Departamento de Sistemas e Computação – FURB
Curso de Ciência da Computação
Disciplina de Teoria dos Grafos

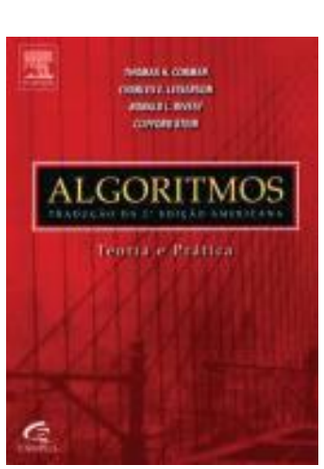
Caminhamento em grafos

Prof. Aurélio Hoppe
aureliof@furb.br
http://www.inf.furb.br/~aurelio/

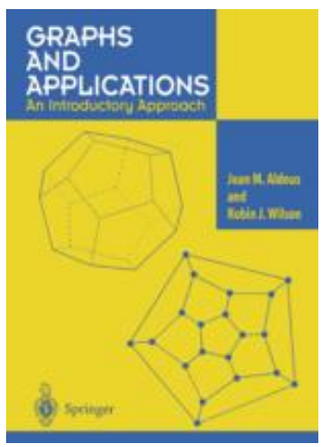
Grupo de Processamento de Imagens,
Análise de dados, Robótica e
Simulação computacional

Bibliografia

Márcia A. Rabuske. **Introdução à Teoria dos Grafos**. Editora da UFSC. 1992



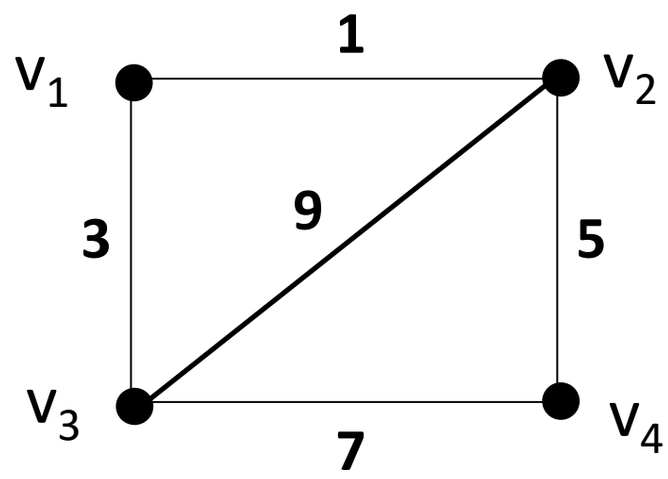
Thomas Cormen et al. **Algoritmos**: teoria e prática. Ed. Campus. 2004.



Joan M. Aldous, Robin J. Wilson. **Graphs and Applications**: as introductory approach. Springer. 2001

Definição 5.1 (Grafo Valorado). Um Grafo Valorado *G=(V, E)* é formado por um conjunto *V* de vértices e um conjunto *E* de **arestas valoradas**, sendo que para cada aresta *(u, v)* $\in E$, temos um valor numérico *w(u, v)* ou w_{uv}, chamado de custo da aresta *(u, v).*

−Este custo representa alguma grandeza numérica relevante para o problema, tal como distância, tempo, valor monetário, etc.



Matriz de custos

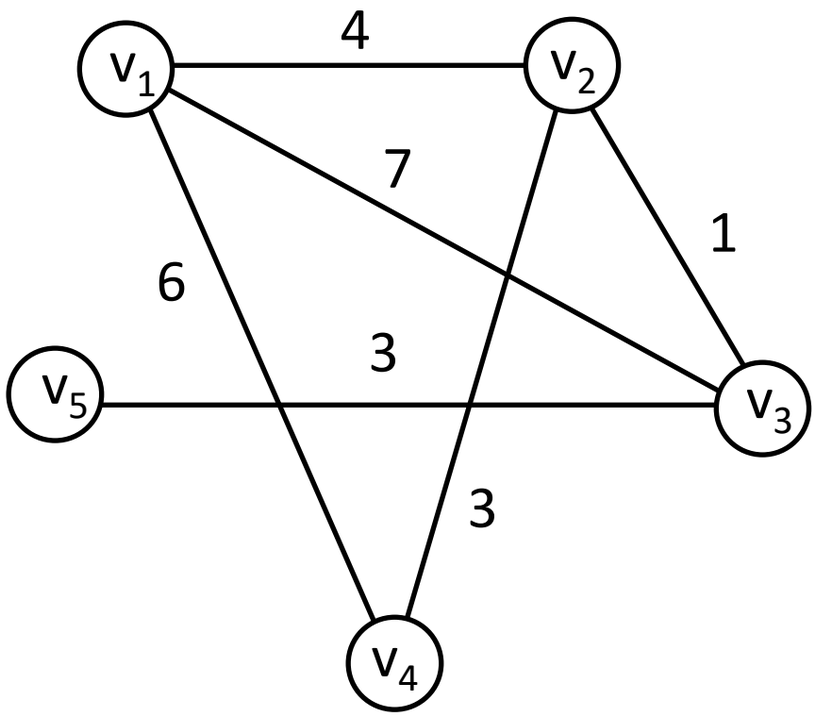
- Os custos das arestas de um grafo valorado G=(V, E) podem ser armazenados em uma matriz W, chamada de matriz de custos do grafo G, definida da seguinte forma:

$$W_{i,j} = \begin{cases} 0, & \text{se } v_i = v_j \\ \infty, & \text{se } (v_i, v_j) \notin E \\ \text{custo}, & \text{se } (v_i, v_j) \in E \end{cases}$$

Exemplo

Dado o grafo valorado abaixo, construa sua matriz de custos [W]:

$$W_{i,j} = \begin{cases} 0, & \text{se } v_i = v_j \\ \infty, & \text{se } (v_i, v_j) \notin E \\ \text{custo}, & \text{se } (v_i, v_j) \in E \end{cases}$$



$$W = \begin{vmatrix} 0 & 4 & 7 & 6 & \infty \\ 4 & 0 & 1 & 3 & \infty \\ 7 & 1 & 0 & \infty & 3 \\ 6 & 3 & \infty & 0 & \infty \\ \infty & \infty & 3 & \infty & 0 \end{vmatrix}$$

Caminho Mínimo

- O **custo** de um caminho p = $\langle v_0, v_1, \ldots, v_k \rangle$ entre dois vértices v_0 e v_k é igual ao somatório dos custos de todas as arestas valoradas do caminho:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Caminho Mínimo

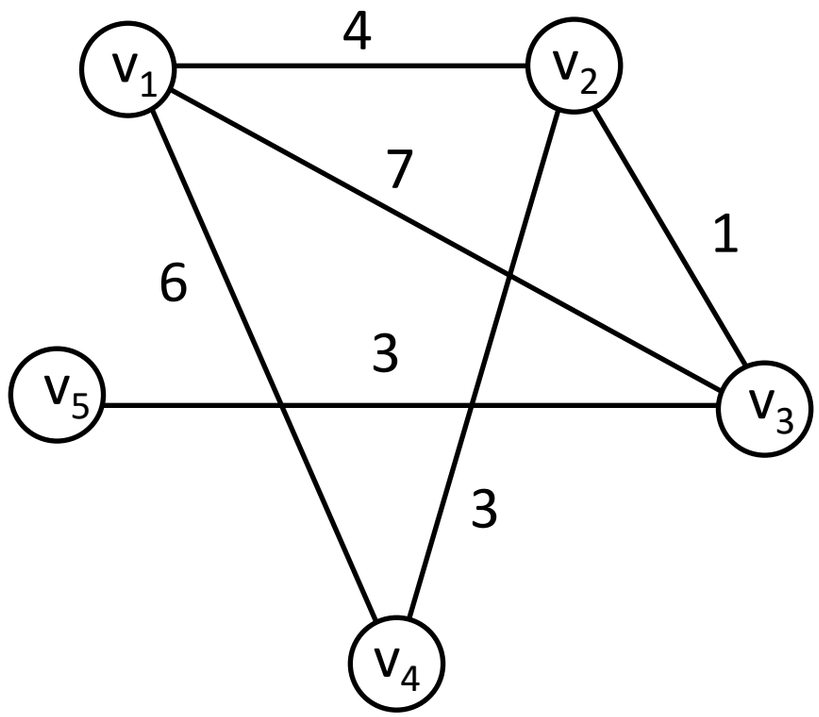
O **custo do caminho mínimo** do vértice u para o vértice v é definido por:

$$\delta_{u,v} = \delta(u,v) = \begin{cases} \min\{\, w(p) : u \rightarrow v \,\}, & \text{se } \exists \text{ caminho de u para v} \\ \infty, & \text{caso contrário} \end{cases}$$

- O **caminho mínimo** entre dois vértices *u* e *v* é definido como qualquer caminho *p* com custo $w(p) = \delta(u, v)$.

Exemplo

- Qual o caminho mínimo entre os vértices v1 e v3?
- Qual o custo deste caminho mínimo?



Problema do caminho mínimo com uma origem

- Dado um grafo valorado *G=(V, E)*, queremos encontrar os caminhos mínimos de um dado vértice origem $s \in V$, até cada um dos vértices $v \in V$.

- O algoritmo que resolve este problema é o **Algoritmo de Dijkstra**

Algoritmo de Dijkstra

- Para armazenar os caminhos entre o vértice inicial *s* e os demais vértices, podemos usar uma árvore de caminhos mínimos, semelhante às árvores de busca.
- Dado o grafo valorado *G=(V, E)*, manteremos para cada vértice $v \in V$ um **predecessor** representado por $\pi[v]$ que contém ou um vértice ou NIL.

Algoritmo de Dijkstra

- Elementos do algoritmo:

Elemento	Descrição
`s`	vértice inicial
`v`	quaisquer outros vértices
`u`	vértice “pivô”, que pode representar uma mudança no caminho mínimo até o vértice v
`d[v]`	custo estimado do caminho mínimo de s até v
`w(u,v)`	custo da aresta (u, v), tem valor ∞ se não existir aresta (u, v)
`π[v]`	vértice predecessor do vértice v
`Q`	fila de prioridades mínimas de vértices (o vértice com menor valor de d[v] tem prioridade para sair da fila)
`S`	conjunto dos vértices cujo caminho mínimo já foi calculado

Algoritmo de Dijkstra

- Inicialização dos atributos dos vértices:

```
INITIALIZE-SINGLE-SOURCE(G,s)
01. for each vertex v ∈ V[G]
02.     do d[v] ← ∞
03.        ∏[v] ← NIL
04. d[s] ← 0;
```

Algoritmo de Dijkstra

- Relaxamento da aresta (u, v): verifica se podemos melhorar a estimativa atual de caminho mínimo até o vértice v, incluindo o vértice v no caminho. Esta operação atualiza os valores de d[v] e *π[v].*

```
RELAX(u, v, w)
01. if d[v] > d[u] + w(u,v)
02.       then d[v] ← d[u] + w(u,v)
03.            ∏[v] ← u
```

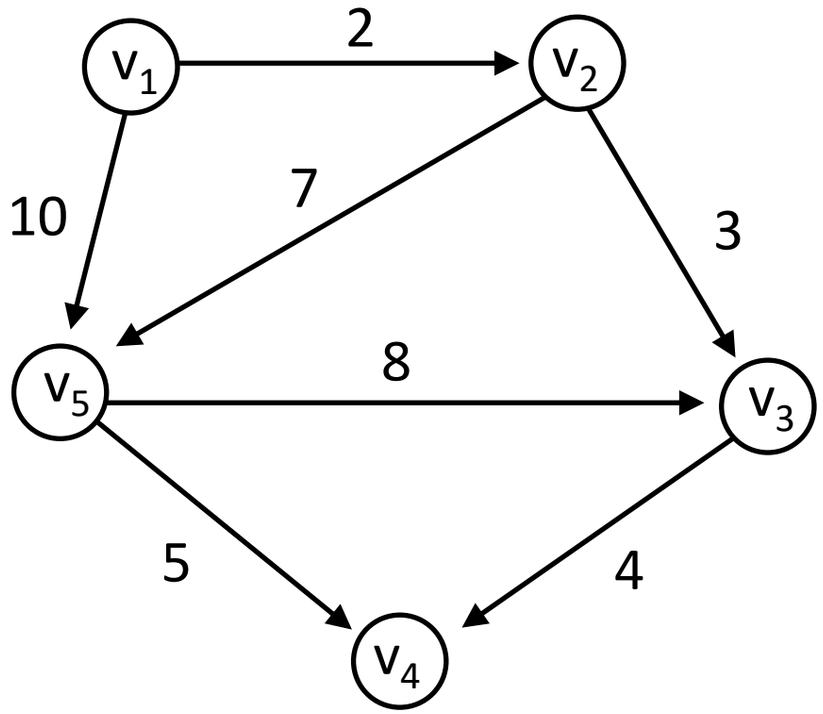
Algoritmo de Dijkstra

```
INITIALIZE-SINGLE-SOURCE(G,s)
01. for each vertex v ∈ V[G]
02.      do d[v] ← ∞
03.         ∏[v] ← NIL
04. d[s] ← 0;
```

```
RELAX(u, v, w)
01. if d[v] > d[u] + w(u,v)
02.      then d[v] ← d[u] + w(u,v)
03.           ∏[v] ← u
```

```
DIJKSTRA(G, w, s)
01. INITIALIZE-SINGLE-SOURCE(G,s)
02. S ← ∅
03. Q ← V[G]
04. while Q ≠ ∅
05.    do u ← EXTRACT-MIN(Q)
06.         S ← S ∪ [u]
07.         for each vertex v ∈ Adj[u]
08.              do RELAX(u,v,w)
```

Exemplo



$$W = \begin{vmatrix} 0 & 2 & \infty & \infty & 10 \\ \infty & 0 & 3 & \infty & 7 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 8 & 5 & 0 \end{vmatrix}$$

```
INITIALIZE-SINGLE-SOURCE(G,s)
01. for each vertex v ∈ V[G]
02.     do d[v] ← ∞
03.        ∏[v] ← NIL
04. d[s] ← 0;
```

```
RELAX(u, v, w)
01. if d[v] > d[u] + w(u,v)
02.     then d[v] ← d[u] + w(u,v)
03.          ∏[v] ← u
```

```
DIJKSTRA(G, w, s)
01. INITIALIZE-SINGLE-SOURCE(G,s)
02. S ← ∅
03. Q ← V[G]
04. while Q ≠ ∅
05.    do u ← EXTRACT-MIN(Q)
06.         S ← S ∪ [u]
07.         for each vertex v ∈ Adj[u]
08.              do RELAX(u,v,w)
```

vértice	v1	v2	v3	v4	v5
d	0	2	5	9	9
∏	nil	V1	V2	V3	V2
Q					
S	x	x	x	x	x

Problema do caminho mínimo com várias origens

- Dado um grafo valorado *G=(V, E)*, queremos encontrar os caminhos mínimos entre todos os pares de vértices
- O algoritmo **Floyd-Warshall** calcula esses caminhos através de operações recursivas em matrizes de distância (D), de tamaho n x n
- O algoritmo calcula n matrizes de distância, e o resutado final com as caminhos mínimos surge na última matriz calculada

Algoritmo de Floyd-Warshall

- Como este algoritmo usa a estrutura de matriz de adjacência, a entrada do algoritmo é a matriz de custos do grafo, definida da seguinte forma:

$$W_{i,j} = \begin{cases} 0, & \text{se } v_i = v_j \\ \infty, & \text{se } (v_i, v_j) \notin E \\ \text{custo}, & \text{se } (v_i, v_j) \in E \end{cases}$$

Algoritmo de Floyd-Warshall

- Inicialização: $D^0 \leftarrow W$
- Demais matrizes:

$$d_{i,j}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0 \\ \min \{d_{i,j} , d_{i,k} + d_{k,j} \} & \text{se } k \geq 1 \end{cases}$$

Onde:

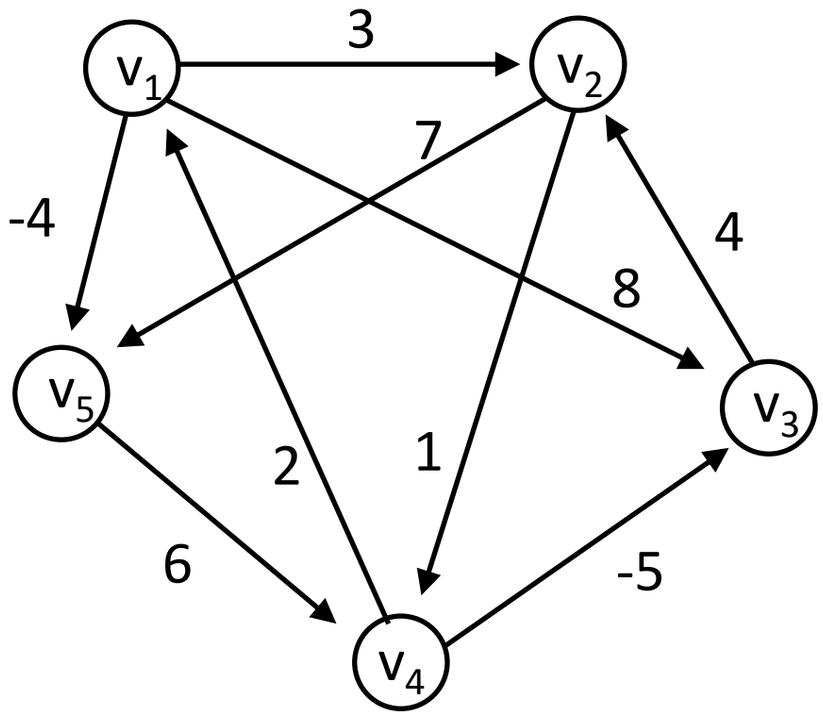
i: linha (índice do vértice origem)
j: coluna (índice do vértice destino)
k: iteração (índice da matriz)

Algoritmo de Floyd-Warshall

```
FLOYD-WARSHALL(G, W)
01. D^0 ← W
02. para cada k ← 1 até n faça
03.    para cada i ← 1 até n faça
04.       para cada j ← 1 até n faça
05.          d_{i,j}^(k)  ← min (d_{i,j} , d_{i,k} + d_{k,j})
06. retorno D^n
```

