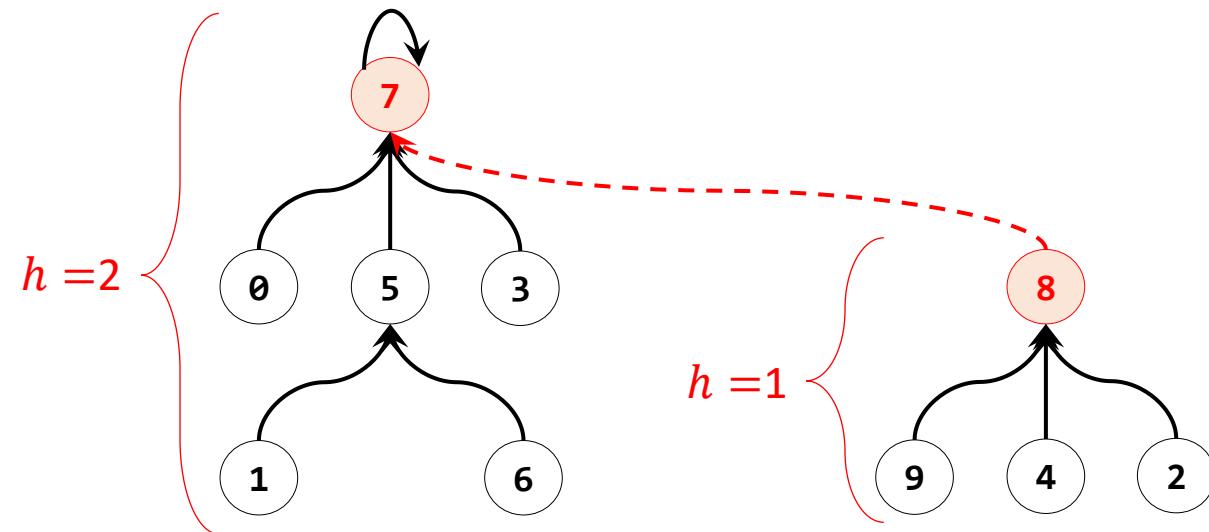
- Usando a implementação *Naïve* ou Ingênua, uma operação $\text{Find}(u)$ pode levar $O(n)$ no pior caso, onde n é o número de elementos.
- No pior caso, a operação $\text{Find}(u)$ leva um tempo proporcional à altura da árvore



- Usando a implementação *Naïve* ou Ingênua, uma operação $\text{Find}(u)$ pode levar $O(n)$ no pior caso, onde n é o número de elementos.
- No pior caso, a operação $\text{Find}(u)$ leva um tempo proporcional à altura da árvore



- Para melhorar a operação $\text{Find}(u)$ a ideia é aprimorar a operação $\text{Union}(u, v)$ segundo duas estratégias
 1. Unir u à raiz de v , ou seja, unir u ao $\text{Find}(v)$
 2. Unir a raiz da árvore menor à raiz da árvore maior (selecionando empates arbitrariamente)



- Para melhorar a operação $\text{Find}(u)$ a ideia é aprimorar a operação $\text{Union}(u, v)$ seguindo duas estratégias
 - Unir u à raiz de v , ou seja, unir u ao $\text{Find}(v)$
 - Unir a raiz da árvore menor à raiz da árvore maior (selecionando empates arbitrariamente)

```
def MakeSet( $\mathcal{G}$ ):  
    for  $u$  in  $\mathcal{V}$ :  
         $\pi[u] = u$   
         $h[u] = 1$ 
```

```
def Union( $u, v$ ):  
     $x = \text{Find}(u)$   
     $y = \text{Find}(v)$   
    if  $x = y$ :  
        return NULL  
    if  $h[x] > h[y]$ :  
         $\pi[y] = x$   
    elif  $h[y] > h[x]$ :  
         $\pi[x] = y$   
    else:  
         $\pi[x] = y$   
         $h[y] = h[y] + 1$ 
```

