A10 Problema do Caminho Mínimo

Universidade Federal de Ouro Preto

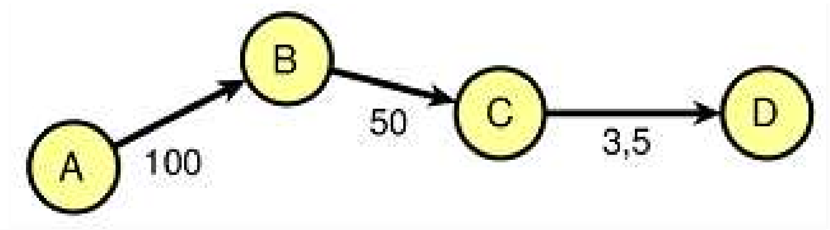
CSI466 – Teoria dos Grafos

Prof. Dr. George H. G. Fonseca
Universidade Federal de Ouro Preto

Introdução

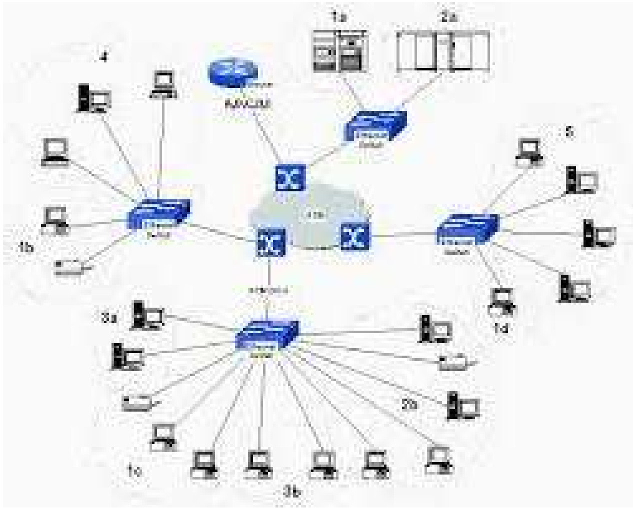
- Considere um grafo G = (V, E) orientado com função peso $w : E \rightarrow R$ que associa cada arco a um valor real $w(u, v)$
  - Obter o caminho de comprimento mínimo entre dois vértices *s* e *t*

- O comprimento de um caminho é a soma dos custos dos arcos que formam o caminho



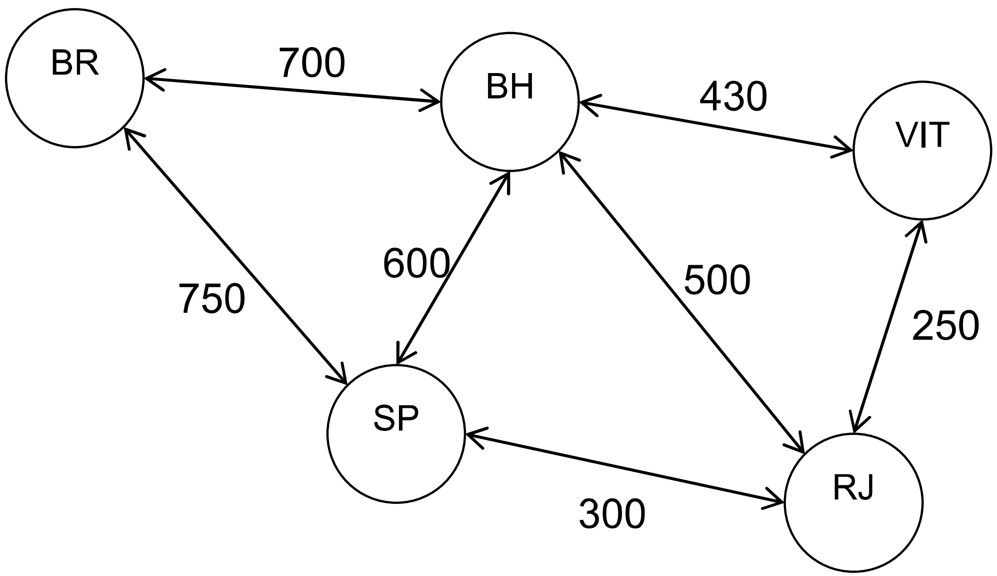
Introdução

- O custo de um arco pode ter várias interpretações de acordo com a aplicação
  - Distâncias
  - Consumo de combustível
  - Tempo
  - Tráfego
  - Custos



Introdução

- Deseja-se encontrar a rota mais rápida de uma cidade A até uma cidade B.
  - Cidades representam vértices
  - Estradas representam arestas
  - Custo de cada aresta indica o tempo necessário para se deslocar pela aresta

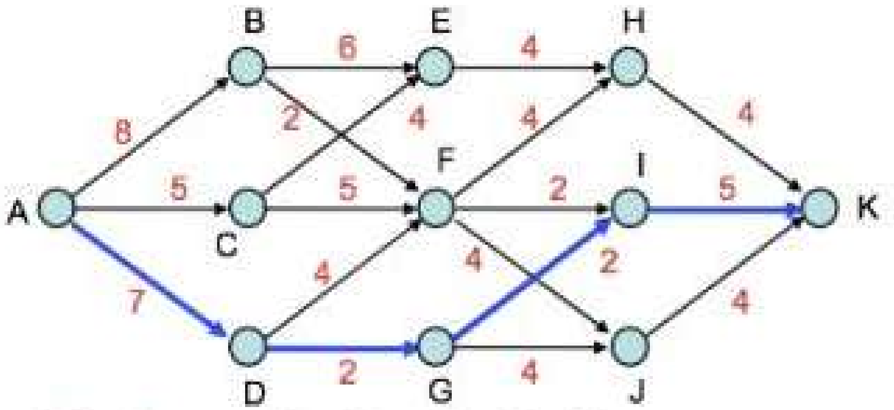


2

Introdução

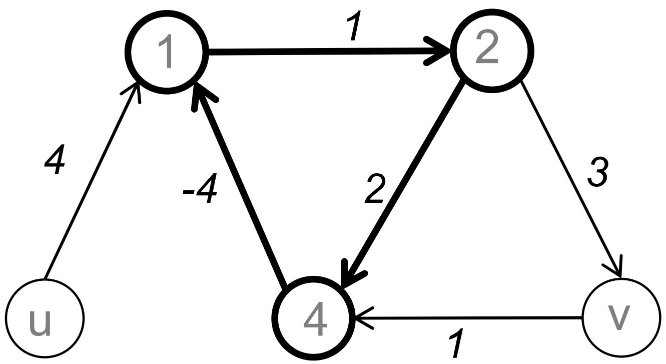
- Construção de estrada entre duas cidades A e K. O grafo abaixo representa os diversos trechos possíveis e o custo de construção de cada um

- Determinar o trajeto ótimo, cujo custo de construção seja mínimo (achar o caminho de menor custo de A a K)



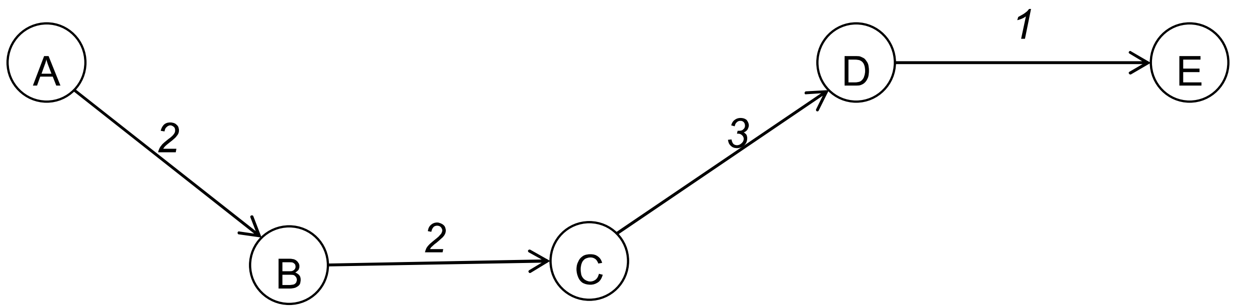
Solução: A – D – G – I – K
custo = 7 + 2 + 2 + 5 = 16

- Condição de existência: para que haja caminho mínimo entre dois vértices u e v, não pode existir no grafo circuito com custo negativo entre os vértices u e v



Teoremas

- Teorema: um subcaminho de um caminho mínimo é também um caminho mínimo



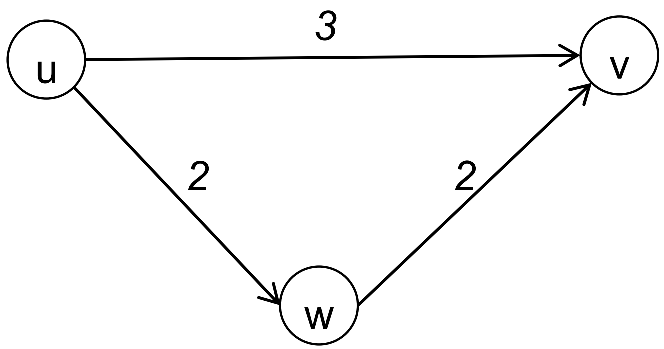
- Assumindo que o caminho entre A e E é mínimo, o caminho entre B e E também é mínimo!

Teoremas

- Teorema (Desigualdade triangular)
  - Uma aresta (u, v) compõe o caminho mínimo se e somente se para todo u, v e w ∊ V:

$$\mathrm{dist}(u, v) \leq \mathrm{dist}(u, w) + \mathrm{dist}(w, v)$$

  - Demonstração
    - Suponha que não seja verdadeiro. Então, a concatenação dos caminhos mínimos (*u*, *w*) e (*w*, *v*) forma um caminho de *u* a *v* menor do que o caminho mínimo de *u* a *v*.
    - Absurdo



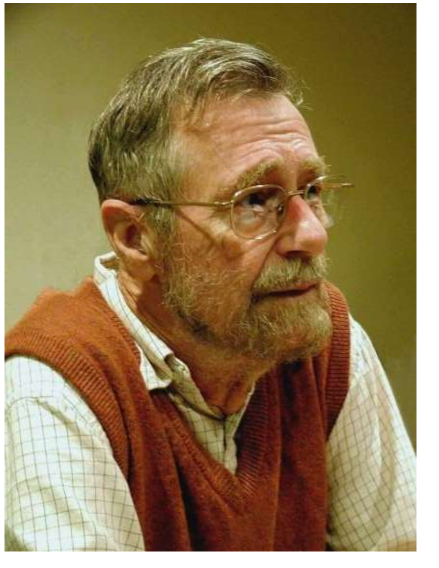
Caminho Mínimo

- As principais variações do problema do caminho mínimo são:
  - **Origem única**: encontrar um caminho mas curto desde uma determinada origem *s* até todo vértice *v*.
  - **Destino único**: encontrar um caminho mas curto até um determinado vértice de destino *t* a partir de cada vértice *v*.
  - **Par único**: encontrar um caminho mais curto de *u* até *v* para determinados vértices *u* e *v*.
  - **Todos pares**: encontrar um caminho mais curto desde *u* até *v*, para todo par de vértices *u* e *v*.

Algoritmo de Dijkstra

- O algoritmo de Dijkstra recebe um grafo orientado G = (V, E, *w*) (sem arestas de peso negativo) e um vértice *s* de G e armazena, para cada vértice v ∊ V, o custo de um caminho mínimo de *s* a *v*.

- Ideia: obter o caminho mínimo para um vértice por iteração até checar todos os vértices



Algoritmo de Dijkstra

- Trabalha com dois vetores:
  - dist[v] distância estimada para cada vértice v
  - pred[v] vértice predecessor ao vértice v no caminho mínimo da estimativa atual

- Abordagem gulosa!
  - Crie um conjunto S de vértices cujas distâncias para o vértice *s* são conhecidas
  - A cada iteração, acrescente a S o vértice $v \in V - S$ cuja distância estimada a *s* é mínima
  - Atualize as distâncias estimadas até *v*

Algoritmo de Dijkstra

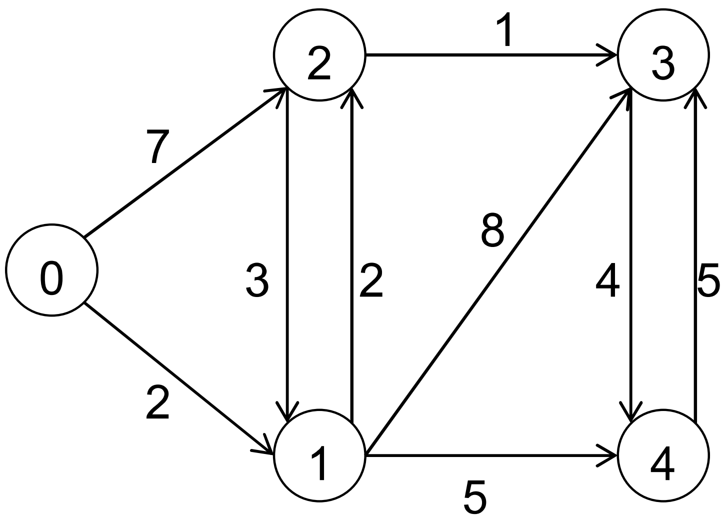
```
DIJKSTRA(G(V, E, w), s)        //Parâmetros: representação de grafo (V, E, w)
                               e vértice origem s
    para cada vértice v em V faça
        dist [v] ← ∞           //dist: vetor que armazena a distância da origem a cada vértice
        pred[v] ← null         //pred: vetor que indica o predecessor de cada vértice no caminho mínimo
                               a partir da origem
    fim-para
    dist[s] ← 0
    Q ← V                      //Q: conjunto (lista) de vértices a processar (inicialmente contem todos os vértices)
    enquanto Q ≠ Ø  faça       //Lista Q não é vazia
        u ← i : min{dist[i], ∀ i ∈ Q}   //u: vértice de menor distância (dist) dentre os vértices de Q
        Q ← Q – {u}            //Remover vértice u de Q (u foi processado)
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então   //w(u, v): peso da aresta (u, v)
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
para cada vértice v em V faça
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ Ø  faça
    u ← i : min{dist[i], ∀ i ∈ Q}
    Q ← Q – {u}
    para cada vértice v adjacente a u faça
        se dist[v] > dist[u] + w(u, v) então
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM
```

Complexidade de tempo $O(|V|^2)$
*pode ser melhorado!