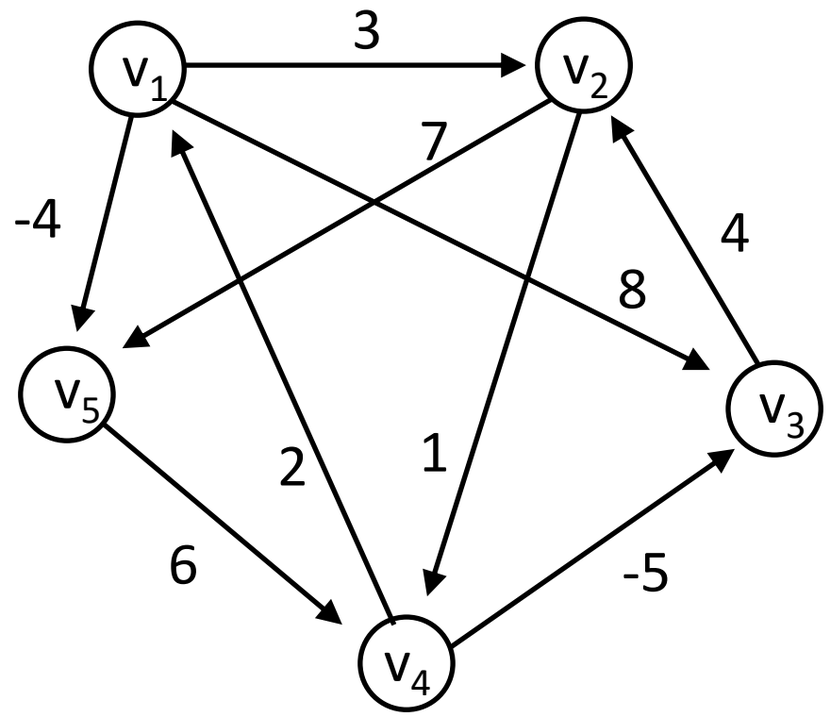
0	**7**	**3**	**15**
7	**0**	**3**	**5**
3	**3**	**0**	**-**
15	**5**	**-**	**0**

Exemplo



$$D(0) = \begin{vmatrix} 0 & 3 & 8 & - & -4 \\ - & 0 & - & 1 & 7 \\ - & 4 & 0 & - & - \\ 2 & - & -5 & 0 & \\ - & - & - & 6 & 0 \end{vmatrix}$$

Exemplo



D(0)

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

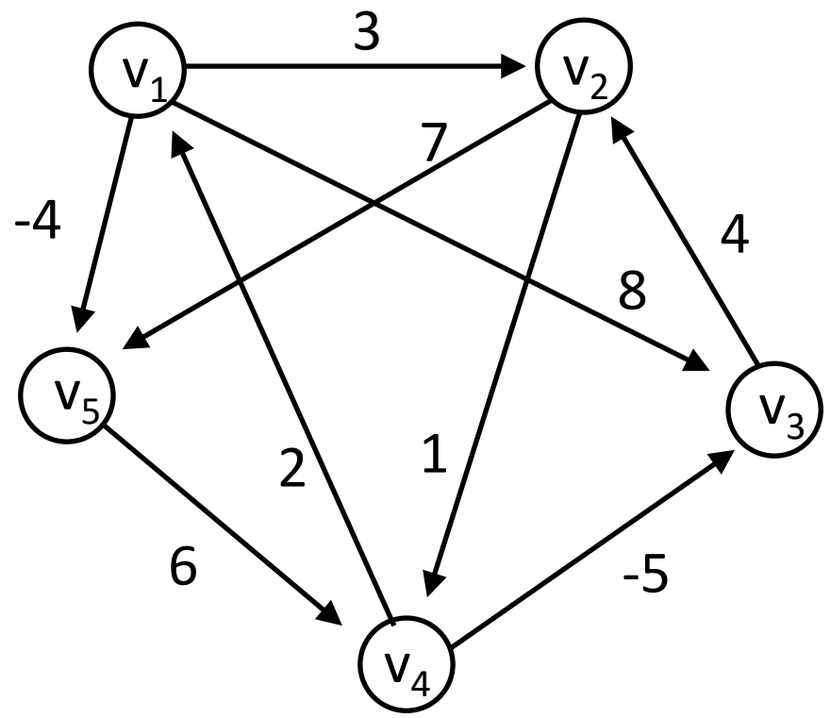
D(1)

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

D(2)

0	3	8	4	-4
-	0	-	1	7
-	4	0	5	11
2	5	-5	0	-2
-	-	-	6	0

Exemplo



D(3)

0	3	8	4	-4
-	0	-	1	7
-	4	0	5	11
2	-1	-5	0	-2
-	-	-	6	0

D(4)

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

D(5)

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

Algoritmo de Floyd-Warshall

- Os caminhos podem ser armazenados em uma **Matrizes de roteamento** (Π), construídas a partir das matrizes de distância (D), da seguinte forma:

- Inicialização:

$$\pi_{ij}^{(0)} = \begin{cases} NIL, & \text{se } v_i = v_j \text{ ou } w_{ij} = \infty, \\ v_i, & \text{se } v_i \neq v_j \text{ e } w_{ij} < \infty. \end{cases}$$

- Demais matrizes:

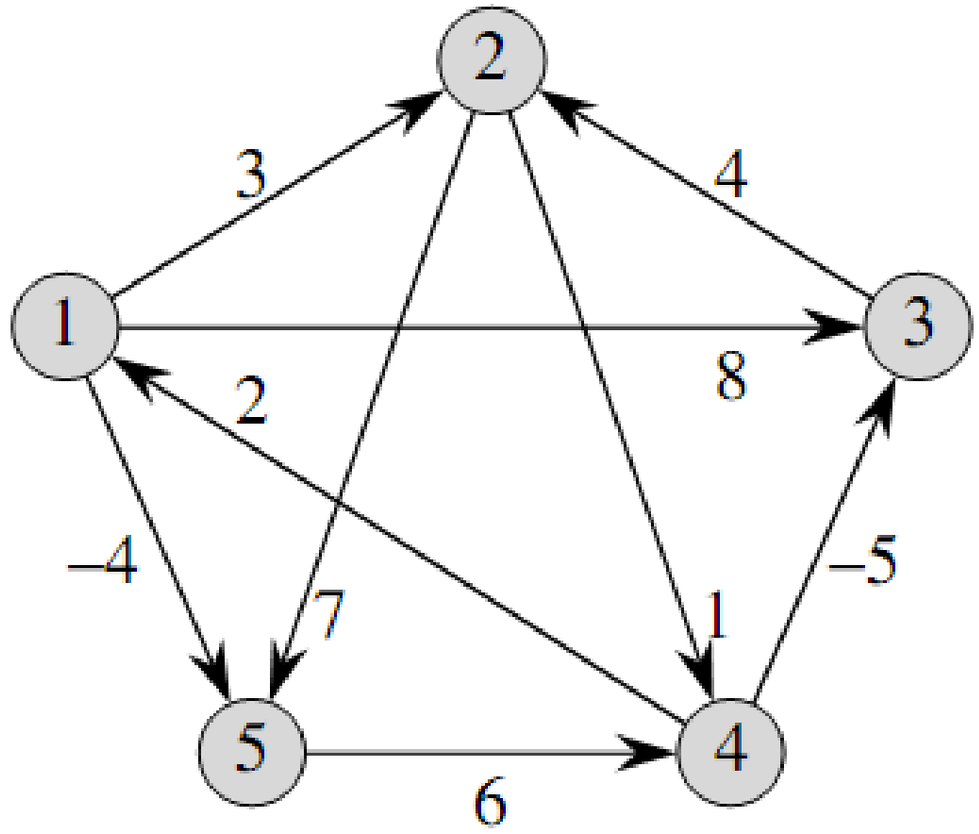
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Obs: π_{ij} armazena o predecessor de v_j num caminho iniciado em v_i

Algoritmo de Floyd-Warshall

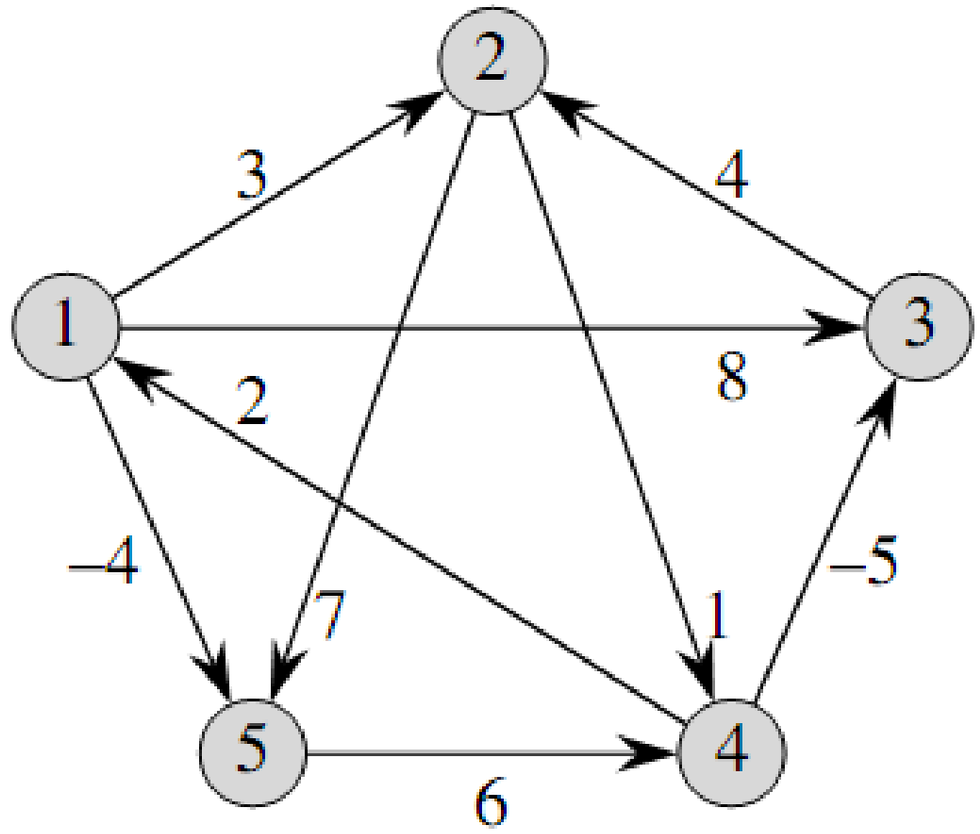
```
FLOYD-WARSHALL(G, W)
01. para cada i ← 1 até n faça
02.   para cada j ← 1 até n faça
03.       D⁰ᵢⱼ ← Wᵢⱼ
04.       se vᵢ ≠ vⱼ e wᵢⱼ < ∞ então
05.           Πᵢⱼ ← vᵢ
06.       senão
07.           Πᵢⱼ ← NIL

08. para cada k ← 1 até n faça
09.   para cada i ← 1 até n faça
10.     para cada j ← 1 até n faça
11.         se (dᵢ,ₖ + dₖ,ⱼ) < dᵢ,ⱼ então
12.           dᵢⱼ ← dᵢ,ₖ + dₖ,ⱼ
13.           Πᵢⱼ ← Πₖⱼ
14. retorno Dⁿ,Πⁿ
```

Exemplo



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Exemplo



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

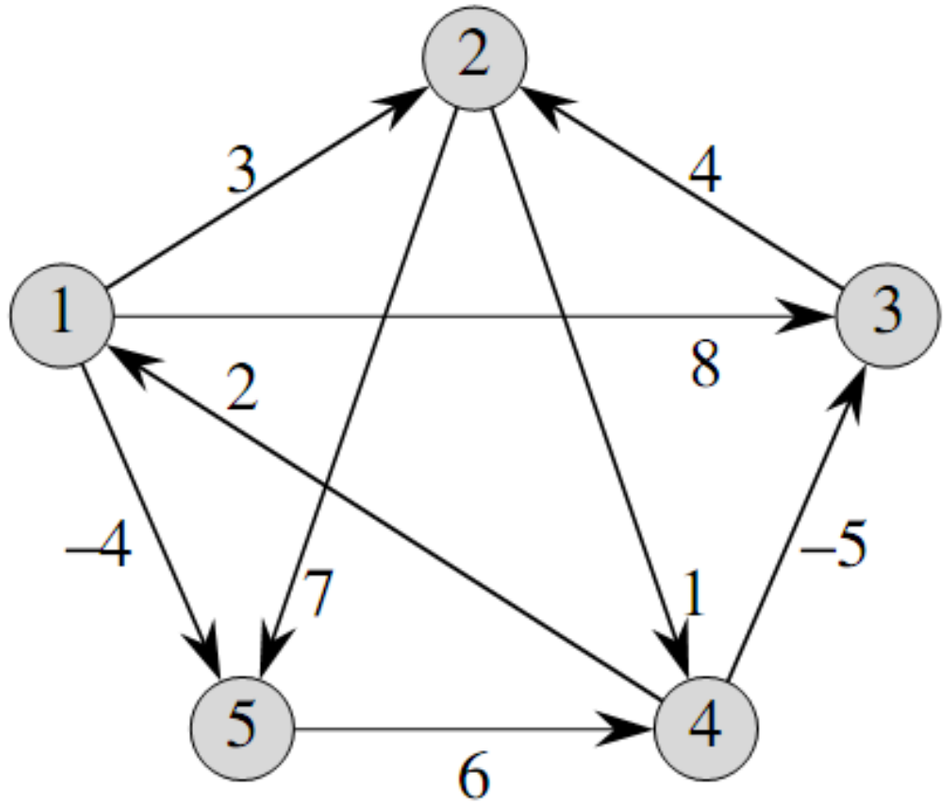
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Exemplo



$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Grafos e ciclos eulerianos e hamiltonianos

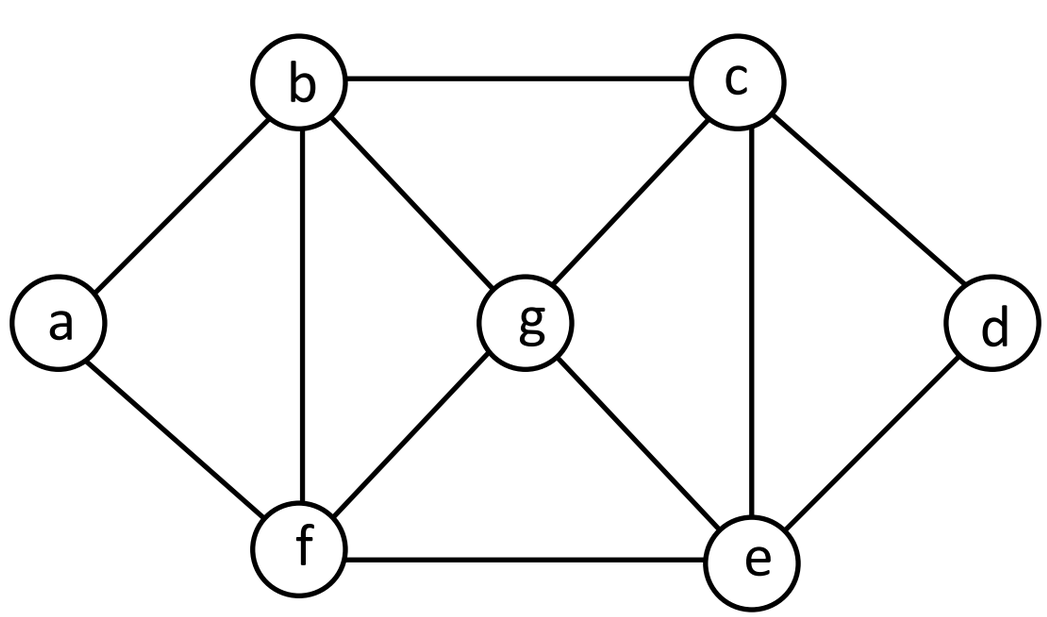
- Começamos explorando dois tipos de problemas a rotas conectando um conjunto de cidades em um mapa:

- **Problema do explorador**: um explorador deseja encontrar um roteiro que passe por todas as estradas do mapa somente uma vez e retorne ao ponto de partida

- **Problema do turista**: um turista deseja encontrar um roteiro que passe em cada cidade somente uma vez, e retorne a cidade de partida.

