SIN110 Algoritmos e Grafos

aula 13

Grafos

- Aplicações Buscas em Largura e Profundidade(E09)
- Árvores Geradoras Mínimas

Aplicações

Busca em largura (BFS)

e

Busca em profundidade(DFS)

Aplicação busca em largura

4. Busca em Largura

Caminhos mínimos

Dados um vértice x de um grafo, encontrar um caminho de comprimento mínimo de " x " a cada um dos demais vértices.

Como todos esses caminhos podem ser representados?

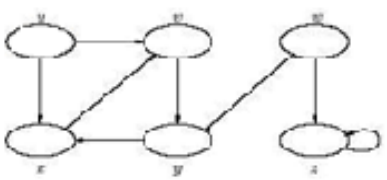
→ *arborescência de caminhos mínimos*

O procedimento imprime os vértices em um caminho mínimo, desde x até v, supondo que BFS já foi executado para calcular a arborescência de caminhos mínimos:

```
Arborescência-BFS(G,x, v)
1.  se v = x
2.       então imprime x
3.       senão se pred(v) = NIL
4.                 então escreve " não há caminho "
5.                 senão Arborescência-BFS(G, x, pred(v))
6.                       escreve "v → "
```

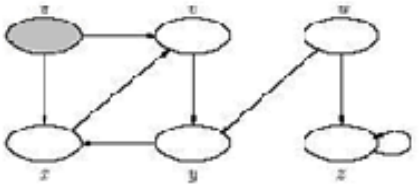
4. Busca em Largura

Exemplo: revendo o andamento do algoritmo BFS em um dígrafo começando a busca por "u"



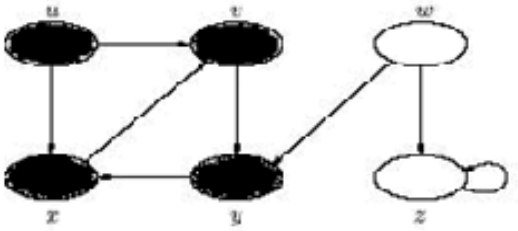
Dígrafo

⇒



(i) Fila: u

⇒



(viii) Fila: ∅

Configuração final dos vetores:

	[u]	[v]	[w]	[x]	[y]	[z]
pred:	-	u	-	u	x	-
dist:	0	1	∞	1	2	∞
	[u]	[v]	[w]	[x]	[y]	[z]

Arborescência para o caminho u – y:

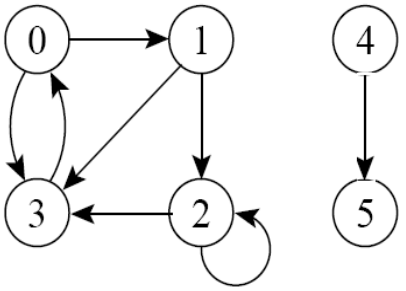
Que é impressa: u → x → y

Busca em profundidade
caminhos e componentes

Aplicando o algoritmo de pesquisa de caminhos

Caminho entre Vértices

Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.




```
Caminho(G, s,t)
1  para cada vétice u em G faça
2     cor[u] ← BRANCO
3  devolva VisitaR(G, s, t).

VisitaR (G, u, t)
1  se u = t
2     então devolve 1
3  cor[u] ← CINZA
4  para cada v em Adj(u) faça
5        se G cor[v] = BRANCO
6           então se VisitaR(G,v,t) = 1
7                    então devolva 1
8  devolva 0
```

Verif caminho 3 → 2:

Caminho(G,3,2)

VisitaR(G,3,2)	cor[3]=Cz
VisitaR(G,0,2)	cor[0]=Cz
VisitaR(G,1,2)	cor[1]=Cz
VisitaR(G,2,2)	cor[2]=Cz

Devolve “1”: existe um caminho simples

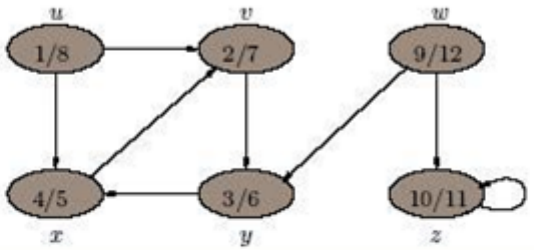
Podemos empregar a busca em profundidade para determinar as componentes de um (dí)grafo, basta uma modificação no algoritmo DFS para contar as componentes:

```
Componentes-DFS(G)
1.  para cada vértice u em G faça
2.        comp[u] = 0
3.  cont ← 0
4.  para cada vértice u em G faça
5.        se comp[u] = 0
6.             então  cont ← cont + 1
7.                    Visita-Componentes-DFS(G, u, cont)
8.  devolve cont

Visita-Componentes-DFS(G, u, cont)
1.  comp[u] ← cont
2.  para cada v em Adj(u) faça
3.        se comp[v] = 0
4.             então  Visita-Componentes-DFS(G, u, cont)
```

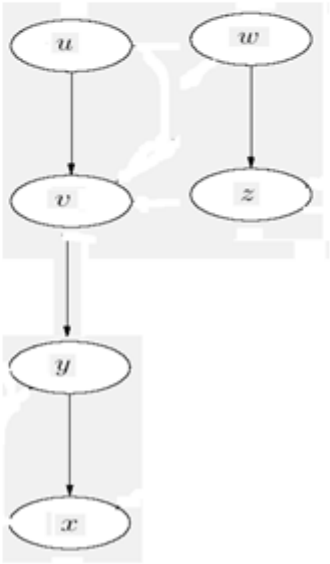
Exemplos: verificando as componentes nos exemplos anteriore

DFS - total



comp:	1	1	2	1	1	2
	[u]	[v]	[w]	[x]	[y]	[z]

Arborescências:



Verificamos 2 componentes (cont = 2)

Ciclos, pontes, articulações

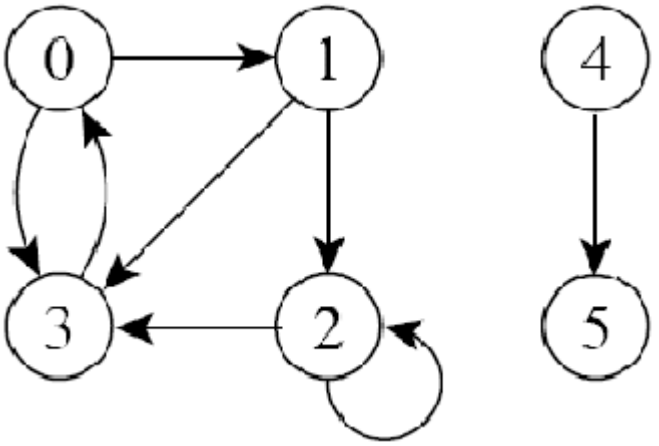
Exemplo: aplicando a pesquisa ao dígrafo:

Pesquisa_Ciclos-dígrafos(G)

```
1. para cada vértice u em G faça
2.       para cada v em Adj(u) faça
3.             ciclo ← Caminho(G, v, u)
4.             se ciclo = 1
5.                   então devolve 1
6. devolve 0
```

```
Caminho(G, s,t)
1  para cada vétice u em G faça
2     cor[u] ← BRANCO
3  devolva VisitaR(G, s, t).

VisitaR (G, u, t)
1  se u = t
2     então devolve 1
3  cor[u] ← CINZA
4  para cada v em Adj(u) faça
5        se G cor[v] = BRANCO
6           então se VisitaR(G,v,t) = 1
7                 então devolva 1
8  devolva 0
```

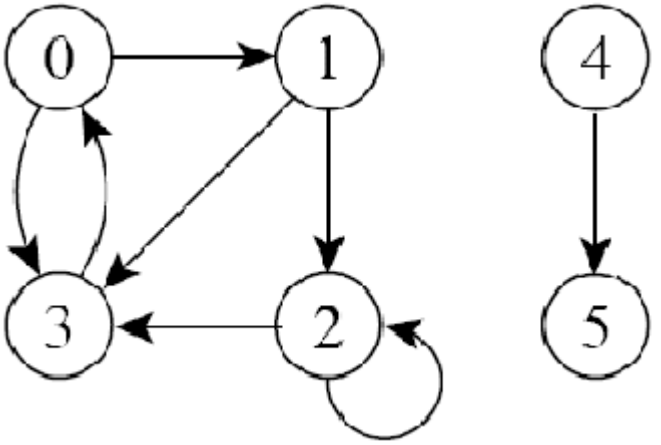


Pesquisa_Ciclos-dígrafos(G)
u ← 0
ciclo ← Caminho(G,1,0)
cor[0,1,2,3,4,5]=Cz
VisitaR(G,1,0)
cor[1]=Cz
VisitaR(G,2,0)
cor[2]=Cz
VisitaR(G,3,0)
cor[3]=Cz
VisitaR(G,0,0)
(devolve 1)
ciclo = 1 aponta “tem ciclo = 1)

Exemplo: aplicando a pesquisa ao dígrafo:

```
Busca_Ciclos-dígrafos(G)
1.  para cada vértice u em G faça
2.        ord(u) ← 0
3.  tempo ← 0
4.  para cada vértice u em G faça
5.        se ord(u) = 0
6.           então  se Visita-d(G,u) = 1
7.                        então devolve 1
8.  devolve 0.

Visita-d(G,u)
1.  tempo ← tempo + 1
2.  ord(u)  ← tempo
3.  para cada v em Adj(u) faça
4.        se ord[v] = 0
5.           então se Visita-d(G,v) = 1
6.                        então devolve 1
7.           senão se ord(v) < ord(u)
8.                        então devolve 1
9.  devolve 0.
```



Busca_Ciclos-dígrafos(G)
ord[0, 1, 2, 3, 4, 5] ← 0 tempo = 0
u ← 0, Visita-d(G,0)

ord[0]=1
v←1 *Visita-d(G,1)*

ord[1]=2
v←2 Visita-d(G,2)

ord[2]=3
v←3 Visita-d(G,3)

ord[3]=4
v←0 ord[0]<ord[3]

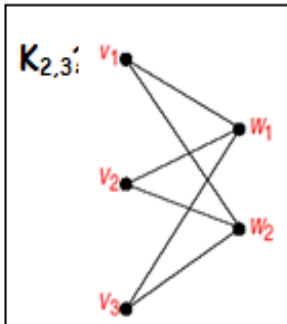
devolve 1

Visita-d(G,0)=1 aponta que "tem ciclo"

Exemplo: aplicando a pesquisa bipartição

```
Pesq_Bipartição(G)
 1.  para cada vértice u em G faça
 2.      cor(u) ← -1
 3.  ref ← 0
 4.  para cada vértice u em G faça
 5.      se cor(u) = -1
 6.          então  se VisitaB-g(G, u, 0) = 0
 7.                    então devolve 0
 8.  devolve 1.
```

```
VisitaB-g(G, u, ref)
 1.  cor(u)  ← 1 - ref
 2.  para cada v em Adj(u) faça
 3.      se cor[v] = -1
 4.          então se VisitaB-d(G, v, 1-ref) = 0
 5.                    então devolve 0
 6.          senão se cor(v) = (1 – ref)
 7.                    então devolve 0
 8.  devolve 1.
```



	V_1	V_2	V_3	W_1	W_2
V_1	-	-	-	1	1
V_2	-	-	-	1	1
V_3	-	-	-	1	1
W_1	1	1	1	-	-
W_2	1	1	1	-	-

Pesq_Bipartição(G)
cor[V1, V2, V3, W1, W3] ← -1 ref ← 0
u ← V1, VisitaB-g(G,V1, 0) cor[V1] ←1-0
VisitaB-g(G,W1,1-0) cor[W1] ←1-1
VisitaB-g(G,V2,1-1) cor[V2] ←1-0
VisitaB-g(G,W2,1-0) cor[W2] ←1-1
VisitaB-g(G,V3,1-1) cor[V3] ← 1-0

Pesq_Bipartição(G) = 1 aponta a “bipartição”

Algoritmo para pesquisar pontes

```
Busca_Pontes(G)
 1.  para cada vértice u em G faça
 2.    ord(u) ← 0
 3.  tempo ← conta ← 0
 4.  para cada vértice u em G faça
 5.        se ord(u) = 0
 6.           então  pred(u) ← u
 7.                  tempo ← tempo + 1
 8.                  VisitaP-g(G, u, tempo, conta).

VisitaP-g(G,u, tempo, conta)
 1.  pré-ord ← ord(u)  ← tempo
 2.  para cada v em Adj(u) faça
 3.        se ord[v] = 0
 4.           então pred(v) ← u
 5.                 tempo ← tempo + 1
 6.                 VisitaP-g(G, v, tempo, conta-ponte)
 7.                 se ord(u) > ord(v)
 8.                    então ord(u) ← ord(v)
 9.                 se pré-ord(v) = ord(v)
10.                    então conta ← conta + 1
11.                          escreve: "ponte", conta, u"–"v
12.           senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
13.                    então pré-ord(u) ← ord(v)
```

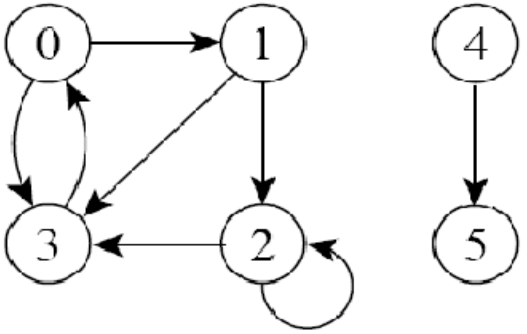
teste da linha 12, " v ≠ pred(u)", garante que *u-v* é um arco de retorno (e não um arco - pai).

Exemplo: aplicando a pesquisa de pontes ao dígrafo:

```
Busca_Pontes(G)
1.  para cada vértice u em G faça
2.     ord(u) ← 0
3.  tempo ← conta ← 0
4.  para cada vértice u em G faça
5.        se ord(u) = 0
6.           então  pred(u) ← u
7.                  tempo ← tempo + 1
8.                  VisitaP-g(G, u, tempo, conta).

VisitaP-g(G,u, tempo, conta)
1.  pré-ord ← ord(u)  ← tempo
2.  para cada v em Adj(u) faça
3.        se ord[v] = 0
4.           então pred(v) ← u
5.                 tempo ← tempo + 1
6.                 VisitaP-g(G, v, tempo, conta-ponte)
7.                 se ord(u) > ord(v)
8.                    então ord(u) ← ord(v)
9.                 se pré-ord(v) = ord(v)
10.                   então conta ← conta + 1
11.                         escreve: "ponte", conta, u"–"v
12.          senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
13.                 então pré-ord(u) ← ord(v)
```

Adj(u)
0 → 1 → 3
1 → 2 → 3
2 → 2 → 3
3 → 0
4 → 5
5

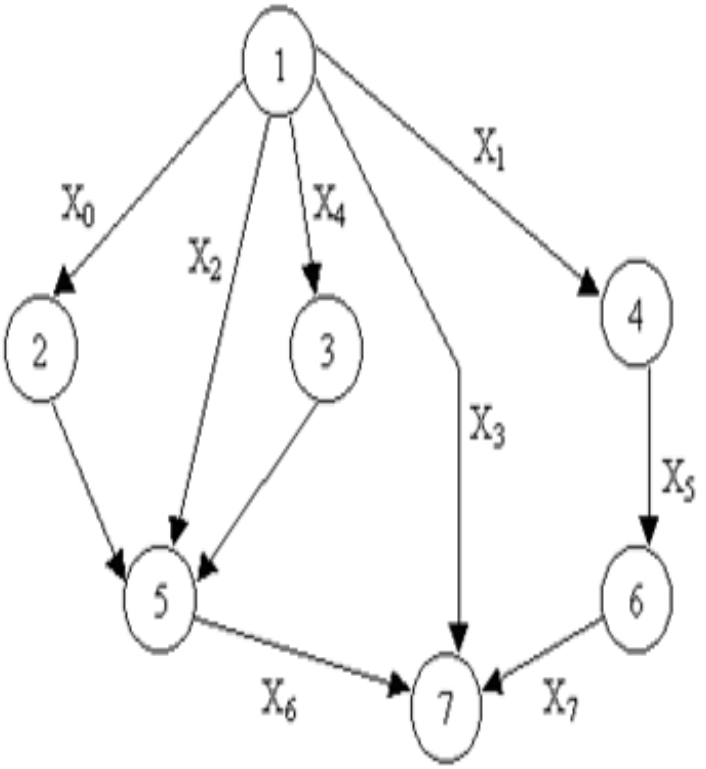


Dígrafo tem 2 componentes …
… e pontes nos arcos 0 -1, 1 -2 e 4-5

Busca_Pontes(G)
ord[0, 1, 2, 3, 4, 5] ← 0 tempo = conta ← 0
u ← 0, tempo = 1 pred[0]=0 VisitaP-g(G,0,1,0) pre-ord[0] = ord[0]=1
v←1 pred[1]=0 tempo=2 VisitaP-g(G,1,2,0) pre-ord[1] = ord[1]=2
v←2 pred[2]=1 tempo=3 VisitaP-g(G,2,3,0) pre-ord[2] = ord[2]=3
v←2 2≠pred[2] E pred-ord[2] = ord[2] … nenhuma ação
v←3 pred[3]=2 tempo=4 VisitaP-g(G,3,4,0) pre-ord[3] = ord[3]=4
v←0 ..ord[0]≠0 … 0≠pred[3] E pre-ord[3] > ord[0] então: pre-ord[3] ← ord[0] = 1
ord[2] < ord[3] não altera … pre-ord[3] < ord[3] não altera.
ord[1]<ord[2] não altera … pre-ord[2]=ord[2] … então ESCREVE **"ponte1 1-2"**
ord[0]< ord[1] não altera … pred-ord[0]=ord[0] … então ESCREVE **"ponte2 0-1"**
u ← 4, tempo = 5 pred[4]=4 **VisitaP-g(G, 4, 5, 0)** pre-ord[4] = ord[4]=5
v←5 pred[5]=4 tempo=6 VisitaP-g(G,5,6,0) pre-ord[5] = ord[5] = 6
ord[4]<ord[5] não altera … pre-ord[4]=ord[4] … então ESCREVE **"ponte3 4-5"**

ORDENAÇÃO TOPOLÓGICA

Um DAG (direct acyclic graph) é um dígrafo acíclico, ou seja, uma travessia em profundidade no dígrafo G não indica nenhum arco para trás. Exemplo de um DAG:

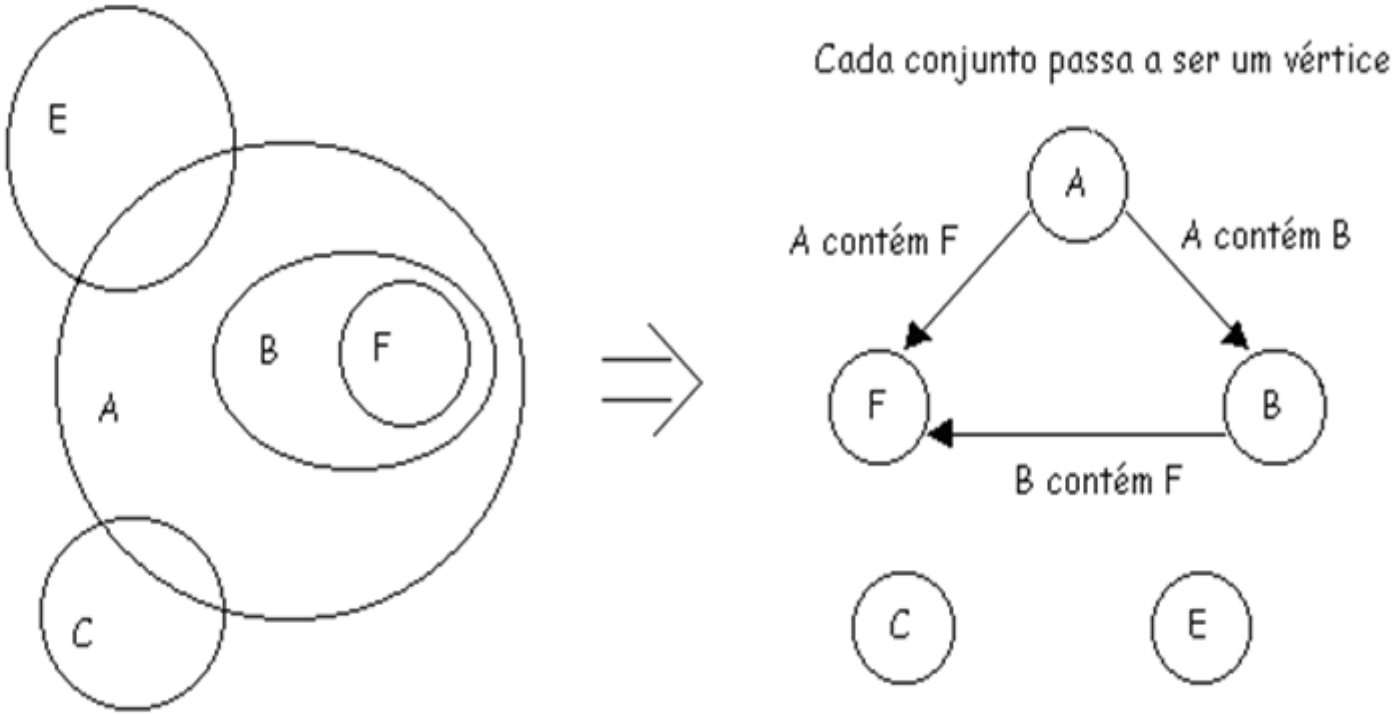


Dígrafo acíclico G

DAGs podem representar: operações de ordem parcial, redes de atividades, compiladores, pré-requisitos, jogos, etc.

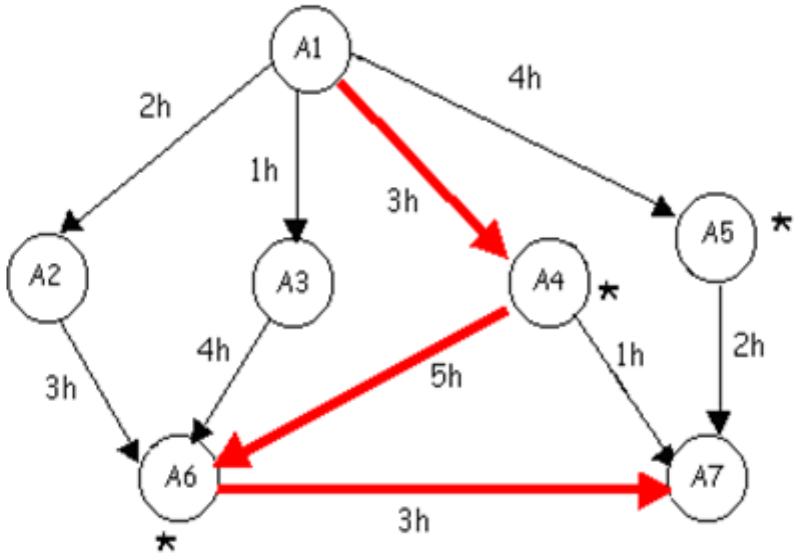
Vejamos alguns exemplos:

- *Operações de Ordem parcial:* exemplo de inclusão:



- *Redes de Atividades*:

- Exemplo: temos uma rede de atividades num processo industrial que pode ser representada na seguinte forma:



Como podemos observar, o peso máximo para chegar à atividade A7 é 11 (caminho a vermelho), ou seja, é a soma dos pesos desde a fonte até à folha, ao qual chamamos de *caminho crítico*.

A ordenação topológica consiste em ordenar os vértices de um grafo acíclico G=(V,E) de forma que, se existe um caminho do vértice v para o vértice u, então v aparece antes de u na ordenação. De notar que uma ordenação topológica não é única e que ela não é possível se o grafo possuir ciclos.