Algoritmo de Dijkstra




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

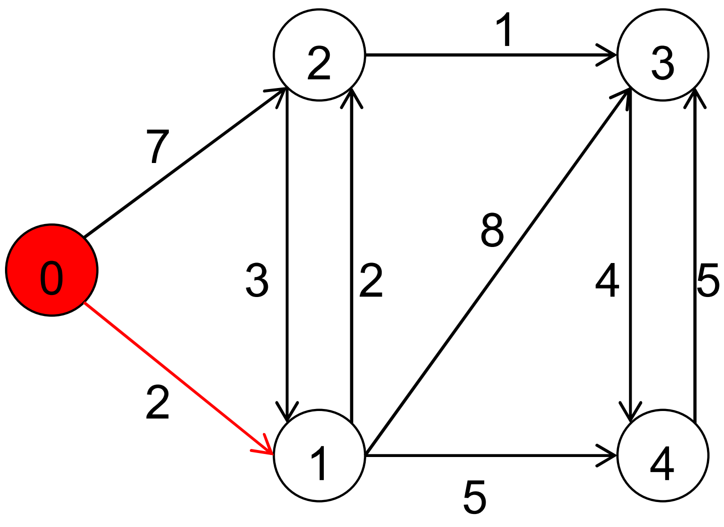
v	0	1	2	3	4
dist[v]	0	∞	∞	∞	∞
pred[v]	-	-	-	-	-

s = 0

Q = {0, 1, 2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



v	0	1	2	3	4
dist[v]	0	∞	∞	∞	∞
pred[v]	-	-	-	-	-

dist[1] > dist[0] + w(0, 1) ?

∞ > 0 + 2

Sim!

Atualize dist[1] e pred[1]

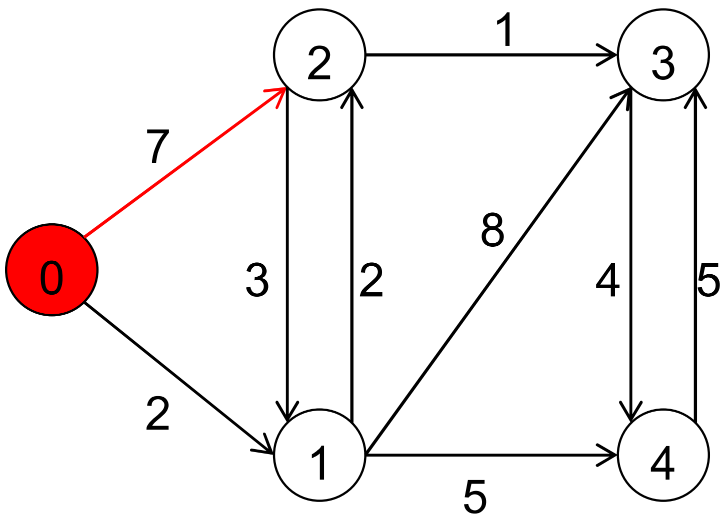
s = 0

u = 0

v = 1

Q = {1, 2, 3, 4}

Algoritmo de Dijkstra




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

dist[2] > dist[0] + w(0, 2) ?

$\infty > 0 + 7$

Sim!

Atualize dist[2] e pred[2]

v	0	1	2	3	4
dist[v]	0	2	∞	∞	∞
pred[v]	-	0	-	-	-

s = 0

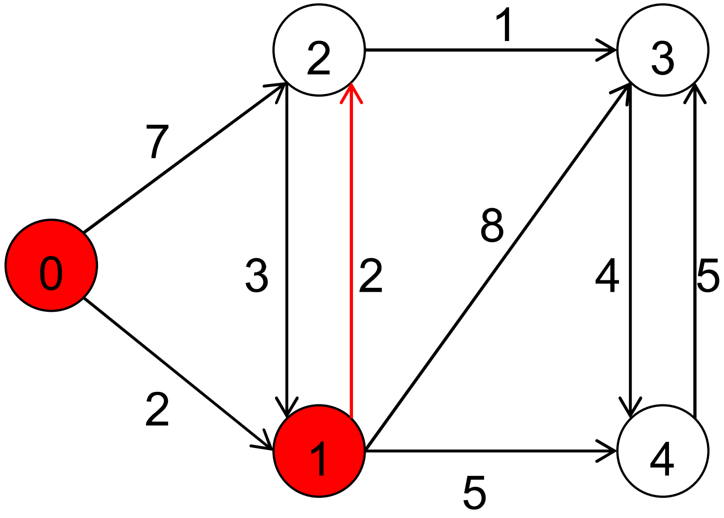
u = 0

v = 2

Q = {1, 2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



dist[2] > dist[1] + w(1, 2) ?

7 > 2 + 2

Sim!

Atualize dist[2] e pred[2]

v	0	1	2	3	4
dist[v]	0	2	7	∞	∞
pred[v]	-	0	0	-	-

s = 0

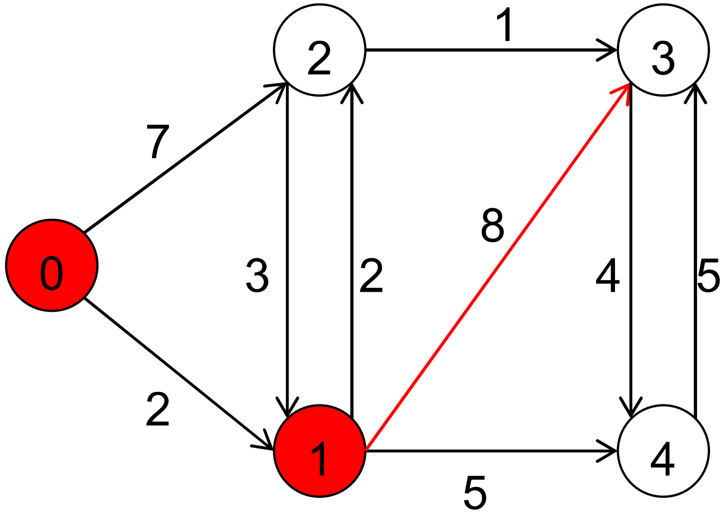
u = 1

v = 2

Q = {2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



v	0	1	2	3	4
dist[v]	0	2	4	∞	∞
pred[v]	-	0	1	-	-

dist[3] > dist[1] + w(1, 3) ?

$\infty > 2 + 8$

Sim!

Atualize dist[3] e pred[3]

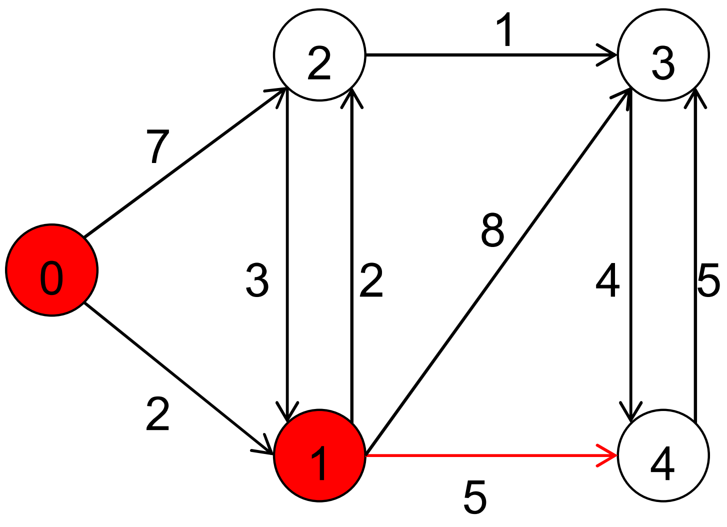
s = 0

u = 1

v = 3

Q = {2, 3, 4}

Algoritmo de Dijkstra




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

dist[4] > dist[1] + w(1, 4) ?

$\infty > 2 + 5$

Sim!

Atualize dist[4] e pred[4]

v	0	1	2	3	4
dist[v]	0	2	4	10	∞
pred[v]	-	0	1	1	-

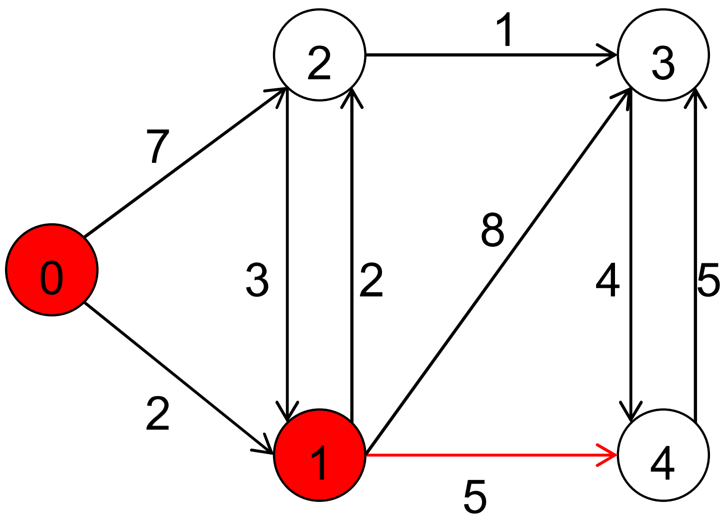
s = 0

u = 1

v = 3

Q = {2, 3, 4}

Algoritmo de Dijkstra



DIJKSTRA(G(V, E, w), s)

para cada vértice v em V **faça**
    dist [v] ← ∞
    pred[v] ← null
fim-para
dist[s] ← 0
Q ← V
enquanto Q ≠ ∅ faça
    u ← i : min{dist[i], ∀ i ∈ Q}
    Q ← Q – {u}
    para cada vértice v adjacente a u **faça**
        se dist[v] > dist[u] + w(u, v) **então**
            dist[v] ← dist[u] + w(u, v)
            pred[v] ← u
        fim-se
    fim-para
fim-enquanto
FIM

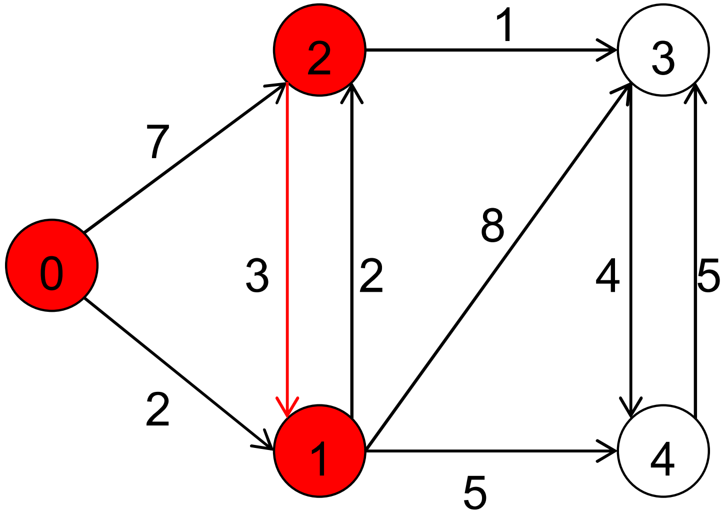
dist[4] > dist[1] + w(1, 4) ?
∞ > 2 + 5
Sim!
Atualize dist[4] e pred[4]

v	**0**	**1**	**2**	**3**	**4**
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

s = 0
u = 1
v = 3
Q = {2, 3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



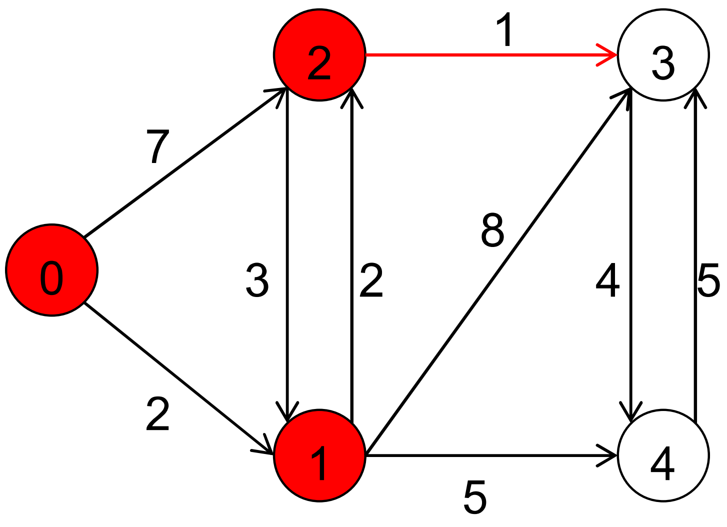
dist[1] > dist[2] + w(2, 1) ?
2 > 4 + 3
Não!