- Para melhorar a operação $\text{Find}(u)$ a ideia é aprimorar a operação $\text{Union}(u, v)$ seguindo duas estratégias
 - Unir u à raiz de v , ou seja, unir u ao $\text{Find}(v)$
 - Unir a raiz da árvore menor à raiz da árvore maior (selecionando empates arbitrariamente)

```
def MakeSet( $\mathcal{G}$ ):  
    for  $u$  in  $\mathcal{V}$ :  
         $\pi[u] = u$   
         $h[u] = 1$ 
```

```
def Union( $u, v$ ):  
     $x = \text{Find}(u)$   
     $y = \text{Find}(v)$   
    if  $x = y$ :  
        return NULL  
    if  $h[x] > h[y]$ :  
         $\pi[y] = x$   
    elif  $h[y] > h[x]$ :  
         $\pi[x] = y$   
    else:  
         $\pi[x] = y$   
         $h[y] = h[y] + 1$ 
```

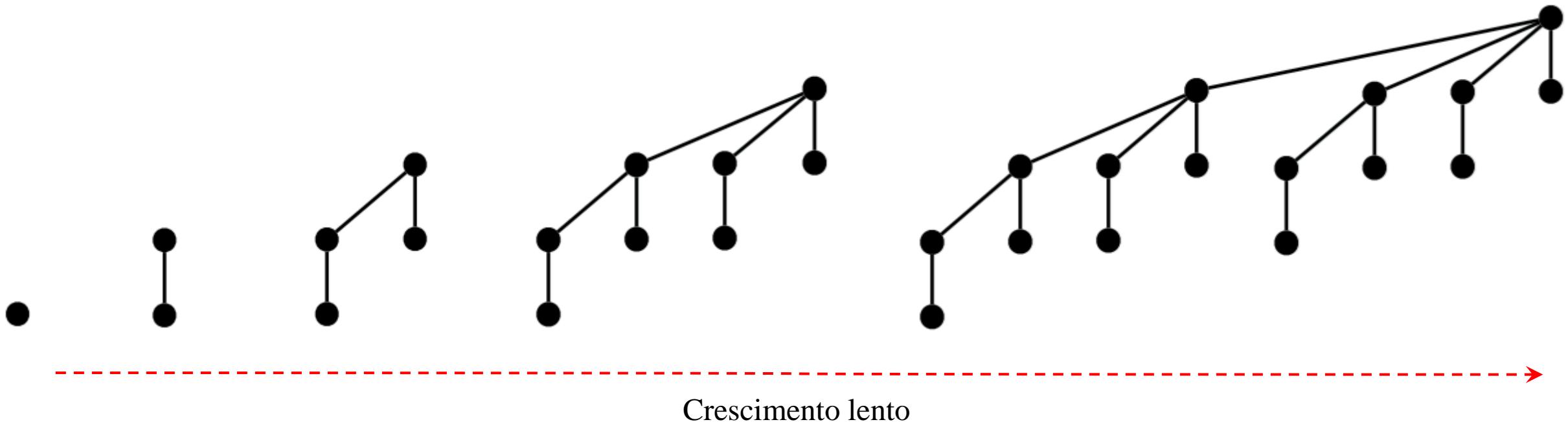
Só aumento um nível se ambos tiverem a mesma altura.

- Para melhorar a operação $\text{Find}(u)$ a ideia é aprimorar a operação $\text{Union}(u, v)$ seguindo duas estratégias
 - Unir u à raiz de v , ou seja, unir u ao $\text{Find}(v)$
 - Unir a raiz da árvore menor à raiz da árvore maior (selecionando empates arbitrariamente)

```
def MakeSet( $\mathcal{G}$ ):  
    for  $u$  in  $\mathcal{V}$ :  
         $\pi[u] = u$   
         $h[u] = 1$ 
```