Um exemplo comum de problemas de ordenação topológica, é a confecção de dicionários, onde desejamos que uma palavra B cuja definição dependa da palavra A, apareça depois de A no dicionário.

Uma forma simples de resolver esse problema é a utilização de uma versão do algoritmo de busca em profundidade que imprime o vértice antes de retornar a chamada. Obteremos assim os vértices em ordem topológica invertida:

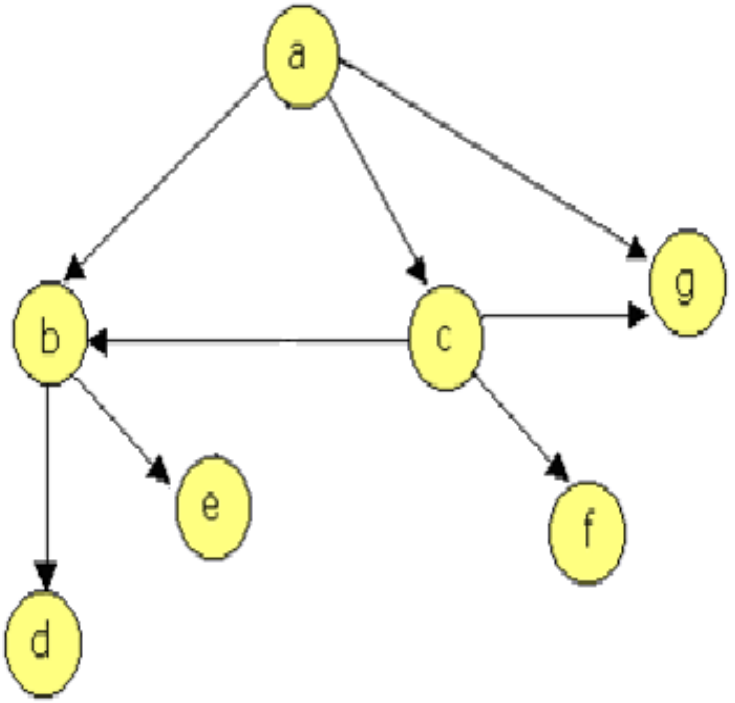
OrdTopo(G)

1. para cada vértice u em G faça
2. cor[u] ← BRANCO
3. topo(u) ← -1
4. k ← |V| (nº vértices de G)
5. para cada vértice u em G faça
6. se cor(u) = BRANCO
7. então Visita(u, topo, k)
8. devolve topo[1 .. |V|]

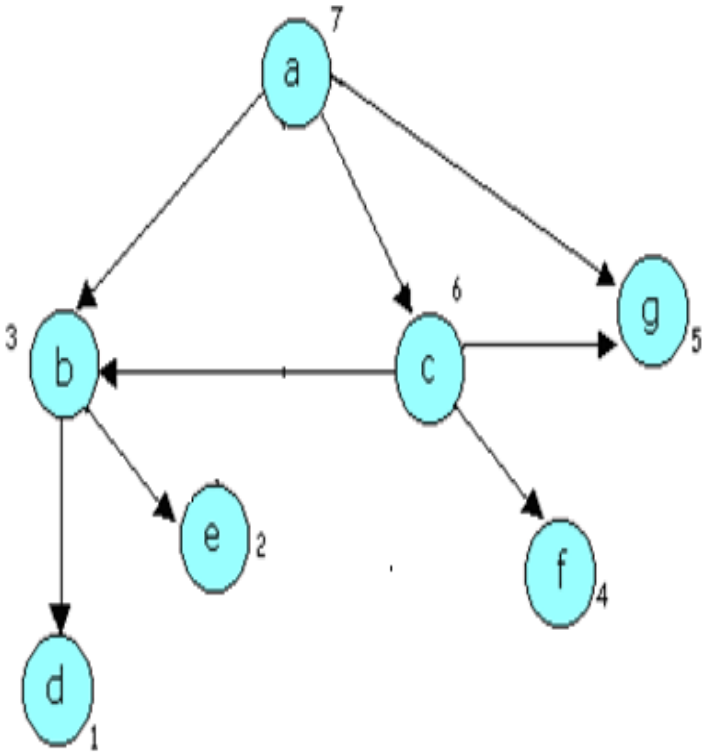
Visita(u, topo, k)

1. cor[u] ← PRETO
2. para cada v em Adj(u) faça
3. se cor[v] = BRANCO
4. então Visita(v)
5. k ← k – 1
6. topo(k) ← u
7. imprime u

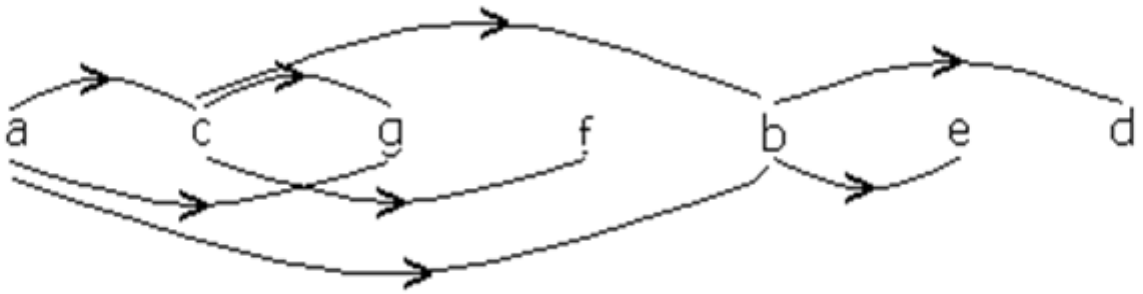
Vejamos um exemplo de ordenação topológica com seguinte DAG:



A execução do algoritmo de busca em profundidade nos dará a ordem de visita de cada vértice, considere que a lista de adjacências deste DAG resulte em:



O objetivo é re - escrever o DAG de forma linear, executando o algoritmo OrdTopo vamos obter os arcos todos na mesma direção:



E o resultado fica registrado no vetor "topo" que apresenta no exemplo o vértice "a" na 1ª posição e o vértice "d" na última.

Há que ter em atenção que a ordenação topológica não é única.

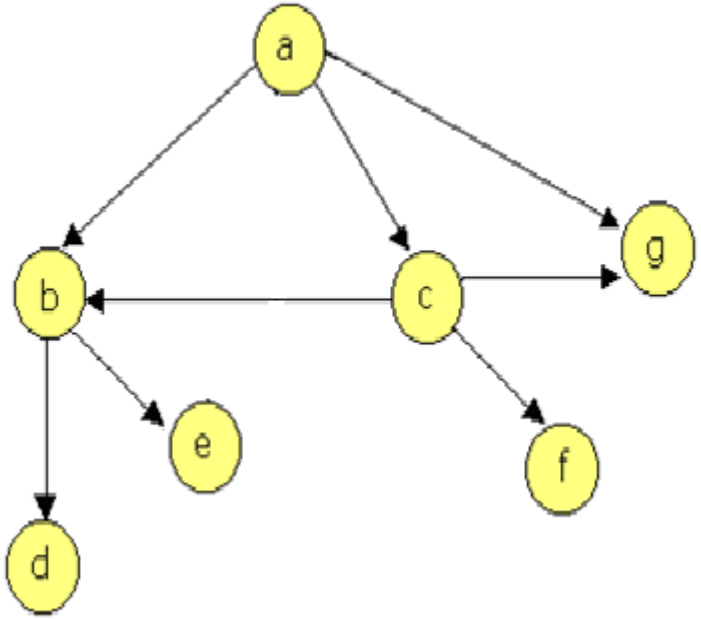
Existe mais de uma forma de escrever um dígrafo por ordem topológica.

No entanto, para realizarmos o algoritmo vamos ter que usar o DFS, sendo o resultado armazenado numa lista.

Outra alternativa para obter uma ordenação topológica é o algoritmo que trabalha com a informação do grau de entrada em cada vértice, denominado algoritmo da eliminação das fontes:

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.  grau-entrada[u] ← 0
 3.  para cada vértice u em G faça
 4.        para cada v em Adj(u) faça     (mapeia o grau de entrada de u)
 5.              grau-entrada[v] ← grau-entrada[v] + 1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (somente fontes são inseridas na fila)
 8.       se grau-entrada = 0
 9.          então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ∅ faça
12.        u ← TiraFila(Q)
13.        topo[i] ← u
14.        i ← i + 1
15.        para cada v em Adj(u) faça
16.              grau-entrada[v] ← grau-entrada[v] - 1
17.              se grau-entrada[v] = 0
18.                 então  InsereFila(Q,v)
19.   devolve topo[1 .. n]
```

Simulando com o DAG:



i) grau de entrada [linhas 1 - 10]

vert: a b c d e f g

grau: 0 2 1 1 1 1 2

Fila: a

ii) Enquanto … [linhas 11 - 19]

u ← a vert: **a** b c d e f g

grau: 0 1 0 1 1 1 1 Fila: c

u ← c vert: **a** b **c** d e f g

grau: **0** 0 **0** 1 1 0 0 Fila: b, f, g

u ← b vert: **a b c** d e f g

grau: **0 0 0** 0 0 0 0 Fila: f, g, e, d

u ← f vert: **a b c** d e **f** g

grau: **0 0 0** 0 0 **0** 0 Fila: g, e, d

u ← g vert: **a b c** d e **f g**

grau: **0 0 0** 0 0 **0 0** Fila: e, d

u ← e vert: **a b c** d **e f g**

grau: **0 0 0** 0 **0 0 0** Fila: d

u ← d vert: **a b c d e f g**

grau: **0 0 0 0 0 0 0** Fila: ⊘

Ordenação:

a → c → b → f → g → e → d

4) Considere um *campeonato de futebol* com **n** times se enfrentando todos contra todos (com uma só partida entre cada par de times), onde nunca há empates (se necessário disputam-se pênaltis). Queremos saber se esse campeonato é consistente, isto é: caso um time **i** tenha perdido para um time **j**, nunca ocorrerá uma seqüência de times $t_i, t_1, t_2, ..., t_k, t_j$ tal que t_i venceu t_1, t_1 venceu t_2, ... e, t_k venceu t_j. Modele o problema de verificar se "o campeonato é consistente", como um problema em grafos e proponha um algoritmo de tempo linear no tamanho do grafo para resolvê-lo.

Modelagem: dígrafo acíclico (arcos do 1º apontam para todos, arcos do 2º apontam para todos menos o 1º, etc.)

Consistência: DFS adaptada para pesquisar ciclos(ex 1c), se tiver um arco reverso o campeonato não é consistente!

Grafos:

- Aplicações Buscas em largura e profundidade

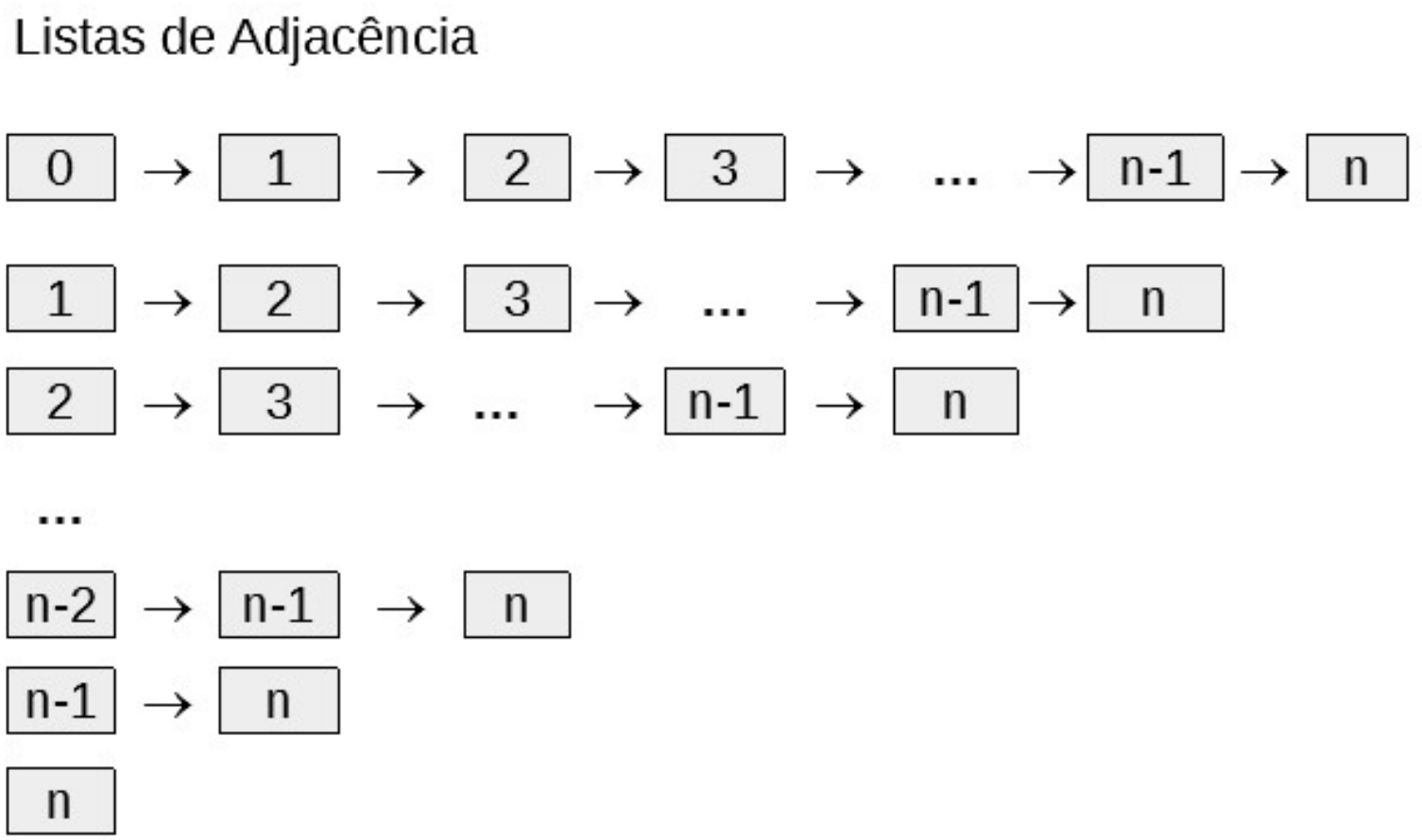
E09 - solução

Adaptando DFS para buscar ciclos em dígrafos … (mostra o primeiro ciclo encontrado….)

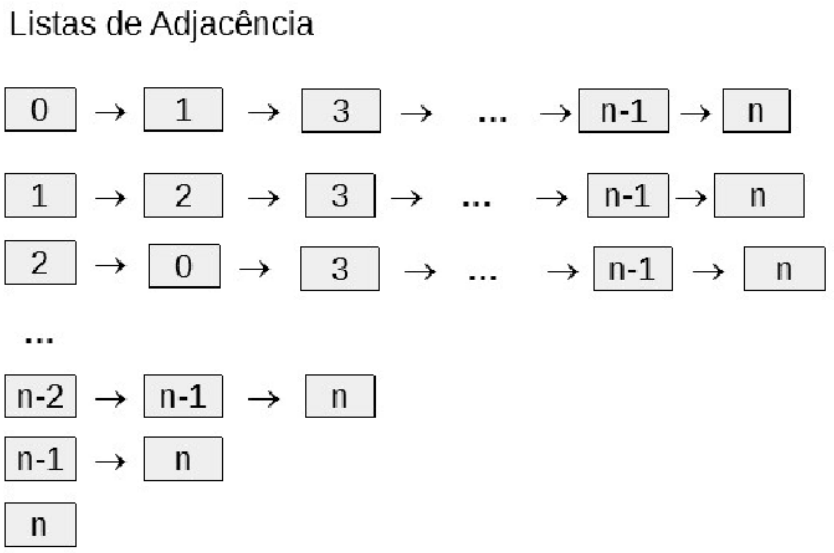
```
DFS(G)
 1.  para cada vértice u em G faça
 2.       cor[u] = BRANCO
 3.       pred[u] ← NIL
 4.  tempo ← 0
 5.  para cada vértice u em G faça
 6.          se cor(u) = BRANCO
 7.              então Visita_DFS(u)
 8.  devolve pred[1..n]
Visita_DFS(u)
 1.   cor[u] ← CINZA
 2.  tempo ← tempo + 1
 3.  d(u)  ← tempo
 4.  para cada v em Adj(u) faça
 5.          se cor[v] = BRANCO
 6.                  então pred[v] ← u
 7.                        Visita_DFS(v)
 7.  cor[u] ← PRETO
 8.  tempo ← tempo + 1
 9.  f(u)  ← tempo
```

```
Busca_Ciclos-dígrafos(G)
 1.  para cada vértice u em G faça
 2.       ord(u) ← 0
 3.  tempo ← 0
 4.  para cada vértice u em G faça
 5.       se ord(u) = 0
 6.          então  se Visita-d(G,u) = 1
 7.                      então devolve 1
 8.  devolve 0.
Visita-d(G,u)
 1.  tempo ← tempo + 1
 2.  ord(u)  ← tempo
 3.  para cada v em Adj(u) faça
 4.       se ord[v] = 0
 5.          então se Visita-d(G,v) = 1
 6.                      então devolve 1
 7.          senão se ord(v) < ord(u)
 8.                      então devolve 1
 9.  devolve 0.
```

Representação em listas de adjacências do dígrafo “campeonato”



Procurando por ciclos… supondo uma inconsistência



Busca_Ciclos-dígrafos(G)

```
1. para cada vértice u em G faça
2.       ord(u) ← 0
3. tempo ← 0
4. para cada vértice u em G faça
5.       se ord(u) = 0
6.          então  se Visita-d(G,u) = 1
7.                       então devolve 1
8. devolve 0.
```

Visita-d(G,u)

```
1. tempo ← tempo + 1
2. ord(u) ← tempo
3. para cada v em Adj(u) faça
4.       se ord[v] = 0
5.          então se Visita-d(G,v) = 1
6.                    então devolve 1
7.          senão se ord(v) < ord(u)
8.                    então devolve 1
9. devolve 0.
```

Busca (G)
ord[0,1,…,n] ← 0 tempo ← 0
u ← 0, Vis(G,0) ord[0] = 1
v ← 1 Vis(G,1) ord[1] = 2
v ← 2 Vis(G,2) ord[2] = 3
v ← 0 … ord[0] ≠ 0 … ord[0] < ord[2] … devolve 1…, ié, tem ciclo (um arco de retorno)

5) Denote por *dist(v,w)* a distância entre dois vértices *v* e *w* num grafo conexo. O diâmetro de um grafo conexo é a valor máximo da expressão *dist(v,w)* com *v* e *w* variando no conjunto de todos os vértices. Escreva uma função que calcule o diâmetro de qualquer grafo conexo.

Função: BFS + pesquisa maior valor dist(v,w)

Adaptando BFS ...

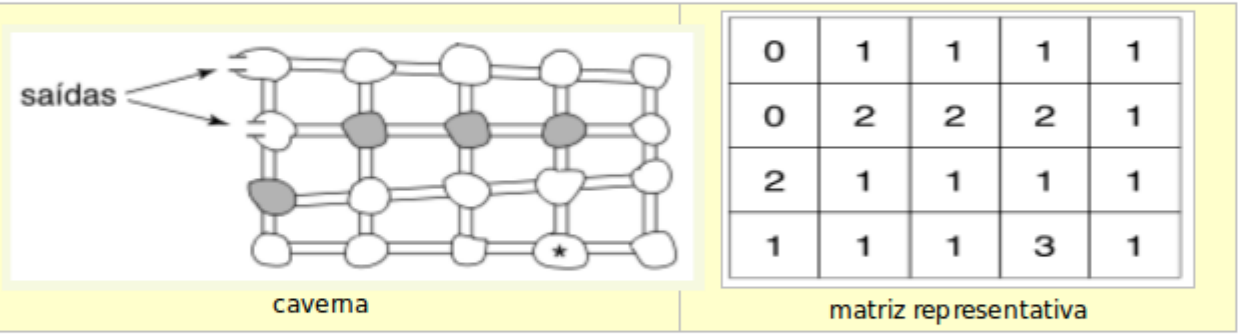
```
BFS(G,x)
 1.  para u ← 1 até n faça
 2.        cor[u] ← BRANCO, dist[u] □ □
 3.  cor[x] ← CINZA, dist[x] = 0
 4.  diâmetro ← dist[x]
 5.  Q ← Inicialiaza-Fila(Q,x)
 6.  enquanto Q≠∅faça
 7.        u ← Primeiro-da-Fila(Q)
 8.    para cada v em Adj[u] faça
 9.           se cor[v] = BRANCO
10.                 então cor[v] ← CINZA, dist[v] ← dist[u]+1
11.                       Insira-na-Fila(Q,v)
12.                 se diâmetro < dist[v]
13.                       então  diâmetro ← dist[v]
14.    Remova-da-Fila(Q)
15.    cor[u] ← PRETO
16.  devolve dist[1..n], diâmetro
```

3) [Adaptado de *spoj.com/problems problema 2608*] "O duende perdido".

Dud, o duende, ficou preso em uma caverna e precisa sair o mais rapidamente possível. A caverna é formada por salões interligados por túneis, na forma de uma grade retangular, com N linhas e M colunas. Alguns dos salões da caverna têm paredes de cristal.

Duendes, *como todos sabem*, não gostam de ficar em ambientes com qualquer tipo de cristal, pois seus organismos entram em ressonância com a estrutura de cristais, e em casos extremos os duendes podem até mesmo explodir!

Compreensivelmente, *Dud* não quer entrar em nenhum salão com parede de cristal. A figura abaixo mostra uma caverna com quatro linhas e cinco colunas de salões; os salões cinza têm paredes de cristal. A posição inicial de *Dud* é indicada com um caractere '*', conforme ilustra a figura:

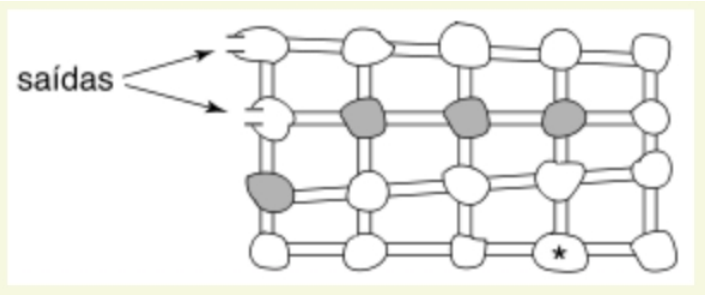


Empregando teoria de grafos, escreva um algoritmo em que, dadas a configuração da caverna e a posição inicial de *Dud* dentro da caverna, calcula qual o número mínimo de salões pelos quais o duende deve passar antes de sair da caverna (não contando o salão em que o duende está inicialmente), mas contando o salão que tem saída para o exterior). Na caverna da figura, o duende deverá percorrer no mínimo 8 salões até encontrar a saída.

A caverna será modelada como uma matriz de duas dimensões, vide figura acima, cujos elementos representam os salões. Um salão que não tem parede de cristal e que tem saída para o exterior da caverna é representado pelo valor 0; um salão que não tem parede de cristal e não tem saída para o exterior é representado pelo valor 1; um salão que tem parede de cristal é representado pelo valor 2; e o salão em que o duende está inicialmente (que não tem saída para o exterior e nem paredes de cristal) é representado pelo valor 3.

Além de projetar, mostre a correção de seu algoritmo e quanto tempo consome para processar a pesquisa.

3) “O duende perdido”.



0	1	1	1	1
0	2	2	2	1
2	1	1	1	1
1	1	1	3	1

Devemos modelar a caverna (representada na matriz) por um grafo e depois realizar uma BFS, a partir da posição inicial do duende (quando M(i,j) = 3).

Transformando a caverna em um grafo, partimos da matriz que mapeia a caverna, nomeando cada elemento da matriz (n linhas X m colunas), que representa um salão como um vértice.

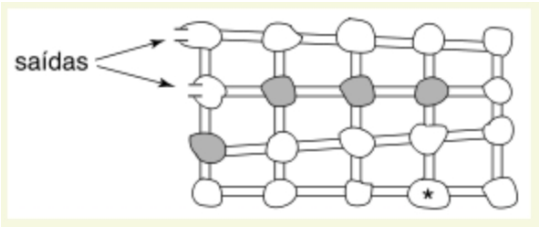
*Para viabilizar a montagem das listas de adjacências, numeramos os vértices considerando a posição do elemento M(i,j) usando a relação n^o vertice = (i-1)*m + j, com $0 \leq i < n$ e $1 \leq j \leq m$, assim M(1,1) = 1,…, M(1,m) = m, M(2,1) = 1 + m, …, M(n,m) = (n-1)*m+m.*

Para montar as listas de adjacências devemos pesquisar o conteúdo dos elementos M(i,j) que corresponderão a conexões se igual a 1, 0 (quando sai da caverna, nas bordas) ou 3 localização inicial do duende.

Se $M(i,j) \neq 2$ pesquisamos sua vizinhança, em M(i+1,j) e M(i,j+1) se também for temos uma conexão para anotar no grafo.

Quando M(i,j) = 0 devemos anotar que é um dos destinos (saída) e, também a posição de inicio da BFS quando M(i,j) = 3.

3) “O duende perdido”.



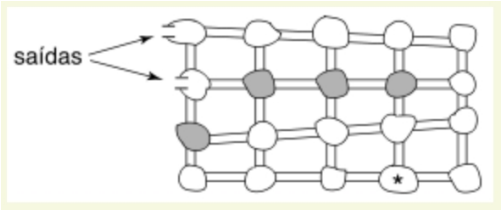
0	1	1	1	1
0	2	2	2	1
2	1	1	1	1
1	1	1	3	1

Com a função “Monta( )” obtemos o grafo e anotamos o inicio da BFS e saídas (vetor fim[..]):

Monta(*M[1..n, 1..m]* ***)***