v	0	1	2	3	4
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

s = 0
u = 2
v = 1
Q = {3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



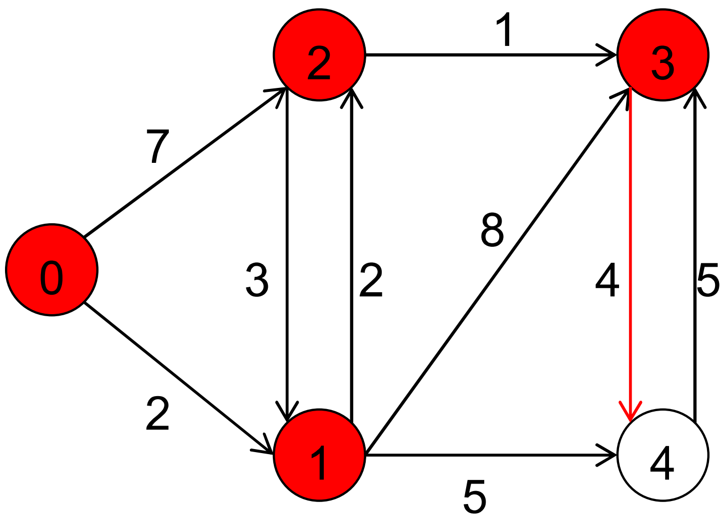
dist[3] > dist[2] + w(2, 3) ?
10 > 4 + 1
Sim!
Atualize dist[3] e pred[3]

v	0	1	2	3	4
dist[v]	0	2	4	10	7
pred[v]	-	0	1	1	1

s = 0
u = 2
v = 1
Q = {3, 4}

Algoritmo de Dijkstra

```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```



dist[4] > dist[3] + w(3, 4) ?

7 > 5 + 4

Não!

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

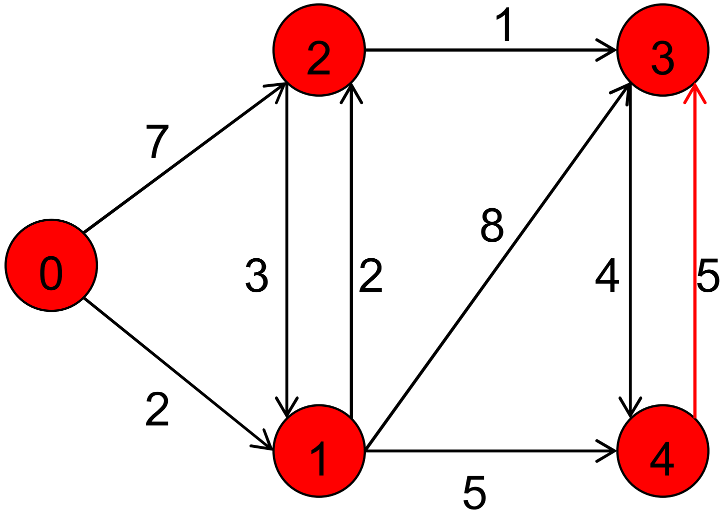
s = 0

u = 3

v = 4

Q = {4}

Algoritmo de Dijkstra



dist[3] > dist[4] + w(4, 3) ?
5 > 7 + 5
Não!

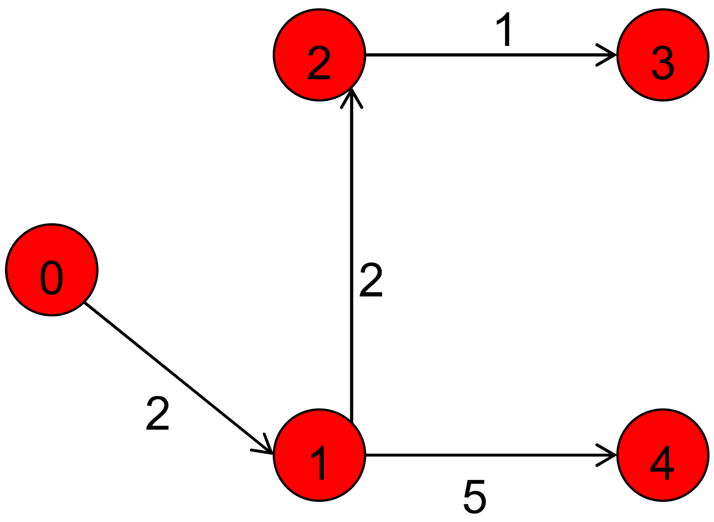
```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
u = 4
v = 3
Q = {}

Algoritmo de Dijkstra

Arestas dos caminhos mínimos:




```
DIJKSTRA(G(V, E, w), s)
    para cada vértice v em V faça
        dist [v] ← ∞
        pred[v] ← null
    fim-para
    dist[s] ← 0
    Q ← V
    enquanto Q ≠ ∅  faça
        u ← i : min{dist[i], ∀ i ∈ Q}
        Q ← Q – {u}
        para cada vértice v adjacente a u faça
            se dist[v] > dist[u] + w(u, v) então
                dist[v] ← dist[u] + w(u, v)
                pred[v] ← u
            fim-se
        fim-para
    fim-enquanto
FIM
```

v	0	1	2	3	4
dist[v]	0	2	4	5	7
pred[v]	-	0	1	2	1

s = 0
u = 4
v = 3
Q = {}   Fim do algoritmo!

BCC204 - Teoria dos Grafos

Marco Antonio M. Carvalho

(baseado nas notas de aula do prof. Haroldo Gambini Santos)
Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

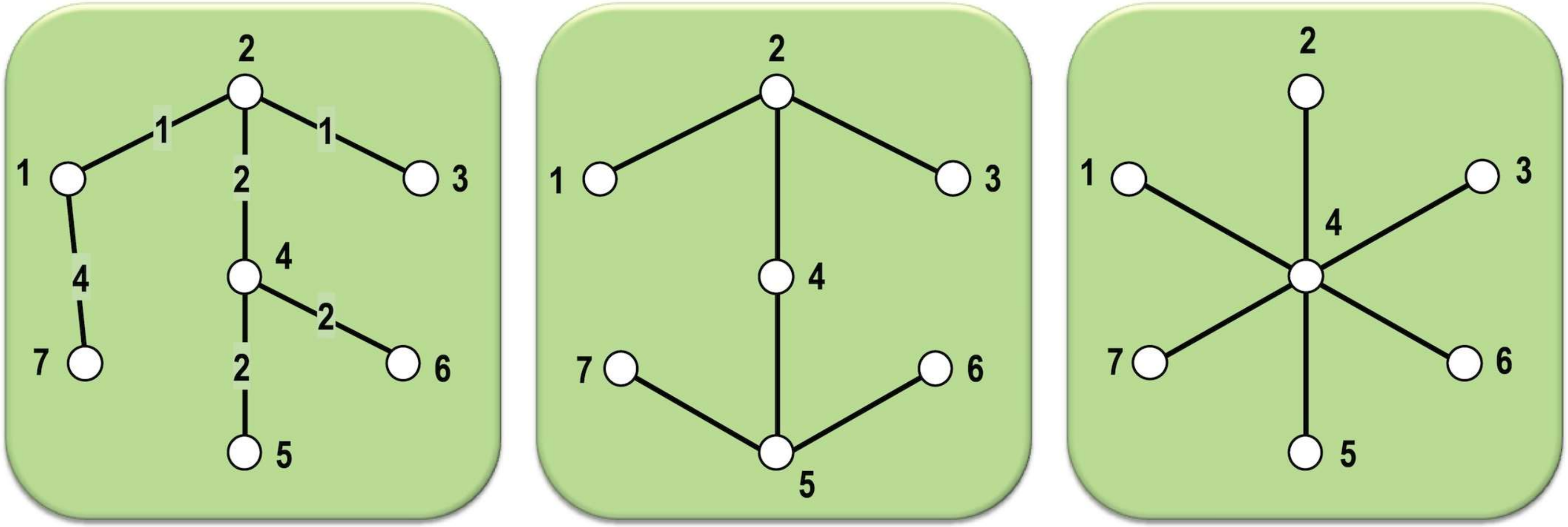
16 de outubro de 2019

Árvores

Definição

Grafo conexo e sem ciclos em que há somente um caminho entre qualquer par de vértices.

Um subgrafo conexo e acíclico de uma árvore é denominado subárvore.

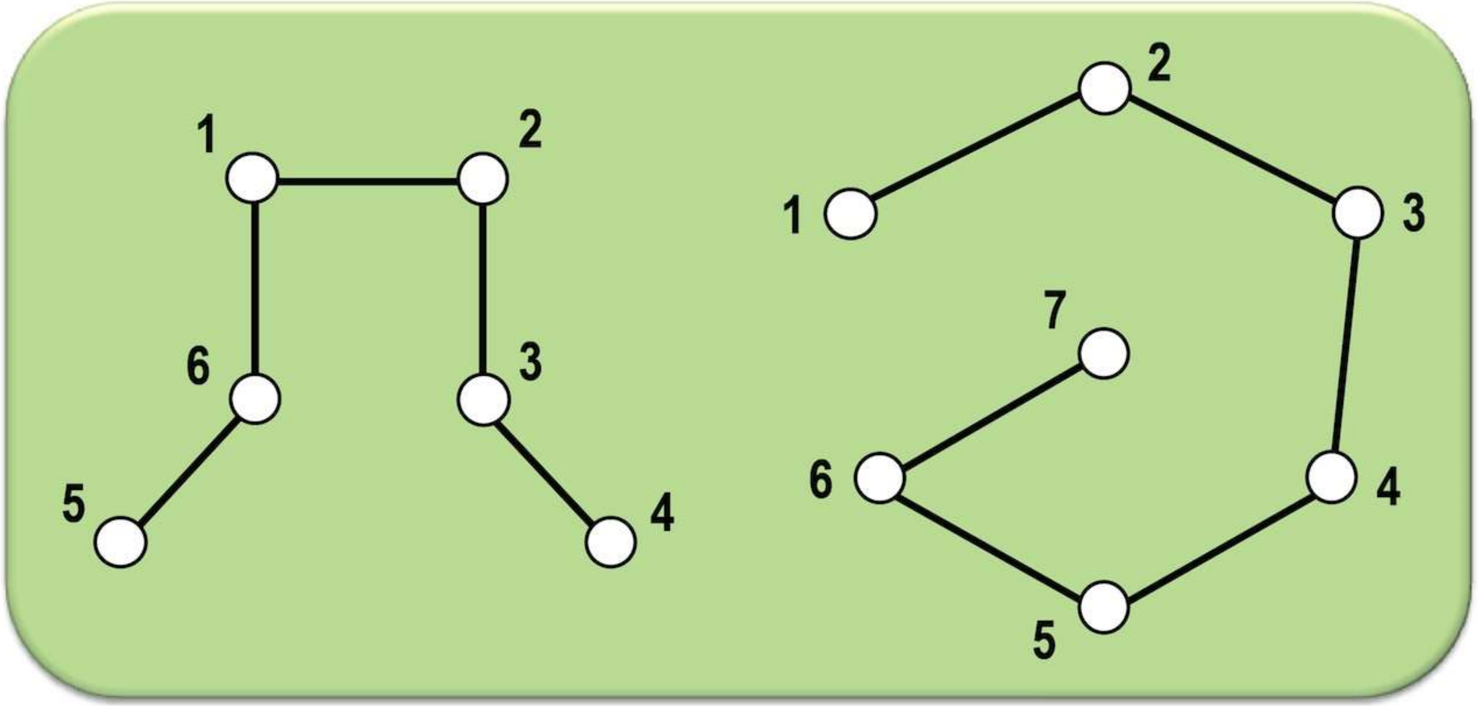


Árvore ponderada, árvore não ponderada e grafo estrela.

Grafo Caminho

Definição

Um grafo caminho (ou grafo linha) é um caso especial de árvore em que todos os vértices têm grau 2 ou 1, havendo apenas dois vértices com grau 1.



Grafos caminho.

Árvores

Características

Seja T uma árvore com n vértices, então:

I. T é conexo e sem ciclos;
II. T possui $n-1$ arestas;
III. Cada aresta de T é uma ponte;
IV. T é um grafo planar;
V. Se $n>1$, então T possui pelo menos dois vértices folhas (ou terminais).