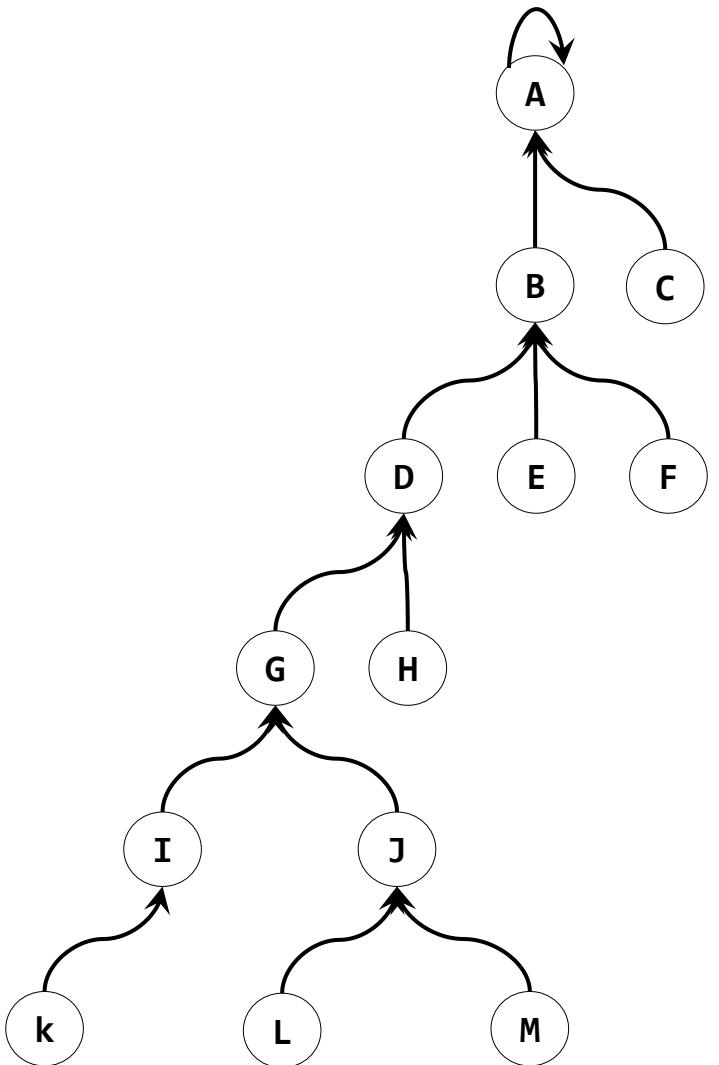
Implementação denominada *Link By Rank*

```
def Union( $u, v$ ):  
     $x = \text{Find}(u)$   
     $y = \text{Find}(v)$   
    if  $x = y$ :  
        return NULL  
    if  $h[x] > h[y]$ :  
         $\pi[y] = x$   
    elif  $h[y] > h[x]$ :  
         $\pi[x] = y$   
    else:  
         $\pi[x] = y$   
         $h[y] = h[y] + 1$ 
```



- Também podemos melhor a operação $\text{Find}(u)$

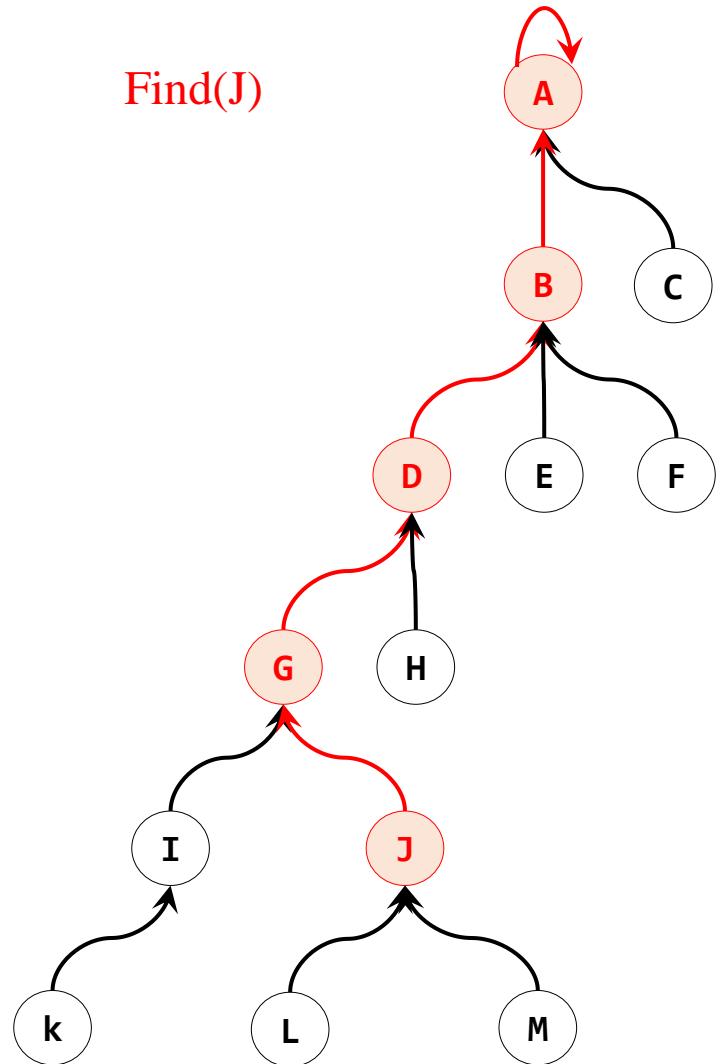
```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```



- Também podemos melhorar a operação $\text{Find}(u)$