```
1. para i ← 1 até n-1 faça
2.       para  j ← 1 até m-1 faça
3.              se M(i,j) ≠ 2
4.                  então  u ← (i-1)*m + j
5.                         se M(i+1, j) ≠ 2  então v ← (i+1-1)*m + j;  Adj(u) ← v;  Adj(v) ← u
6.                         se M(i, j+1) ≠ 2  então v ← (i-1)*m + (j+1);  Adj(u) ← v;  Adj(v) ← u
7.                         se M(i,j) = 3  então  ini ← u
8.      se M(i,m) ≠ 2 E M(i+1,m) ≠ 2
9.                  então u ← (i-1)*m + m; v ← (i+1-1)*m + m;  Adj(u) ← v;  Adj(v) ← u
10. para  j ← 1 até m-1 faça
11.      se M(n,j) ≠ 2 E M(n,j+1) ≠ 2 então u←(n-1)*m+j; u ← (n-1)*m+(j+1); Adj(u)←v;  Adj(v)←u
12. k ←1
13. para  i ← 1 até n faça
14.       se M(i,  1) = 0  então fim(k) ← (i-1)*m + 1; k ←k + 1
15.       se M(i, m) = 0  então fim(k) ← (i-1)*m + m; k ←k + 1
16. para  j ← 1 até m faça
17.       se M(1,  j) = 0  então fim(k) ← 1*m + j; k ←k + 1
18.       se M(n,  j) = 0  então fim(k) ← (n-1)*m + j; k ←k + 1
19. devolve G(V,E), fim[1..k-1], k-1
```

3) “O duende perdido”.



0	1	1	1	1
0	2	2	2	1
2	1	1	1	1
1	1	1	3	1

Aplicando a função “Monta( )” a matriz acima obtemos o grafo:

<u>Listas de Adjacências</u>:

1 → 2 → 6
2 → 1 → 3
3 → 2 → 4
4 → 3 → 5
5 → 4 → 10
6 → 1
7
8
9
10 → 5 → 15
11
12 → 13 → 17
13 → 12 → 14 → 18
14 → 13 → 15 → 19
15 → 14 → 20
16 → 17
17 → 12 → 16 → 18
18 → 13 → 17 → 19
19 → 14 → 18 → 20
20 → 15 → 19

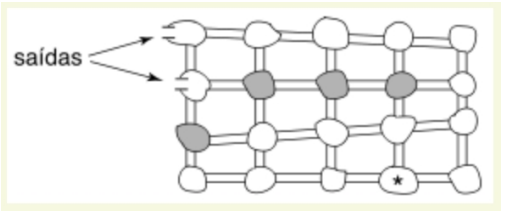
ini = 19

fim(1) = 1

fim(2) = 6

k-1 = 2

3) “O duende perdido”.



0	1	1	1	1
0	2	2	2	1
2	1	1	1	1
1	1	1	3	1

Listas de Adjacências:

1 → 2 → 6
2 → 1 → 3
3 → 2 → 4
4 → 3 → 5
5 → 4 → 10
6 → 1
7
8
9
10 → 5 → 15
11
12 → 13 → 17
13 → 12 → 14 → 18
14 → 13 → 15 → 19
15 → 10 → 14 → 20
16 → 17
17 → 12 → 16 → 18
18 → 13 → 17 → 19
19 → 14 → 18 → 20
20 → 15 → 19

BFS(G,x)

```
1.  para u ← 1 até n faça
2.        cor[u] ← BRANCO
3.        dist[u] ← ∞
4.        pred[u] ← NIL
5.  cor[x] ← CINZA
6.  dist[x] = 0
7.  Q ← Inicializa-Fila(Q,x)
8.   enquanto Q ≠ ⊘ faça
9.        u ← Primeiro-da-Fila(Q)
10.       para cada v em Adj[u] faça
11.           se cor[v] = BRANCO
12.               então cor[v] ← CINZA
13.                  dist[v] ← dist[u]+1
14.                  pred[v] ← u
15.                  Insira-na-Fila(Q,v)
16.       Remova-da-Fila(Q)
17.       cor[u] ← PRETO
18. devolve dist[1..n], pred[1..n]
```

Executando BFS no grafo exemplo...obtemos:

dist[1..n] = {8, 7, 6, 5, 4, 9, ∞, ∞, ∞, 3, ∞, 3, 2 ,1, 2, 3, 2, 1, 0, 1}

pred[1..n] = {2, 3, 4, 5, 10, 1, NIL, NIL, NIL, 15, NIL, 17, 14, 19, 14, 17, 18, 19, -, 19}

3) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9
CALÇA 2 → 3 → 8 → 9
CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

Modelamos a sequência em dígrafo definindo arcos conforme descrição.

cada arco (u,v): u antecede v

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

Aplicando :

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.       para cada v em Adj(u) faça
 5.            grau-entr[v] ← grau-entr[v] + 1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.       u ←  TiraFila(Q), topo[i] ← u, i ← i + 1
13.       para cada v em Adj(u) faça
14.            grau-entr[v] ← grau-entr[v] - 1
15.            se grau-entr[v] = 0
16.                 então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	1	2	0	1	2	1	3	8
TOPO									

Q: 1, 4

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	2	0	1	2	0	2	7
TOPO	**1**								

Q: 4, 2, 7,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	1	0	0	2	0	2	6
TOPO	**1**	**4**							

Q: 2, 7, 5

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	2	0	1	5
TOPO	1	4	2						

Q: 7, 5, 3,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	2	0	0	4
TOPO	**1**	**4**	**2**	**7**					

Q: 5, 3, 8,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	1	0	0	3
TOPO	1	4	2	7	5				

Q: 3, 8,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	0	0	0	2
TOPO	**1**	**4**	**2**	**7**	**5**	**3**			

Q: 8, 6,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8 → 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	0	0	0	1
TOPO	**1**	**4**	**2**	**7**	**5**	**3**	**8**		

Q: 6,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8→ 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	0	0	0	0
TOPO	**1**	**4**	**2**	**7**	**5**	**3**	**8**	**6**	

Q: 9,

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

Listas de Adjacências:

CUECA 1 → 2 → 7 → 8 → 9

CALÇA 2 → 3 → 8 → 9

CINTO 3 → 6 → 9
CAMISA 4 → 3 → 5 → 9
GRAVATA 5 → 6 → 9
PALETÓ 6 → 9
MEIAS 7 → 8 → 9
SAPATO 8 → 9
RELÓGIO 9

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.          grau-entr[u] ← 0
 3.  para cada vértice u em G faça
 4.      para cada v em Adj(u) faça
 5.          grau-entr[v] ← grau-entr[v] +
1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (fontes)
 8.      se grau-entrada = 0
 9.        então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ⊘ faça
12.      u ←  TiraFila(Q), topo[i] ← u, i ← i +
1
13.      para cada v em Adj(u) faça
14.          grau-entr[v] ← grau-entr[v] - 1
15.          se grau-entr[v] = 0
16.              então  InsereFila(Q,v)
17.  devolve topo[1 .. n]
```

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	0	0	0	0
TOPO	**1**	**4**	**2**	**7**	**5**	**3**	**8**	**6**	**9**

Q: ⊘

1) *Mr Bean* é muito metódico. Todos os dias pela manhã, segue o mesmo *ritual* para se vestir. Faz parte do seu vestuário: cueca, calça, cinto, camisa, gravata, paletó, meias e sapato, além de um vistoso relógio de pulso. Ele sempre veste a cueca antes de por as meias e a calça. Os sapatos são calçados após o professor ter vestido a cueca, calça e meias. O cinto vai depois da calça e da camisa. O relógio pode ser colocado em qualquer momento. O paletó só é vestido depois do cinto e da gravata que é colocada depois da camisa. Utilize a teoria dos grafos para ajudar *Mr Bean*, determinando em que sequência deve vestir as peças para que o seu *ritual* seja cumprido.

vert	1	2	3	4	5	6	7	8	9
grau	0	0	0	0	0	0	0	0	0
TOPO	**1**	**4**	**2**	**7**	**5**	**3**	**8**	**6**	**9**

Listas de Adjacências:

CUECA **1 → 2 → 7 → 8 → 9**

CALÇA **2 → 3 → 8 → 9**

CINTO **3 → 6 → 9**
CAMISA **4 → 3 → 5 → 9**
GRAVATA **5 → 6 → 9**
PALETÓ **6 → 9**
MEIAS **7 → 8 → 9**
SAPATO **8 → 9**
RELÓGIO **9**

Sequência: 1 → 4 → 2 → 7 → 5 → 3 → 8 → 6 → 9

Árvores Geradoras

Florestas e Árvores

arborescência da busca em profundidade é uma árvore que reporta seqüência de visitas

floresta é um grafo sem ciclos não - triviais

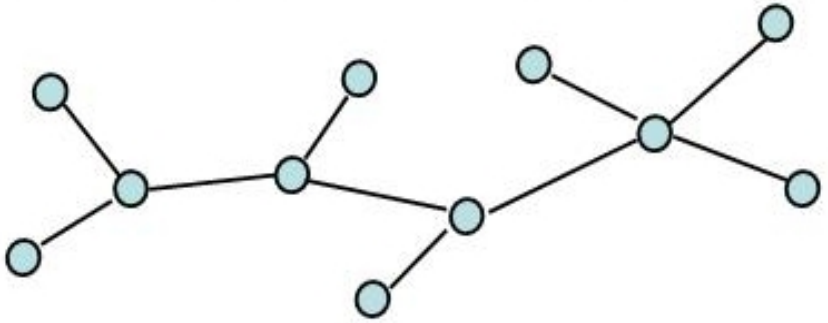
(arco v-w da floresta, o caminho v-w-v é um ciclo, más não é trivial).

árvore é uma floresta conexa, portanto, cada componente de uma floresta é uma árvore.

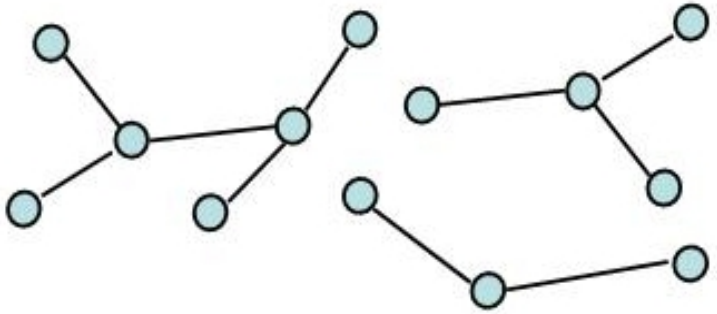
Não confunda árvores com arborescências.

Uma árvore é um dígrafo simétrico, enquanto uma arborescência é um dígrafo não-simétrico.

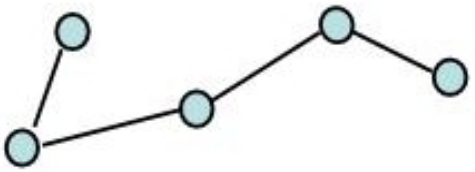
- **Árvore:** grafo conexo sem circuitos



- **Floresta:** grafo cujas componentes conexas são árvores

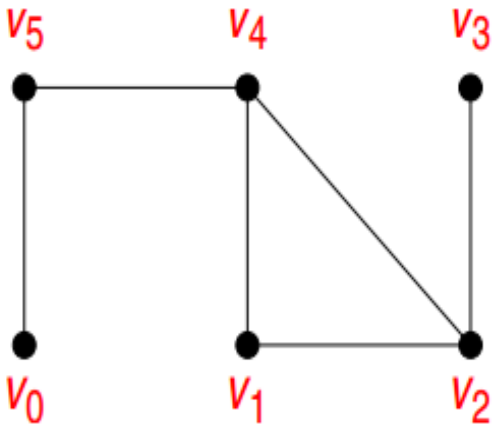


Um caminho é uma árvore?



Sim!

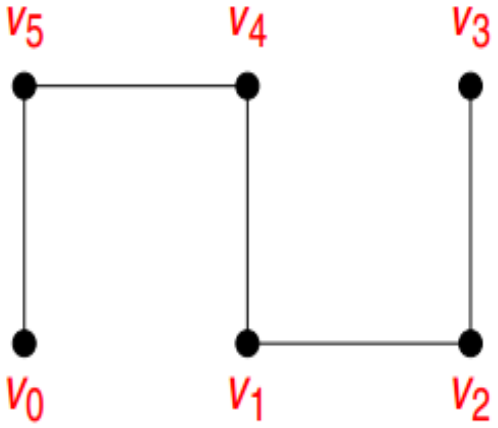
Seja o grafo G abaixo

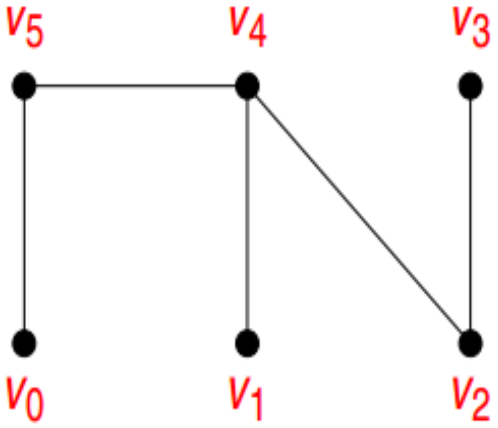


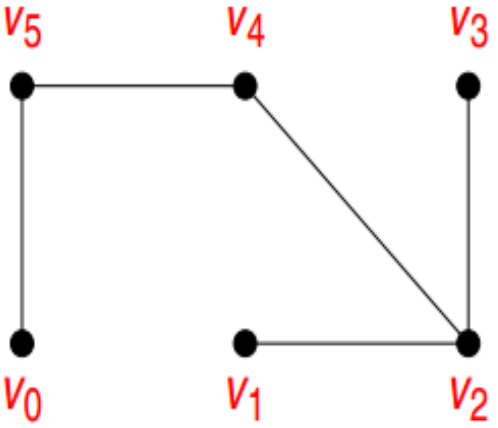
Este grafo possui o circuito $v_2v_1v_4v_2$.

A remoção de qualquer uma das três arestas do circuito leva a uma árvore.

Assim, todas as três árvores geradoras são:







Árvore Geradora Mínima

- **Termo em inglês:**
  - **Minimum Spanning Tree (MST)**

- **Alguns sinônimos encontrados:**
  - **Árvores de Expansão Mínima**
  - **Árvores Geradora Mínima**
  - **Árvores de Cobertura Mínima**
  - **Árvores Espalhadas Mínimas**

Aplicações

- **Projetos de redes de telecomunicações:**
  - **Redes de computadores**
  - **Redes de fibra ótica**
  - **Redes de telefonia**
  - **Redes de televisão a cabo**

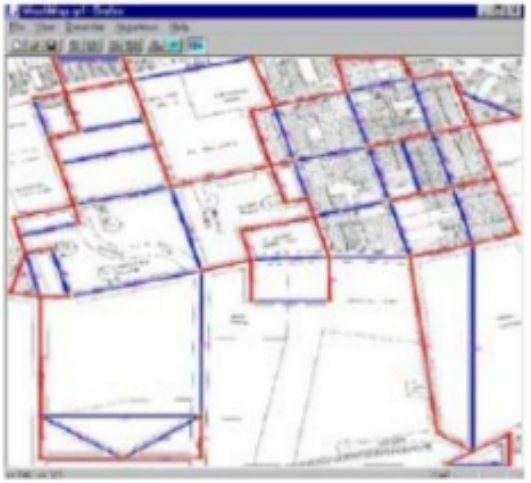
- **Projetos na área de transportes:**
  - **Rodovias, ferrovias, gasoduto**
  - **Irrigação – "transporte de água"**
  - **Logística (distribuição)**

- **Projetos de redes de distribuição de energia**

- **Mineração de dados em *clustering***

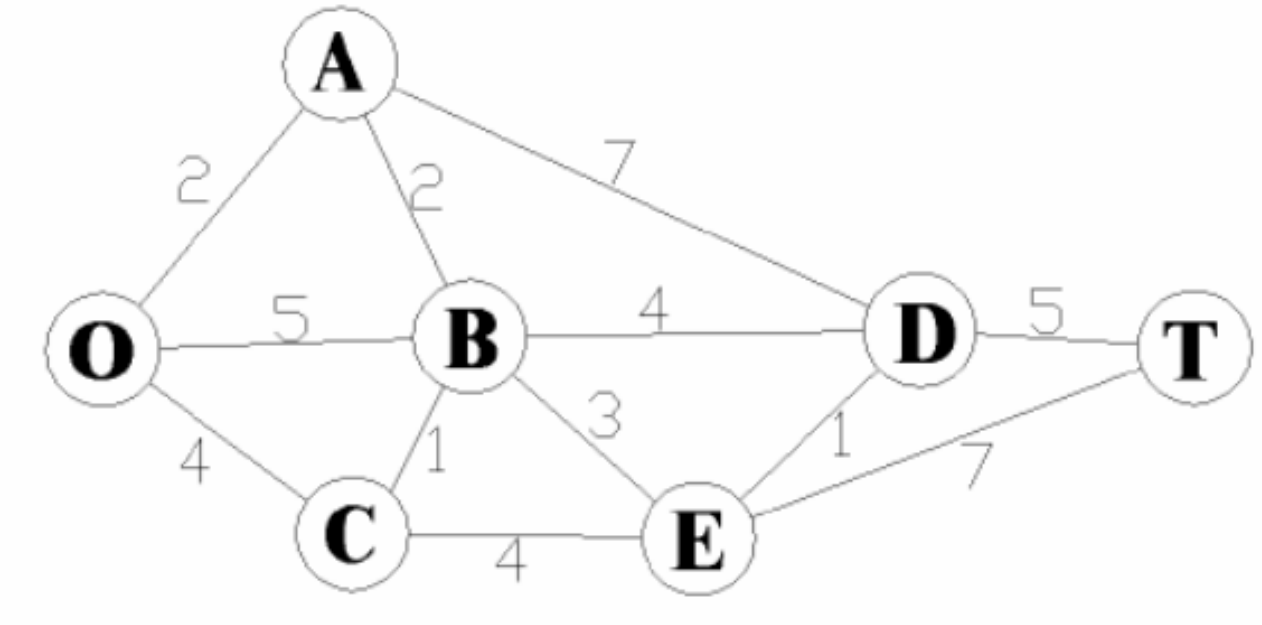
Aplicações

Sistemas de Informações Geográficas



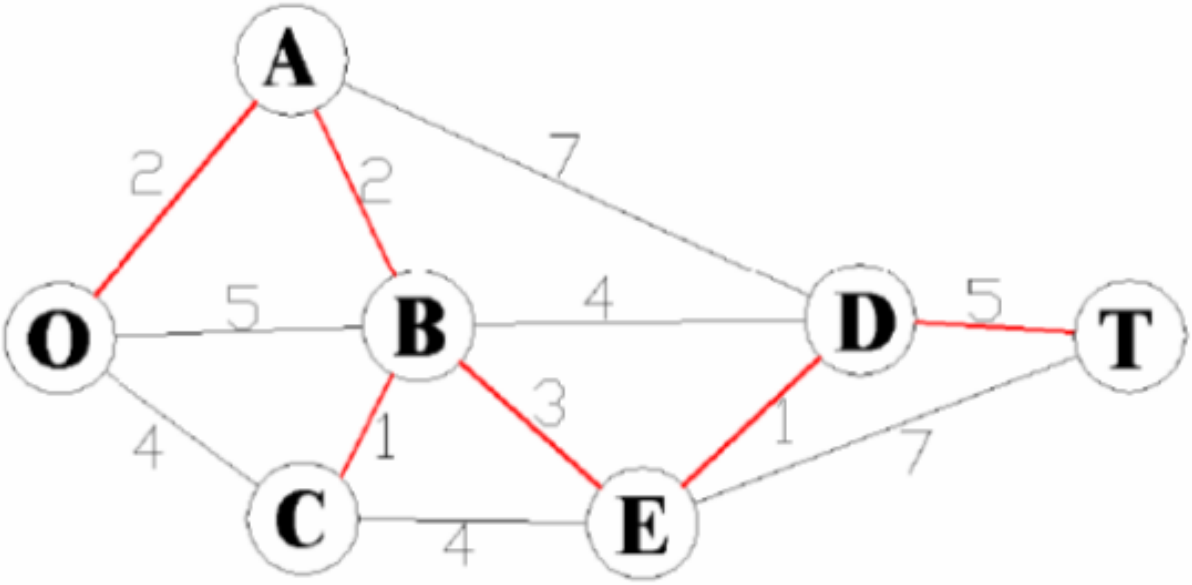
conceito

- **Árvore que interliga todos os vértices do grafo utilizando arestas com um custo total mínimo**