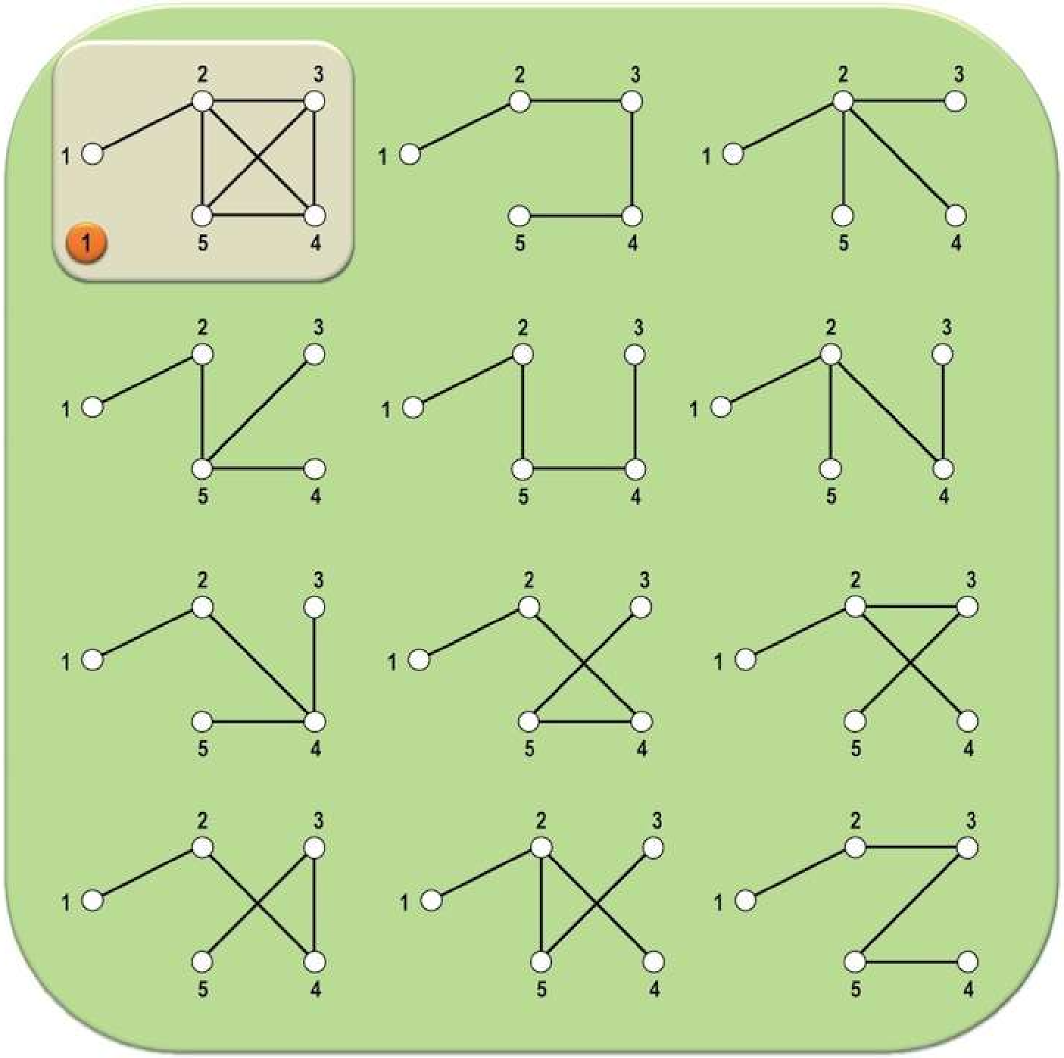
Definição

Todo grafo G conexo possui pelo menos uma árvore que contém todos os seus vértices.

Uma árvore geradora de um grafo G é um subgrafo conexo e acíclico que possui todos os vértices originais de G e um subconjunto das arestas originais de G.

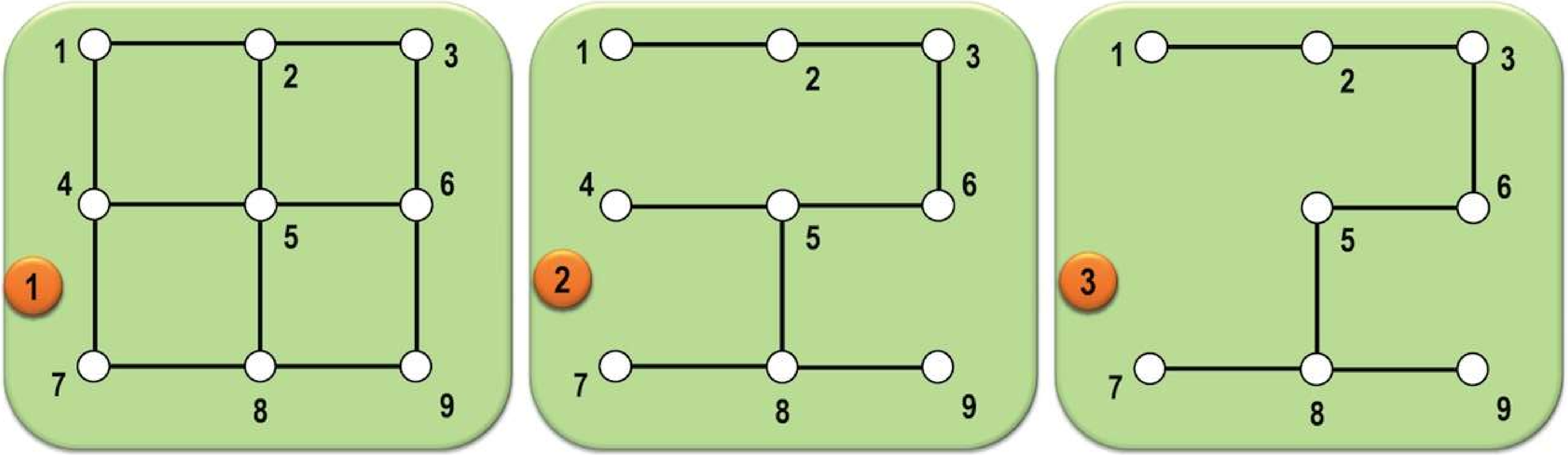
Como consequência das propriedades de uma árvore, todo grafo conexo possui pelo menos uma árvore geradora.

Árvores Geradoras



Grafo de exemplo e árvores geradoras.

Árvores Geradoras



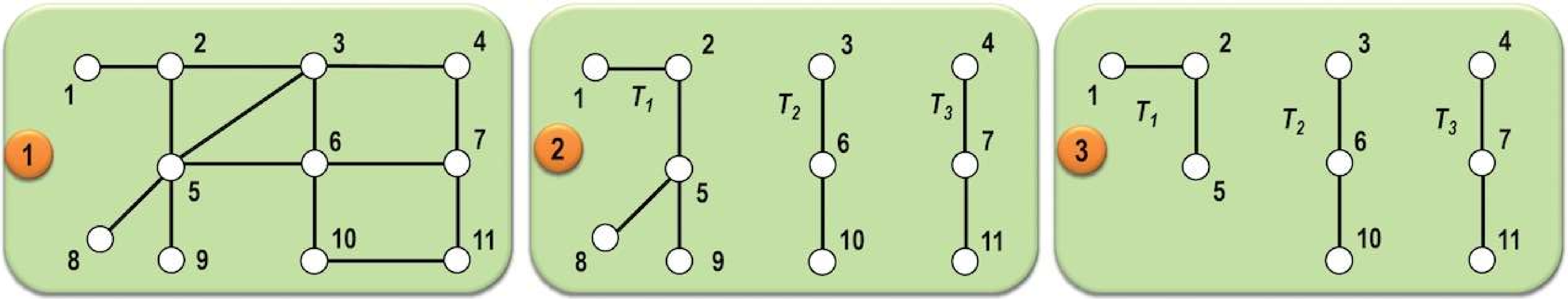
Grafo de exemplo, árvore geradora e uma árvore não geradora.

Florestas

Definições

Uma floresta é um conjunto de árvores sem vértices em comum.

Uma floresta geradora é uma floresta que contém todos os vértices de um grafo.



Grafo de exemplo e florestas. A primeira floresta é geradora.

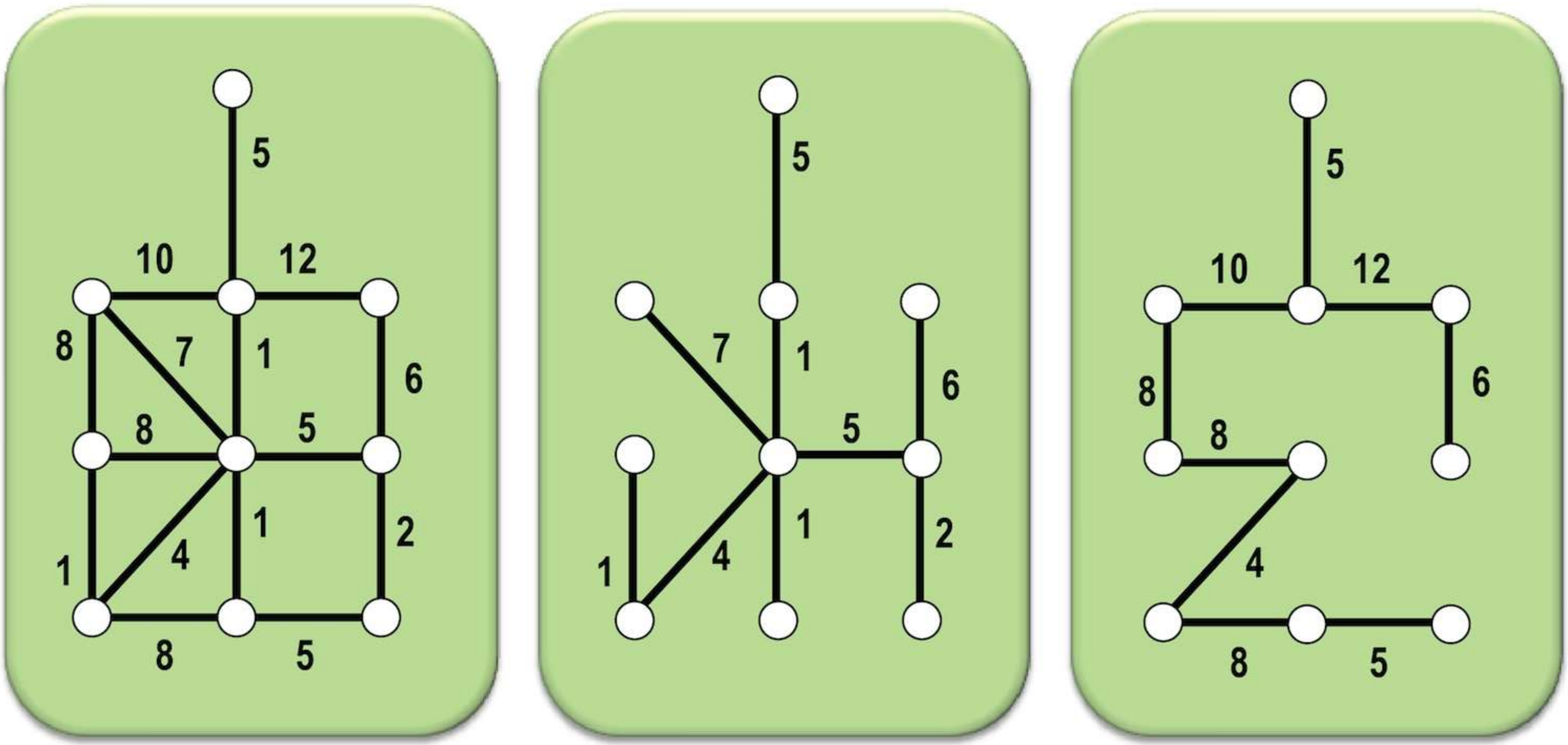
Árvore Geradora de Custo Mínimo e Máximo

Definição

A árvore geradora de custo mínimo é a árvore geradora de menor custo dentre todas as possíveis em um grafo.

Analogamente, árvore geradora de custo máximo é a árvore geradora de maior custo dentre todas as possíveis em um grafo.

A determinação de ambas as árvores descritas pode ser feita em tempo **determinístico polinomial** por algoritmos gulosos.



Grafo de exemplo, árvore geradora de custo mínimo e árvore geradora de custo máximo.

Algoritmos

Evolução da Complexidade

Um dos primeiros algoritmos para determinação de árvores geradoras mínimas data do ano de 1928.

De lá para cá, a complexidade dos algoritmos evoluiu de $O(m \log n)$ para $O(m)$, cuja implementação data de 2008.

Os Básicos

Dois dos algoritmos mais populares para determinação de árvores geradoras mínimas, ambos gulosos, remetem ao final da década de 50: o algoritmo de **Prim** e o Algoritmo de **Kruskal**.

O Algoritmo de Prim

Histórico

Este algoritmo foi proposto originalmente em 1930 pelo matemático tcheco Vojtěch Jarník, posteriormente pelo cientista da computação americano Robert C. Prim ($\star$ 1921 † 2009) em 1957 e redescoberto posteriormente pelo holandês Edsger Dijkstra em 1959.