```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```

Find(J)

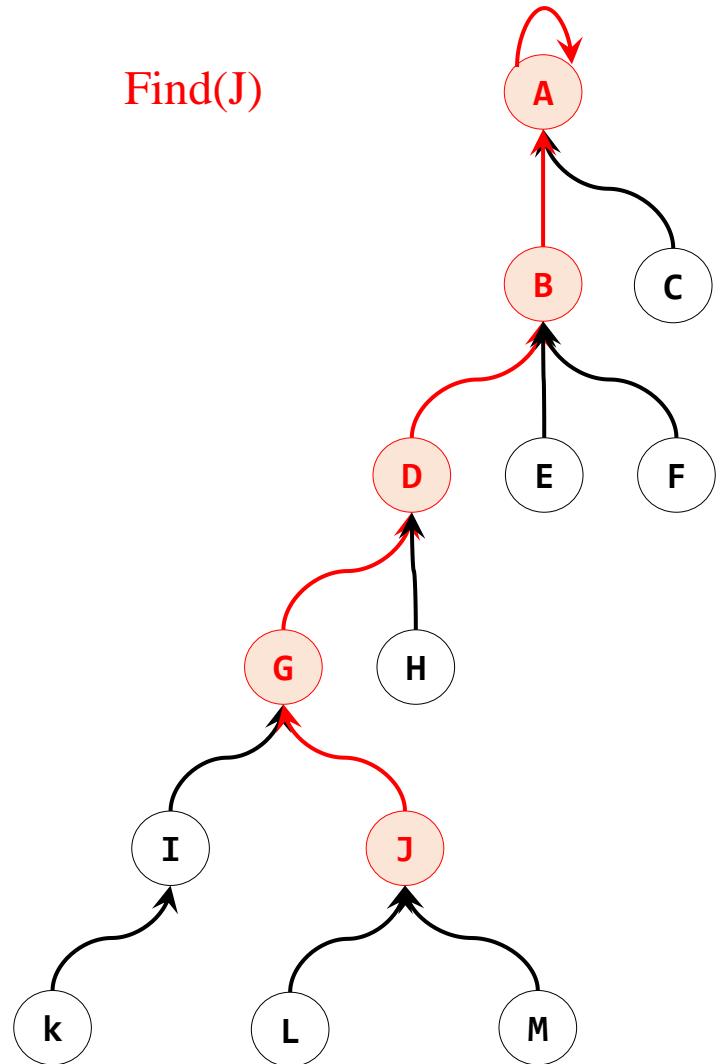


- Também podemos melhor