Grafos e ciclos eulerianos e hamiltonianos

- Exemplo:



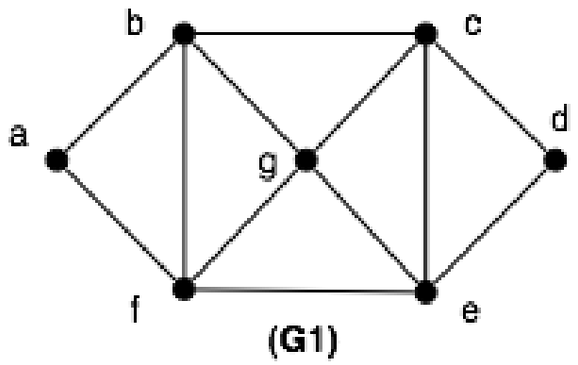
Grafos e ciclos eulerianos e hamiltonianos

Definições:

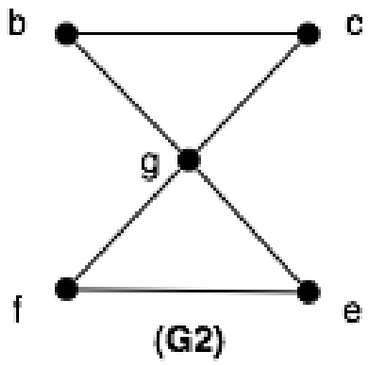
- Um grafo conexo é **Euleriano** se ele contiver um caminho simples fechado que inclua cada uma de suas arestas. Tal caminho é chamado de **Ciclo euleriano**
- Um grafo conexo é **Hamiltoniano** se contiver um ciclo que inclua cada um dos vértices do grafo. Tal ciclo é chamado de **Ciclo hamiltoniano**

Exercício

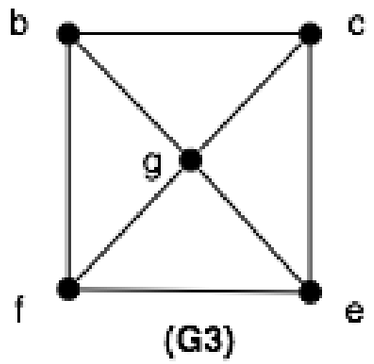
- Dados os grafos abaixo, verifique se cada um deles é Euleriano e/ou Hamiltoniano. Escreva os ciclos eulerianos e hamiltonianos sempre que possível:



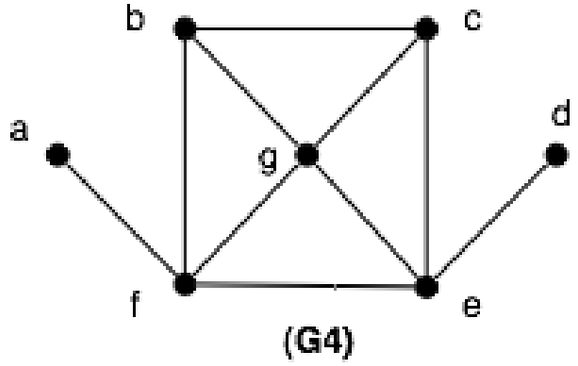
(G1)



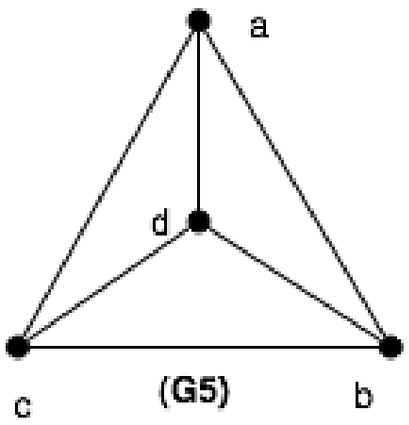
(G2)



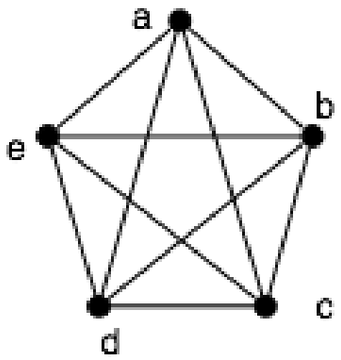
(G3)



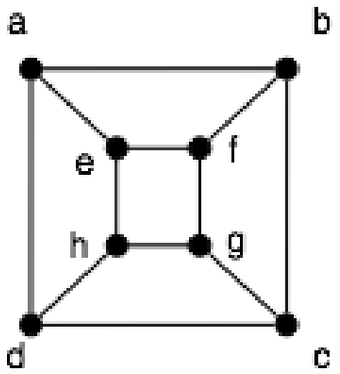
(G4)



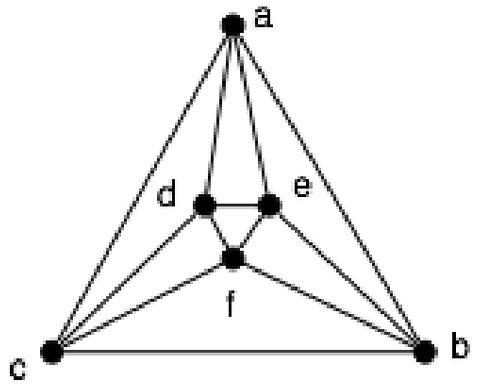
(G5)



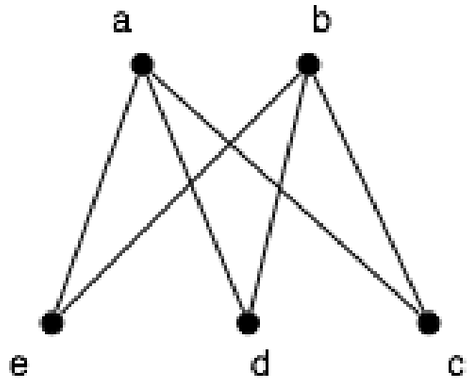
(G6)



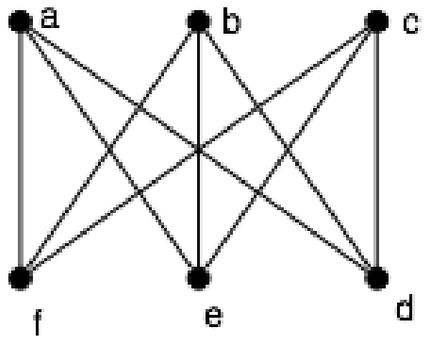
(G7)



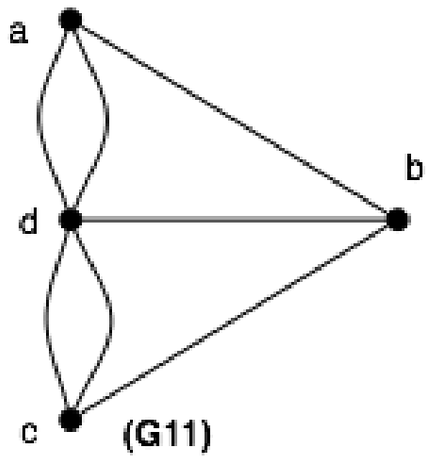
(G8)



(G9)



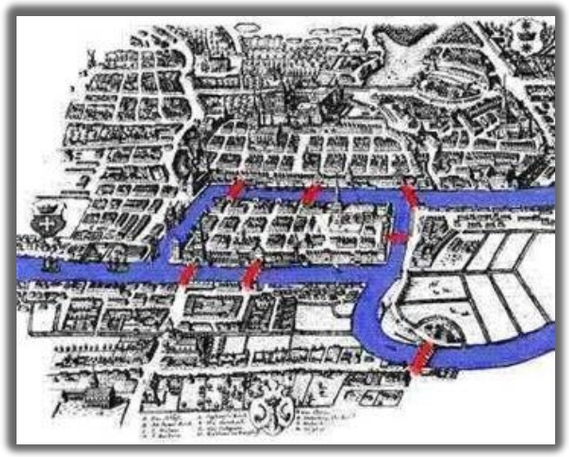
(G10)

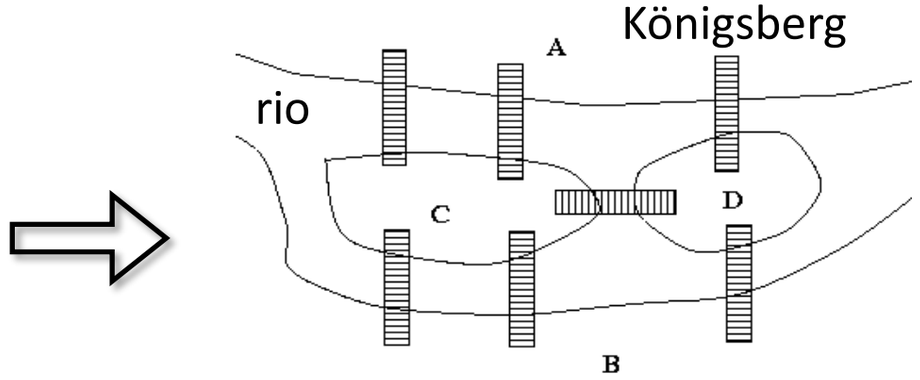


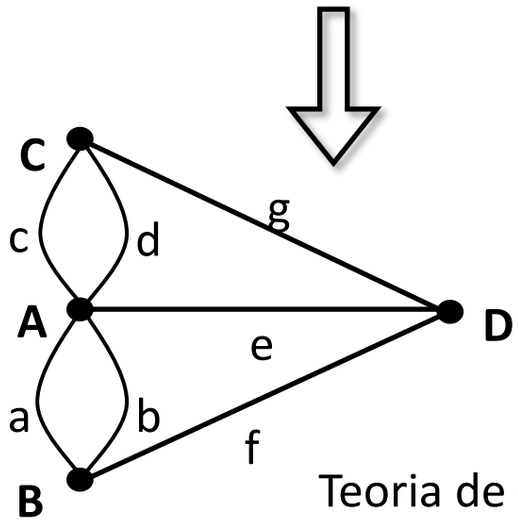
(G11)

Grafos e Ciclos Eulerianos

Histórico: Leonhard Euler (1736), problema das pontes de Könisberg







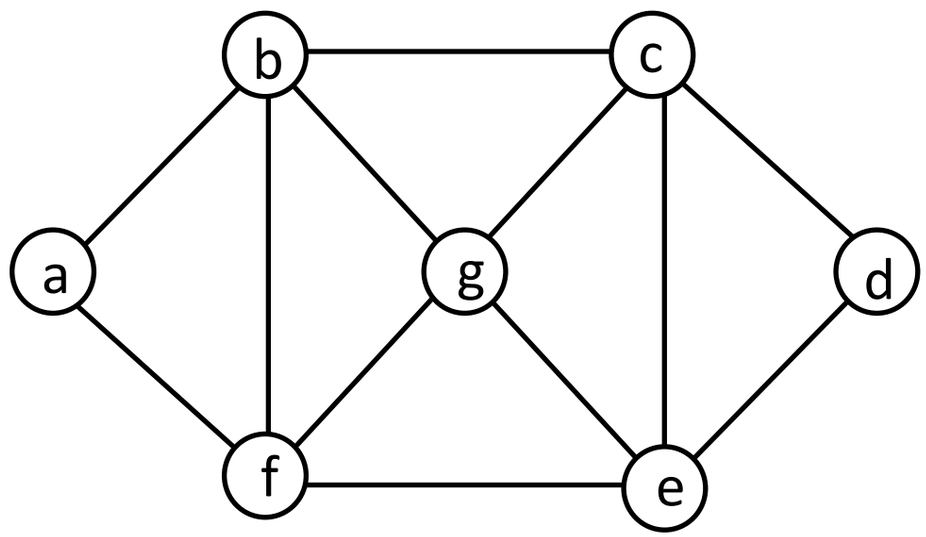
Teoria de Grafo

(Euler, 1736)

Grafos e Ciclos Eulerianos

Teorema 5.2:
Seja G um grafo no qual todos os vértices possuem grau par. Então, G pode ser decomposto em ciclos de forma que não haja dois ciclos compartilhando arestas (ciclos disjuntos).

Dado o grafo abaixo, mostre que ele pode ser decomposto em ciclos disjuntos. Como esses ciclos podem ser combinados para formar um ciclo euleriano?



Grafos e Ciclos Eulerianos

Teorema 5.3:
Um grafo conexo é euleriano se e somente se todos os seus vértices possuirem grau par.

Um grafo conexo dirigido é euleriano se e somente se o grau de entrada de cada vértice for igual ao seu grau de saída.

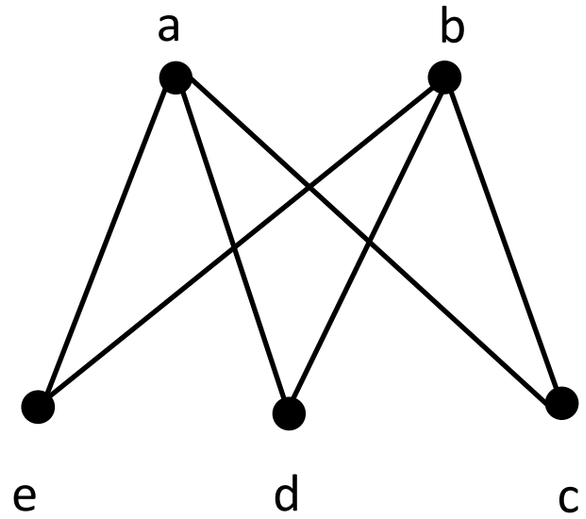
Grafos Semi-eulerianos

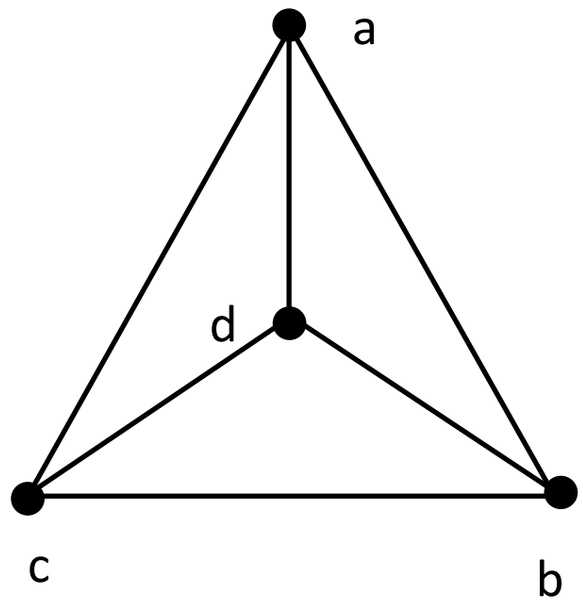
Definição:
Um grafo conexo é semi-euleriano se existe um caminho simples aberto que inclua todas as suas arestas. Tal caminho é chamado de caminho semi-euleriano.

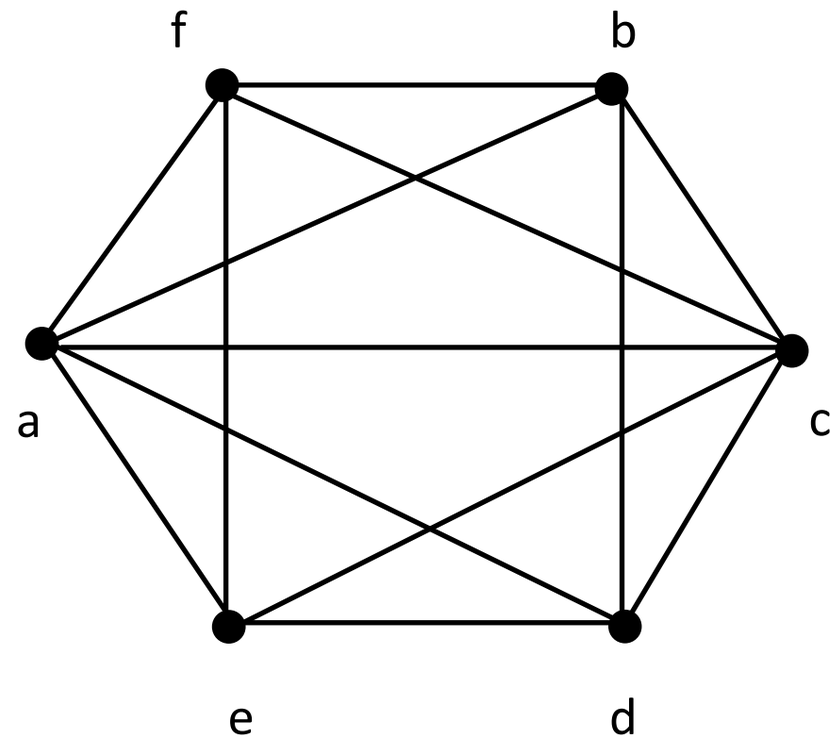
Teorema 5.4:
Um grafo conexo é semi-euleriano se e somente se ele tiver exatamente dois vértices de grau ímpar.

Exercícios

Exercício: pelo teorema 5.4, determine quais dos grafos a seguir são semi-eulerianos e escreva o caminho semi-euleriano se possível:







Algoritmo de Fleury

Dado um grafo euleriano, o algoritmo de Fleury encontra um ciclo euleriano

Inicie um caminho a partir de um vértice qualquer, e atravesse as arestas seguindo as regras abaixo:

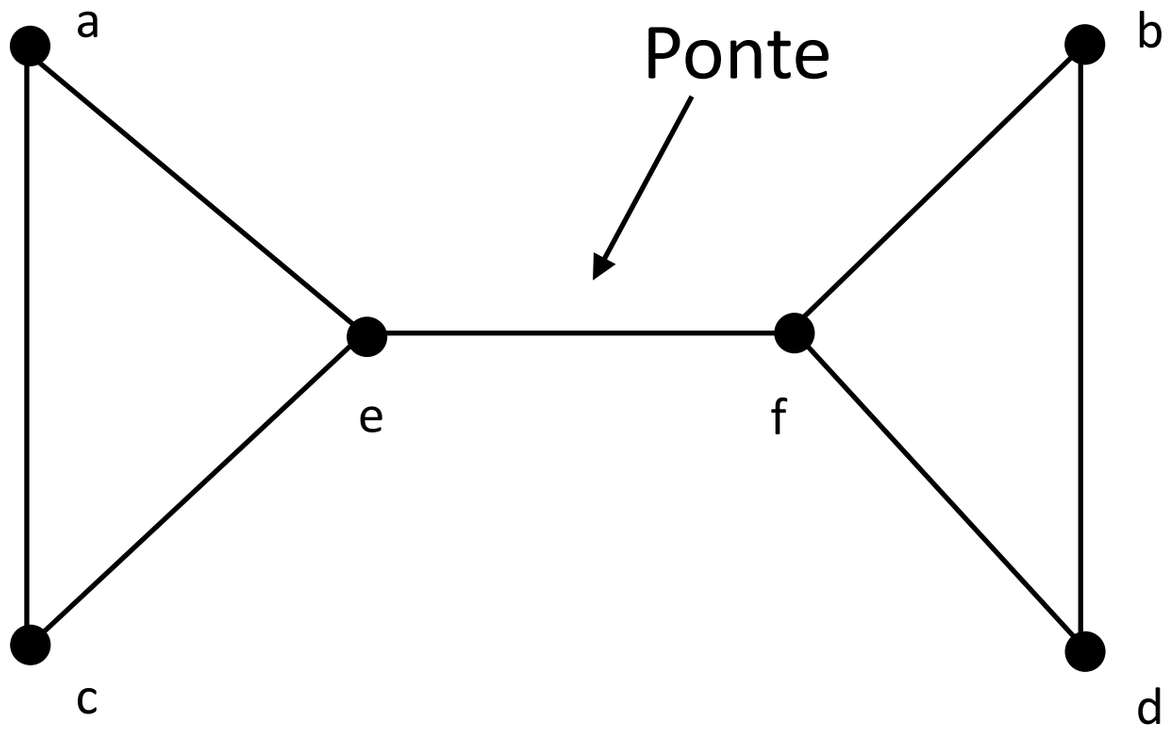
R1. Remova a aresta que acabou de ser percorrida. Se algum vértice ficar isolado, remova-o também;

R2. Atravesse uma “ponte”, somente se não houver outra alternativa.