Princípio

Incluir, de forma gulosa, um a um, os vértices da árvore geradora mínima.

O algoritmo parte de qualquer vértice do grafo e, a cada passo, acrescenta a aresta de menor peso incidente ao conjunto de vértices que já foram selecionados e que possui uma extremidade em vértices no conjunto de não selecionados.

Algoritmo de Prim

Terminologia

- T_{min}: Conjunto de arestas que define a árvore geradora mínima;
- T: Conjunto dos vértices já selecionados pelo algoritmo;
- N: Conjunto dos vértices não selecionados pelo algoritmo;
- $\setminus$: subtração em conjuntos.

Algoritmo de Prim

Entrada: Grafo $G = (V, A)$ e matriz de pesos D=$\{d_{ij}\}$ para todas as arestas $\{i, j\}$

Escolha qualquer vértice $i \in V$;

$T \leftarrow \{i\}$;

$N \leftarrow V \setminus i$;

$T_{min} \leftarrow \emptyset$;

enquanto $|T| \neq n$ **faça**

  Encontre a aresta $\{j, k\} \in A$ tal que $j \in T$, $k \in N$ e d_{jk} é mínimo;

  $T \leftarrow T \cup \{k\}$;

  $N \leftarrow N \setminus \{k\}$;

  $T_{min} \leftarrow T_{min} \cup \{j, k\}$;

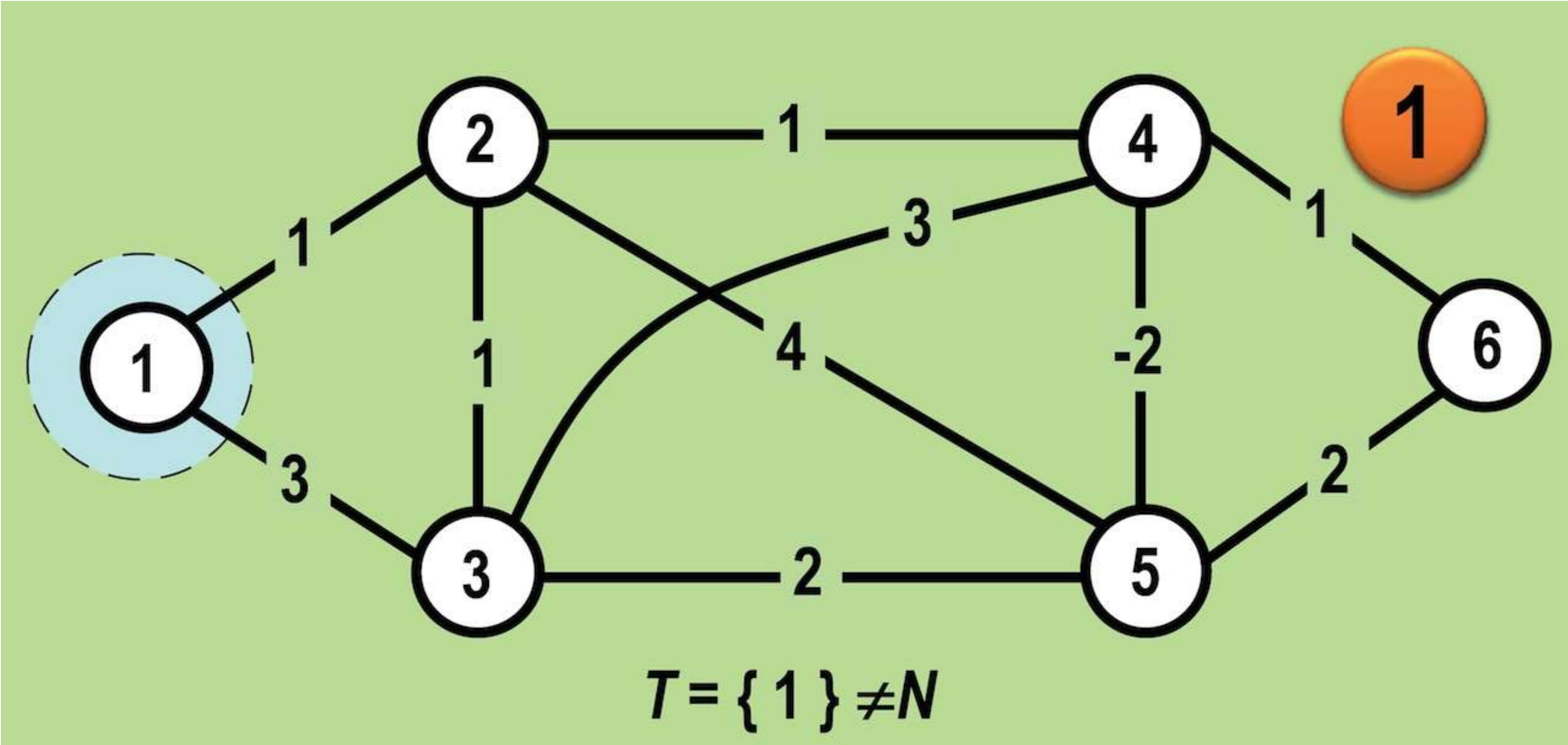
fim

Algoritmo de Prim

Complexidade

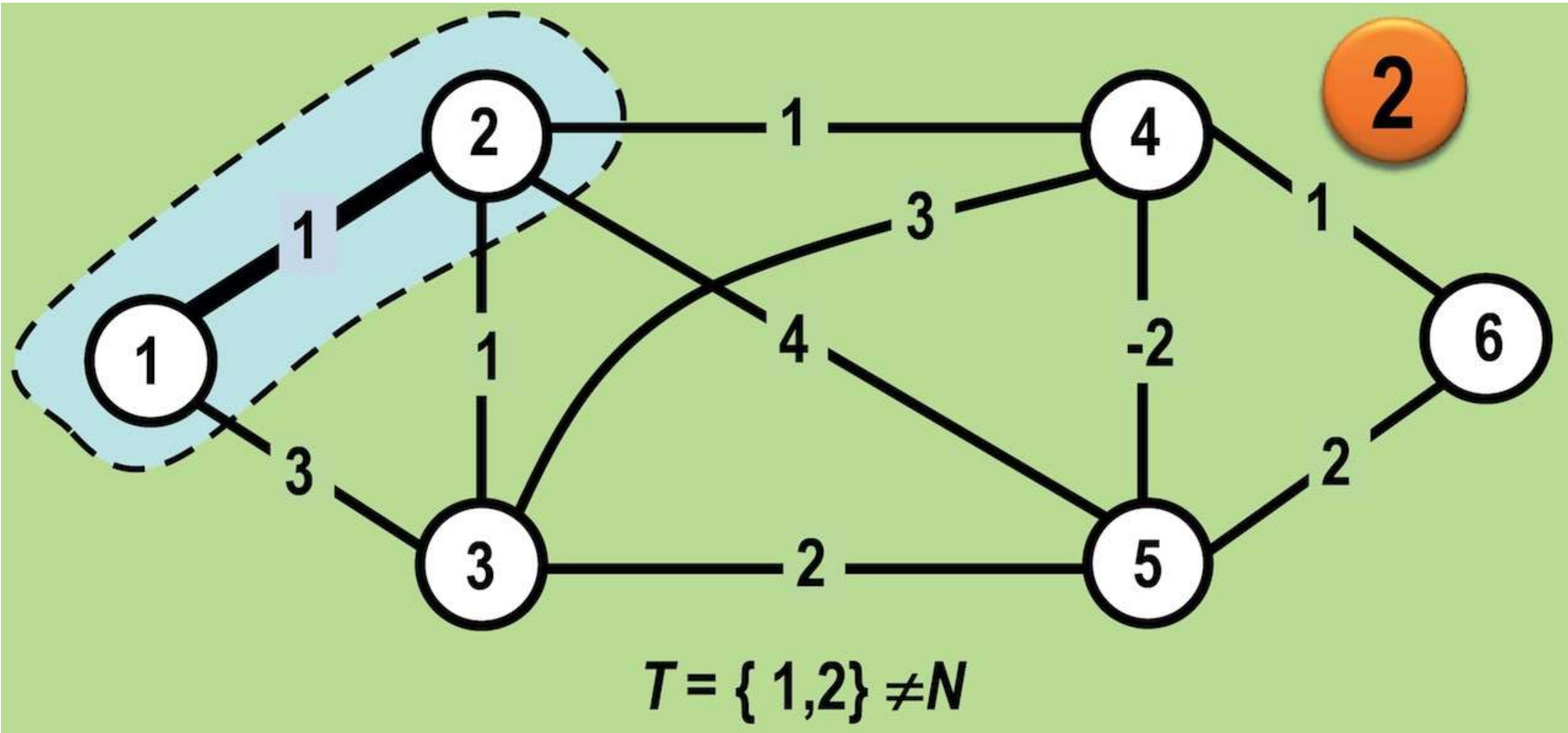
- Utilizando-se uma matriz de adjacências e uma busca linear na mesma, a complexidade é $O(n^2)$, por conta da aplicação repetidas vezes do procedimento que encontra a aresta de peso mínimo;
- Usando heaps binárias, o algoritmo pode ser implementado em $O(m \log n)$;
- Usando heaps de Fibonacci, o algoritmo pode ser implementado em $O(n \log n + m)$.

Exemplo

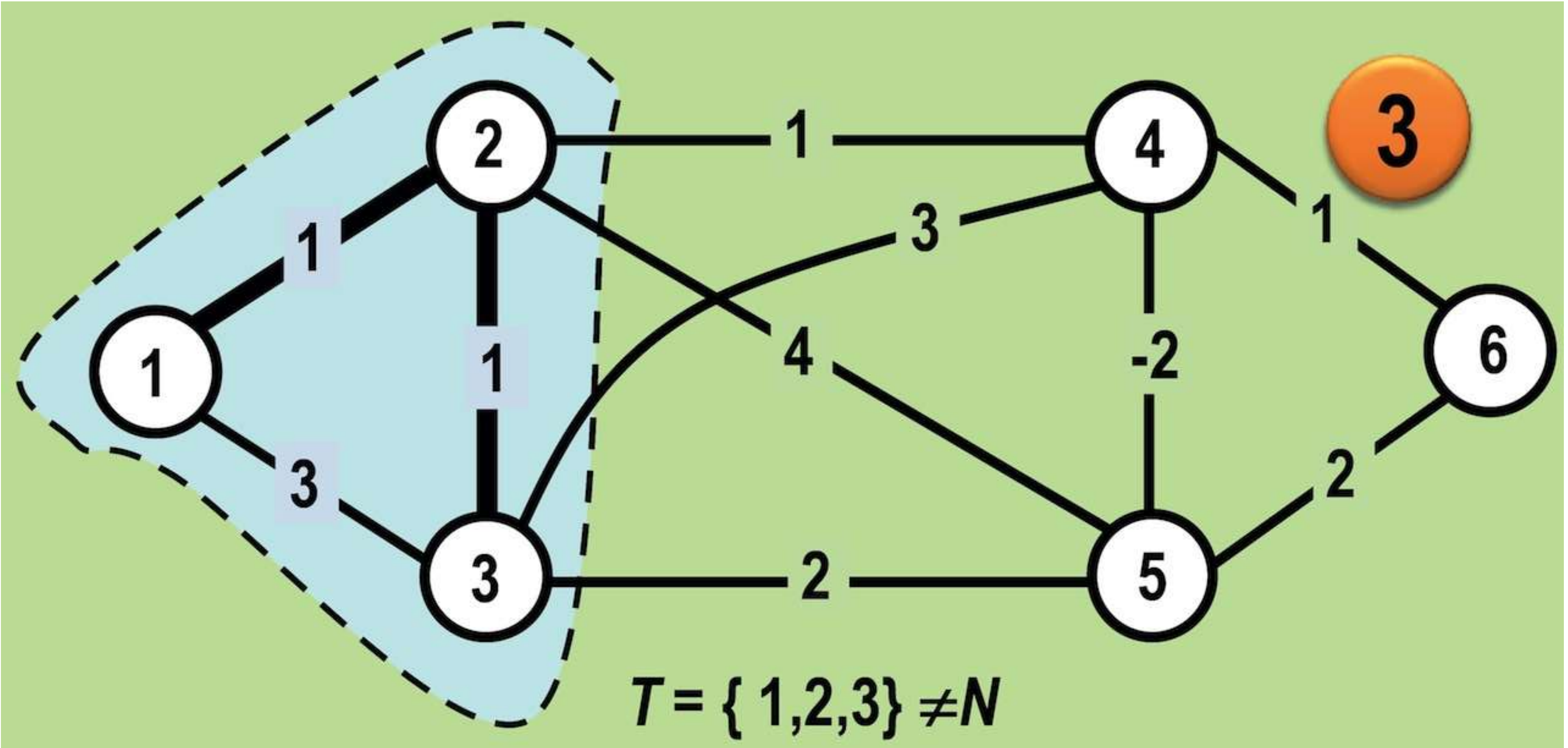


Grafo de exemplo. O vértice 1 é o primeiro a ser escolhido.