a operação $\text{Find}(u)$

```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```

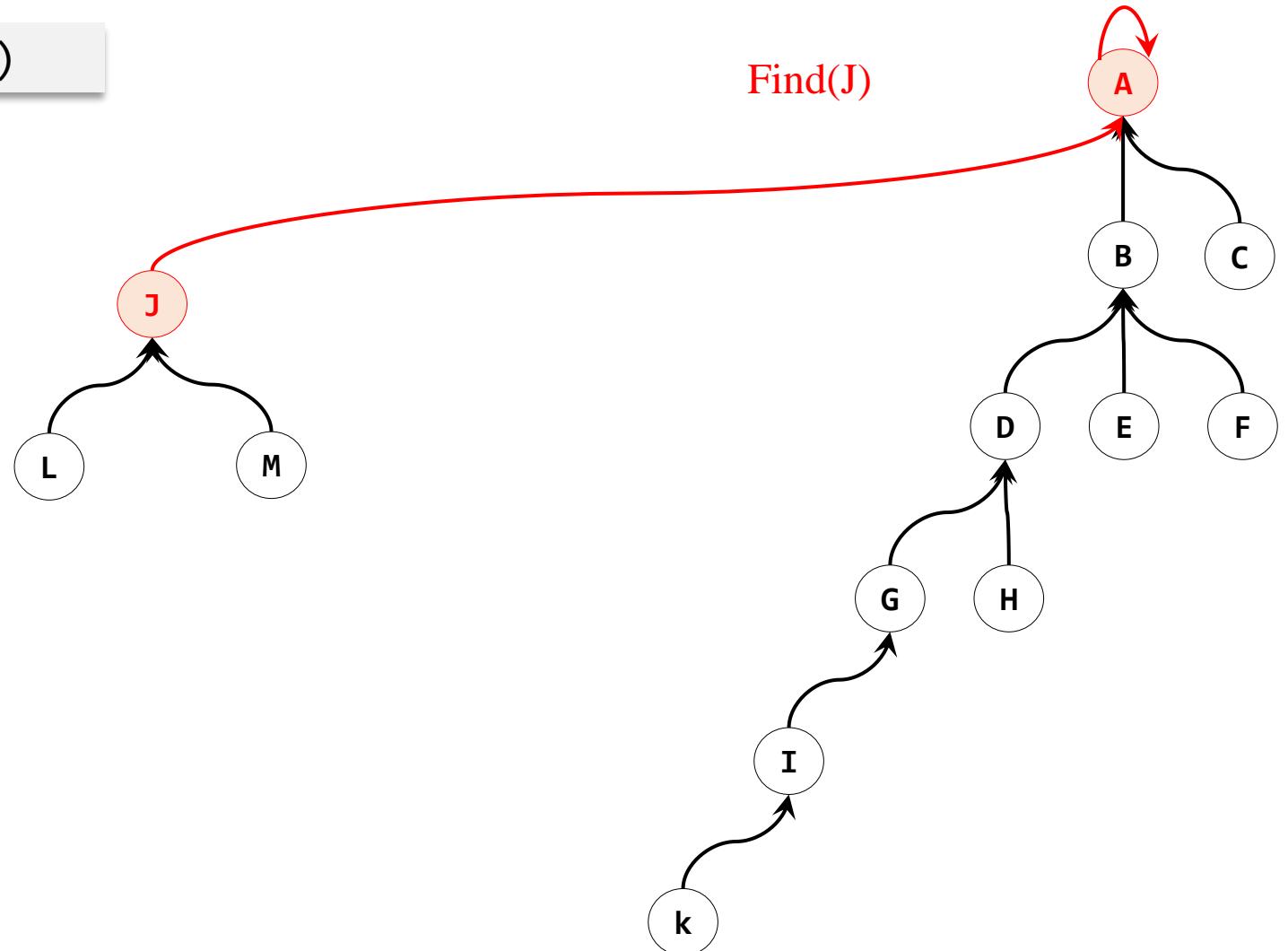