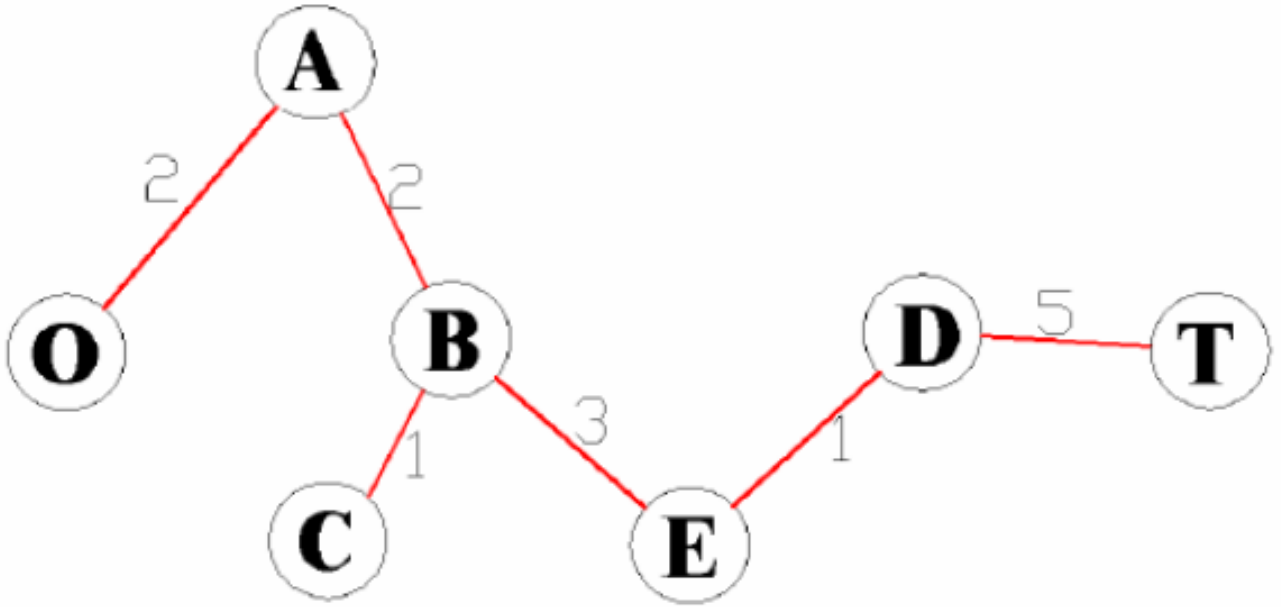
conceito

- **Árvore que interliga todos os vértices do grafo utilizando arestas com um custo total mínimo**



conceito

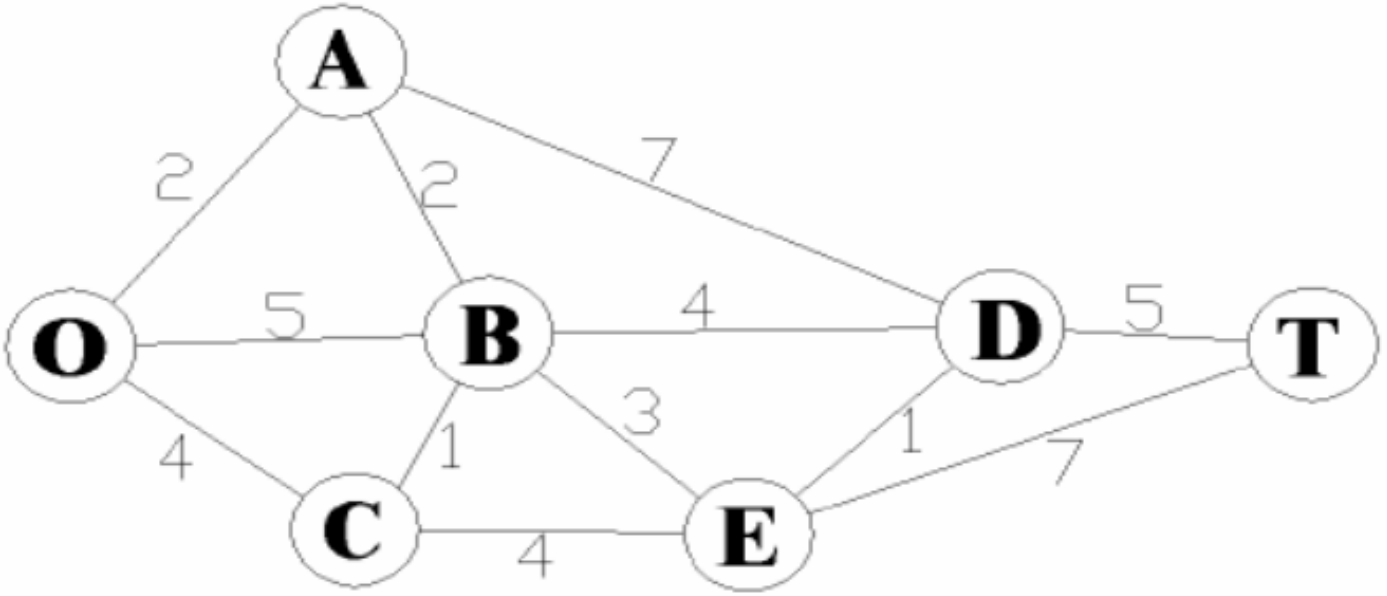
- **Árvore que interliga todos os vértices do grafo utilizando arestas com um custo total mínimo**



conceito

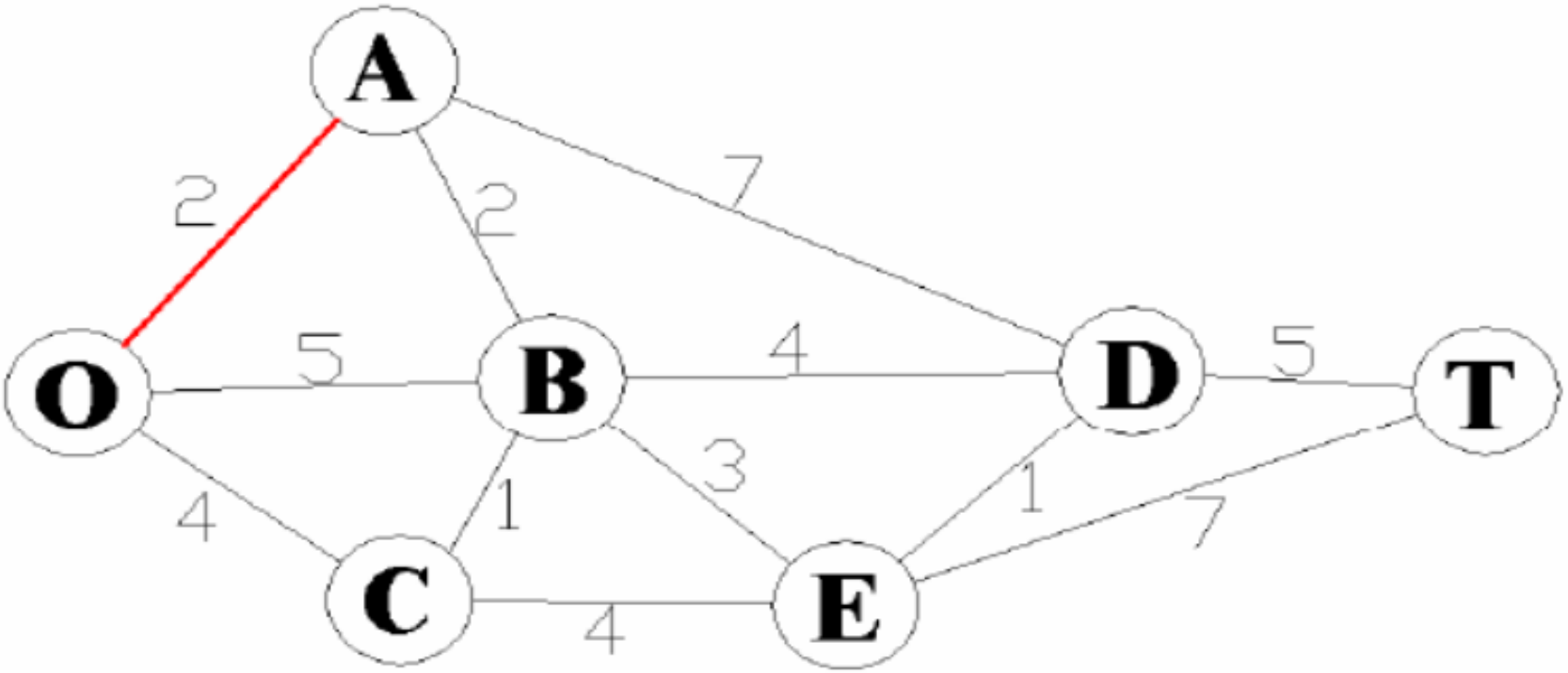
- **Dado um grafo não dirigido ponderado G, desejamos encontrar uma árvore T que contenha todos os vértices de G e minimize sua soma.**

A rede do Parque Seervada necessita interligar todos os postos de guarda por uma linha telefônica.

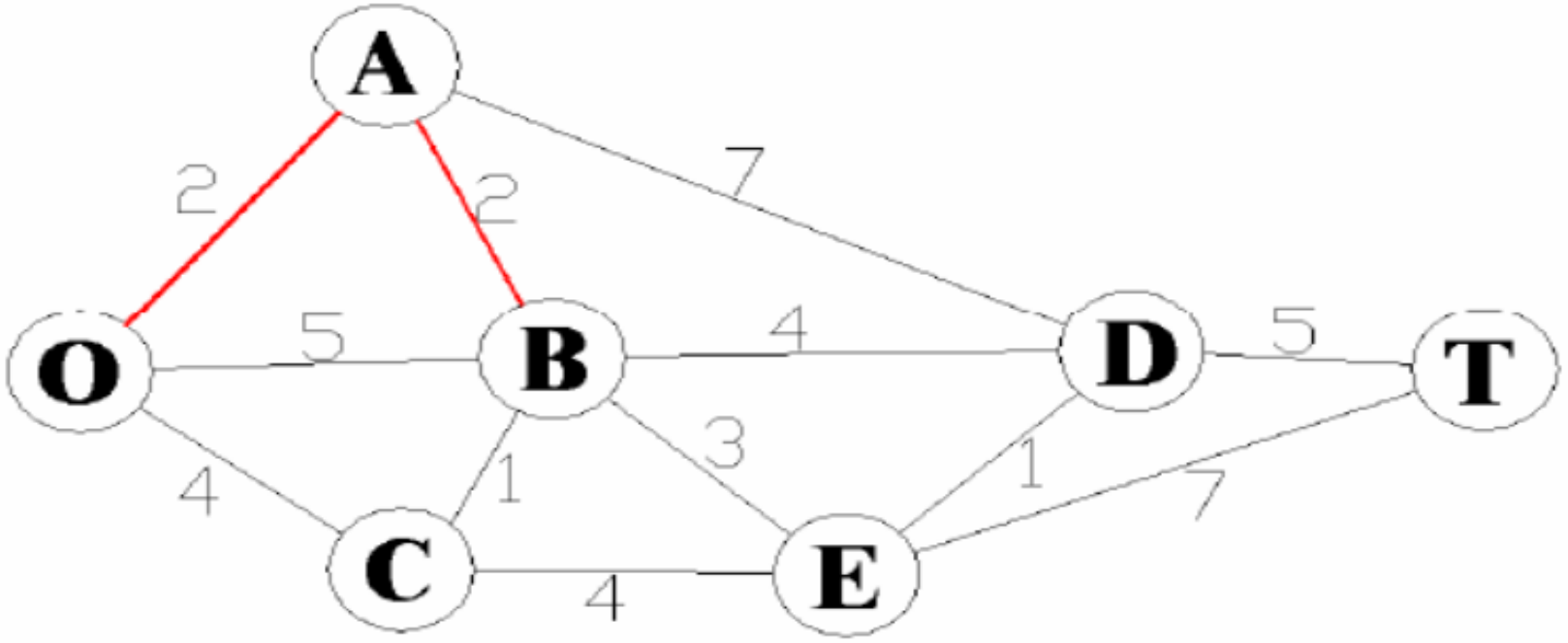


As linhas representam “arcos potenciais”.

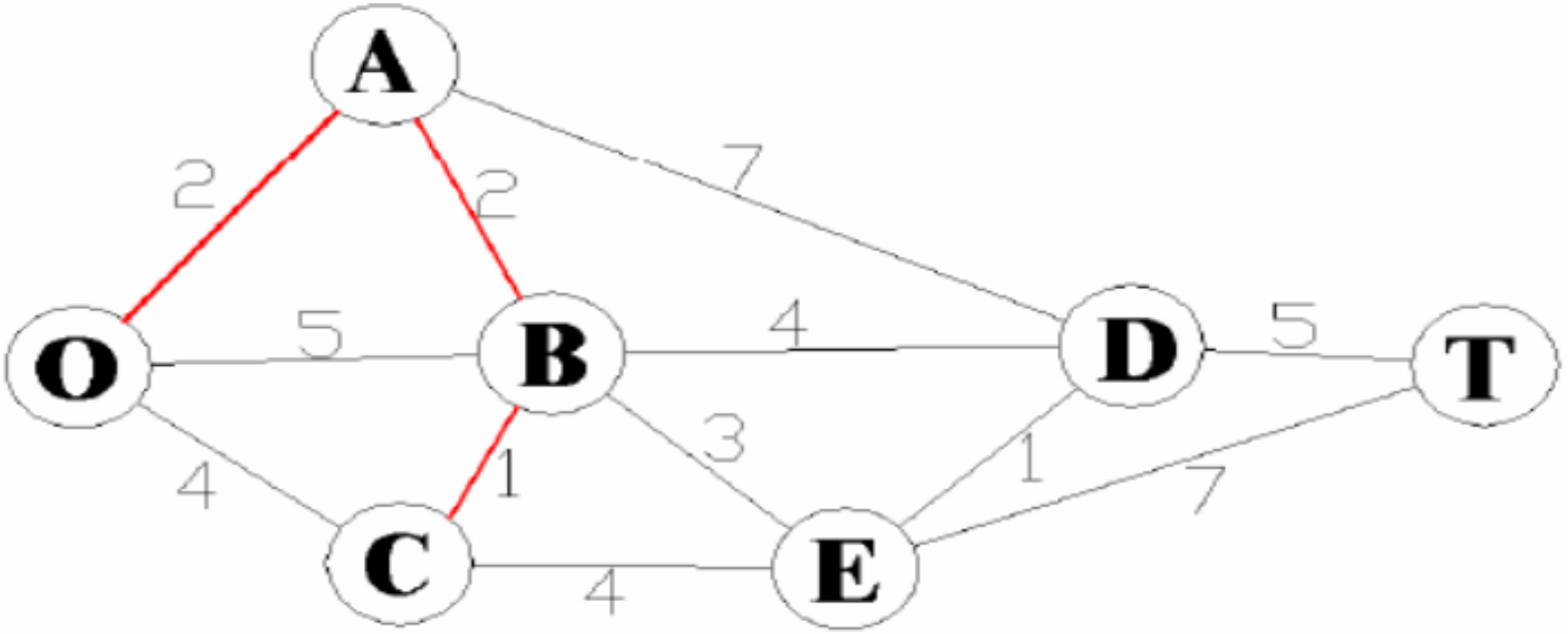
Arbitrariamente, seleciona o nó O para iniciar (poderia ter sido qualquer outro nó). O nó não conectado mais próximo de O é A. Conectar o nó A para o nó O.



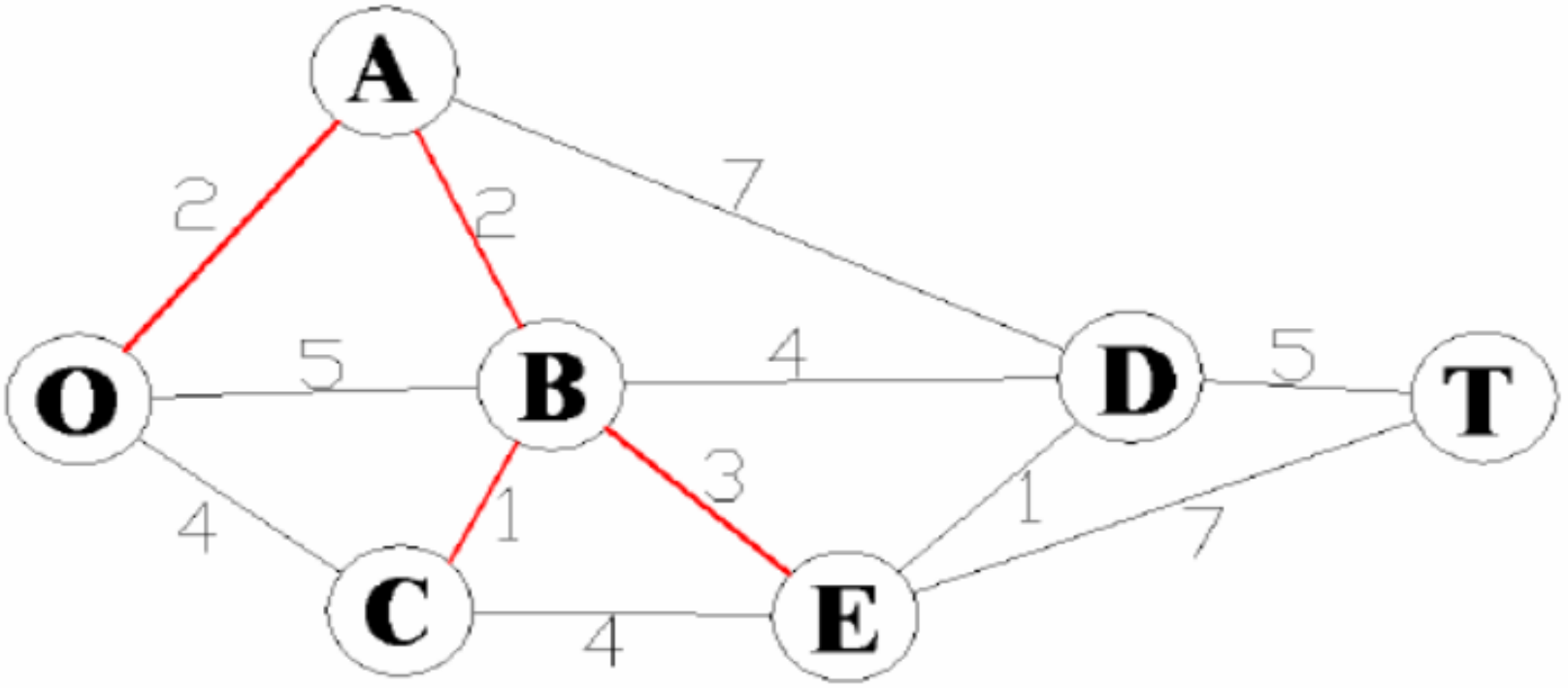
O nó não conectado mais próximo dos nós O e A é o nó B (mais próximo de A). Conectar o nó B para o nó A.



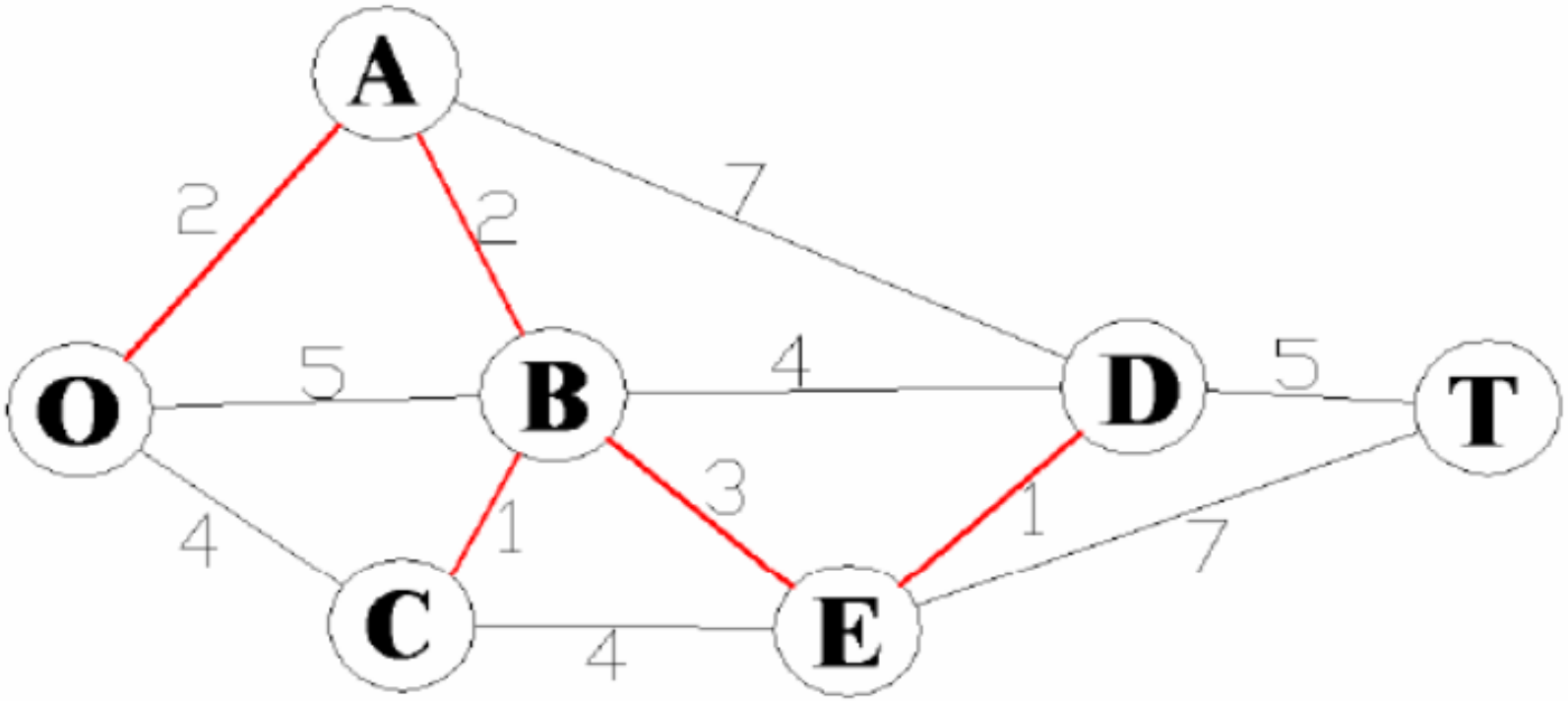
O nó não conectado mais próximo dos nós O, A e B é o nó C (mais próximo de B). Conectar o nó C para o nó B.



O nó não conectado mais próximo dos nós O, A, B e C é o nó E (mais próximo de B). Conectar o nó E para o nó B.

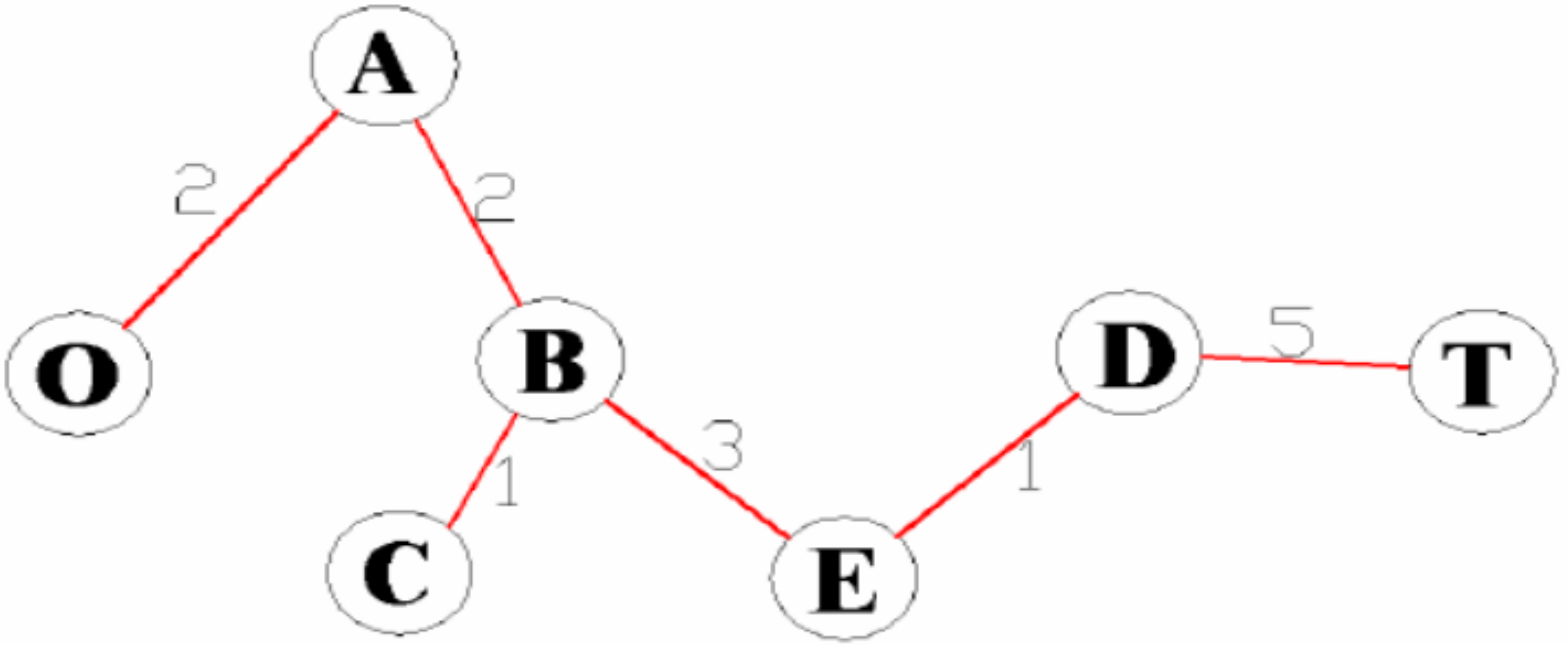


O nó não conectado mais próximo dos nós O, A, B, C e E é o nó D (mais próximo de E). Conectar o nó D para o nó E.