Exemplo

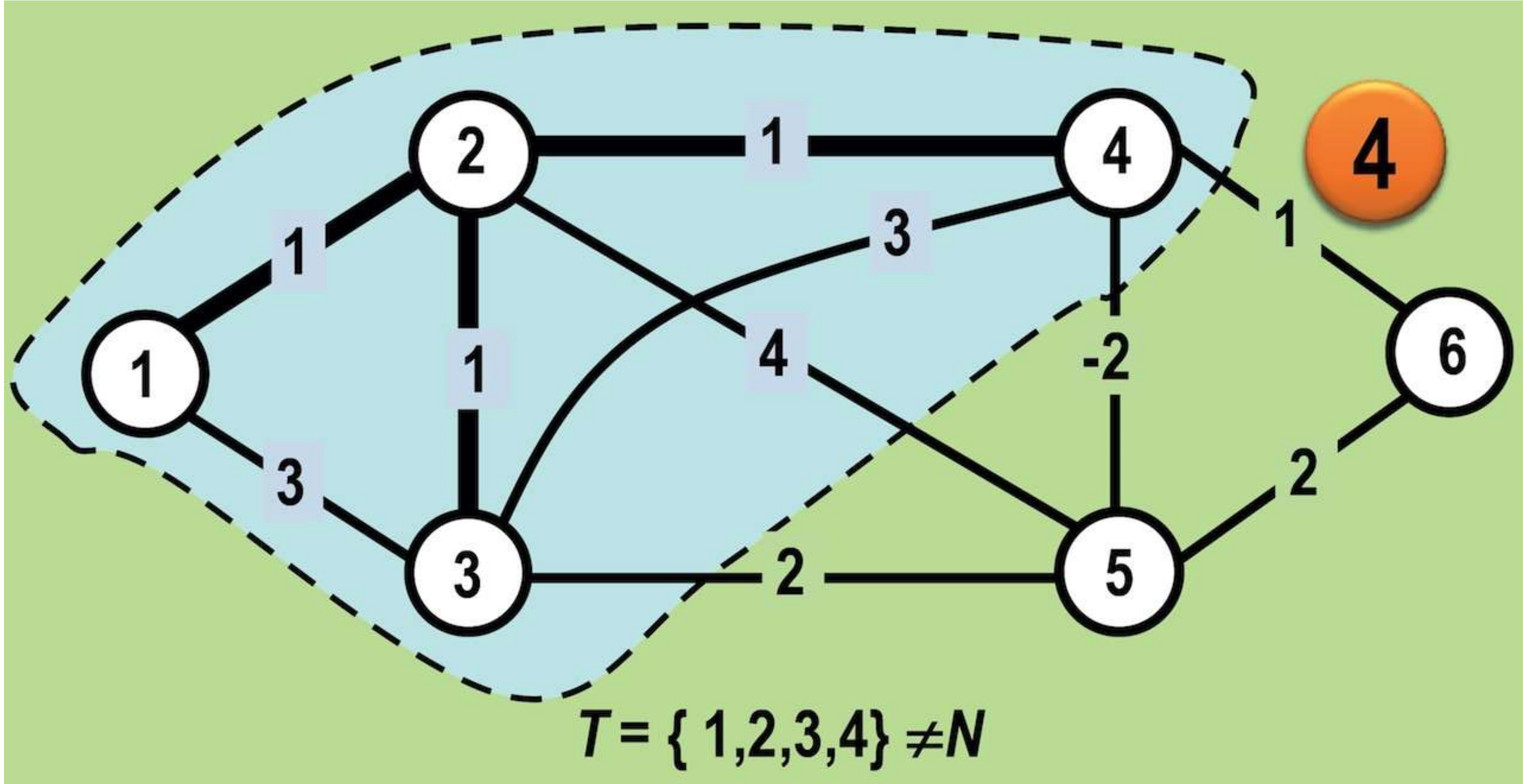


Inserção do vértice 2 e da aresta $\{1, 2\}$.
A região em azul indica os vértices escolhidos.



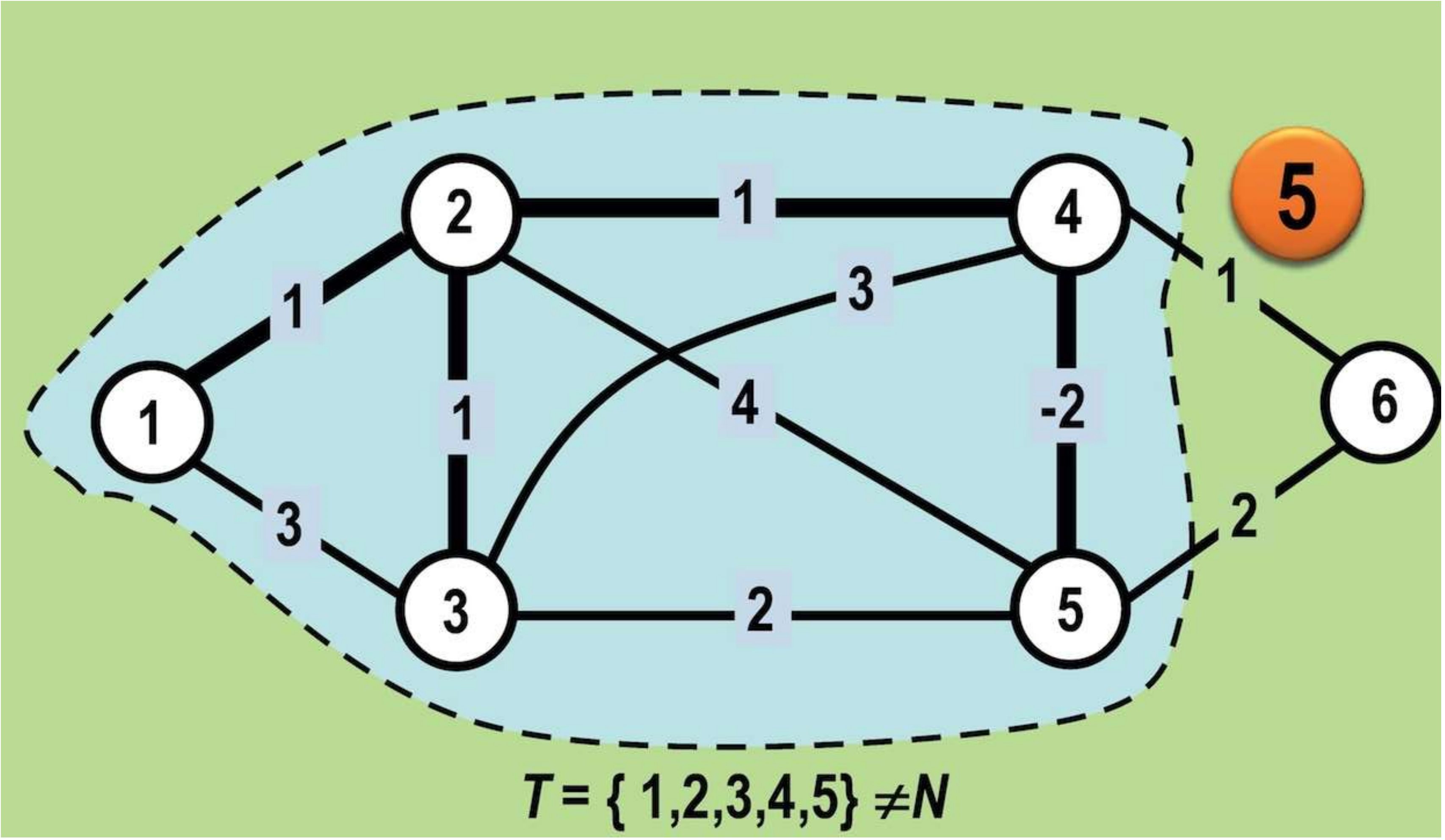
Inserção do vértice 3 e da aresta {2, 3}.

Exemplo



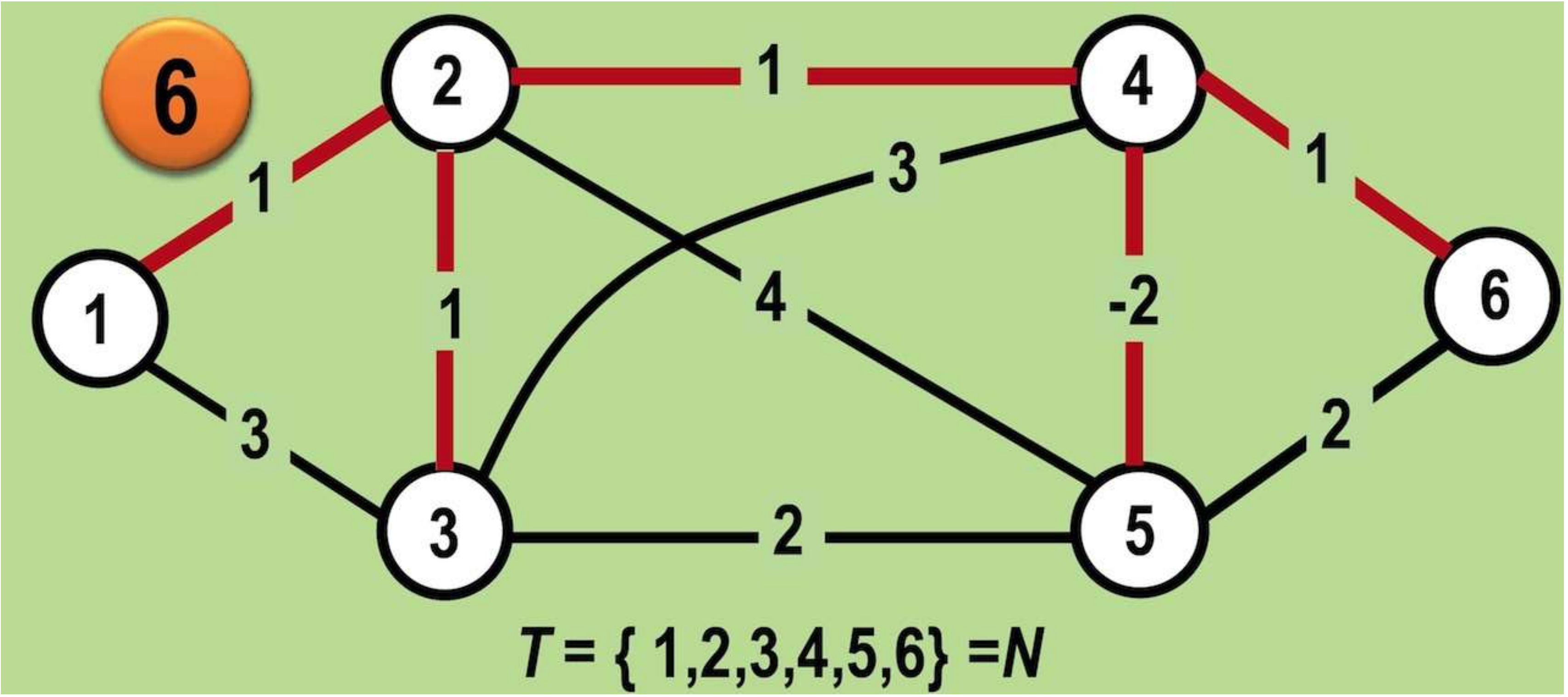
Inserção do vértice 4 e da aresta {2, 4}.

Exemplo



Inserção do vértice 5 e da aresta {4, 5}.

Exemplo



Inserção do vértice 6 e da aresta $\{4, 6\}$.
A árvore geradora mínima foi determinada.

O Algoritmo de Kruskal

Histórico

Este algoritmo foi proposto em 1956 por Joseph Bernard Kruskal Jr. ($\star$ 1928 † 2010), estatístico, matemático, cientista da computação e psicometrista americano.

Princípio

Incluir na árvore, a cada iteração, a aresta de menor custo que não formar ciclo.

Consequemente, processar $n - 1$ iterações;

O raciocínio está voltado para a formação da árvore a partir da inclusão de arestas, e não de vértices, como no algoritmo de Prim.

Algoritmo de Kruskal

Terminologia

- H: Vetor de arestas, ordenadas de acordo com os pesos;
- T: Conjunto de arestas que define a árvore geradora mínima;
- $\cup$: união em conjuntos.