Qual a situação ideal?

$\text{Find}(J)$



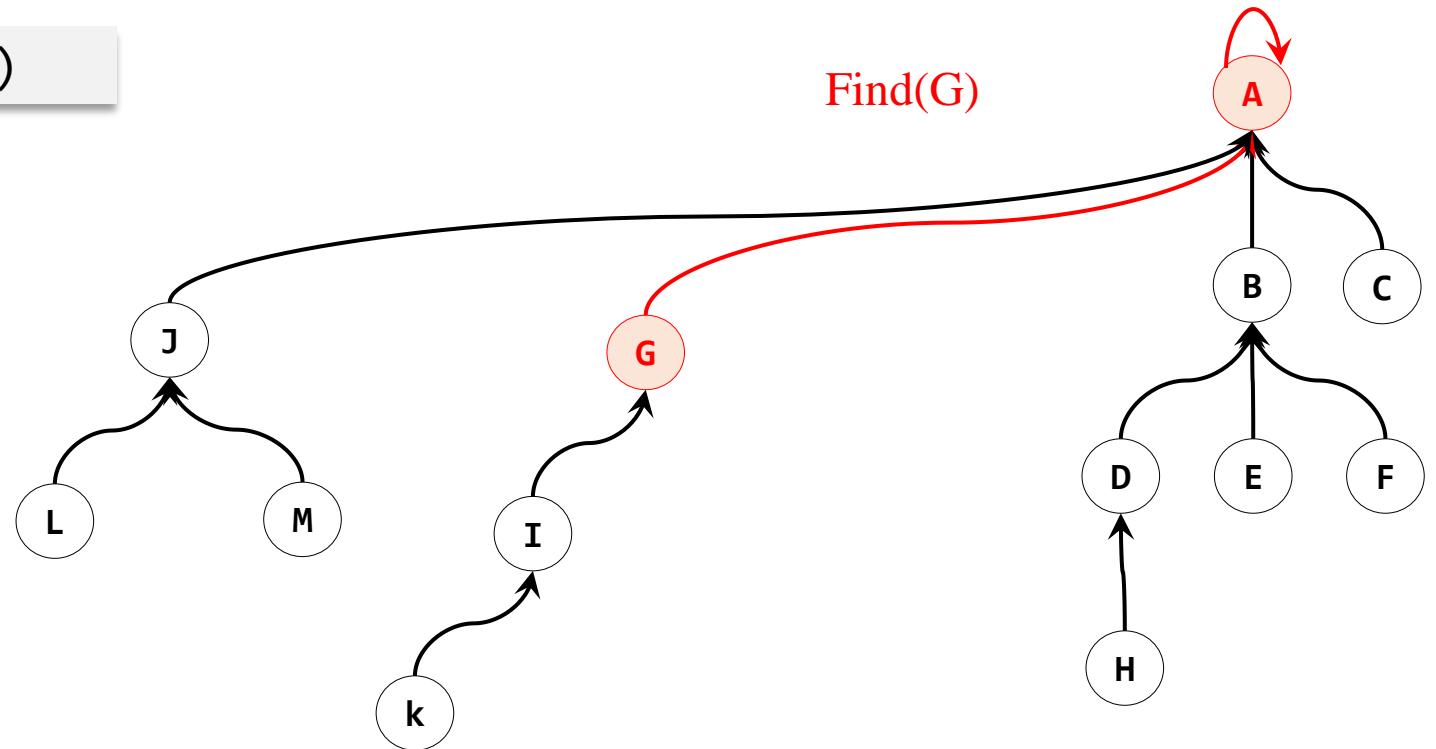
- Também podemos melhor a operação $\text{Find}(u)$