Pontes

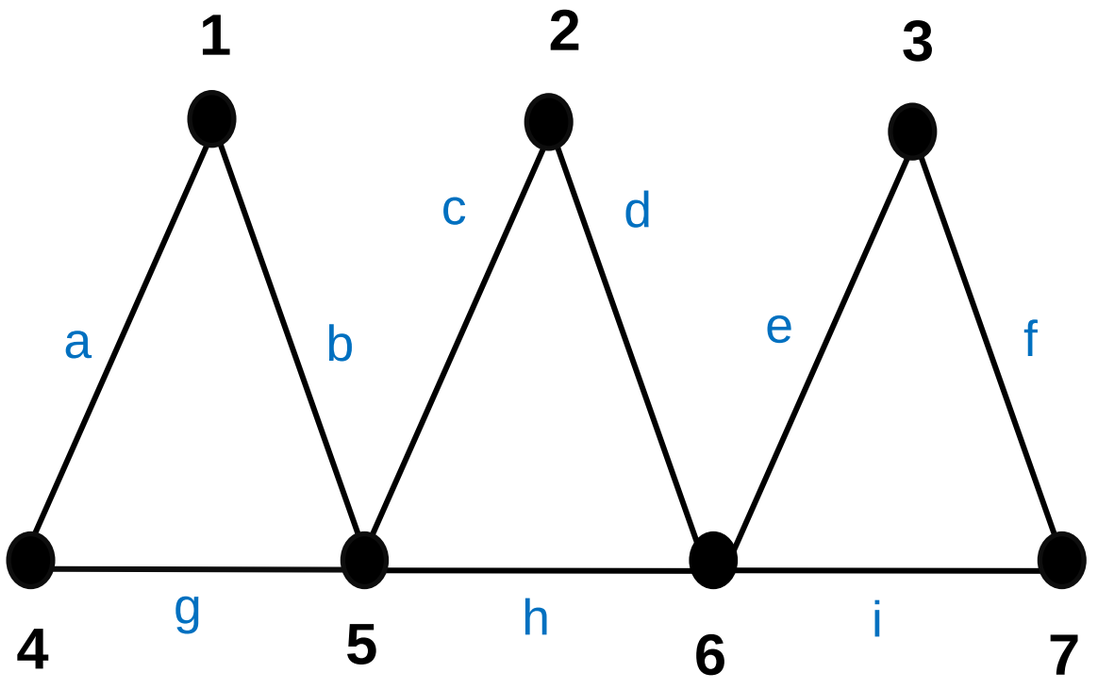
Uma **Ponte** é uma aresta cuja remoção “quebra” a conexidade do grafo.

- Exemplo:



Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

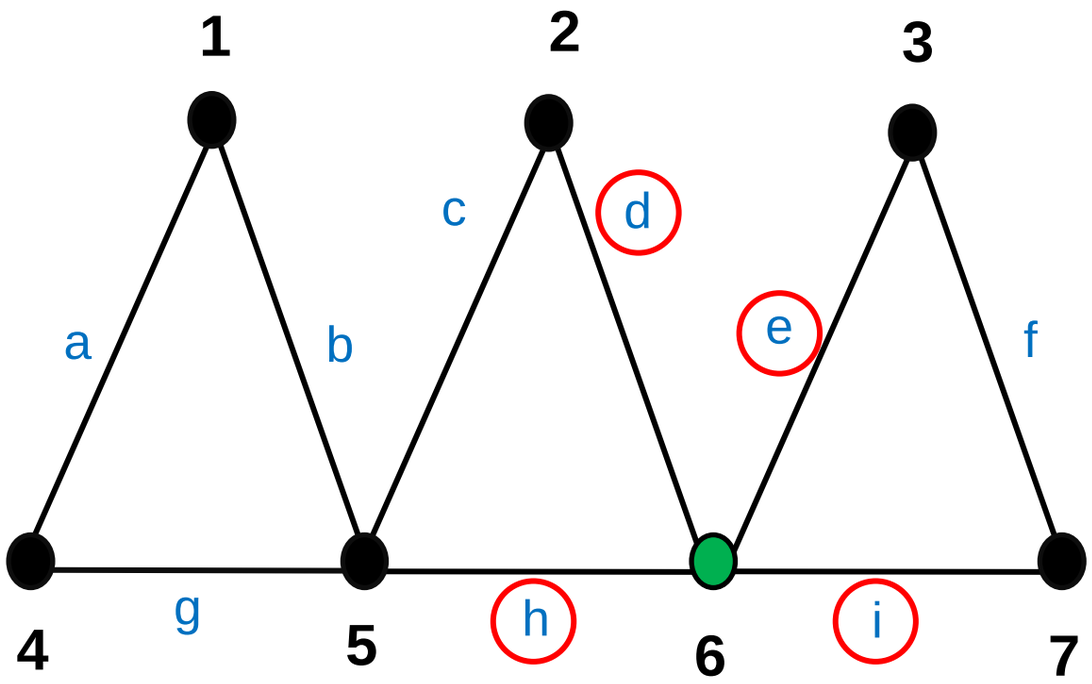
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

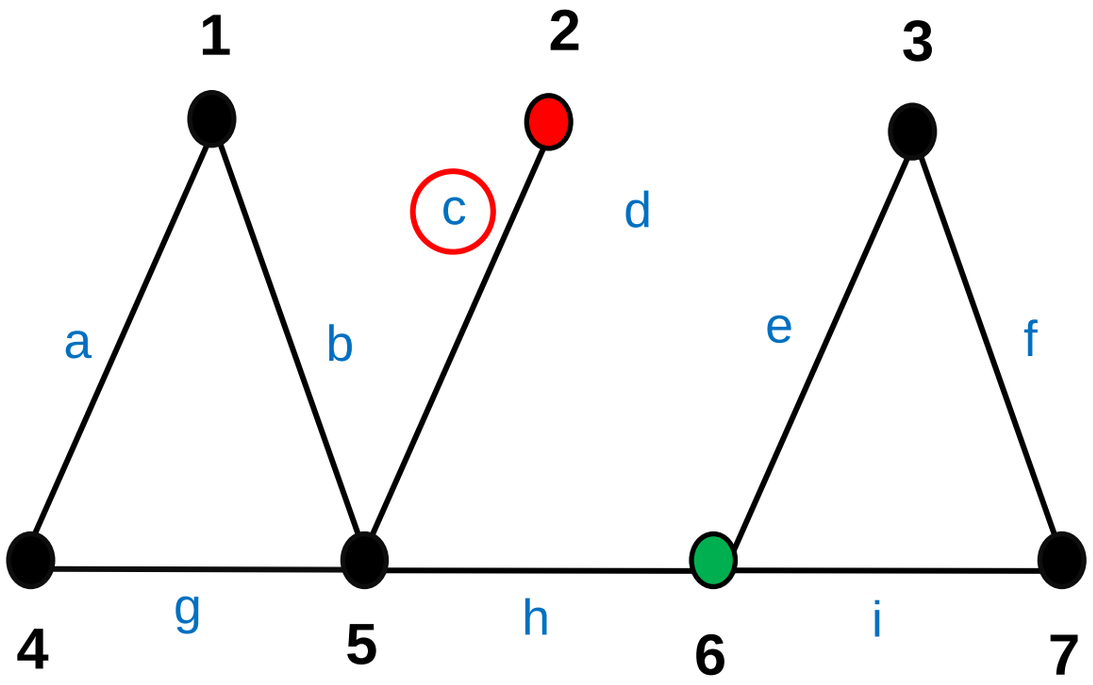
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

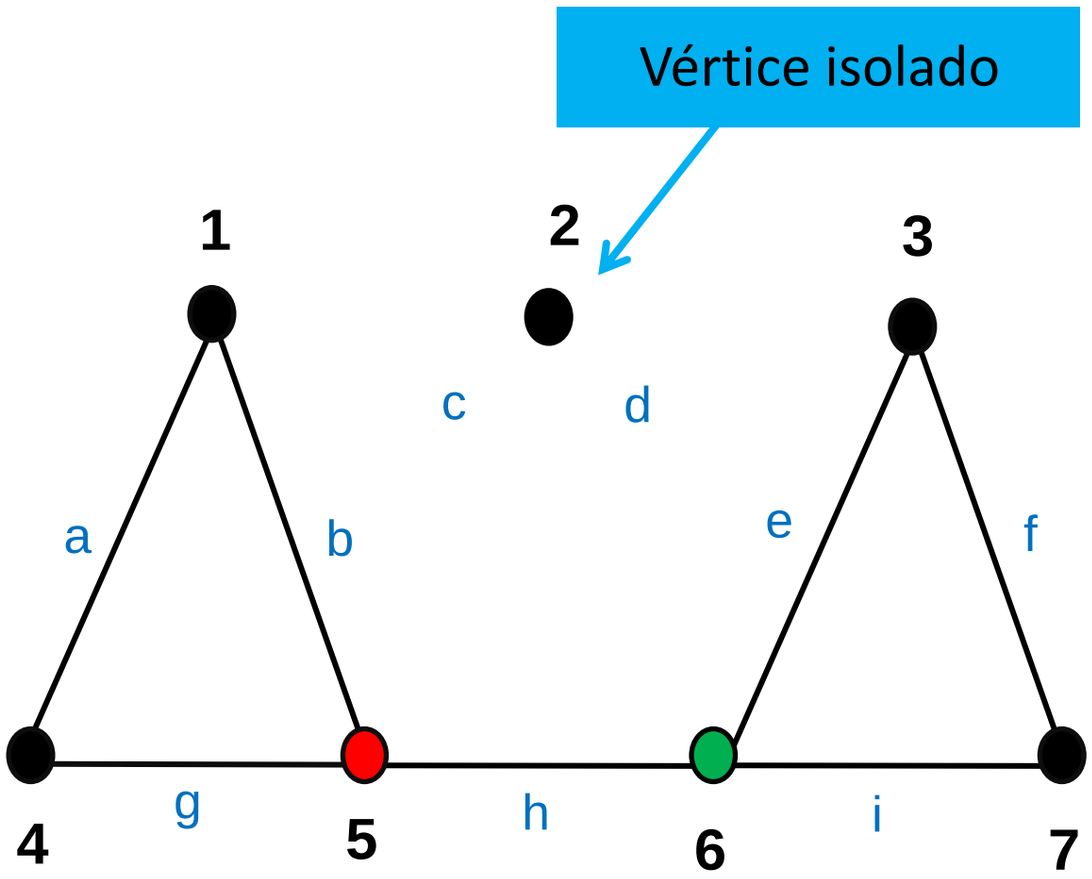
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

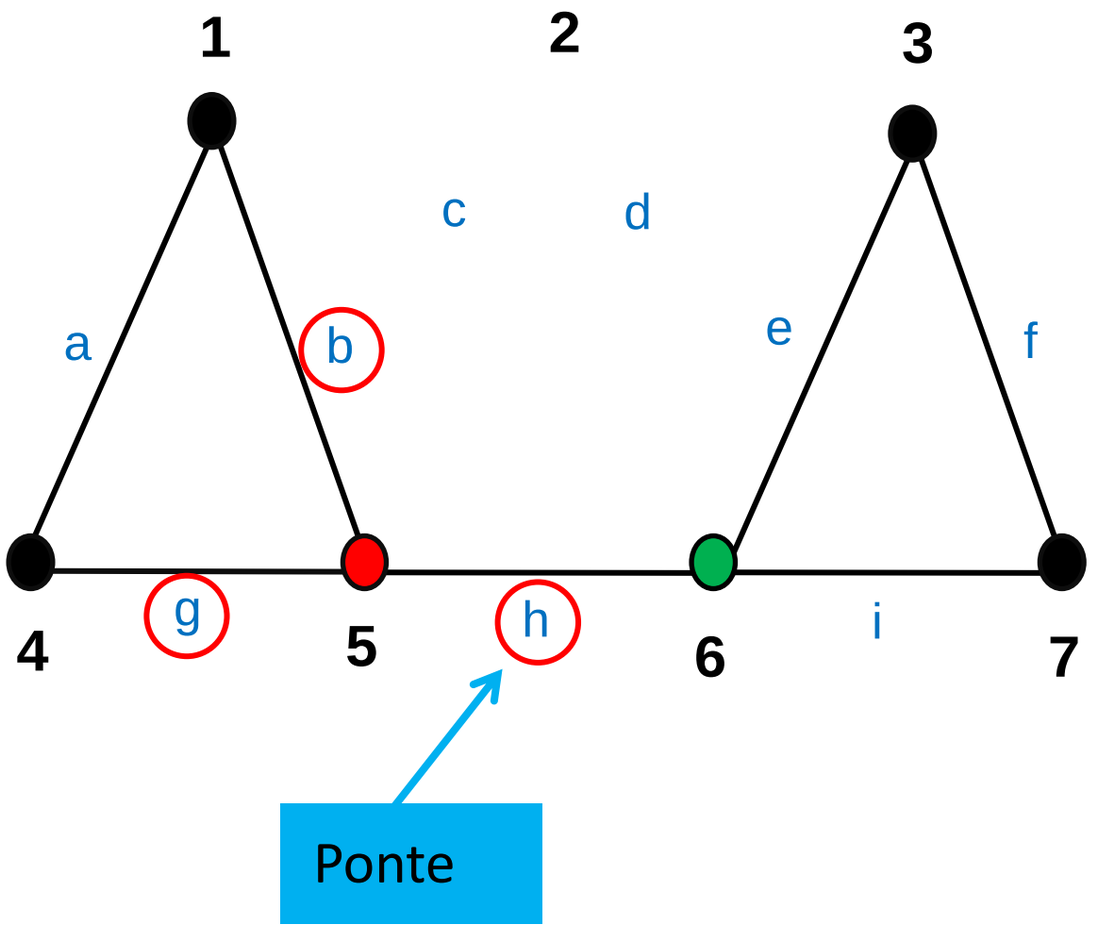
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

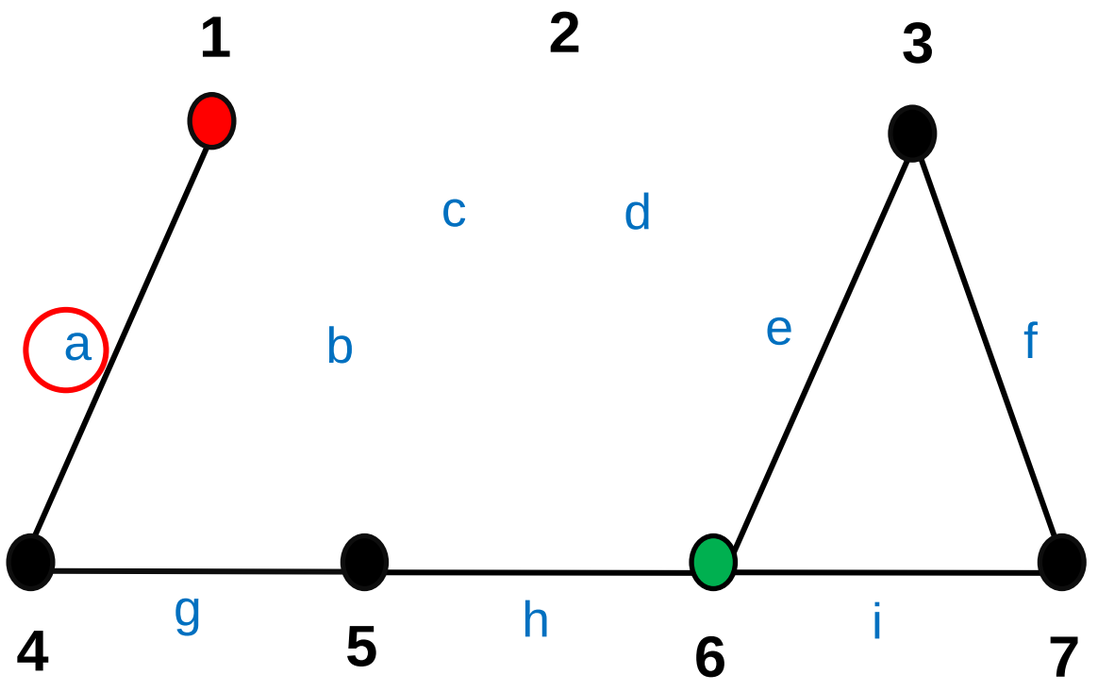
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

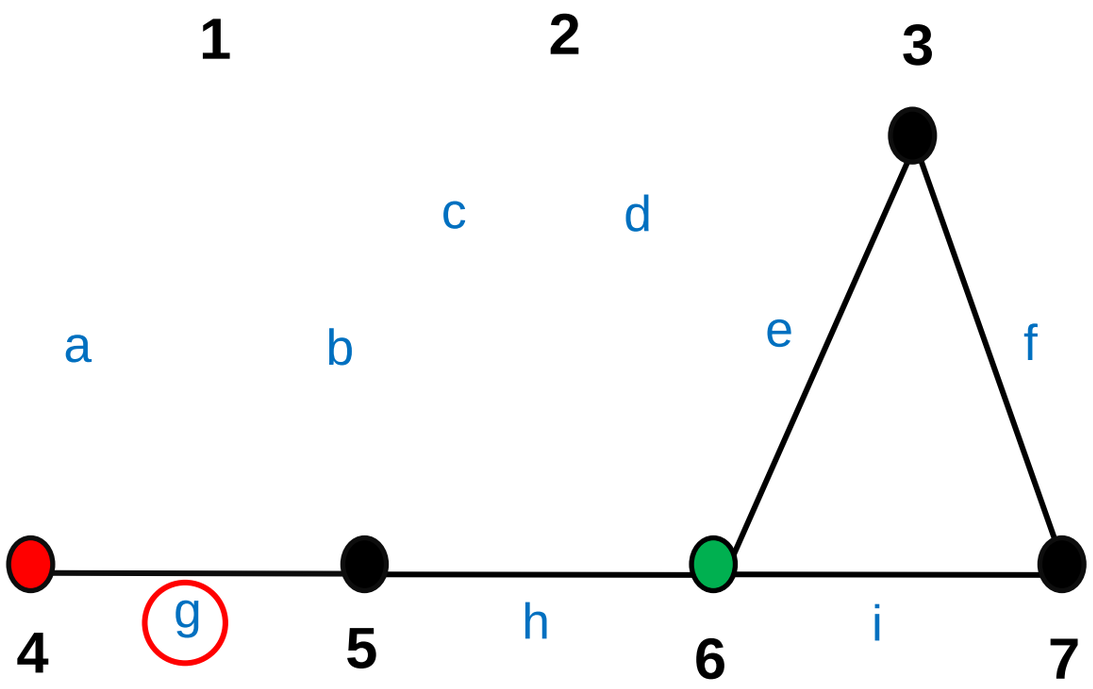
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