A solução final possui uma distância total de 14 milhas. Através desta rede, é possível ir de um posto a qualquer outro posto.

Independente do nó inicial, a solução será a mesma.



Construção de uma árvore geradora

Problema: *dado um grafo e um vértice x, encontre uma árvore geradora com raiz x.*

A solução emprega um algoritmo que no início de cada iteração tem duas classes de vértices: vértices PRETOs e vértices BRANCOs, e uma árvore com raiz *x* que "cobre" o conjunto dos vértices PRETOs.

```
Árvore(G, x)
 1. para u ←1 até n faça
 2.       cor[u] ← BRANCO
 3.       pred[u] ← NIL
 4. cor[x] ← CINZA
 5. enquanto existe u tal que cor[u] = CINZA faça
 6.          para cada v em Adj[u] faça
 7.                se cor[v] = BRANCO
 8.                   então cor[v] ← CINZA
 9.                         pred[v] ← u
10.          cor[u] ← PRETO
11. devolve pred
```

Árvores geradoras de peso mínimo (Minimum - Spanning Trees)

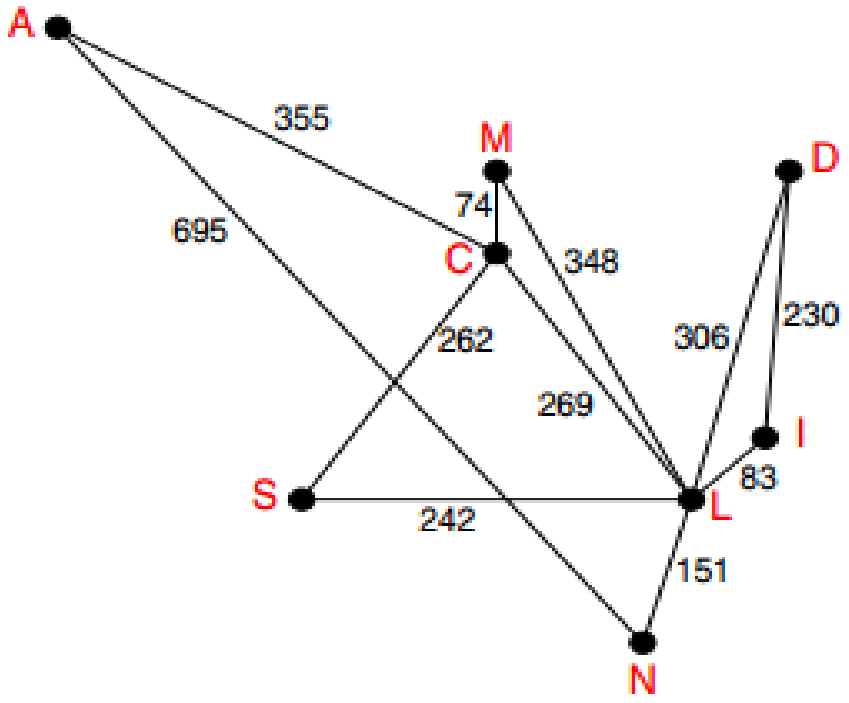
Imagine que cada aresta *(u,v)* de um *grafo* tem um **peso** numérico *w(u,v)*, que pode ser positivo, negativo ou nulo. O **peso** de um conjunto *E* de arestas é a soma dos pesos das arestas em *E*; o peso de *E* será denotado por *w(E)*.

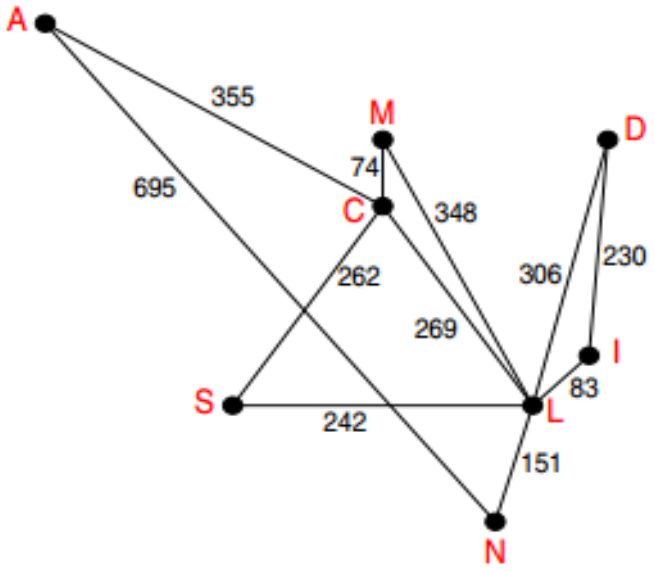
Onde esses pesos ficam armazenados? → Os valores de *w* ficam dentro das listas de adjacência: na lista *Adj[u]*, junto com a célula que contém um vértice *v*, fica também o número *w(u,v)* e, na matriz de adjacências fica na célula (u,v) que indica a existência da aresta quando não é nula.

> Problema: *dado um grafo, um vértice x, e uma função w que atribui um peso a cada arco, encontrar uma árvore geradora com raiz x que tenha peso mínimo.* (Obs.: o peso de uma árvore é a soma dos pesos de suas arestas.)

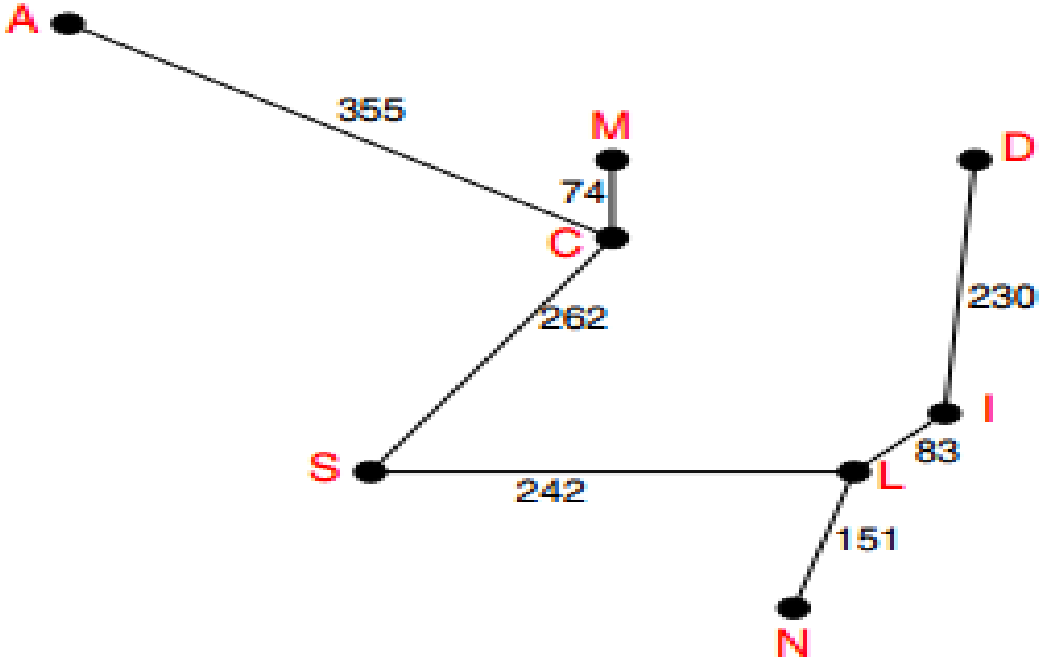
No grafo qualquer árvore geradora com raiz *x* "cobre" todos os vértices do componente que contém *x*, ou seja, para cada vértice *v* do componente existe um (e um só) caminho de *x* a *v* na árvore. Se existe uma árvore geradora com raiz *x* e peso *P* então, para qualquer vértice *v* no componente que contém *x*, existe uma árvore geradora de peso *P* e raiz *v*.

O grafo de rotas da companhia aérea que recebeu permissão para voar pode ser "rotulado" com as distâncias entre as cidades:





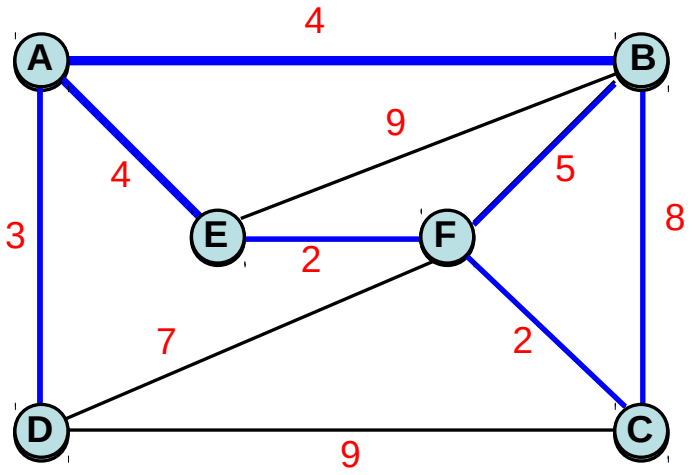
Suponha que a companhia deseja voar para todas as cidades mas usando um conjunto de rotas que minimiza o total de distâncias percorridas:



Este conjunto de rotas interconecta todas as cidades.

Árvore Geradora de Peso Mínimo

+ Exemplo:



4 4

4 5 4

3 8 3 2

2

árvore geradora
peso = 24

árvore geradora
peso = 15

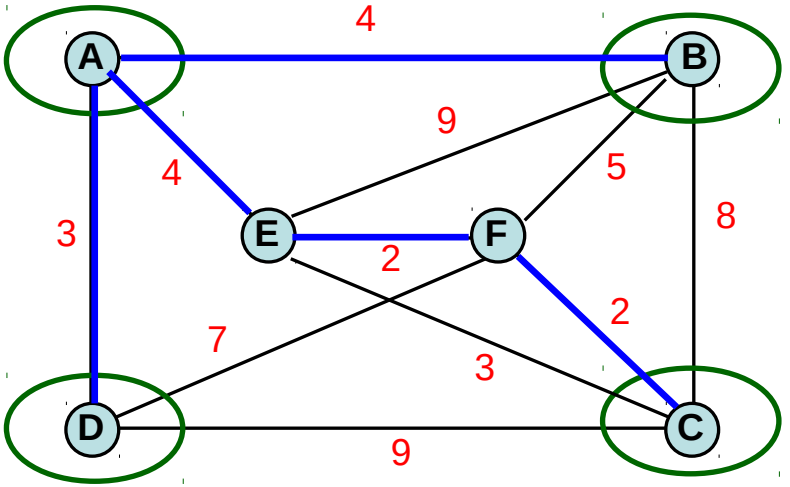
Algoritmo de Prim

O algoritmo de Prim usa uma "fila" *de vértices*, com prioridade ditada por uma chave

```
MST_Prim(G, x)
 1.  para u ← 1 até n faça
 2.        cor[u] ← BRANCO
 3.        pred[u] ← NIL
 4.        chave[u] ← ∞
 5.  Q ← Crie-Fila-Vazia()
 6.  para u ← 1 até n faça
 7.         Insira-na-Fila(u,Q)
 8.  chave[x] ← 0
 9.  pred[x] ← x
10.  enquanto Q ≠ ∅ faça
11.        u ← Retire-Mínimo(Q)
12.        para cada v em Adj[u]faça
13.              se cor[v] = BRANCO e w(u,v) < chave[v]
14.                  então chave[v] ← w(u,v)
15.                        Refaça-Fila(v,Q)
16.                        pred[v]←u
17.         cor[u] ← PRETO
18.  devolve pred
```

AGM obtida com algoritmo de Prim

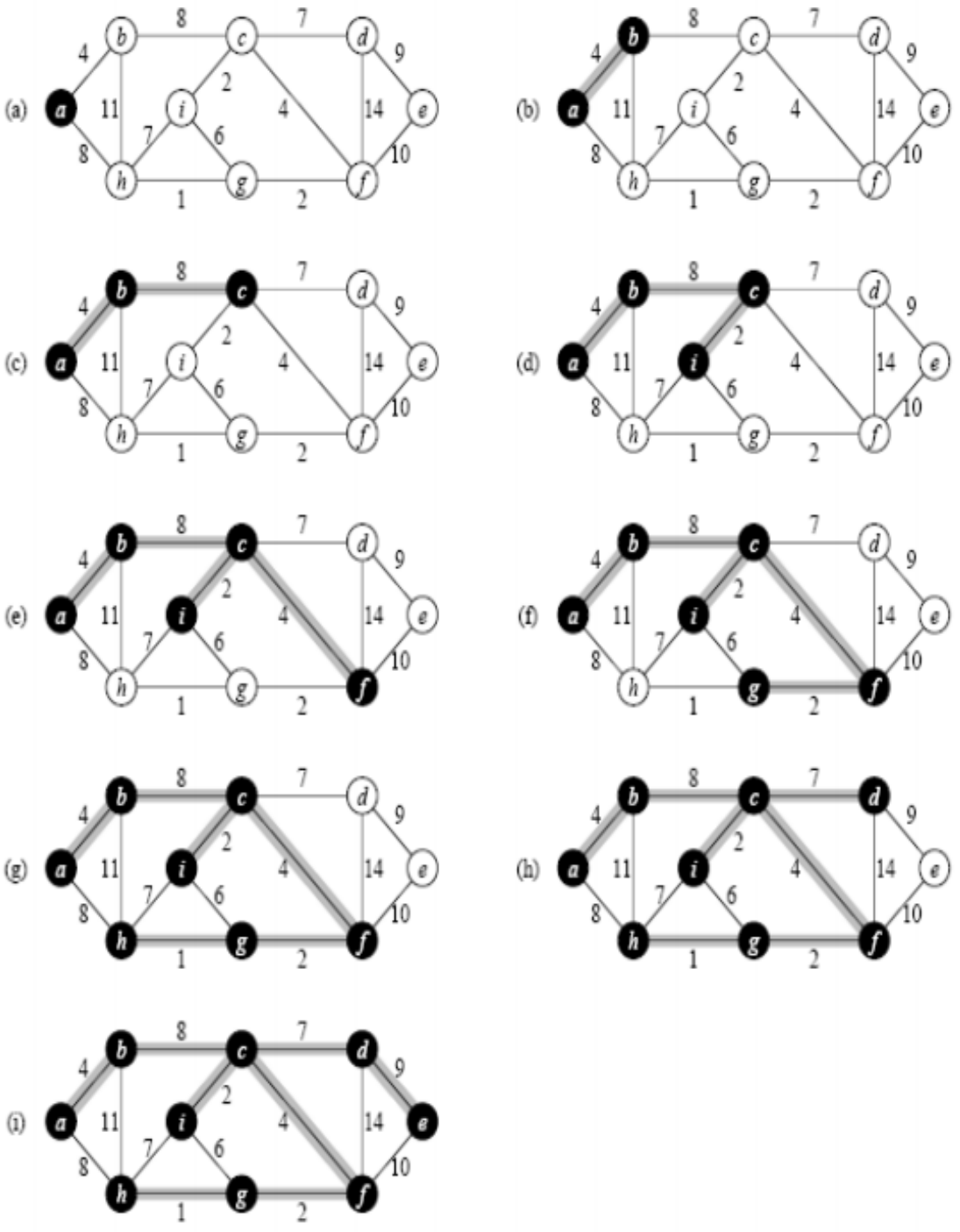
Exemplo:



c(F) = 15

Algoritmo de Prim

- Idéia básica:
  - Tomando como vértice inicial A, crie uma fila de prioridades classificada pelos pesos das arestas conectando A.
  - Repita o processo até que todos os vértices tenham sido visitados.



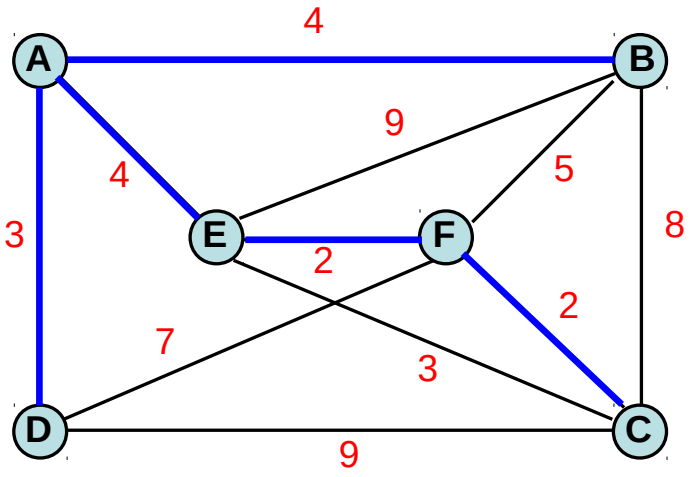
Algoritmo de Kruskal

O algoritmo de Kruskal começa com uma floresta de árvores formada por apenas um vértice e procura, a cada passo, uma aresta que, além de conectar duas árvores distintas da floresta, possua peso mínimo.

```
MST_Kruskal(G,w)
1.  A ← ∅
2.  para u ← 1 até n faça Construa_Conjunto(u)
3.  Ordene(E)                          {arestas de E em ordem crescente de peso (w)}
4.  para cada aresta (u,v) faça  {tomadas em ordem crescente de peso (w)}
5.        se Conjunto(u) ≠ Conjunto(v)
6.              então A ← A ∪ {(u,v)}
7.                    Une_Árvores(u,v)
8.  devolve A
```

AGM obtida com algoritmo de Kruskal

Exemplo:



c(F) = 15

Subárvores

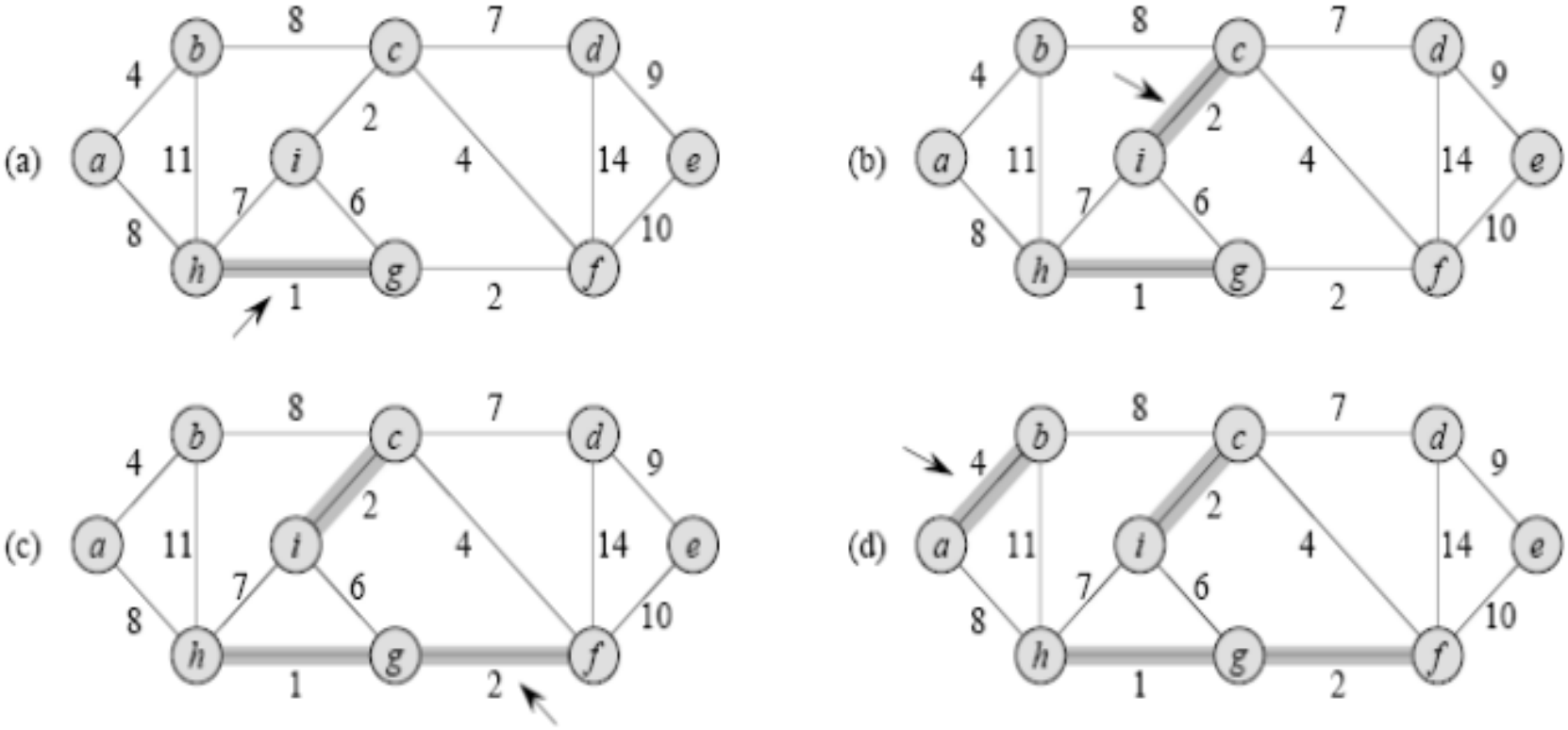
{ A, B, C, D, E, F }

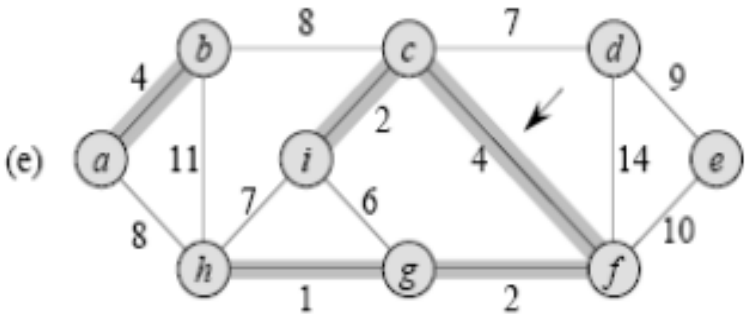
Lista L

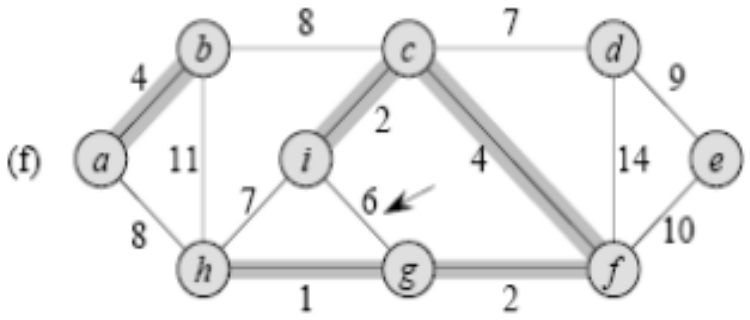
e	c(e)
(C,F)	2
(E,F)	2
(A,D)	3
(C,E)	3
(A,B)	4
(A,E)	4
(B,F)	5
(D,F)	7
(B,C)	8
(B,E)	9
(C,D)	9

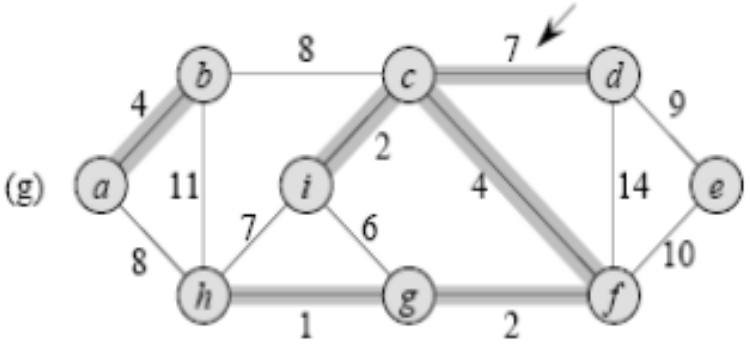
Algoritmo de Kruskal

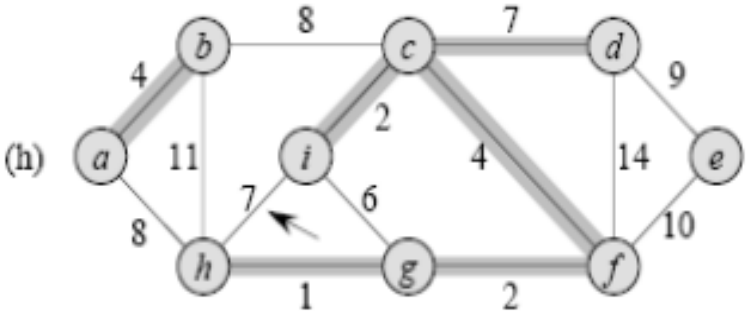
- Idéia básica:
  - Seleciona a aresta de menor peso que conecta duas árvores de uma floresta.
  - Repita o processo até que todos os vértices estejam conectados sempre preservando a invariante de se ter uma árvore.