```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```



- Também podemos melhorar a operação $\text{Find}(u)$

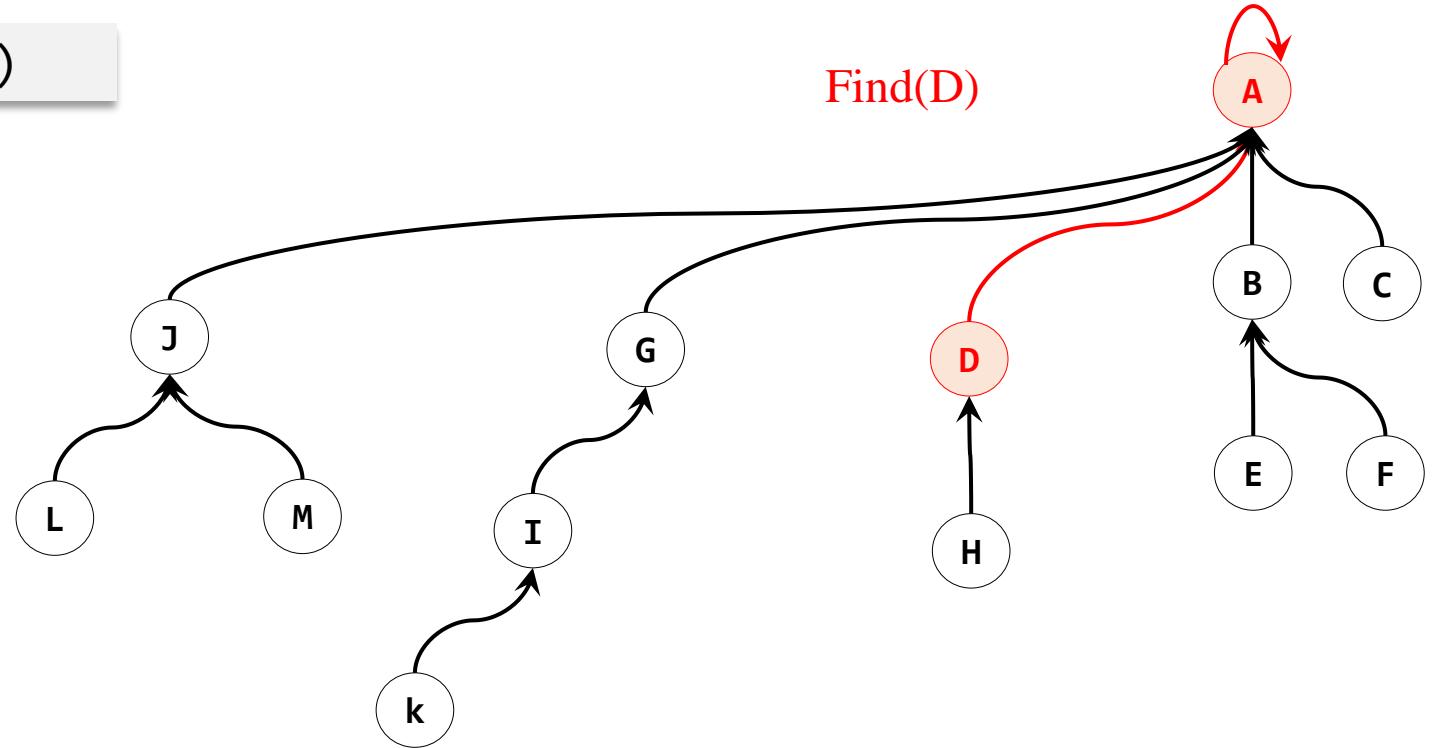
```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```



- Também podemos melhor a operação $\text{Find}(u)$

```
def Find(u):  
    while π[u] != u:  
        u = π[u]  
    return(u)
```

Find(D)



- Também podemos melhor a operação $\text{Find}(u)$
 - Implementação da operação $\text{Find}(u)$ com compressão de caminho ou *Path compression*

```
def Find(u):
    if π[u] != u:
        π[u] = Find(π[u])
    return(π[u])
```

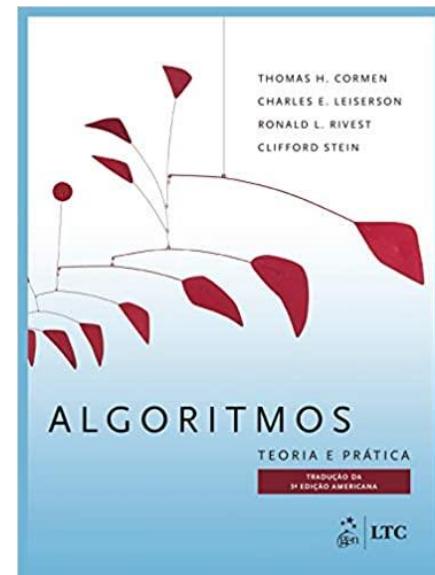
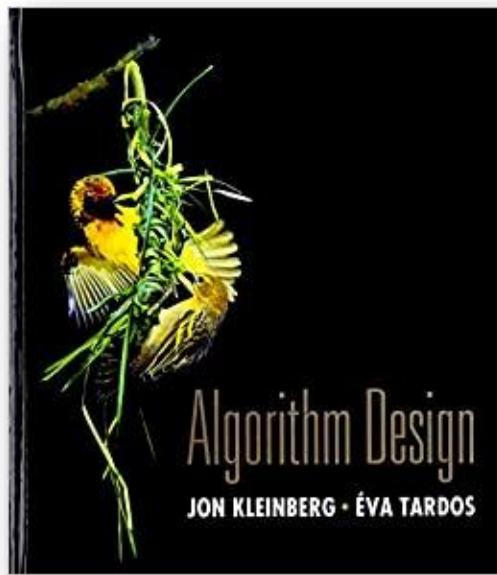
- Também podemos melhor a operação $\text{Find}(u)$
 - Implementação da operação $\text{Find}(u)$ com compressão de caminho ou *Path compression*