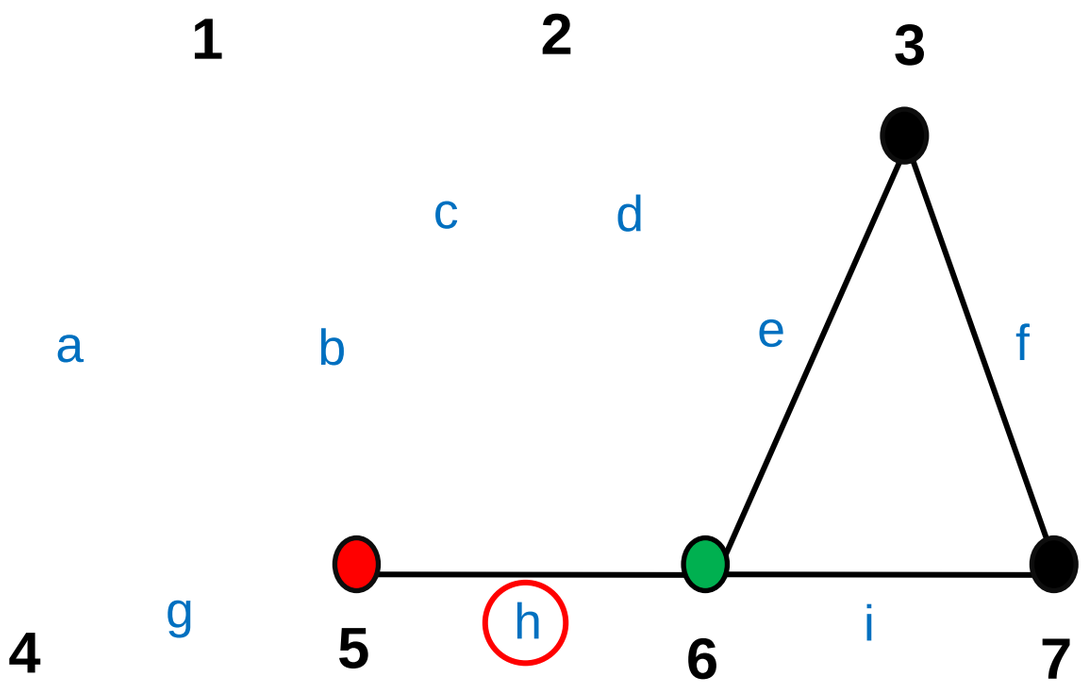
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

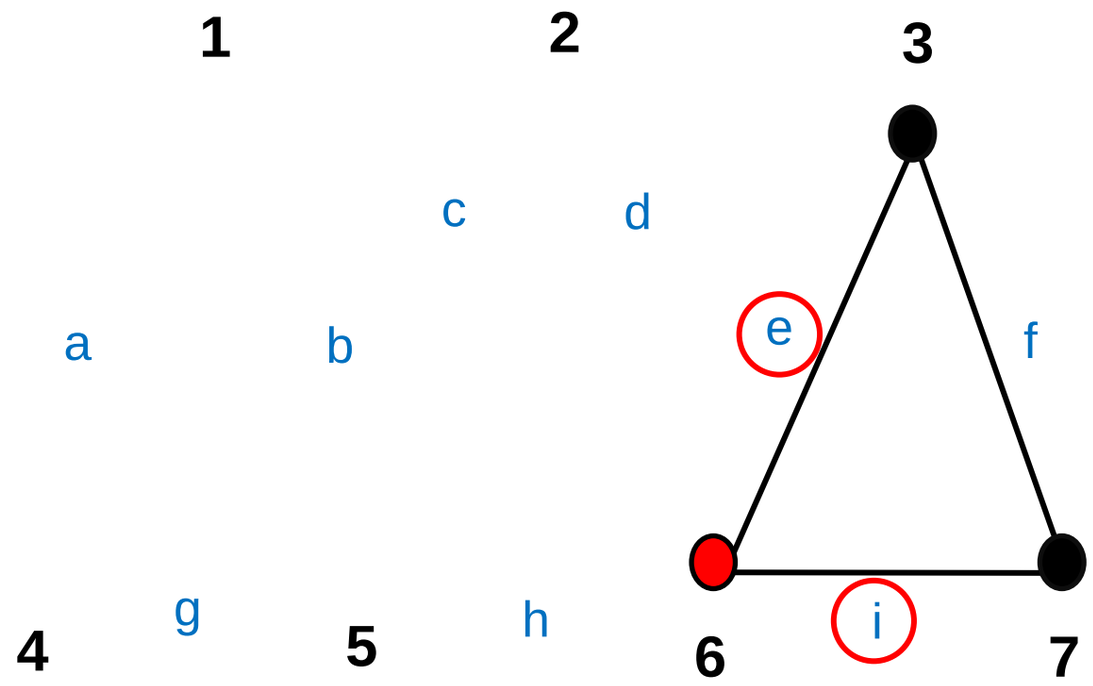
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

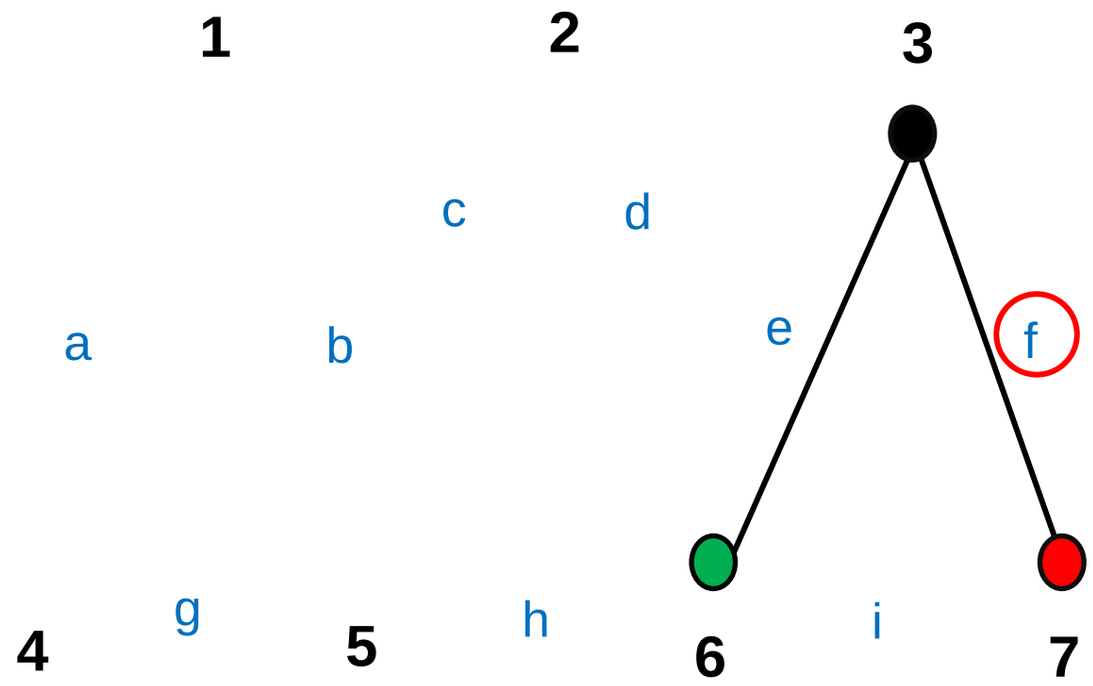
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

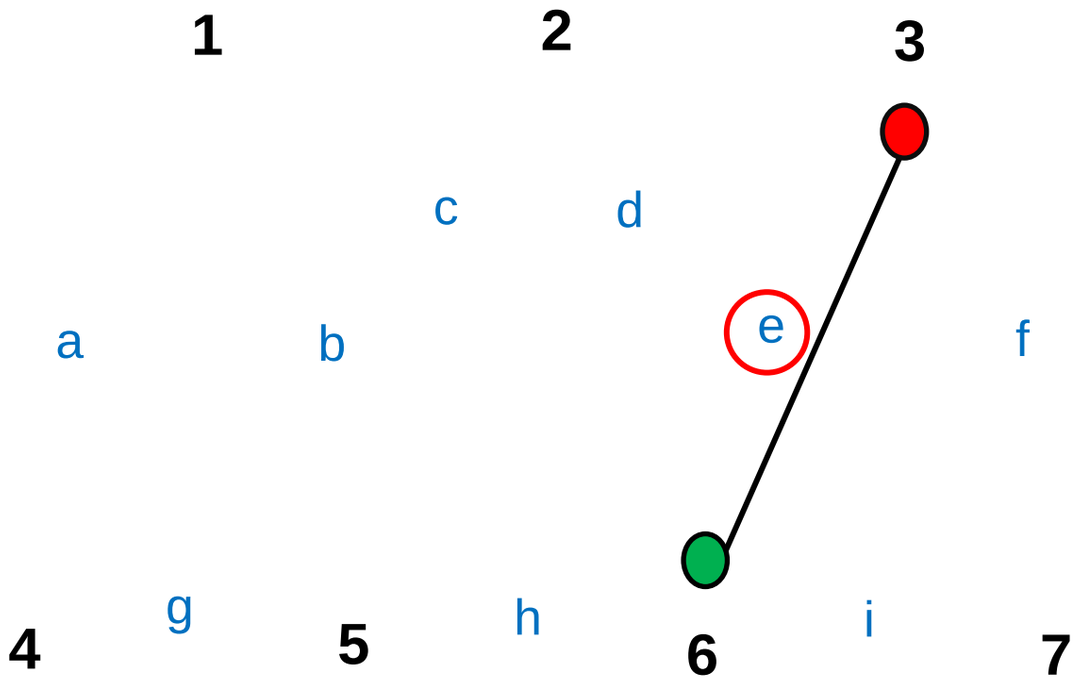
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

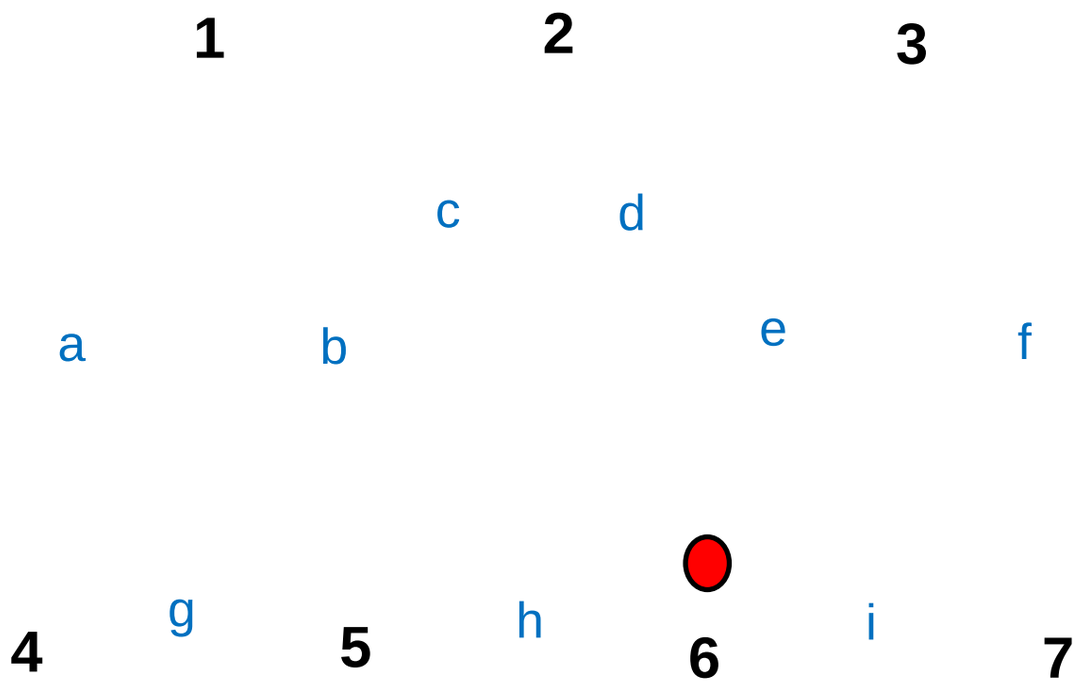
Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Algoritmo de Fleury

Exemplo



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

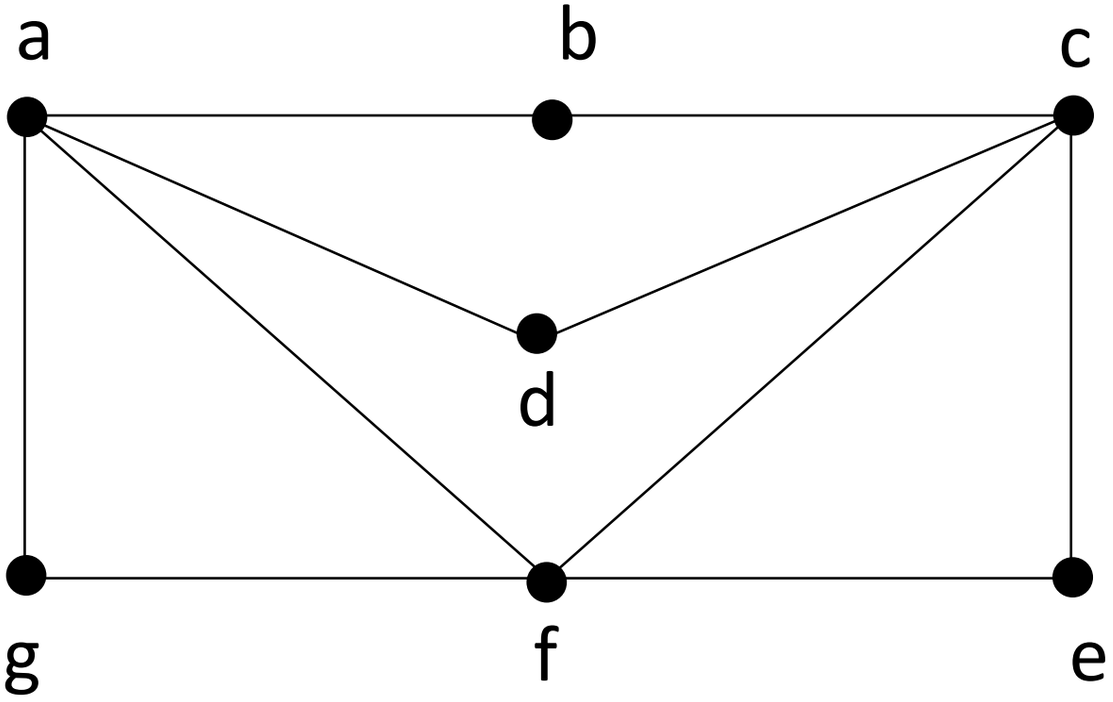
Regra III: passe por uma ponte somente se não houver outra alternativa

Ciclo Euleriano:

6,2,5,1,4,5,6,7,3,6

Exercícios

Descubra o ciclo euleriano do grafo abaixo:



Comece em qualquer vértice e percorra as arestas de forma aleatória, seguindo sempre as seguintes regras:

Regra I: apague as arestas depois de passar por elas

Regra II: se aparecer algum vértice isolado, apague-o também

Regra III: passe por uma ponte somente se não houver outra alternativa

Problema do carteiro chinês

Suponha que no grafo de rede rodoviária necessita-se determinar o custo do menor caminho partindo de um vértice inicial, passando por todas as arestas uma única vez e retornando ao ponto de origem.

- **Solução 1**: se o grafo for de Euler, então o caminho pode ser achado, e consequentemente seu respectivo custo, através do **algoritmo de Fleury**. O custo é dado pela soma dos custos de todas as arestas do grafo.
- **Solução 2**: Se o grafo não for de Euler, então algumas arestas terão ser repetidas. Para resolver o problema, sugere-se o **algoritmo do carteiro chinês**

Como Eulerizar um grafo

Problema:
Se não tivermos nem um caminho nem um ciclo euleriano como podemos percorrer o grafo repetindo o menor número de arestas possíveis?

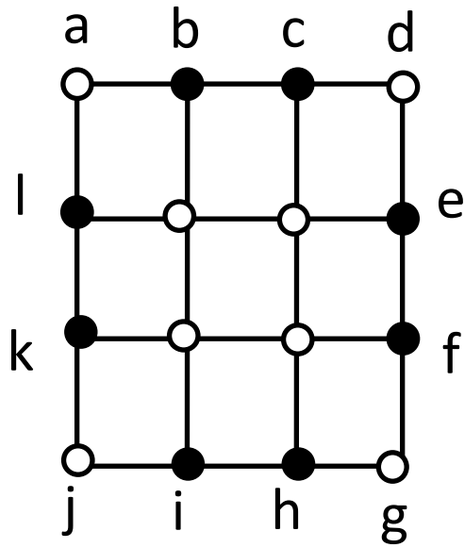
A **solução** deste problema consiste na eulerização de um grafo.

Eulerizar um grafo é juntar as arestas estritamente necessárias para que todos os vértices de grau ímpar se tornem vértices de grau par.

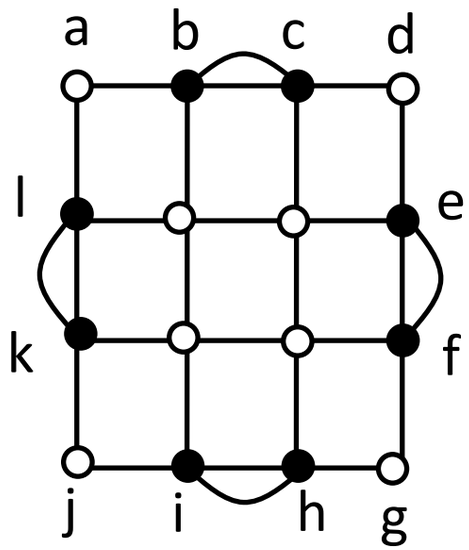
No entanto não podemos acrescentar arestas não existentes, apenas podemos duplicar as já existentes.