Algoritmo de Kruskal

Entrada: Grafo $G = (V, A)$ e matriz de pesos D=$\{d_{ij}\}$ para todas as arestas $\{i, j\}$

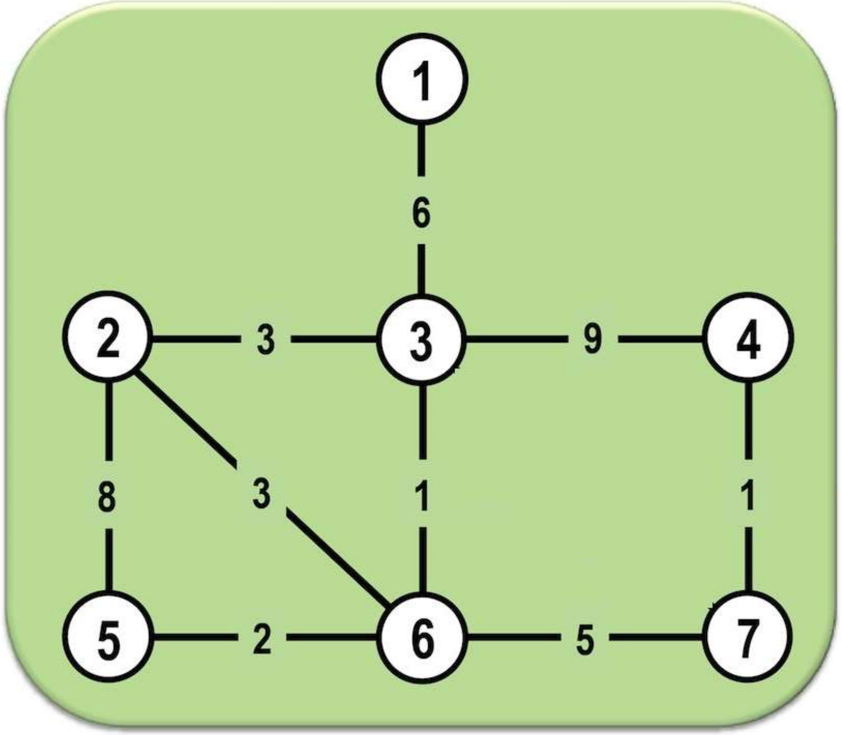
```
Ordene as arestas em ordem não decrescente de pesos d_ij no vetor H;
T ← h_1;
i ← 2;
enquanto j < n − 1 faça
    se T ∪ h_i é um grafo acíclico então
        T ← T ∪ h_i;
        j ← j + 1;
    fim
    i ← i + 1;
fim
```

Algoritmo de Kruskal

Complexidade

- A ordenação das arestas pode ser feita em $O(m \log m)$;
- A escolha das arestas é realizada $O(m)$ vezes;
- A verificação se o grafo é acíclico exige complexidade $O(m)$;
- Logo, em problemas sem características particulares, a complexidade é $O(m \log m)$.

Exemplo

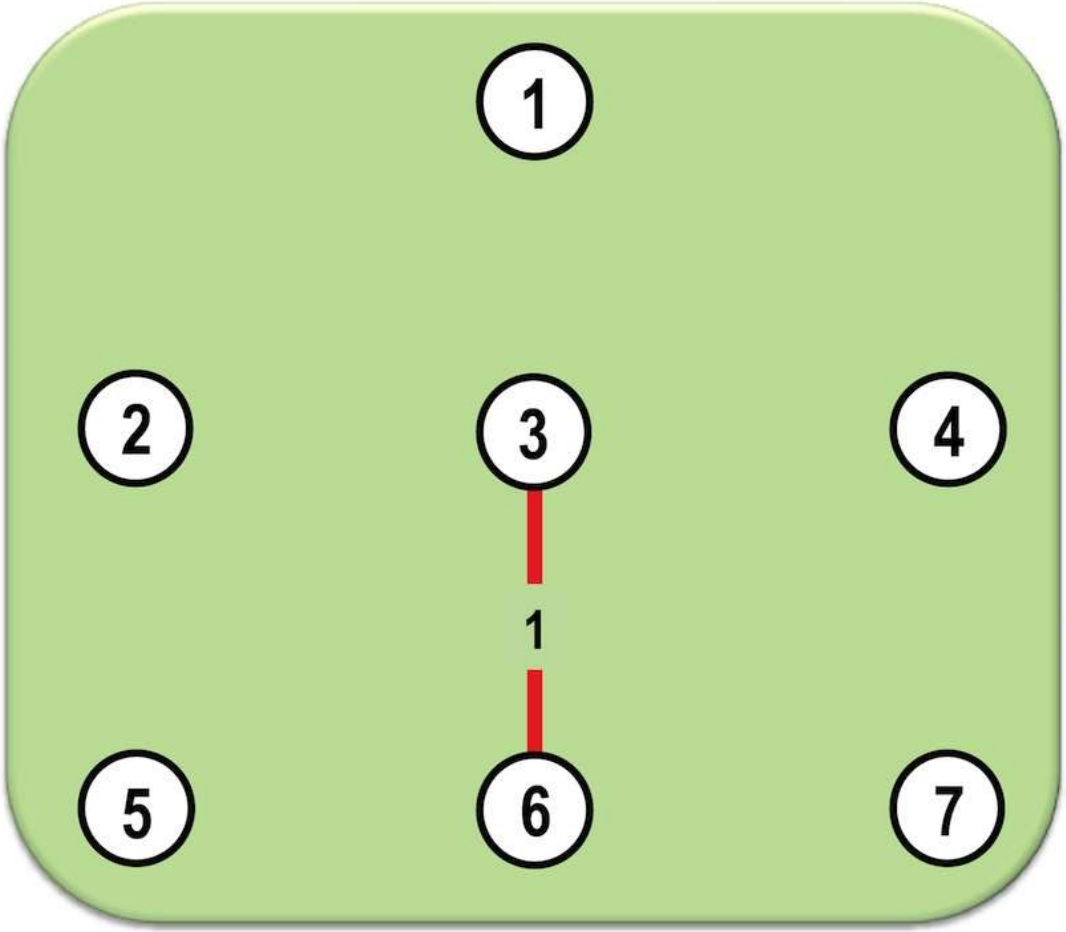


Vetor *H*

h_1 - (3,6) = 1; h_2 - (4,7) = 1; h_3 - (5,6) = 2;
h_4 - (2,3) = 3; h_5 - (2,6) = 3; h_6 - (6,7) = 5
h_7 - (1,3) = 6; h_8 - (2,5) = 8; h_9 - (3,4) = 9

Grafo de exemplo e vetor *H* desordenado.

Exemplo

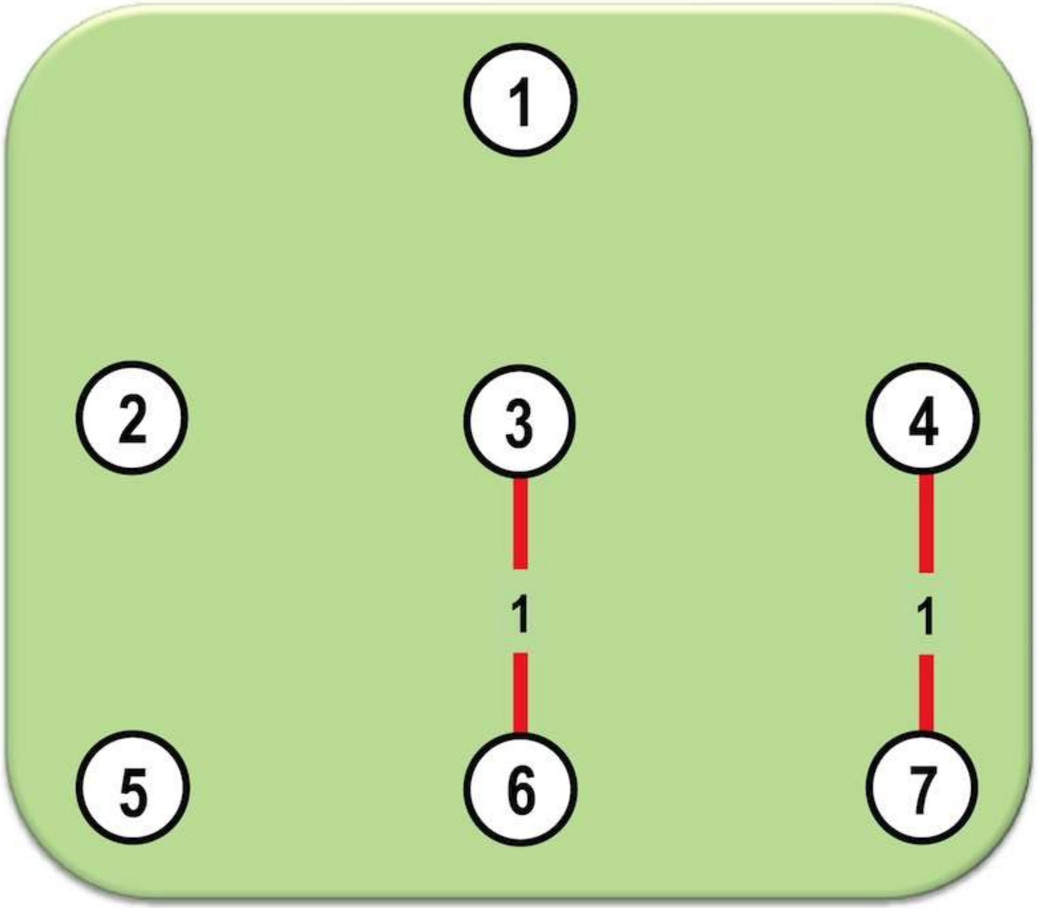


Vetor H

h_1 - (3,6) = 1
h_2 - (4,7) = 1
h_3 - (5,6) = 2
h_4 - (2,3) = 3
h_6 - (6,7) = 5
h_7 - (1,3) = 6
h_8 - (2,5) = 8
h_9 - (3,4) = 9

Inserção da primeira aresta em T.

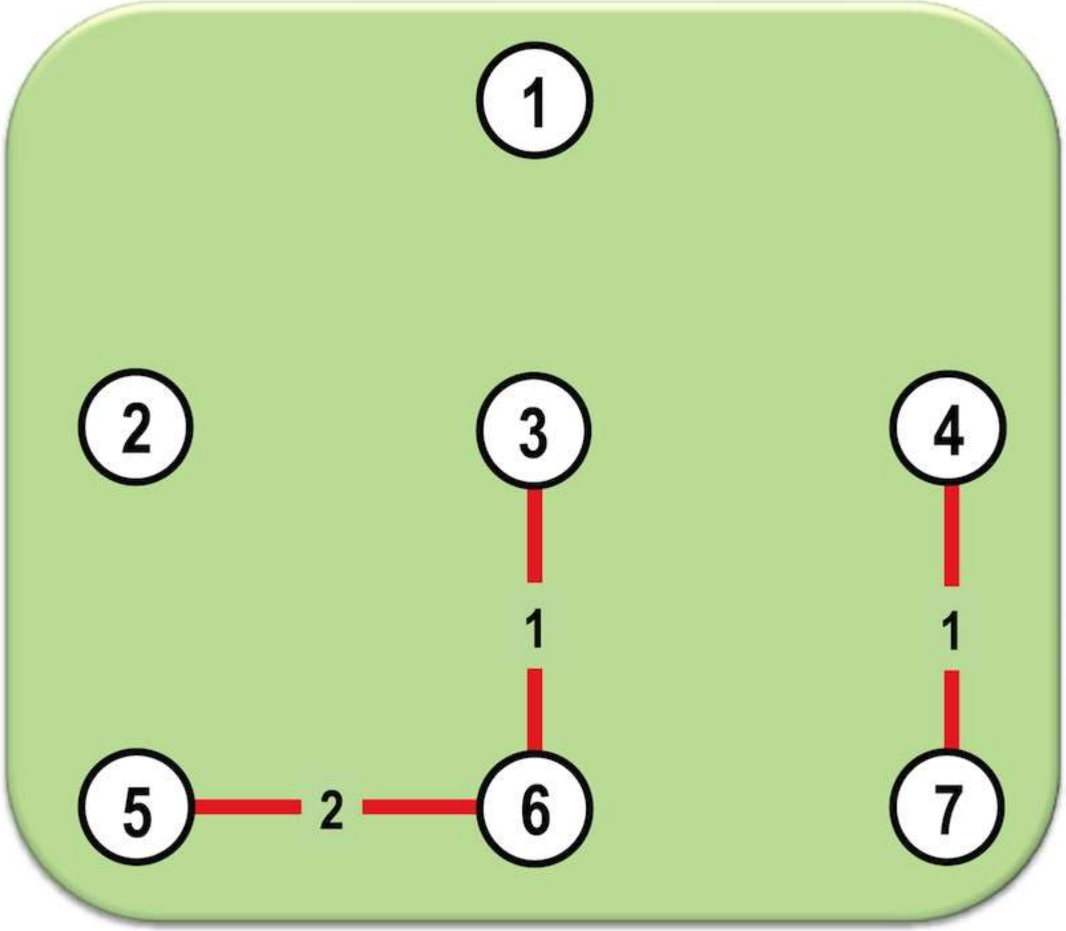
Exemplo

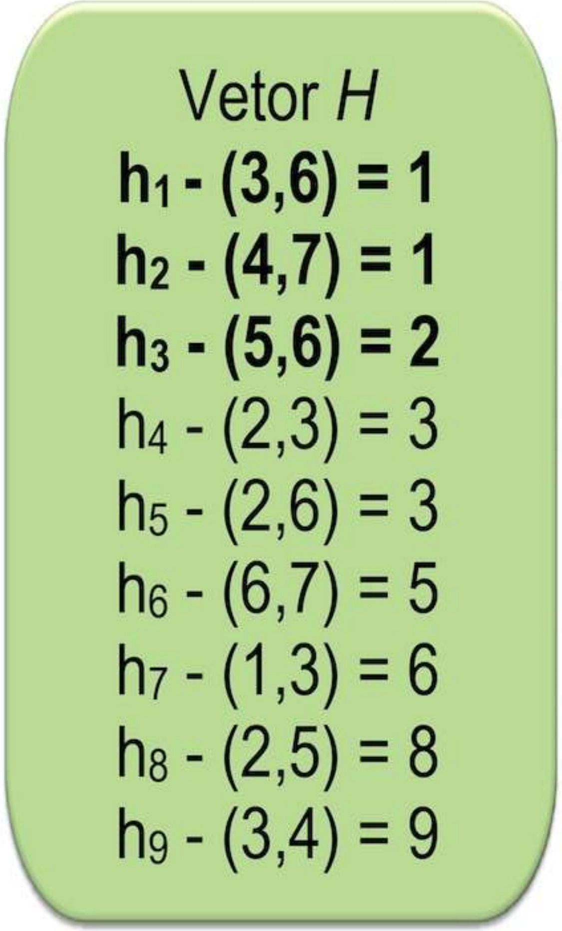


Vetor *H*

h_1 - (3,6) = 1
h_2 - (4,7) = 1
h_3 - (5,6) = 2
h_4 - (2,3) = 3
h_5 - (2,6) = 3
h_6 - (6,7) = 5
h_7 - (1,3) = 6
h_8 - (2,5) = 8
h_9 - (3,4) = 9