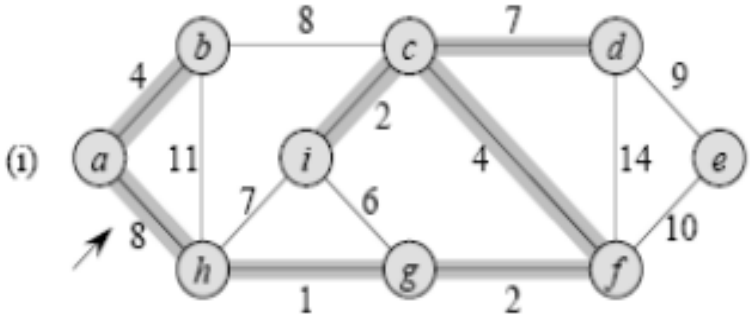


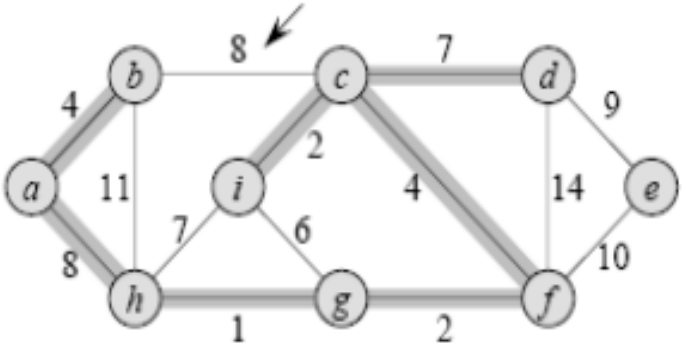

(e)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
7
6
10
1
2


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

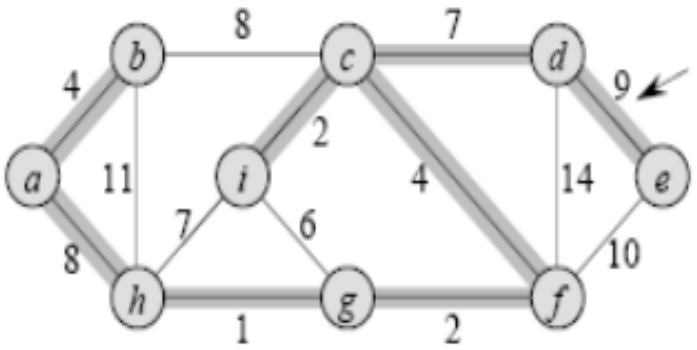

(g)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
7
6
10
1
2


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

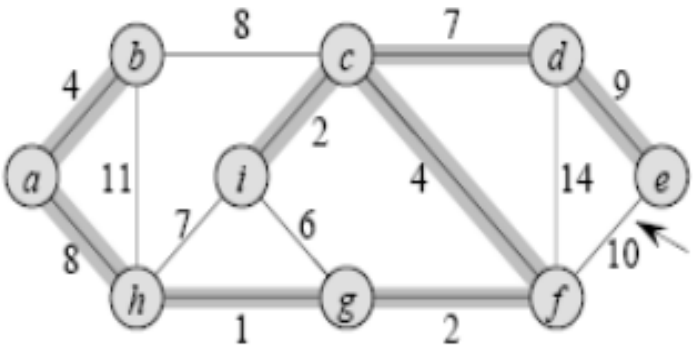

(i)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
7
6
10
1
2


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

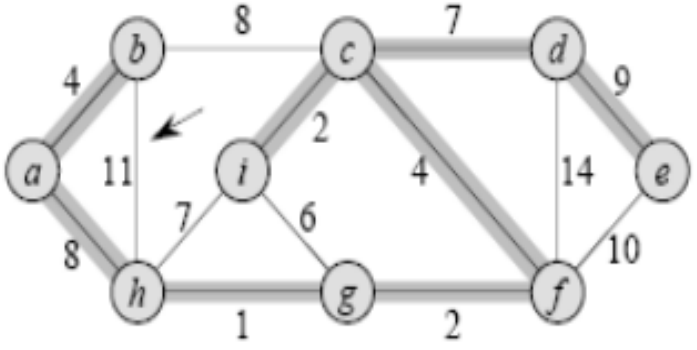
(k)


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

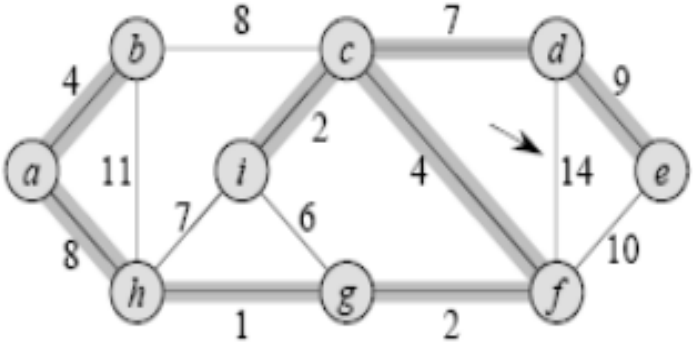
(l)


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

(m)

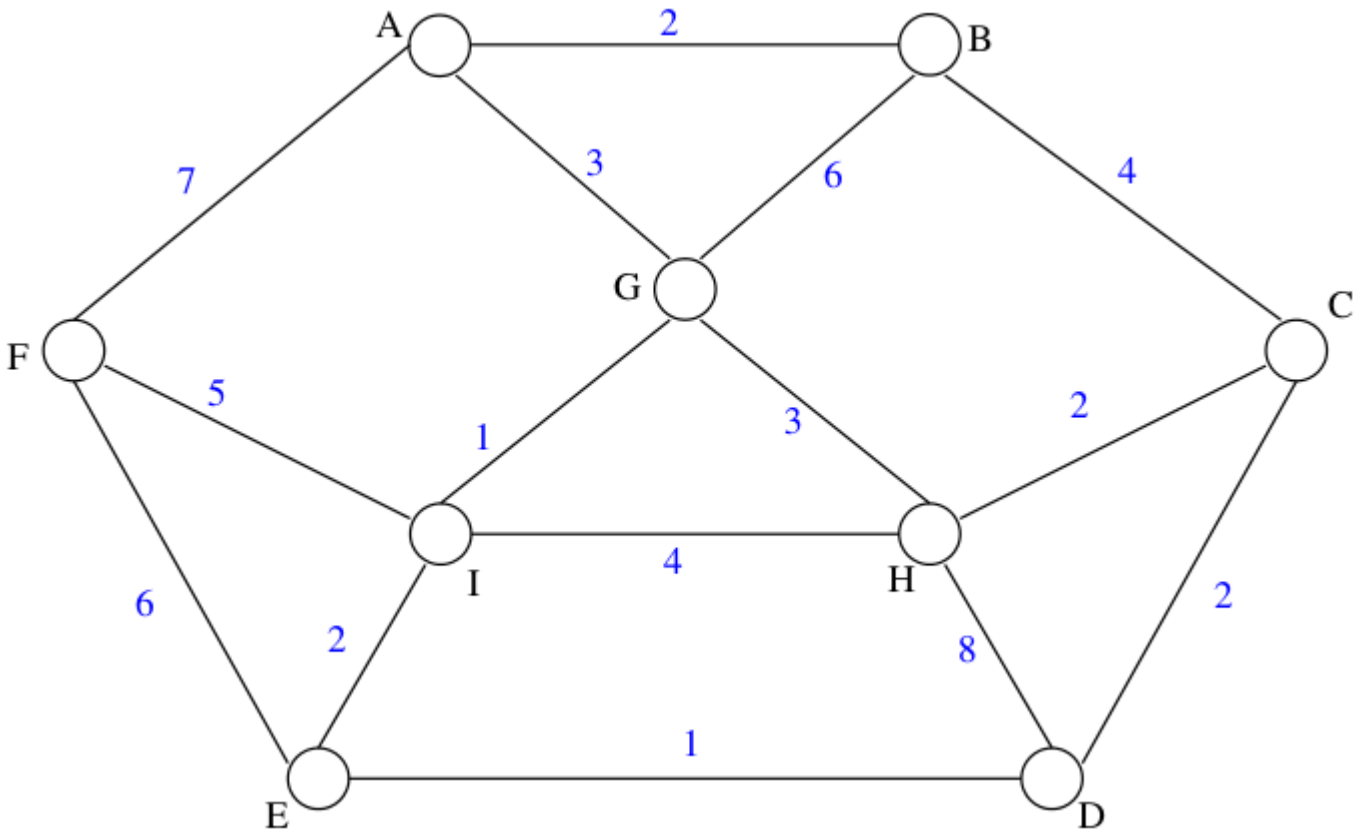

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
4
14
9
6
10
1
2

(n)

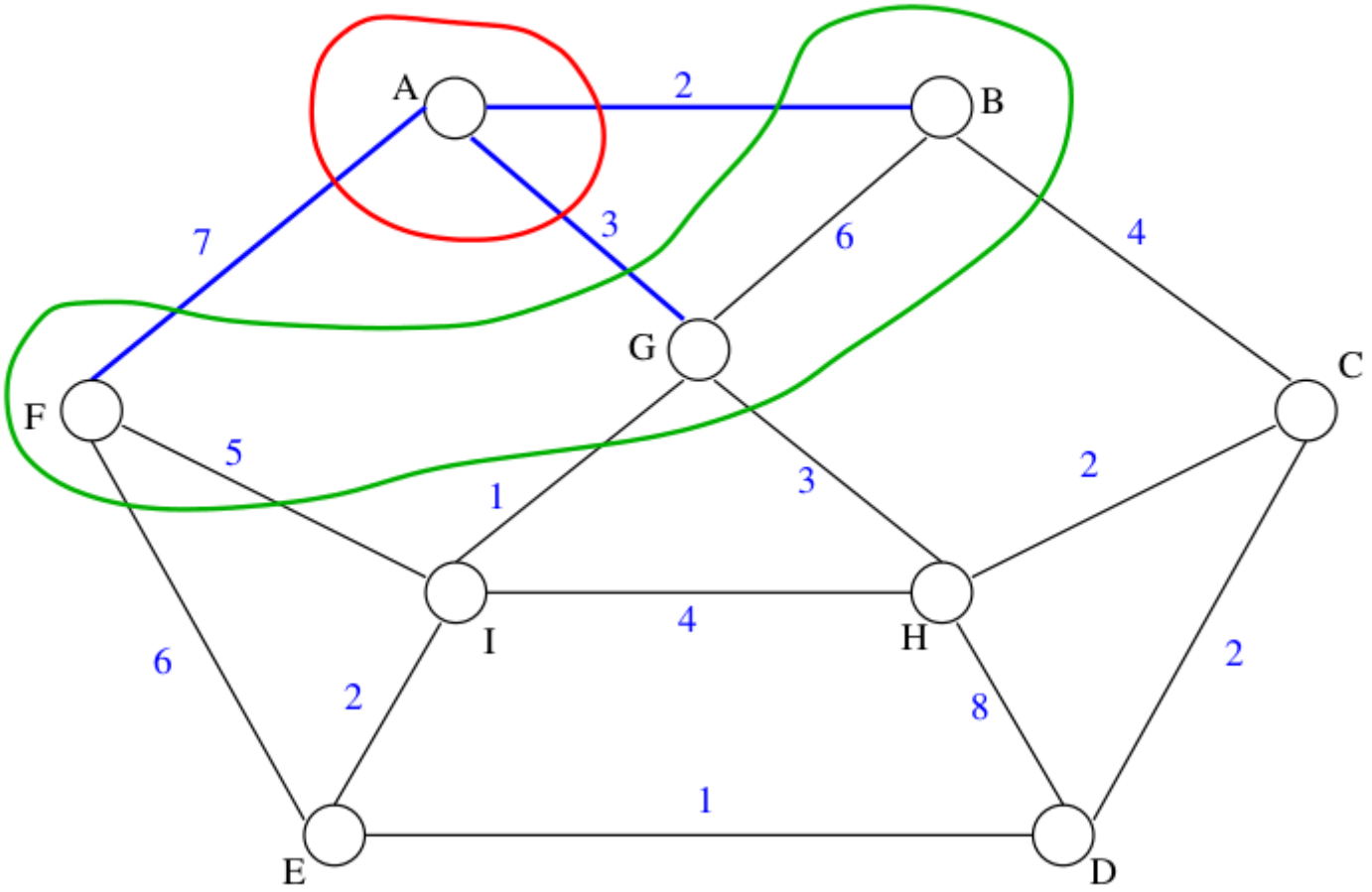

a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
4
14
9
6
10
1
2

Exercitando

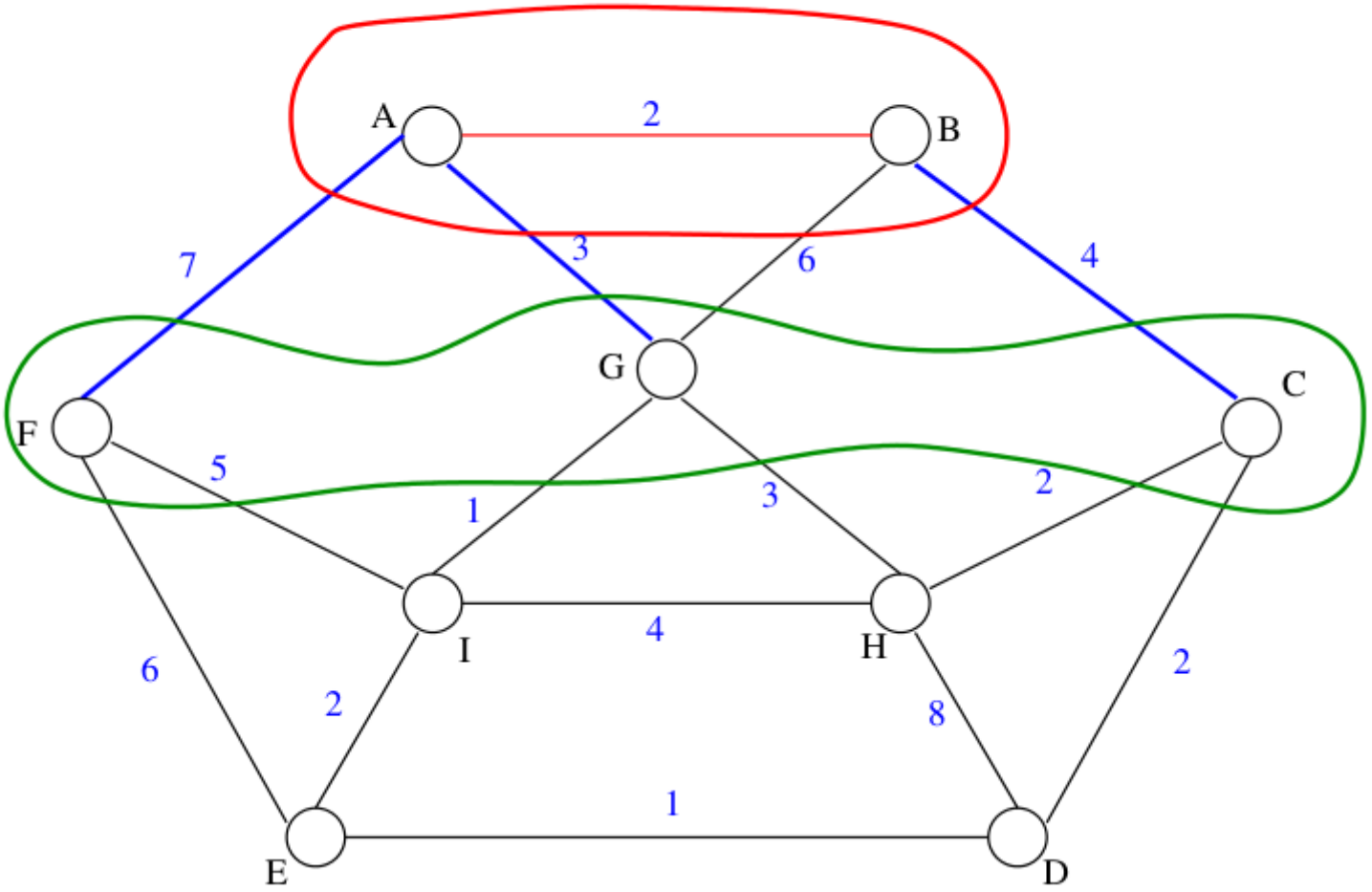
obter AGM com algoritmo de Prim, Kruskal