Como Eulerizar um grafo

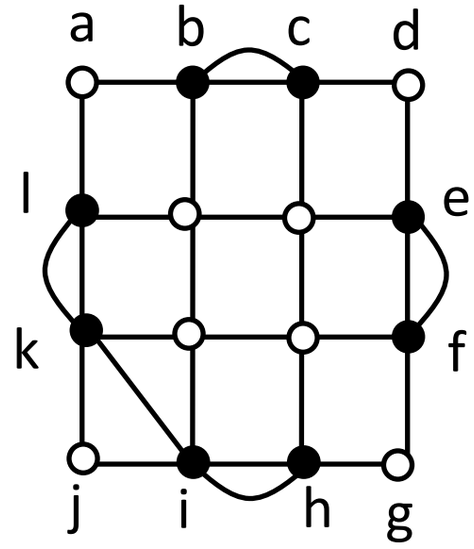
Exemplo:



grafo original



versão eulerizada



eulerização ilegal

Algoritmo do carteiro chinês

P1. Determine os vértices de grau ímpar

P2. Construa a matriz de distância D, com apenas os vértices de grau ímpar; [utilize o algoritmo de Dijkstra para determinar a matriz D]

P3. Determine, através da matriz D, o par de vértices v_i e v_j que contém o menor caminho

P4. Construa um caminho artificial de vi para vj com o custo encontrado de P3; [Este caminho artificial representa as arestas de menor custo que serão repetidas entre v_i e v_j]

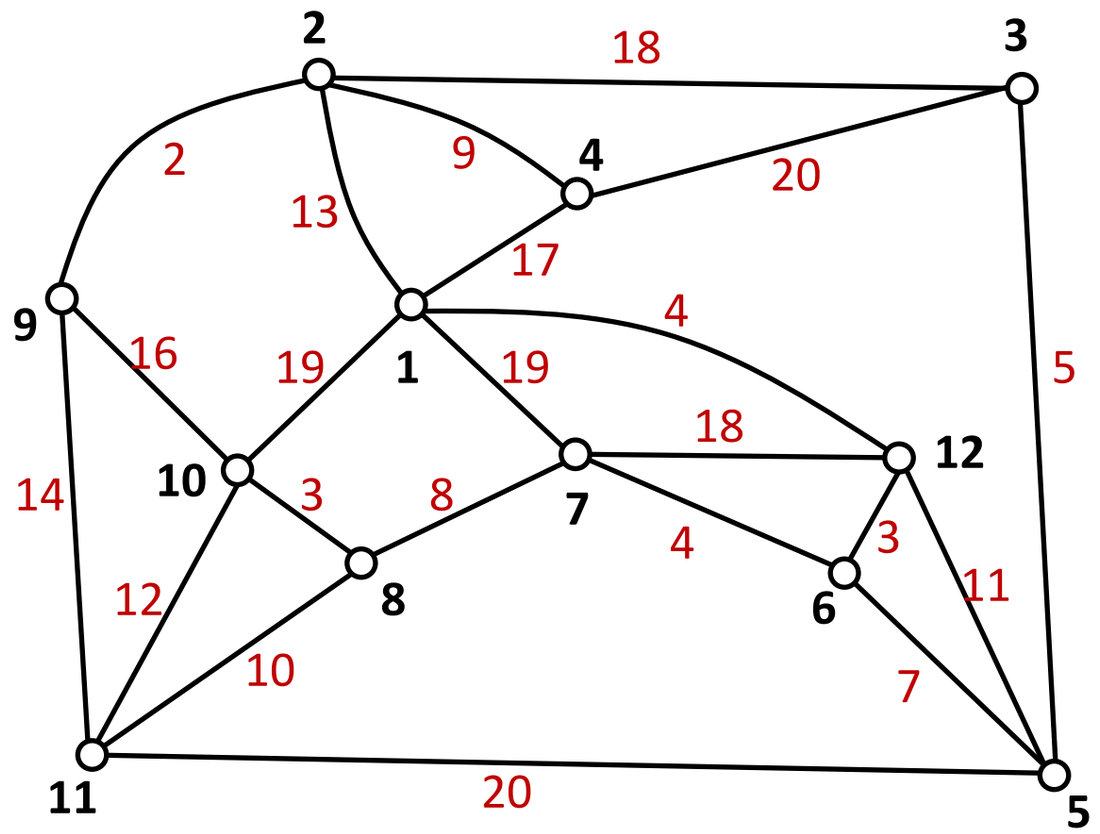
P5. Elimine da matriz D as linhas colunas correspondentes a v_i e v_j

P6. Se ainda houver linha e coluna, então volte para P3

P7. Oriente o grafo

P8. O custo será igual à soma dos custos de todas as arestas acrescida dos custos das arestas encontradas em P3

Algoritmo do carteiro chinês



O grafo em questão não é de Euler; logo, algumas arestas terão que ser repetidas

P1. Os vértices de grau ímpar são $V_1 = \{1, 3, 4, 6, 8, 9\}$

Algoritmo do carteiro chinês

P2. A matriz de distâncias com os vértices de grau ímpar dado por v1 é:

	1	**3**	**4**	**6**	**8**	**9**
1	0	19	17	7	19	15
3	19	0	20	12	24	20
4	17	20	0	24	30	11
6	7	12	24	0	12	22
8	19	24	30	12	0	19
9	15	20	11	22	19	0

P3. Determinam-se os pares vi e vj, em E, de vértices que produzem o menor caminho na matriz D. São:

1 com 6 – caminho 1-12-6 - custo 7

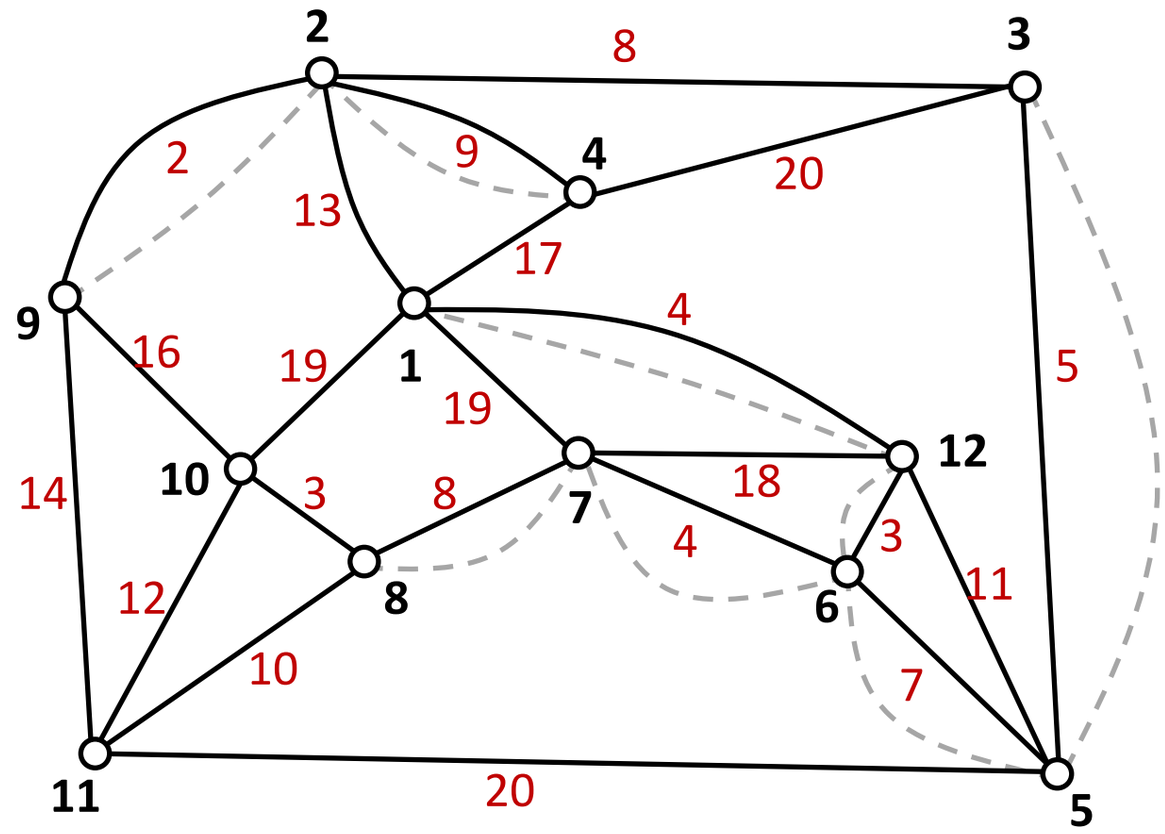
3 com 8 – caminho 3-5-6-7-8 - custo 24

9 com 4 – caminho 9-2-4 - custo 11

Resultado, portanto, no custo **7+24+11= 42**

Algoritmo do carteiro chinês

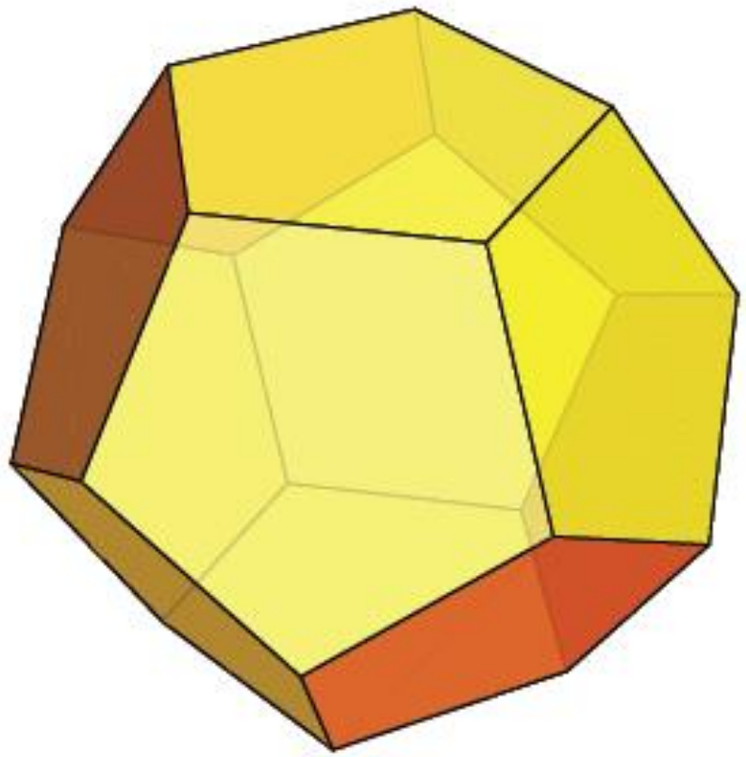
P4. Construindo um caminho artificial entre os vértices de grau ímpar, vi para vj, com o custo igual ao encontrado na matriz D, obtém-se a figura abaixo. As linhas tracejadas, além de estarem representando a construção dos caminhos artificiais



P5. Agora poderá ser usado um algoritmo para orientar o grafo, e determinar o custo total do caminho, que será dado por 42 acrescido do custo de todas as arestas.

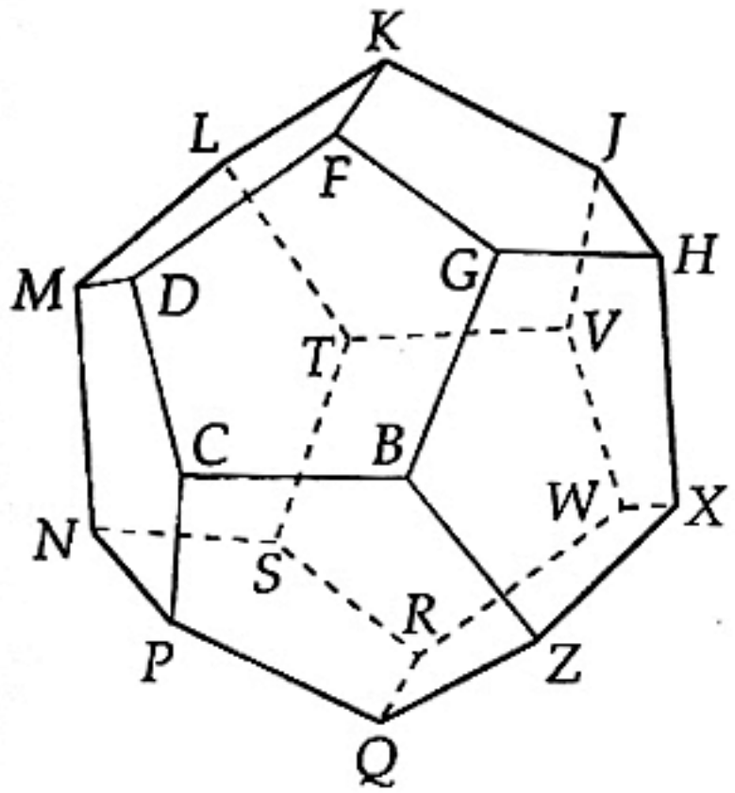
Grafos e Ciclos Hamiltonianos

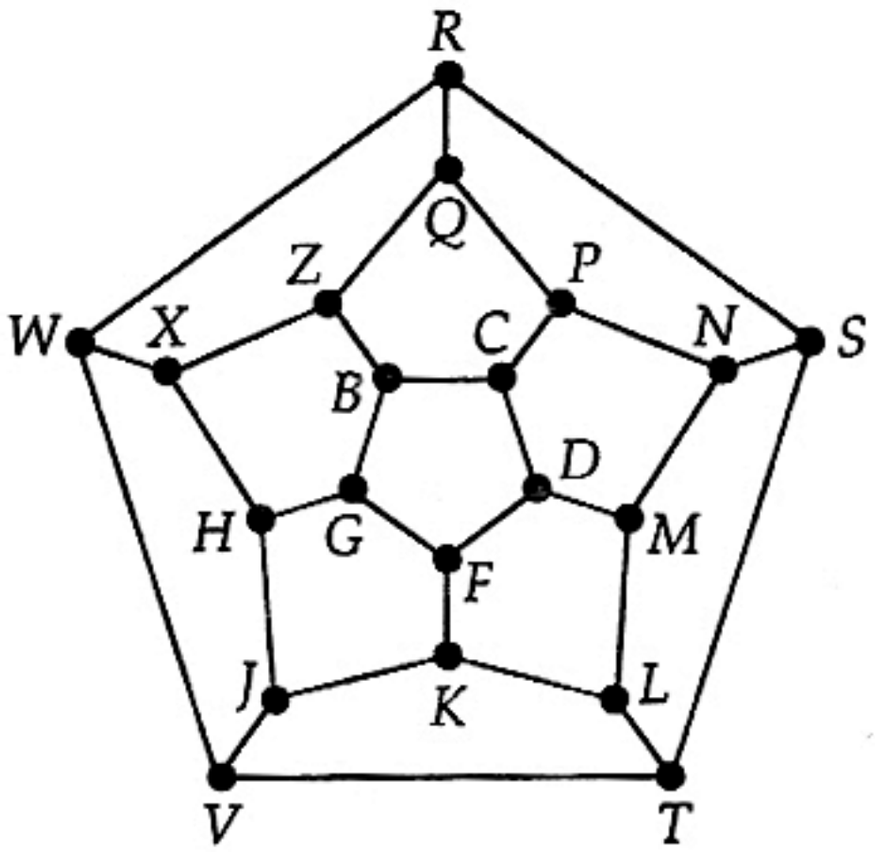
- Histórico: Sir William Rowan Hamilton (1805-1865).



Grafos e Ciclos Hamiltonianos

No “Jogo Icosiano” de Hamilton, os vértices do dodecaedro representam cidades no globo terrestre. O desafio é encontrar uma rota que dê a volta ao mundo passando somente uma vez em cada cidade.





Grafos Hamiltonianos

Um grafo conexo é **Hamiltoniano** se contiver um ciclo que inclua cada um dos vértices do grafo. Tal ciclo é chamado de **Ciclo hamiltoniano**

- Como podemos saber se um grafo é hamiltoniano?

  1. Teorema de Ore
  2. Teorema de Dirac
  3. Podemos tentar encontrar um ciclo hailtoniano utilizando algum algoritmo (Roberts e Flores).

Grafos Hamiltonianos

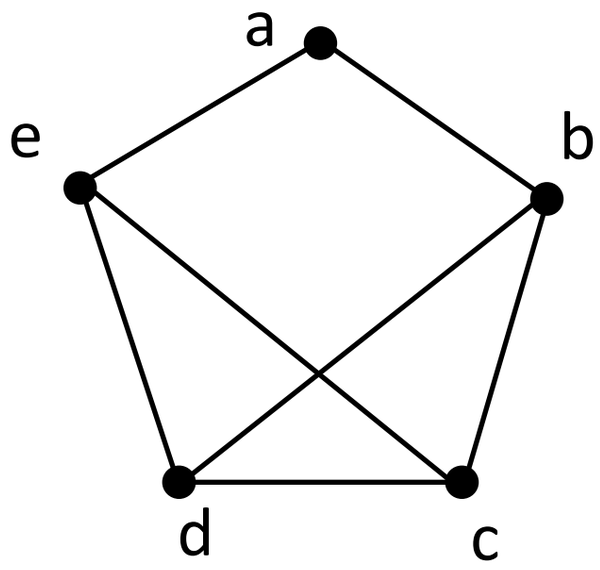
- Se tivermos um grafo hamiltoniano e adicionarmos a ele uma nova aresta, obtemos um novo grafo hamiltoniano, visto que continuamos tendo o mesmo ciclo hamiltoniano do grafo inicial.
- Assim, grafos mais densos têm maior probabilidade de serem hamiltonianos. Partindo desse princípio, Oysten Ore provou o seguinte teorema, em 1960:

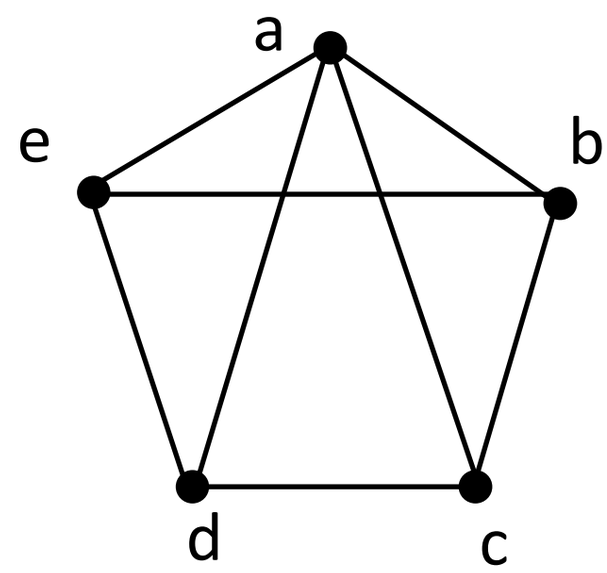
Teorema de Ore

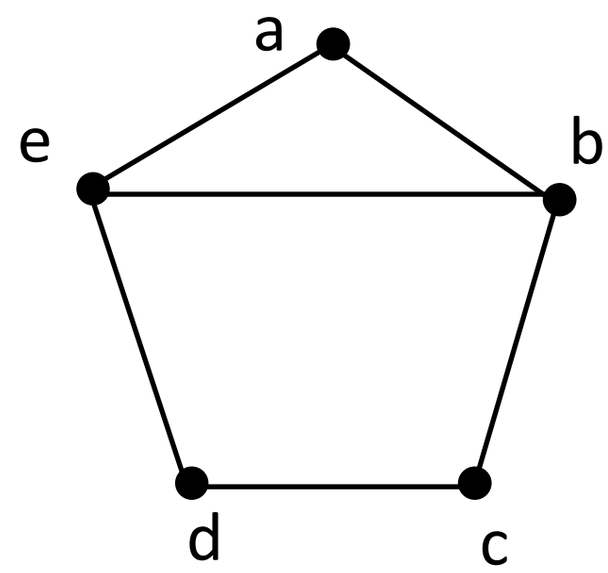
Seja G um grafo simples conexo com **n** vértices, onde n≥3 e **grau(u) + grau(v) ≥n**, para cada par de vértices não adjacentes u e v. Então G é hamiltoniano.