```
def Find(u):
    if π[u] != u:
        π[u] = Find(π[u])
    return(π[u])
```

Essa operação muda a estrutura da árvore

E a complexidade?

- O consumo de tempo amortizado dessa implementação do union-find é de $O(\log^*_2 n)$ unidade de tempo por operação. (Como sempre, n é o número de vértices do grafo.) A função $\log^*_2 n$ cresce muito devagar com n . Portanto, do ponto de vista prático, o desempenho desta implementação é essencialmente igual a $O(1)$.
- A prova da cota $O(\log^*_2 n)$ é deveras complexa.



https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/union-find.html

Kruskal(\mathcal{G})

$\mathcal{H} = \emptyset$

MakeSet(\mathcal{G})

Ordene as arestas de \mathcal{G} em ordem não-decrescente de peso

Para cada aresta $(u, v) \in \mathcal{E}$ Faça

 Se $\text{Find}(u) \neq \text{Find}(v)$ Então
 $\text{Union}(u, v)$

Kruskal(\mathcal{G})

$\mathcal{H} = \emptyset$

MakeSet(\mathcal{G}) } n

Ordene as arestas de \mathcal{G} em ordem não-decrescente de peso } $m \log_2 n$

Para cada aresta $(u, v) \in \mathcal{E}$ Faça

 Se $\text{Find}(u) \neq \text{Find}(v)$ Então
 Union(u, v)

} Entro no laço m vezes

Kruskal(\mathcal{G})

$\mathcal{H} = \emptyset$

MakeSet(\mathcal{G}) } n

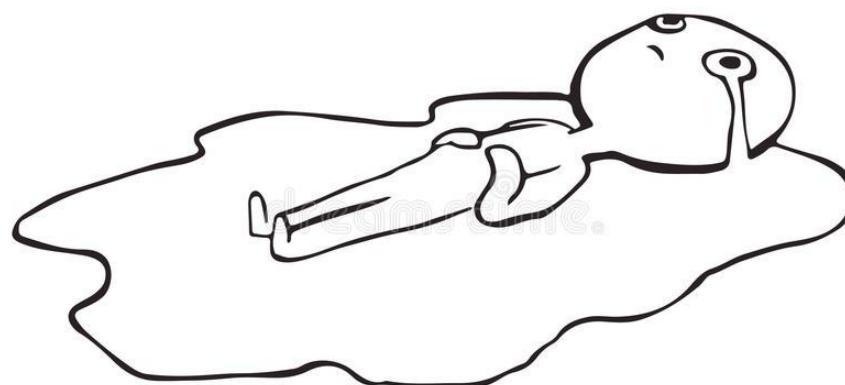
Ordene as arestas de \mathcal{G} em ordem não-decrescente de peso } $m \log_2 n$

Para cada aresta $(u, v) \in \mathcal{E}$ Faça

 Se $\text{Find}(u) \neq \text{Find}(v)$ Então
 Union(u, v)

} Entro no laço m vezes

Grafos Acabou



- k-Clusterização com Espalhamento Máximo (Material extra no AVA)

- Programação dinâmica

Obrigado



Dúvidas

Email: alanvalejo@ufscar.br

Acessar o fórum no Moodle