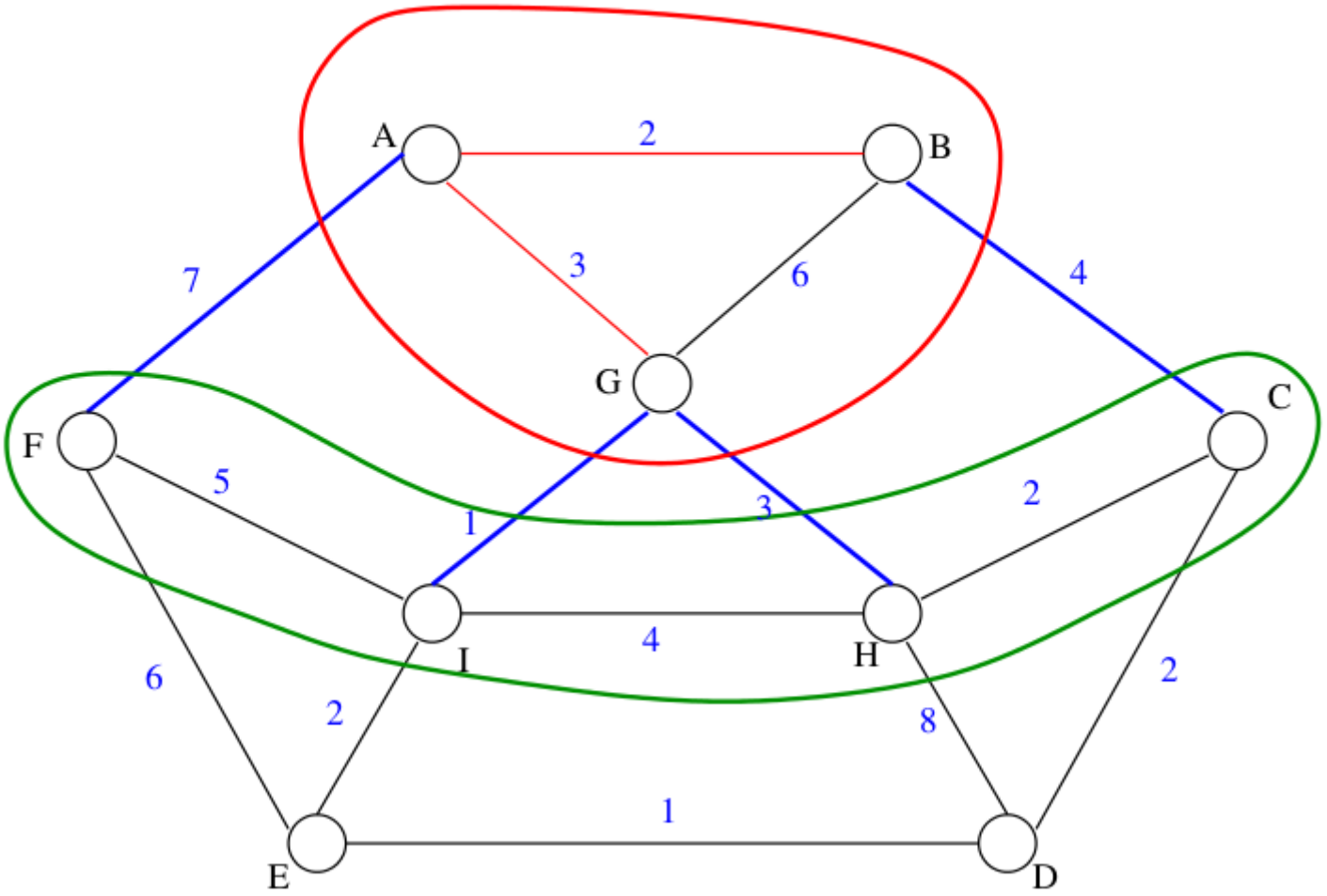
Algoritmo MST de Prim – Exemplo



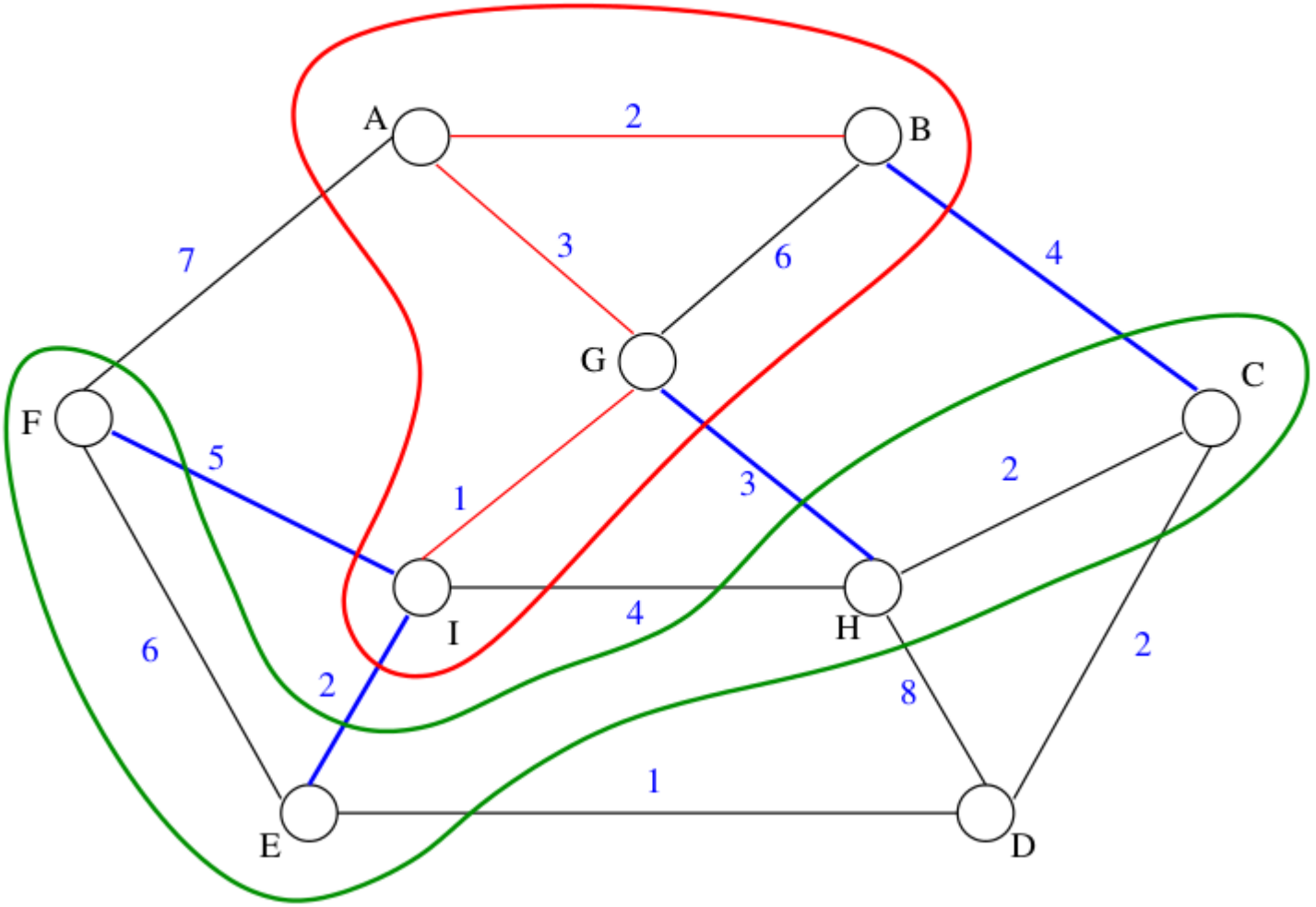
Algoritmo MST de Prim – Exemplo



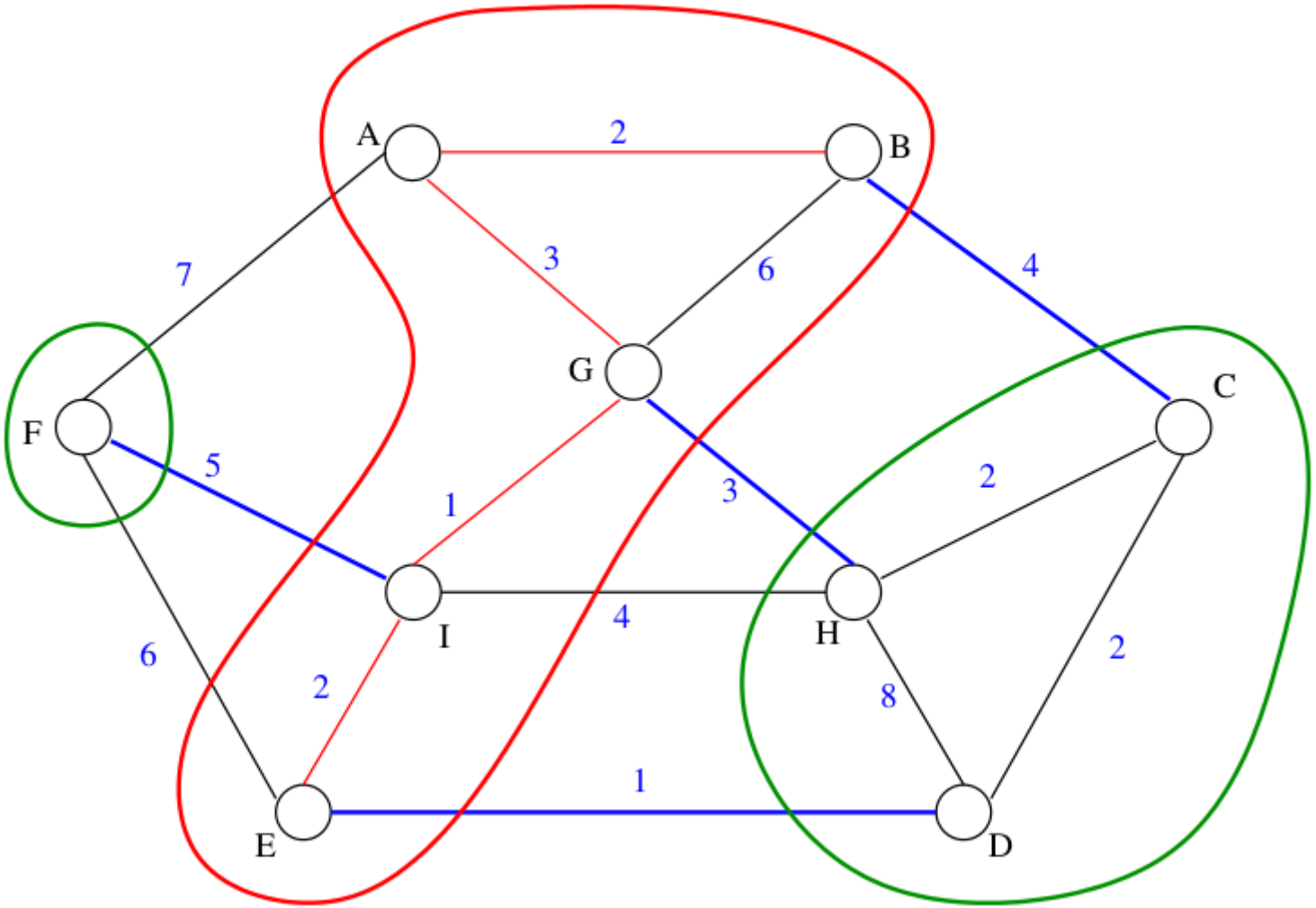
Algoritmo MST de Prim – Exemplo



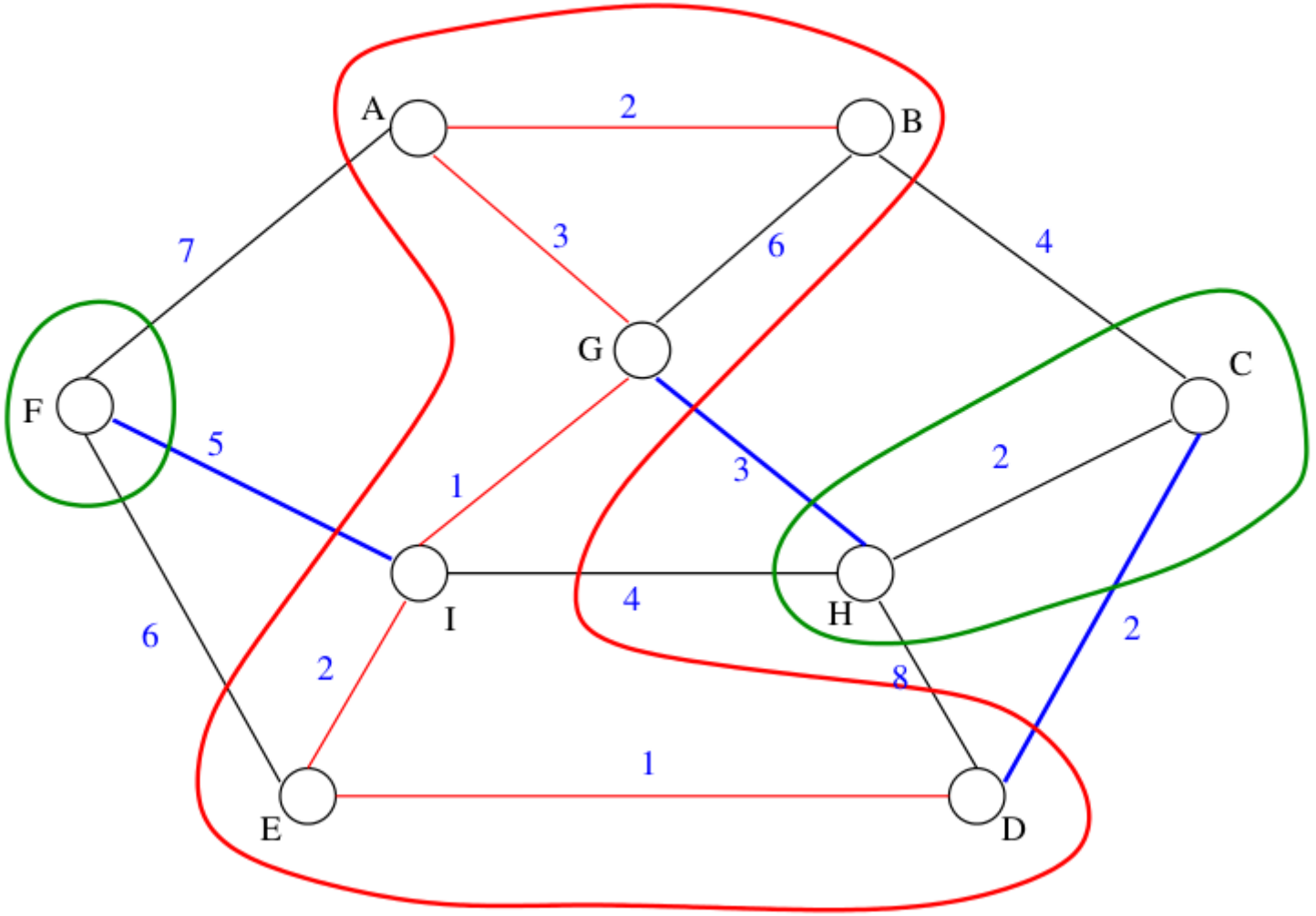
Algoritmo MST de Prim – Exemplo



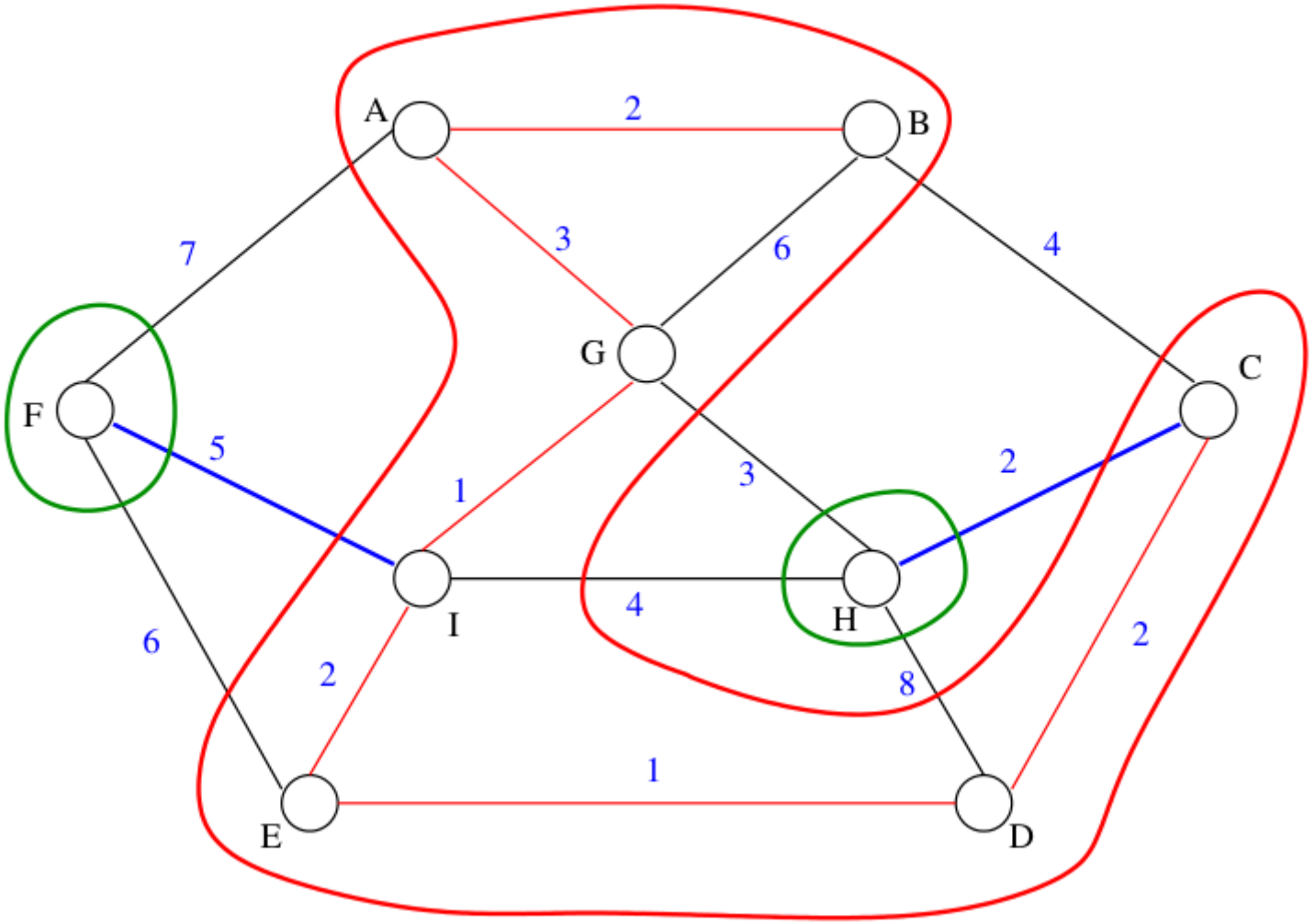
Algoritmo MST de Prim – Exemplo



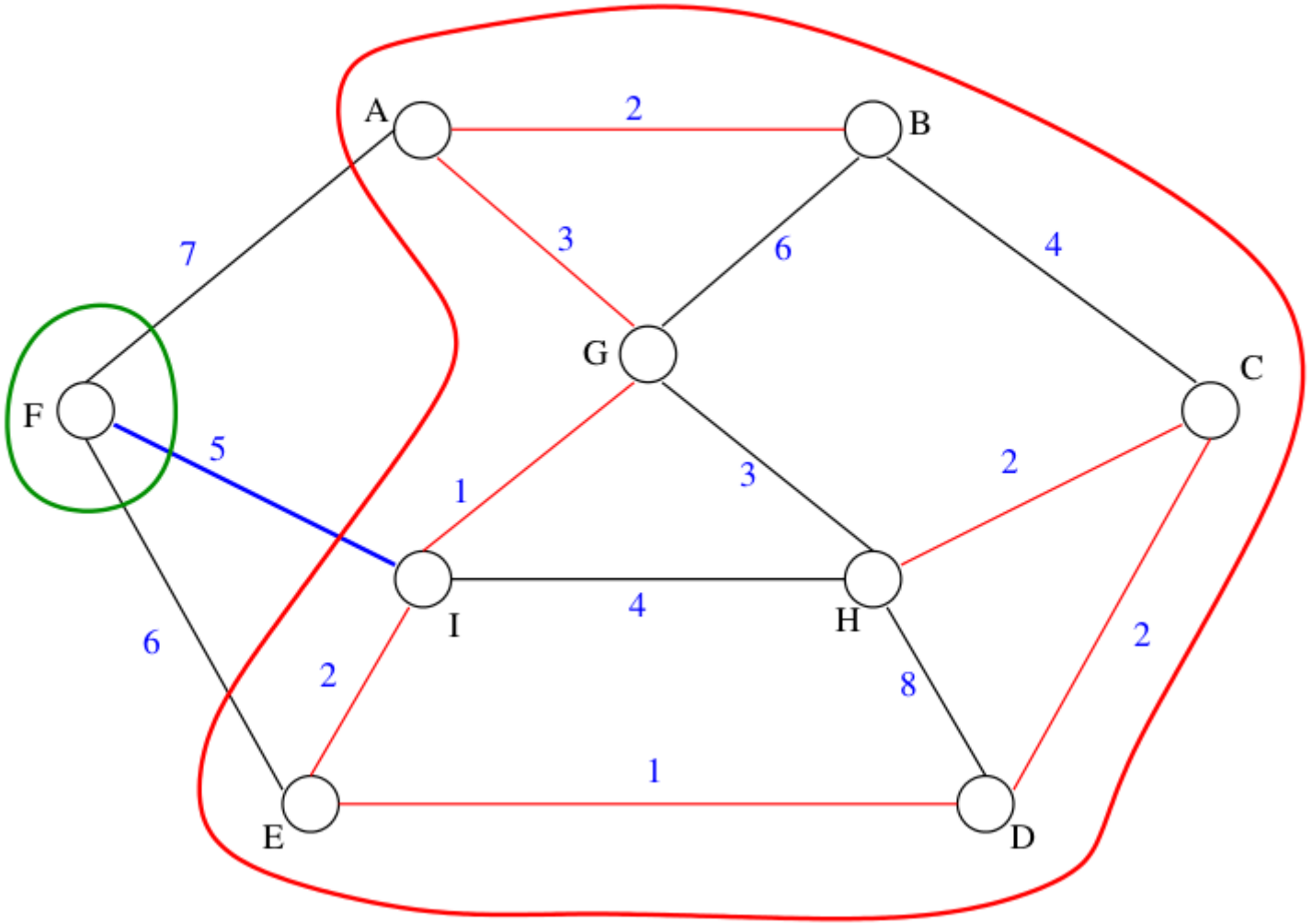
Algoritmo MST de Prim – Exemplo



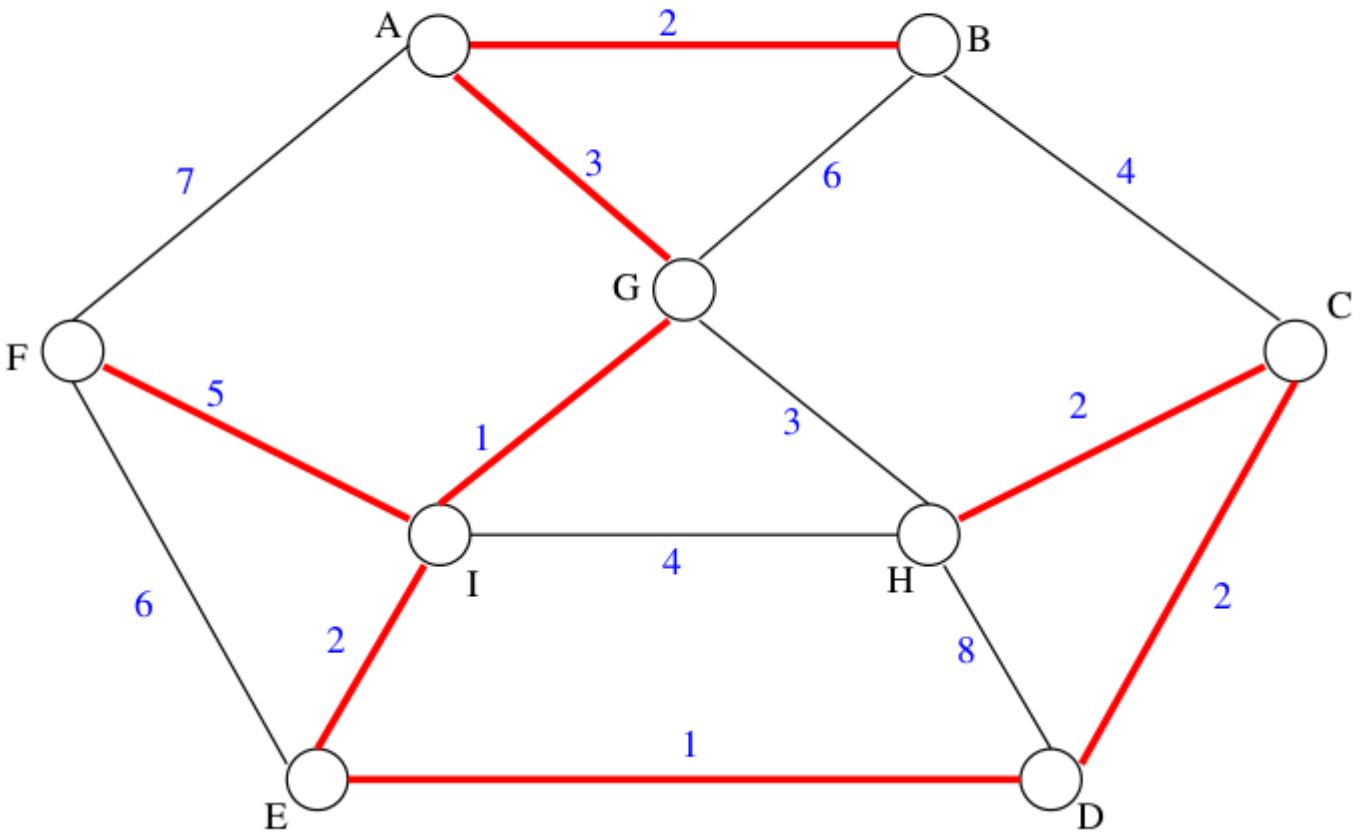
Algoritmo MST de Prim – Exemplo



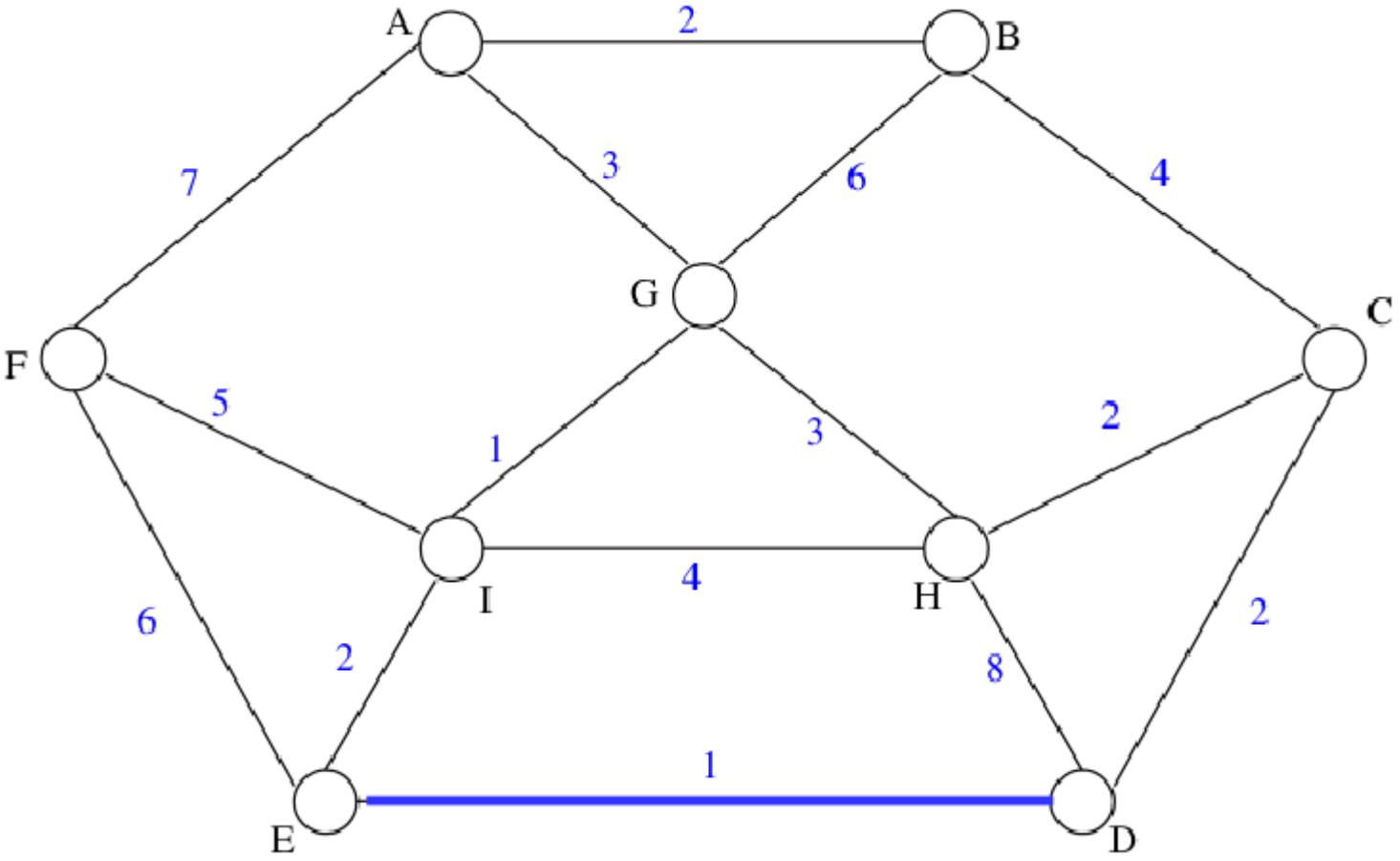