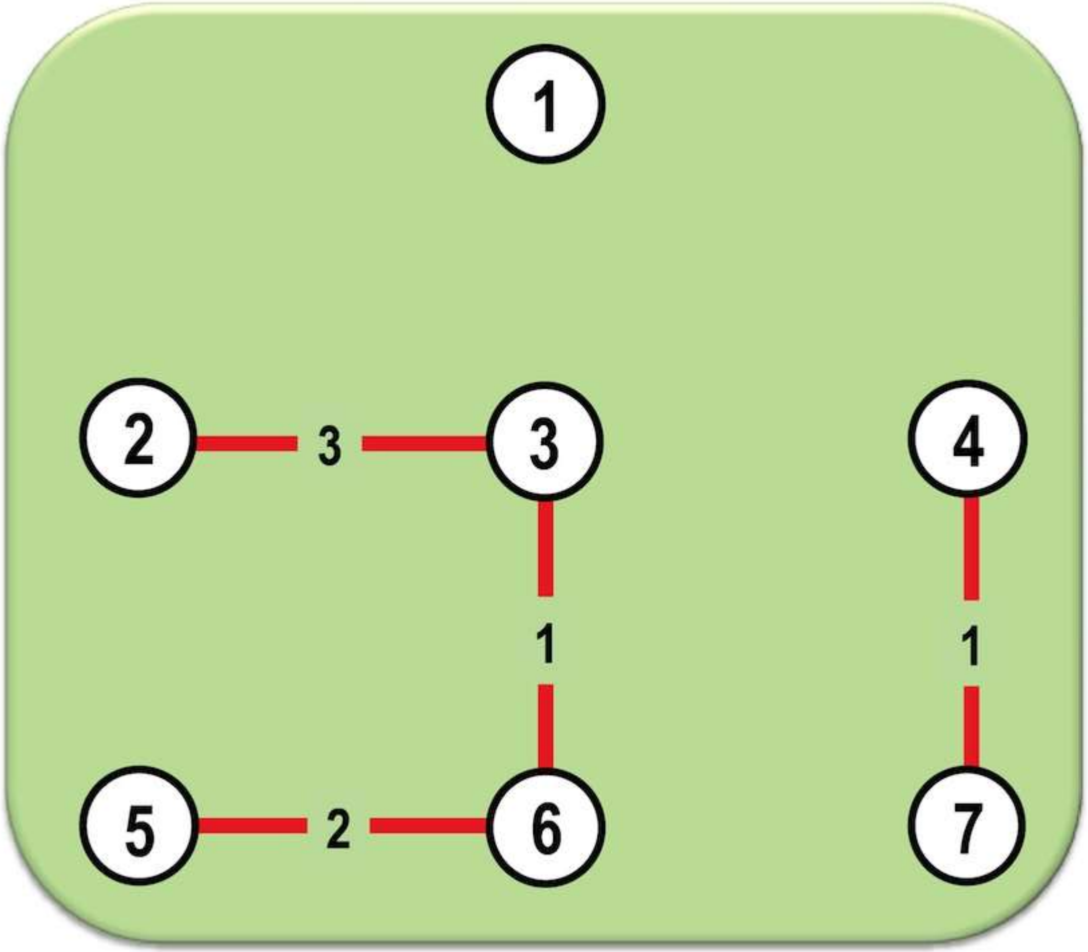
Inserção da segunda aresta em T.





Inserção da terceira aresta em T.

Exemplo

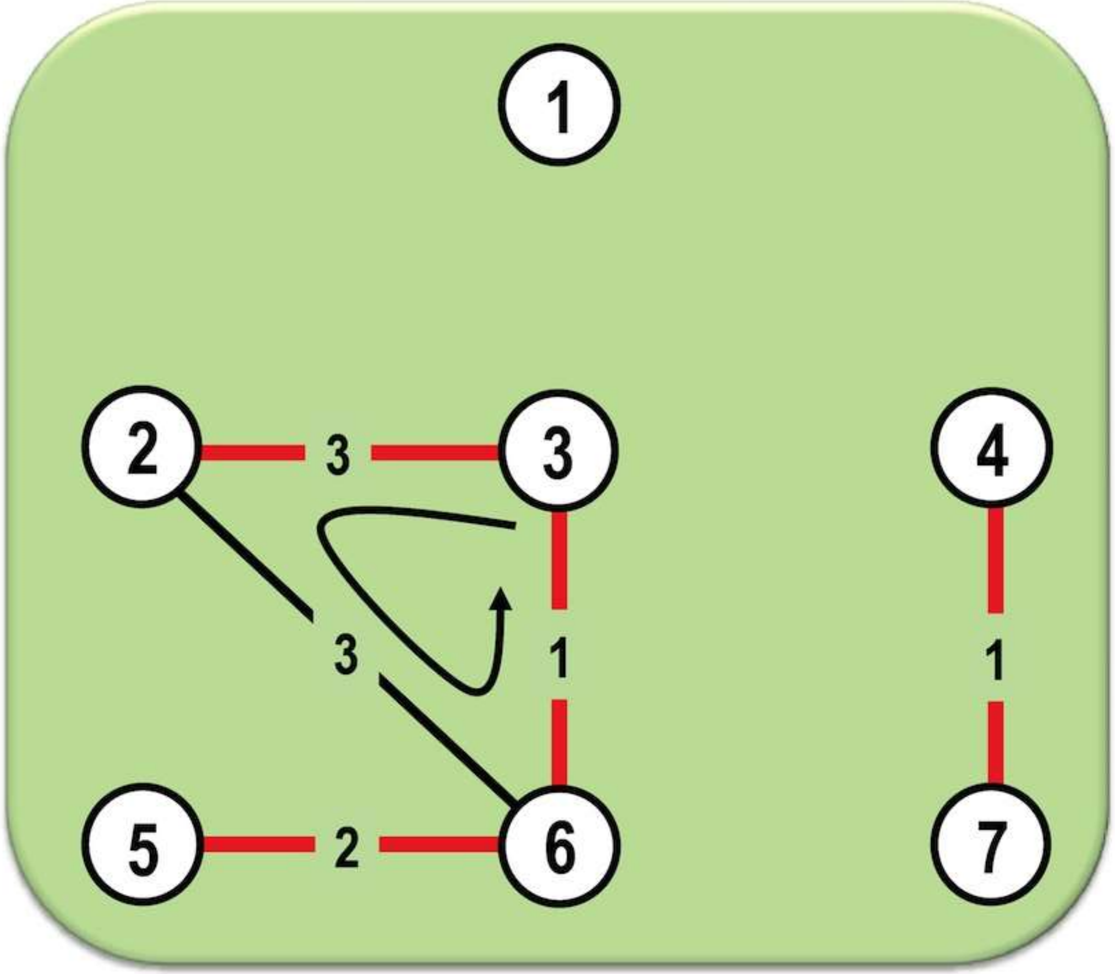


Vetor H

h_1 - (3,6) = 1
h_2 - (4,7) = 1
h_3 - (5,6) = 2
h_4 - (2,3) = 3
h_5 - (2,6) = 3
h_6 - (6,7) = 5
h_7 - (1,3) = 6
h_8 - (2,5) = 8
h_9 - (3,4) = 9

Inserção da quarta aresta em T.

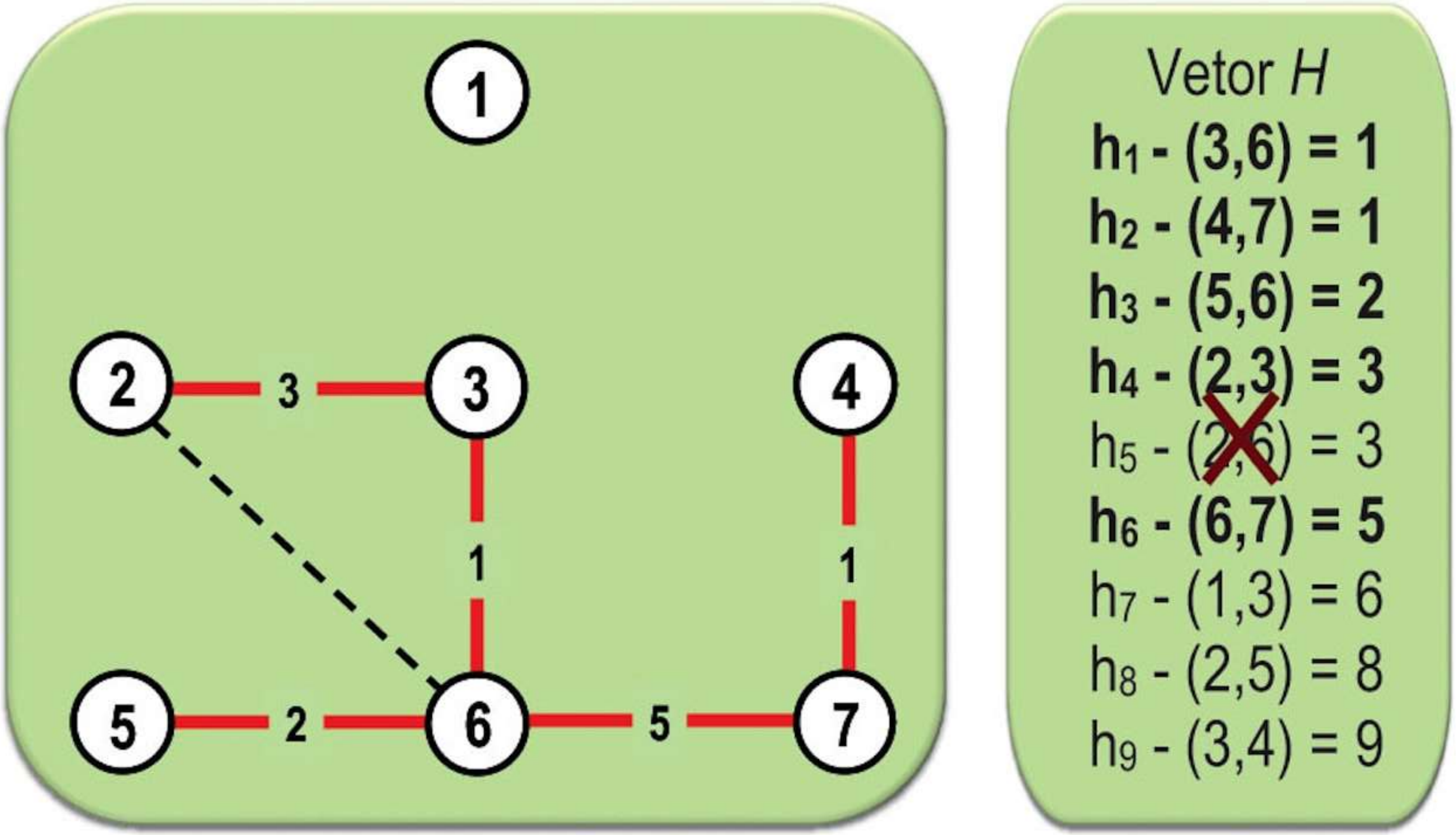
Exemplo



Vetor H
h_1 - (3,6) = 1
h_2 - (4,7) = 1
h_3 - (5,6) = 2
h_4 - (2,3) = 3
h_5 - (2,6) = 3
h_6 - (6,7) = 5
h_7 - (1,3) = 6
h_8 - (2,5) = 8
h_9 - (3,4) = 9

Tentativa de inserção da quinta aresta em T.

Exemplo



Inserção da quinta aresta em T.