Algoritmo MST de Prim – Exemplo



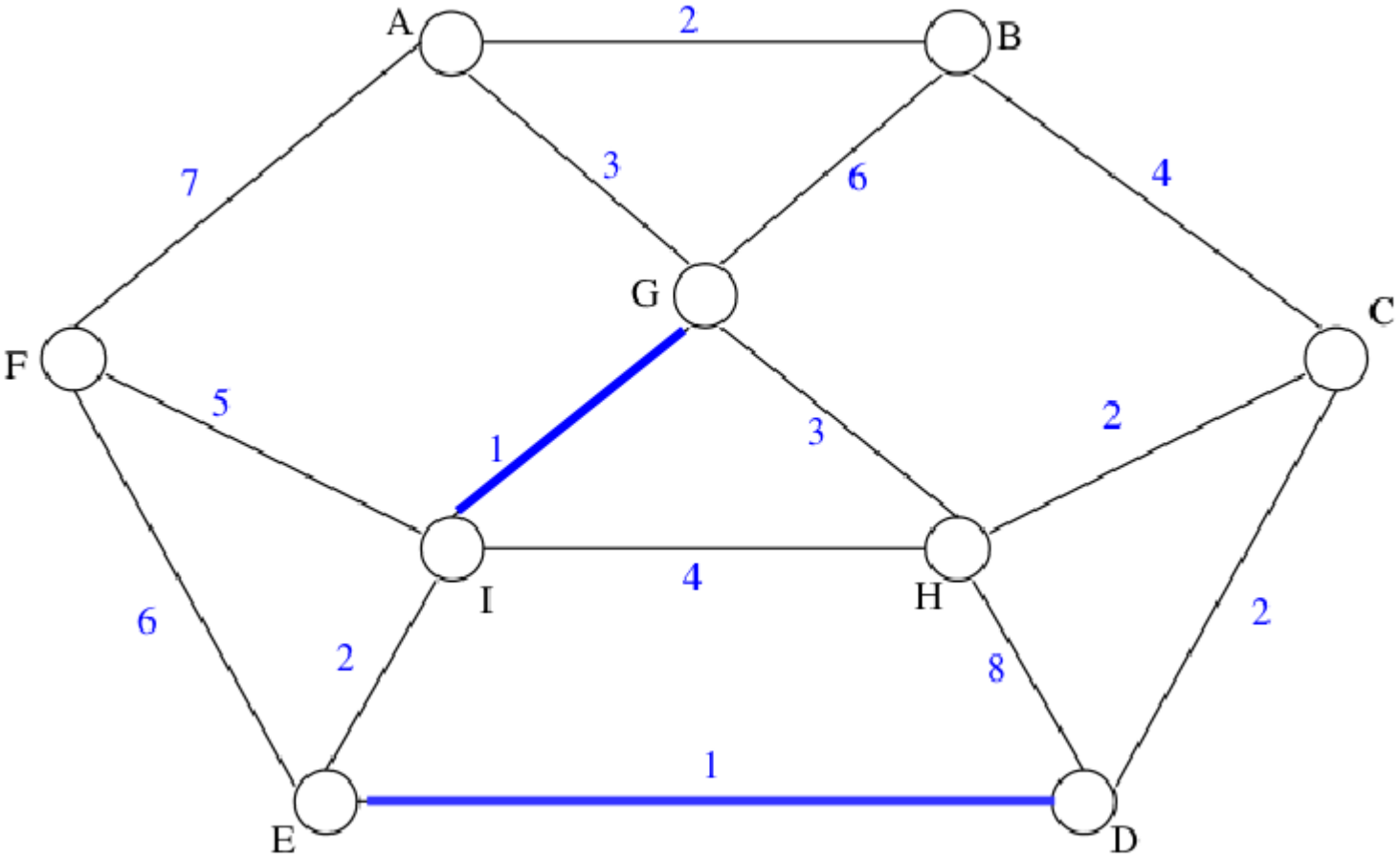
Algoritmo MST de Prim – Exemplo



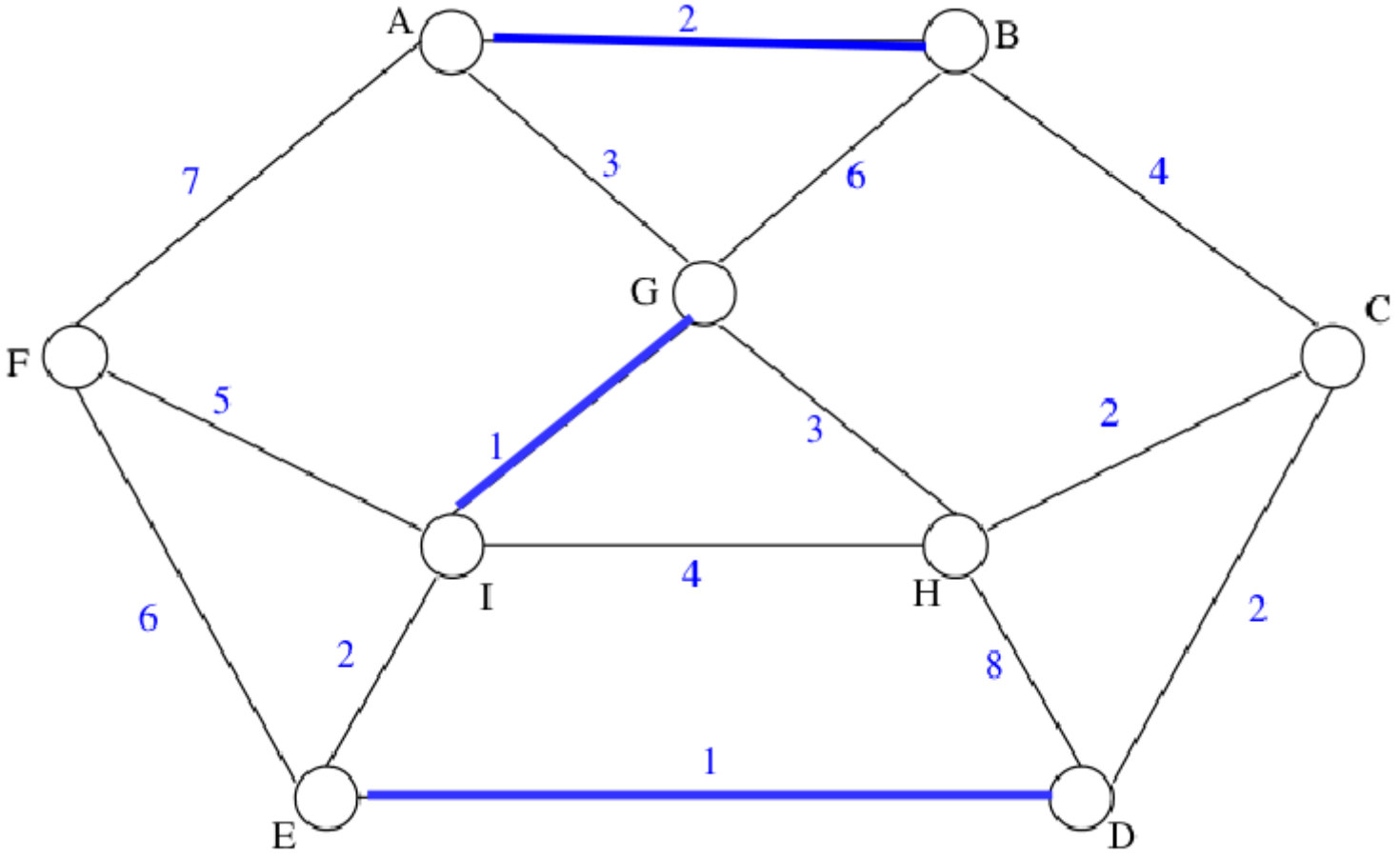
Algoritmo de Kruskal exemplo



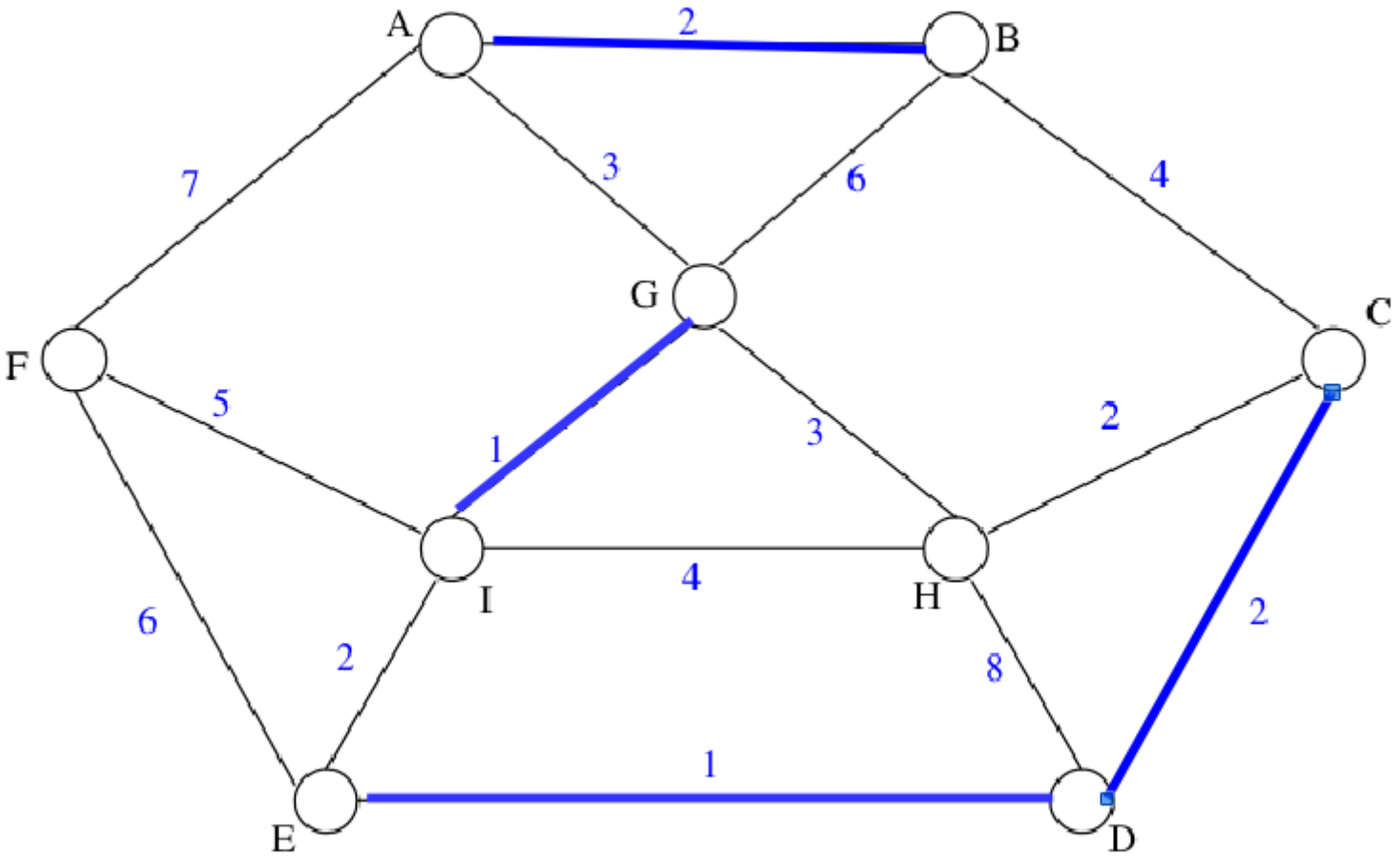
Algoritmo de Kruskal exemplo



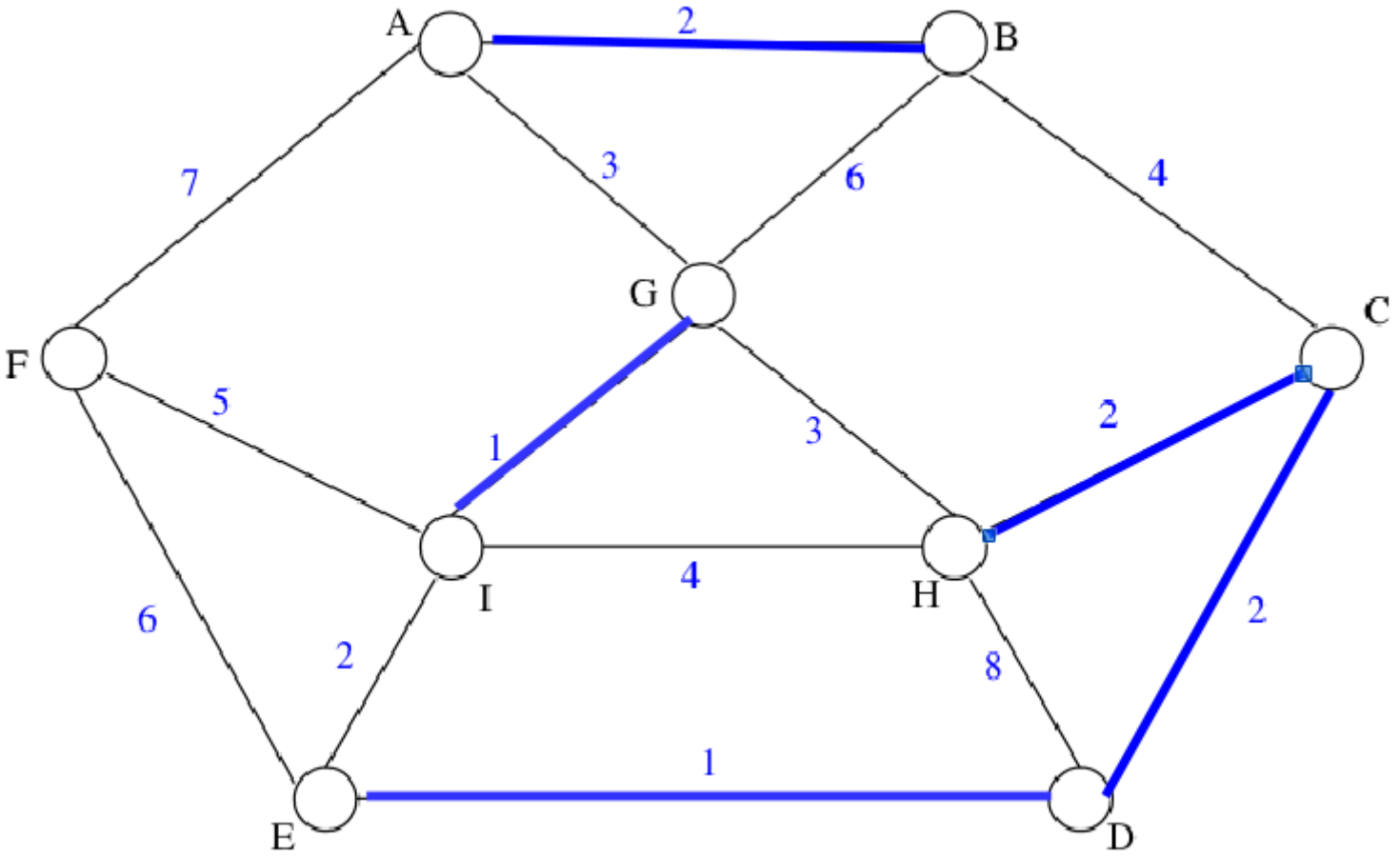
Algoritmo de Kruskal exemplo



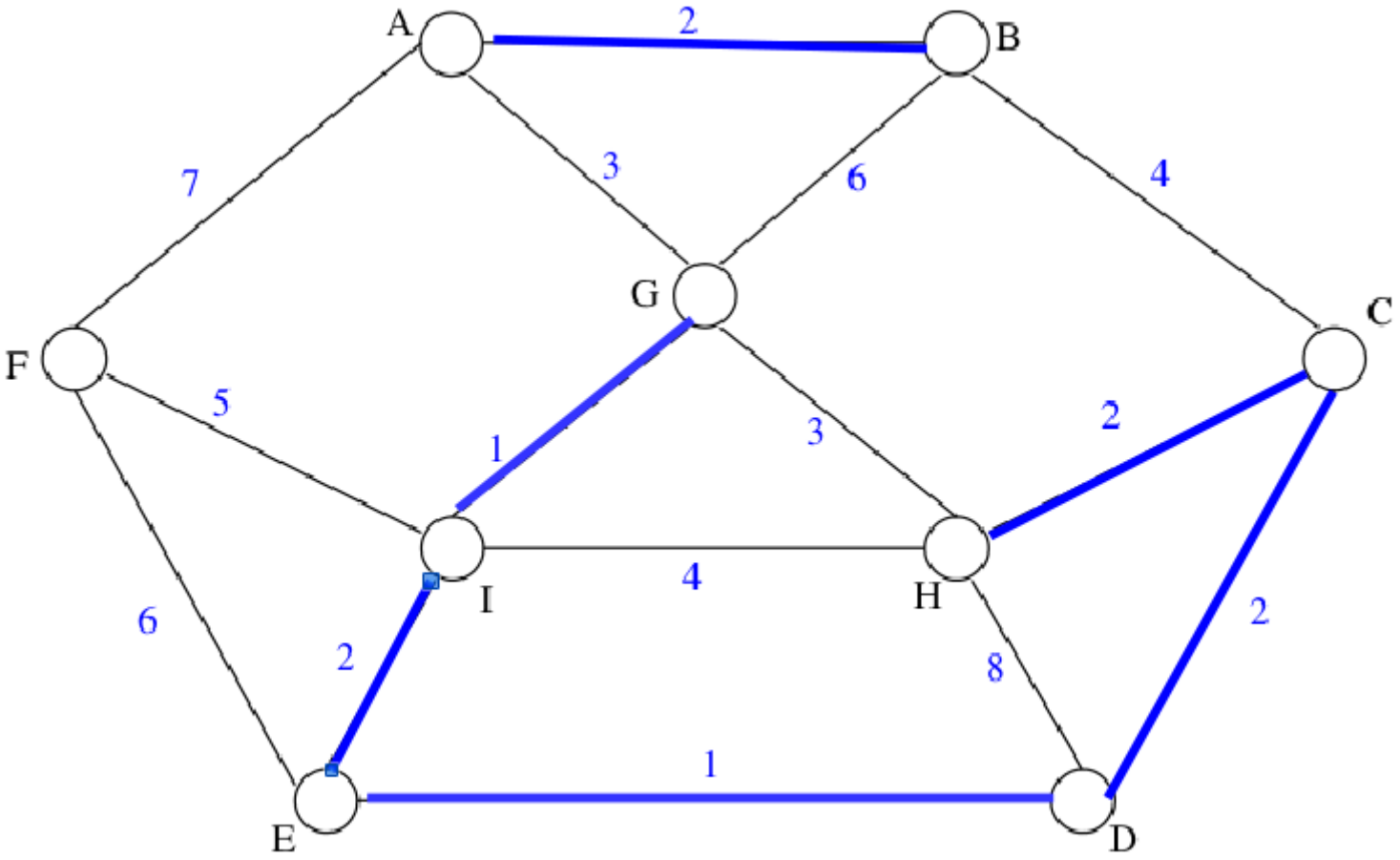
Algoritmo de Kruskal exemplo



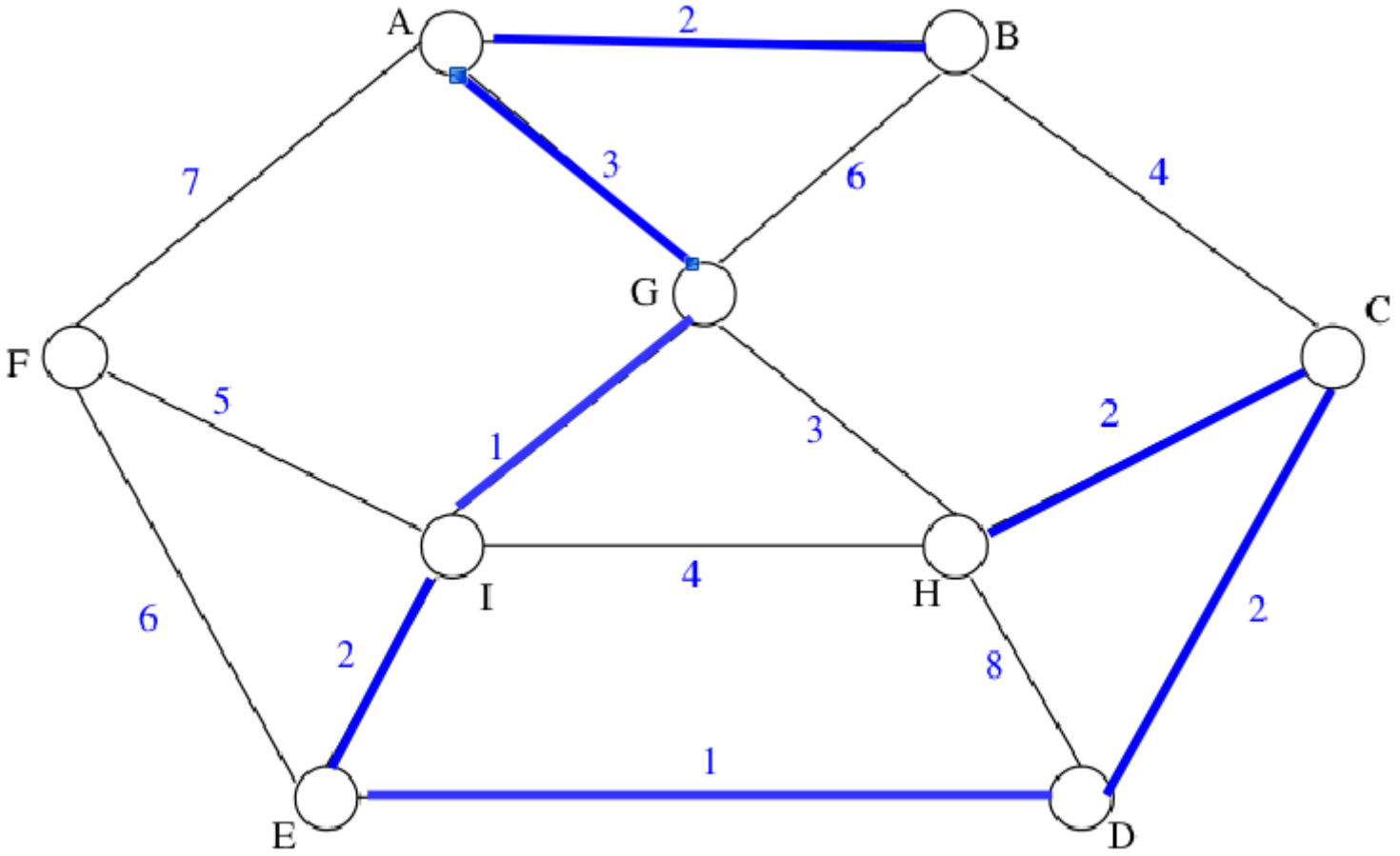
Algoritmo de Kruskal exemplo



Algoritmo de Kruskal exemplo



Algoritmo de Kruskal exemplo



Algoritmo de Kruskal exemplo