Exercícios

1. Verifique se os grafos abaixo são hamiltonianos utilizando o Teorema de Ore;







2. Dê um exemplo de grafo hamiltoniano que não satisfaz o Teorema de Ore.

Grafos Hamiltonianos

Teorema de Dirac

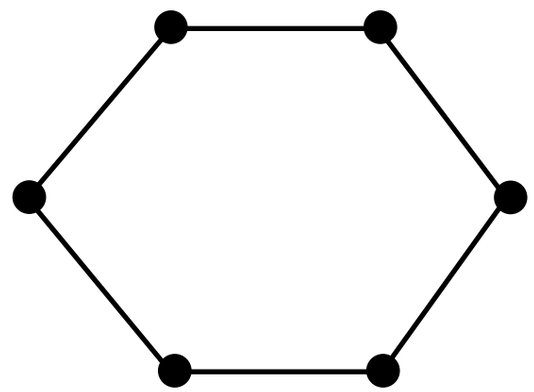
- Uma condição suficiente (mas não necessária) para que um grafo G seja hamiltoniano é que o grau de todo vértice de G seja no mínimo n/2, onde n é o número de vértices em G.

- Deste teorema podemos então concluir que: se o grau de todo vértice de G é no mínimo n/2, onde n é o número de vértices em G, então um grafo G é hamiltoniano

Grafos Hamiltonianos

Teorema de Dirac

Contra-exemplo: O grafo da figura é hamiltoniano e não respeita a condição do teorema B. O grau de cada vértice é 2, o que é menor que 6/2 = 3.

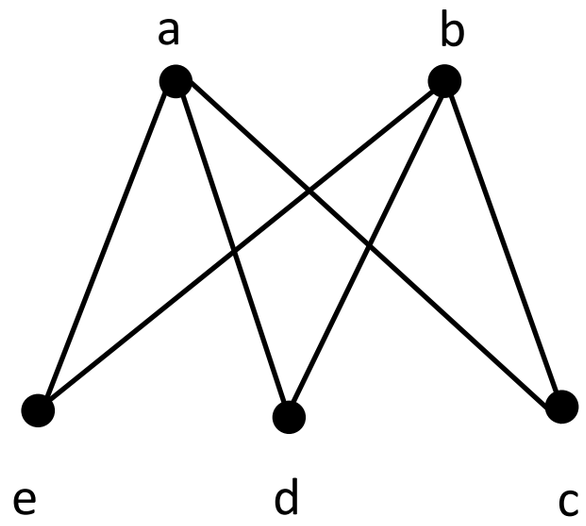


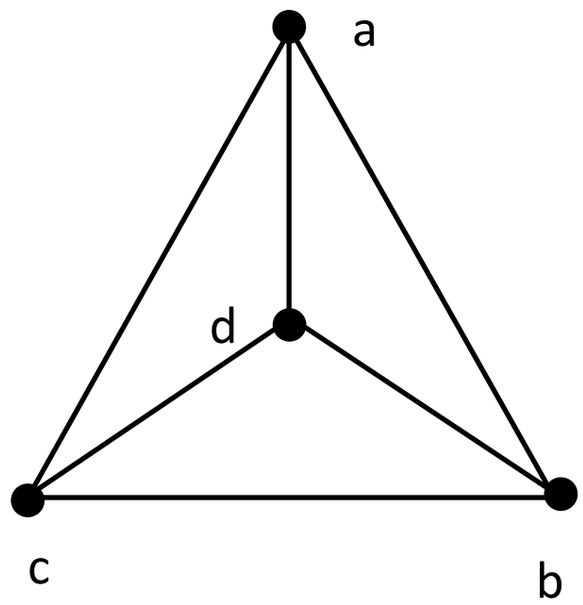
Conclusão: esses teoremas não são muito informativos. O que eles dizem, intuitivamente, é que se um grafo contém muitas arestas e que elas são bem distribuídas, ele é hamiltoniano. No limite, temos um grafo completo

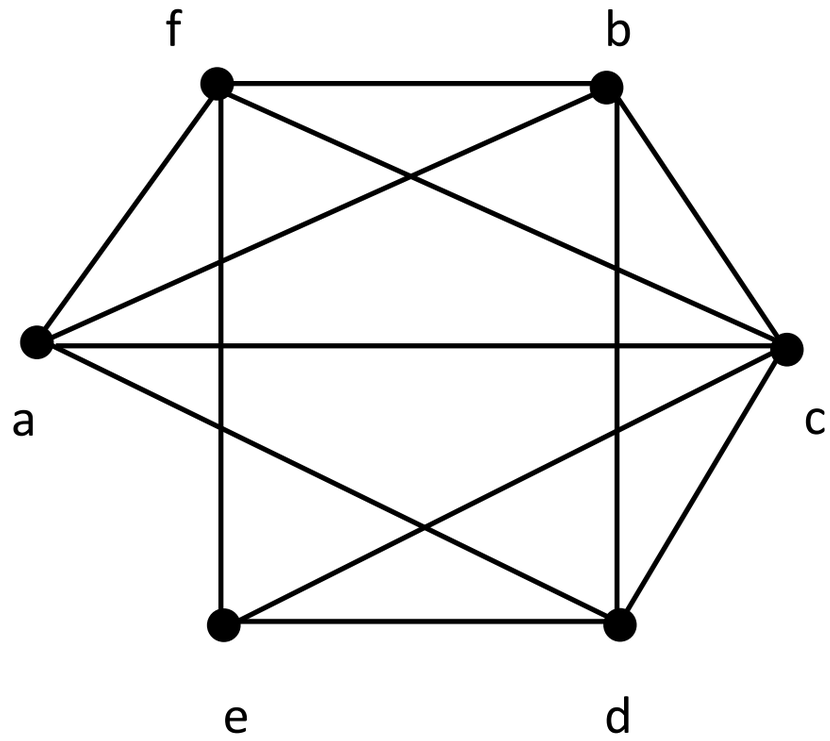
Grafos Semi-hamiltonianos

Definição:
Um grafo conexo é semi-hamiltoniano se existe um caminho simples aberto, mas não um ciclo, que inclua cada um dos seus vértices somente uma vez. Tal caminho é chamado de caminho semi-hamiltoniano.

Exercício: determine quais dos grafos a seguir são semi-hamiltonianos e escreva o caminho semi-hamiltoniano se possível:







Algoritmo de Roberts e Flores

Ideia

- O algoritmo parte de um vértice inicial e determina um caminho, se possível, que leva até o próximo vértice
- Se um vértice viável é encontrado, a busca reinicia a partir deste vértice
- Senão, realiza uma operação de “backtracking” na tentativa de encontrar um caminho possível
- A busca encerra quando todos os possíveis caminhos forem explorados

Algoritmo de Roberts e Flores

P1. Escolher um vértice inicial vi

P2. Incluir vi no conjunto de vértices visitados: S = { v_i }

P3. Adicionar a S o primeiro vértice viável ($v_j \notin S$) destino do último vértice inserido em S (v_i, na primeira iteração)

P4. Repetir o passo P3 enquanto houver vértice viável que seja destino do último vértice em S

P5. Se S contém todos os vértices de G, então a seqüência encontrada é um caminho Hamiltoniano: {v_i, v_j, ..., v_r}. Se existir uma aresta (v_r, v_i), existe um ciclo Hamiltoniano que deve ser armazenado.

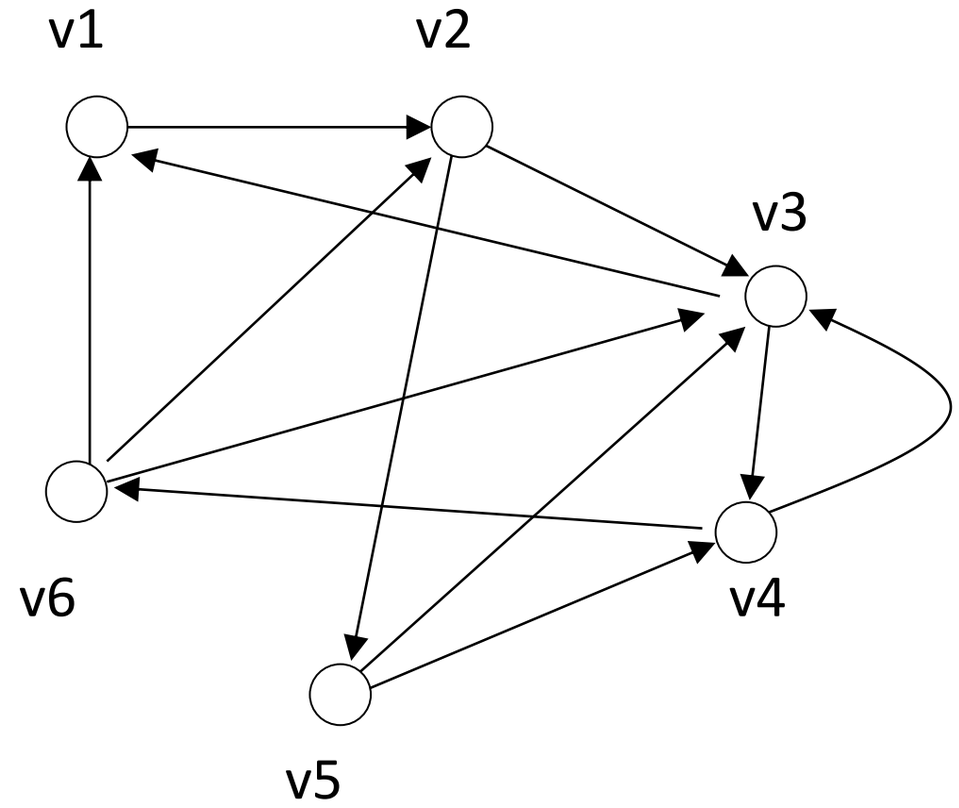
P6. Quando não houver vértice viável, realizar um back-tracking, removendo de S o último vértice adicionado. Retornar ao passo P3. O processo termina quando S contém apenas o vértice v_i e não existir mais nenhum vértice viável a ser adicionado a S.

Exemplo

- Encontrar o menor caminho passando por todos os vértices do grafo abaixo uma única vez e retornar à origem (v1)
- Vértice inicial: v1
- Adjacências

M=

v1	**v2**	**v3**	**V4**	**v5**	**V6**
v2	V3 v5	V1 v4	V3 v6	V3 v4	V1 V2 V3



Algoritmo de Roberts e Flores

P1. Considere o vértice inicial v1 S={v1};

P2. Adicione a S o primeiro vértice viável na coluna de v1; S = { v1, v2}

P3. Continuando o processo tem-se: S = {v1,v2,v3}, S = {v1,v2,v4} e S = {v1,v2,v3,v4,v6}

P4. Observe que na coluna de v6 não existe vértice que seja viável, então faz-se "backtracking" e obtém-se S={v1,v2,v3,v4}

P5. Não existe vértice viável na coluna do v4. Fazendo "backtracking" resultará S = {v1,v2,v3}

P6. Não existe vértice viável na coluna do v3, então o "backtracking" dará S={v1,v2}

P7. Adicionando a S o vértice viável pertencente à coluna de v2, resulta S={v1,v2,v5}

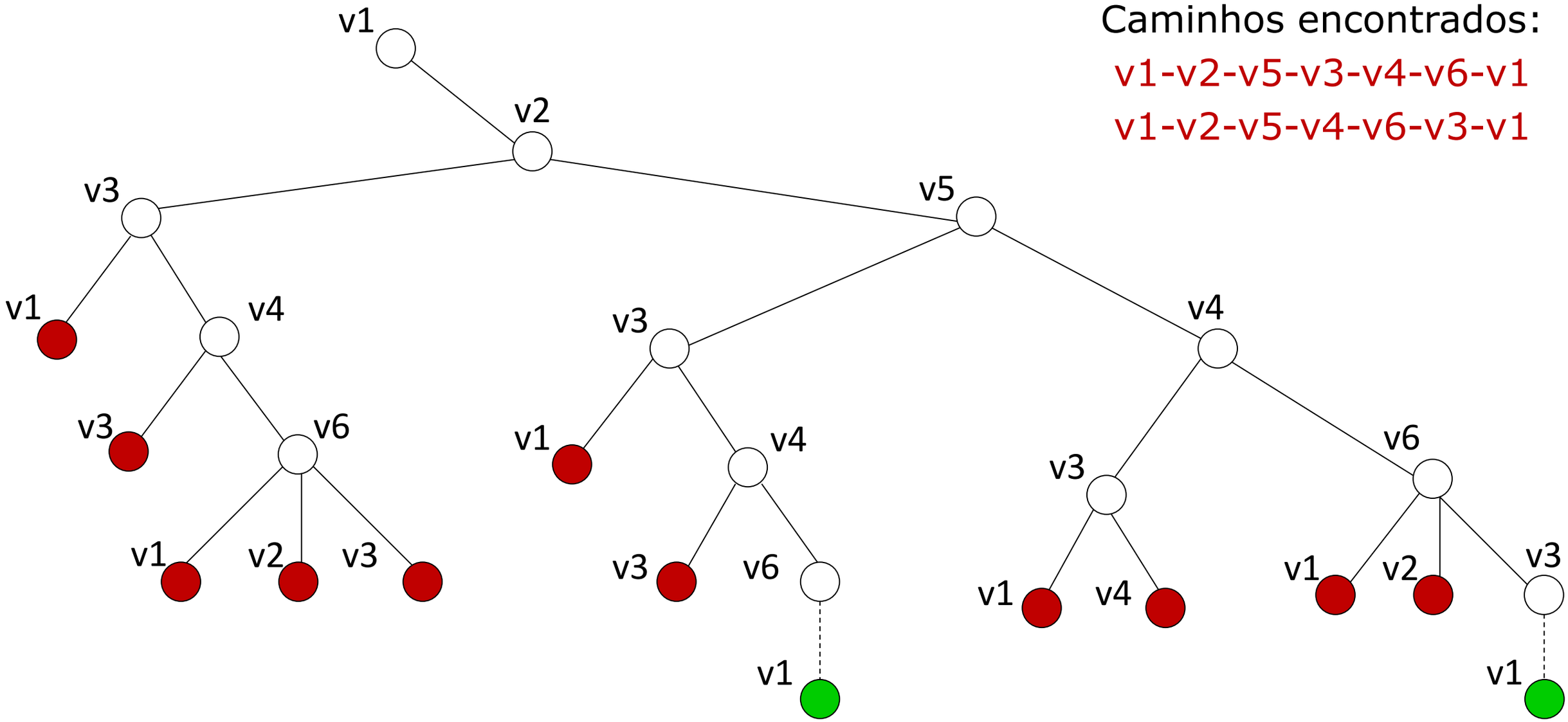
P8. Continuando o processo tem-se: S={v1,v2,v5,v3}, S={v1,v2,v5,v3,v4} e S={v1,v2,v5,v3,v4,v6}

Este último é um ciclo hamiltoniano porque existe a aresta {v6,v1}

Fazendo "backtracking" sucessivos pode-se determinar os outros ciclos hamiltonianos, se existirem

Execução Passo a Passo

- O algoritmo monta uma árvore de busca passando por todos os possíveis caminhos



Caminhos encontrados:

v1-v2-v5-v3-v4-v6-v1

v1-v2-v5-v4-v6-v3-v1

Problema do caixeiro viajante

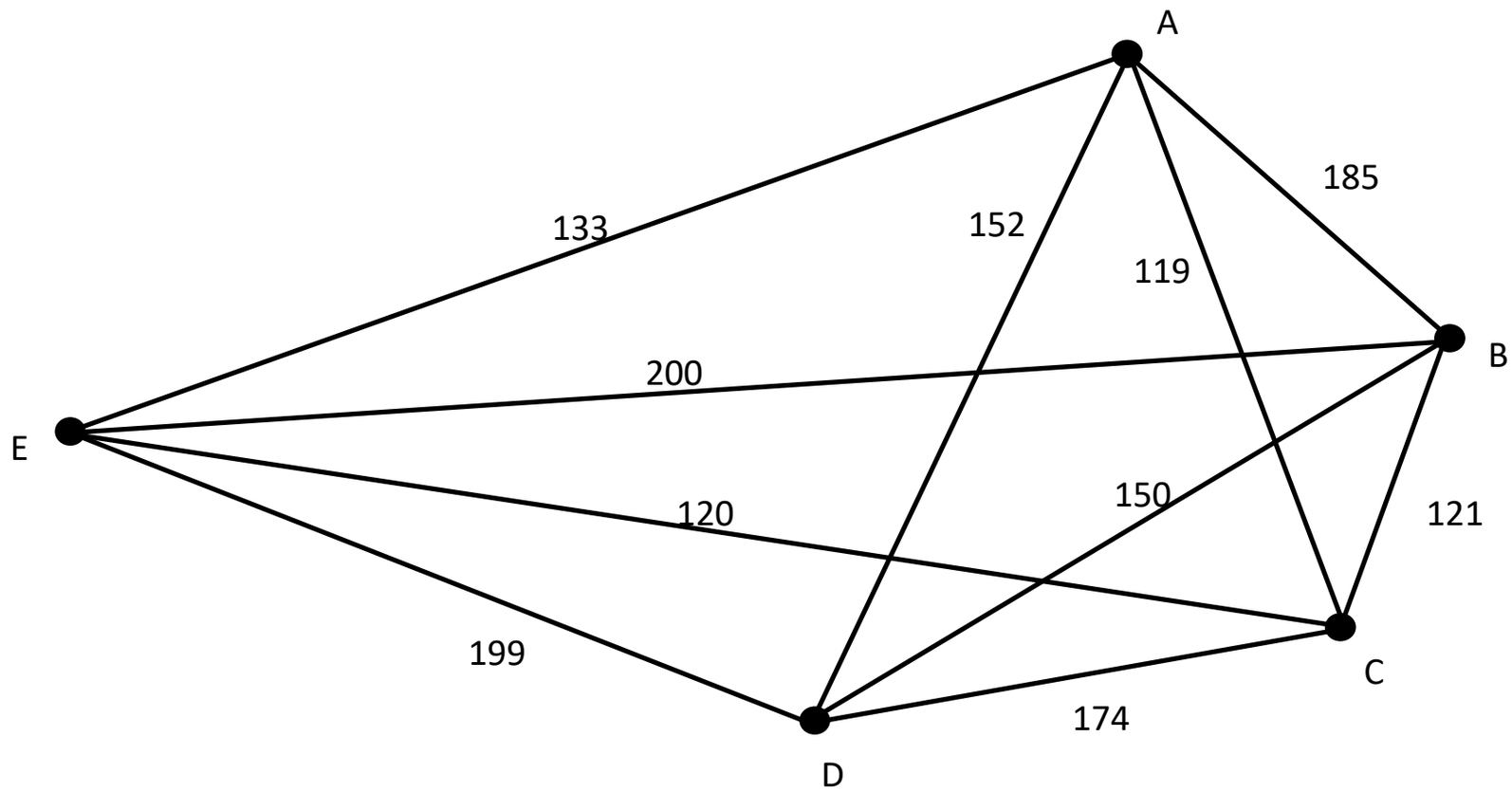
Definição:

- Geralmente chamado de TSP (**Travelling Salesman Problem**), é um problema de otimização que pode ser modelado por grafos.
- Consiste na procura por um circuito (começa e termina no mesmo vértice) que possua a menor distância, que, partindo de uma cidade, visita cada cidade precisamente uma vez e regresse à cidade inicial

Problema do caixeiro viajante

Formulação combinatório:

- Dado um conjunto de cidades C = {c_1, ... , c_n} e uma matriz de distâncias $P_{i,j}$ de tamanho n x n, encontrar a permutação entre os caminhos possíveis no qual a soma dos pesos seja a mínima



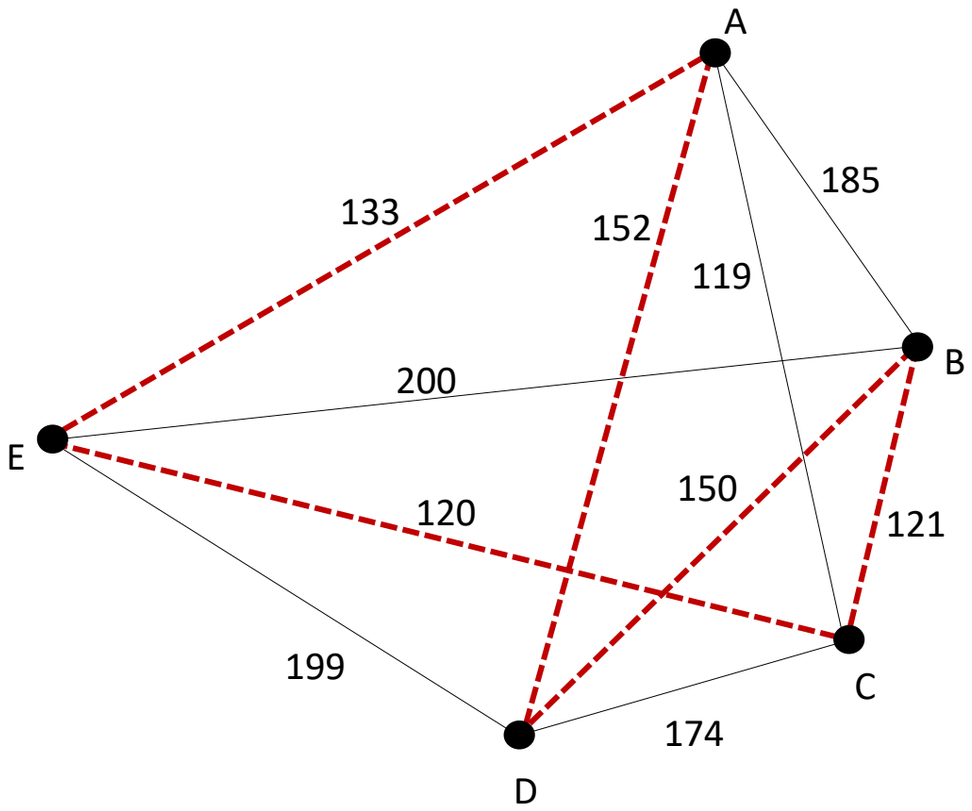
Método exaustivo

Ideia

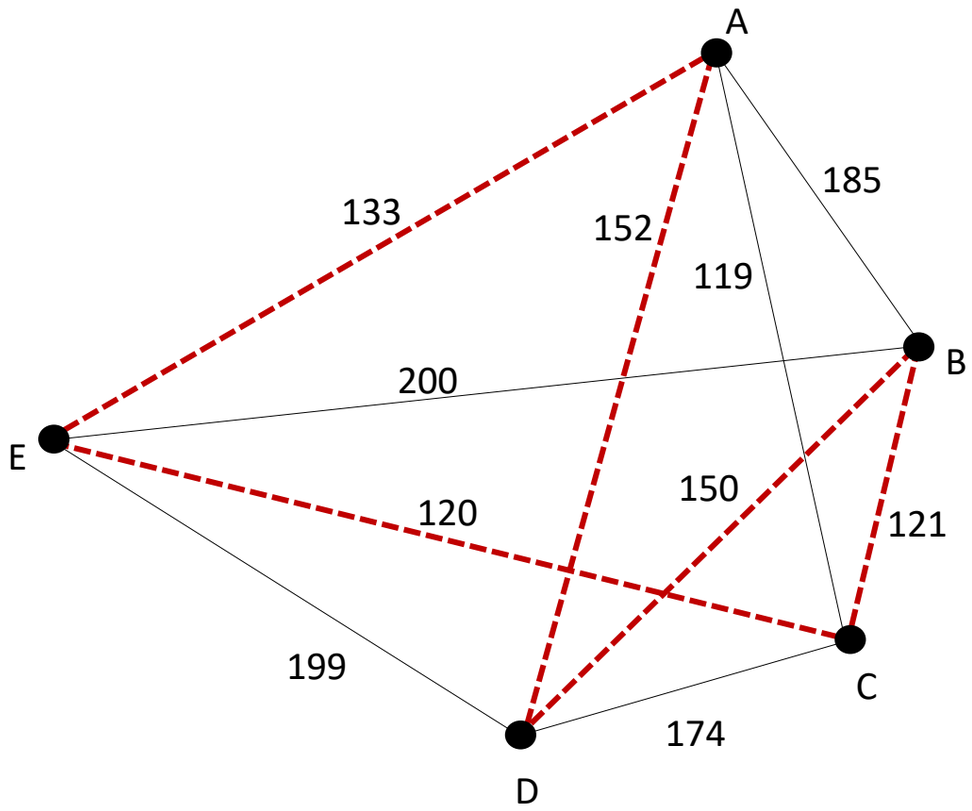
- Faz-se uma lista de todos os circuitos de Hamilton possíveis.
- Calcula-se o custo total de cada circuito, adicionando o peso de todas as arestas do circuito.
- Escolhemos um circuito com o custo total mínimo de acordo com a formulação do problema. A este circuito chamamos **Circuito Ótimo de Hamilton**.

Método exaustivo

Estas são as melhores soluções (custo 676)



A - D - B - C - E - A



A - E - C - B - D - A

Método exaustivo

	Circuito Hamiltoniano	Custo	Circuito inverso
1	A,B,C,D,E,A	185 + 121 + 174 + 199 + 133 = 812	A,E,D,C,B,A
2	A,B,C,E,D,A	185 + 121 + 120 + 199 + 152 = 777	A,D,E,C,B,A
3	A,B,D,C,E,A	185 + 150 + 174 + 120 + 133 = 762	A,E,C,D,B,A
4	A,B,D,E,C,A	185 + 150 + 199 + 120 + 119 = 773	A,C,E,D,B,A
5	A,B,E,C,D,A	185 + 200 + 120 + 174 + 152 = 831	A,D,C,E,B,A,
6	A,B,E,D,C,A	185 + 200 + 199 + 174 + 119 = 877	A,C,D,E,B,A
7	A,C,B,D,E,A	119 + 121 + 150 + 199 + 133 = 722	A,E,D,B,C,A
8	A,C,B,E,D,A	119 + 121 + 200 + 199 + 152 = 791	A,D,E,B,C,A
9	A,C,D,B,E,A	119 + 174 + 150 + 200 + 133 = 776	A,E,B,D,C,A
10	A,C,E,B,D,A	119 + 120 + 200 + 150 + 152 = 741	A,D,B,E,C,A
11	A,D,B,C,E,A	152 + 150 + 121 + 120 + 133 = 676	A,E,C,B,D,A
12	A,D,C,B,E,A	152 + 174 + 121 + 200 + 133 = 780	A,E,B,C,D,A

Problema do caixeiro viajante

Complexidade:

- O tamanho do espaço de procura aumenta exponencialmente com relação ao número de cidades, pois os circuitos possíveis são: **(n-1!) / 2**
- Pertence à categoria NP-Completo, com complexidade exponencial
- Não existe um algoritmo eficiente que encontra a solução ótima, sendo necessárias aproximações e heurísticas, como os métodos de construção de circuitos

Heurísticas para o PCV

Construtivas

- Algoritmo do vizinho mais próximo
- Algoritmo repetitivo do vizinho mais próximo
- Algoritmo da ligação mais econômica

Melhorativas

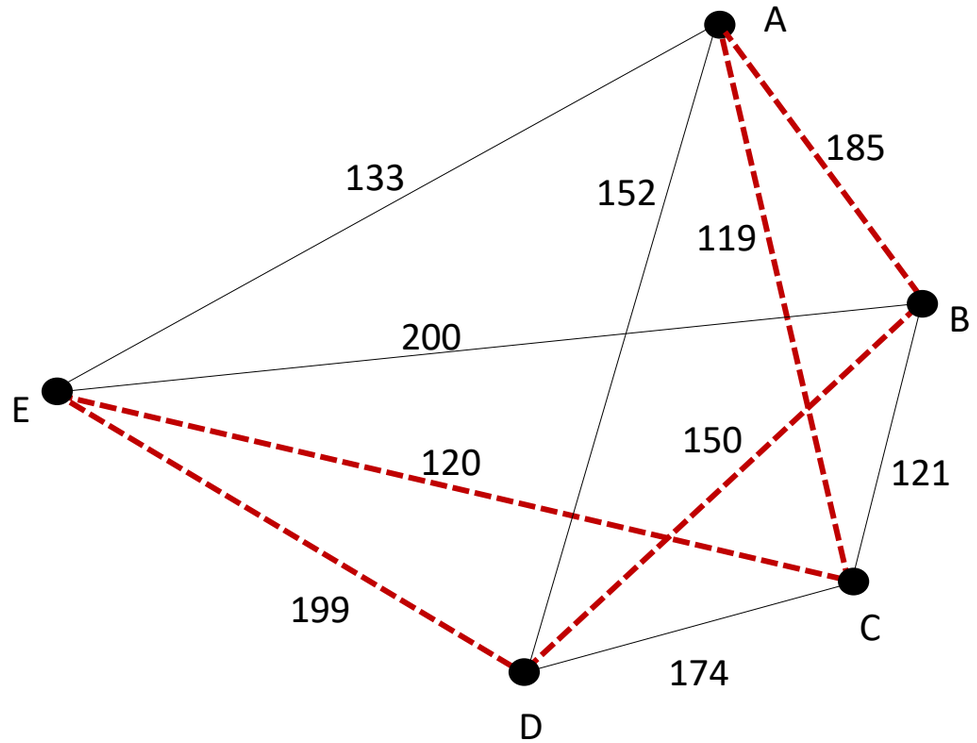
- Simulated Annealing
- Algoritmos Genéticos
- Busca Tabu

Método do Vizinho mais próximo

Ideia

- Escolha um vértice inicial
- Dentre as arestas adjacentes escolhemos a que tem menor peso e partimos para o vértice correspondente, a que chamamos ***vizinho mais próximo***
- Continuamos a construir o circuito, um vértice de cada vez, indo sempre para o vértice que representa o vizinho mais próximo, escolhendo este último entre os vértices que ainda não foram visitados. E prosseguimos assim sucessivamente até que todos os vértices sejam visitados
- Chegados ao último vértice retornamos ao vértice inicial

Método do Vizinho mais próximo



P1. Olhamos para o grafo e procuramos a viagem mais barata partindo de **A**, ou seja, a deslocação para a cidade **C**

P2. Repetindo o processo vamos da cidade **C** para **E**.

P3. A aresta com menor peso partindo de **E** seria a que liga **E** a **A**, no entanto não a podemos escolher pois fecharia um circuito (**A, C, E, A**) havendo ainda vértices por visitar. Então a aresta menor ponderada será a que liga **E** a **D**.

P4. Seguidamente vamos de **D** para **B**, e por fim retornamos à cidade de partida **A**.

Usando esta estratégia obtemos o circuito **A, C, E, D, B, A** exibido na figura a vermelho com um custo total de **773**.

Exaustivo X Vizinho mais próximo

O ***Algoritmo Exaustivo ou Força Bruta*** é um exemplo clássico do que é formalmente conhecido como um *algoritmo ineficiente*

- É um algoritmo cujo número de passos necessários para obter uma solução cresce desproporcionalmente com o tamanho do problema
- Sendo um algoritmo ineficiente a sua aplicação prática restringe-se a pequenos problemas.

O ***Algoritmo do Vizinho mais Próximo*** é um exemplo de um *algoritmo eficiente* porque em cada passo há uma melhor escolha a ser feita com base num critério apropriado e bem conhecido. Infelizmente, o resultado pode estar longe de ser um circuito ótimo de Hamilton

Algoritmos Aproximados

Para tentar sanar os problemas:

- da **lentidão de obtenção da solução ótima** pelo algoritmo da força bruta

- da obtenção, pelo algoritmo do vizinho mais próximo, de uma **solução rápida** mas por vezes apenas **aproximada da solução ótima**

Foram criados os algoritmos aproximados:

1. *Algoritmo repetitivo do vizinho mais próximo*
2. *Algoritmo da ligação mais econômica*

Algoritmo repetitivo do vizinho mais próximo

Este algoritmo é uma variação do algoritmo do vizinho mais próximo no qual se repete várias vezes integralmente o processo de construção do circuito do vizinho mais próximo.

Porquê repetir?

- Porque o circuito de Hamilton obtido depende do vértice de partida, e ao alterarmos este vértice podemos obter um circuito Hamiltoniano diferente e com alguma sorte até mais aproximado da solução ótima

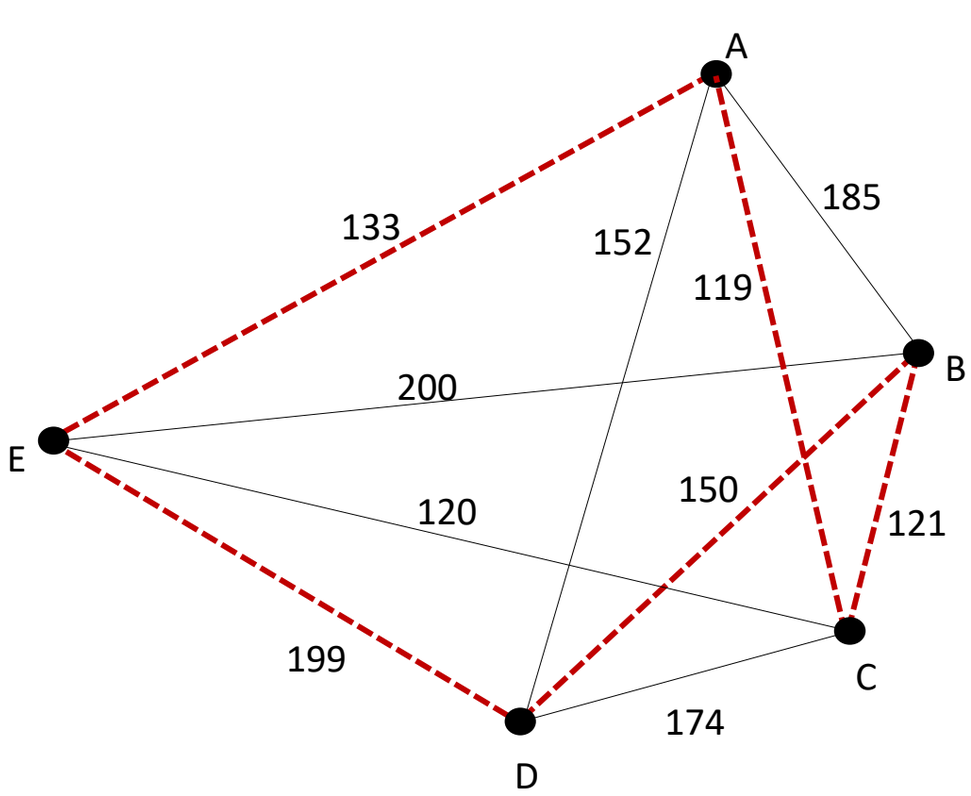
Algoritmo repetitivo do vizinho mais próximo

Ideia

- Seja X um vértice qualquer. Apliquemos o algoritmo do vizinho mais próximo usando X como vértice inicial e calculemos o custo total do circuito obtido
- Repetimos o processo usando os outros vértices do grafo como vértice inicial
- Dos circuitos hamiltonianos obtidos escolhemos o melhor, ou seja, o que tem custo mínimo. Se à partida estiver definido um vértice especifico como vértice inicial e este não coincidir com o do circuito hamiltoniano escolhido então tem-se que rescrever este mesmo circuito tomando como vértice inicial o vértice previamente estabelecido

Algoritmo repetitivo do vizinho mais próximo

Pelo Algoritmo do vizinho mais próximo tínhamos obtido como circuito óptimo A, D, B, C, E, A com o custo total de 773.



Repetimos agora o algoritmo do vizinho mais próximo usando como vértice inicial os restantes vértices do grafo:

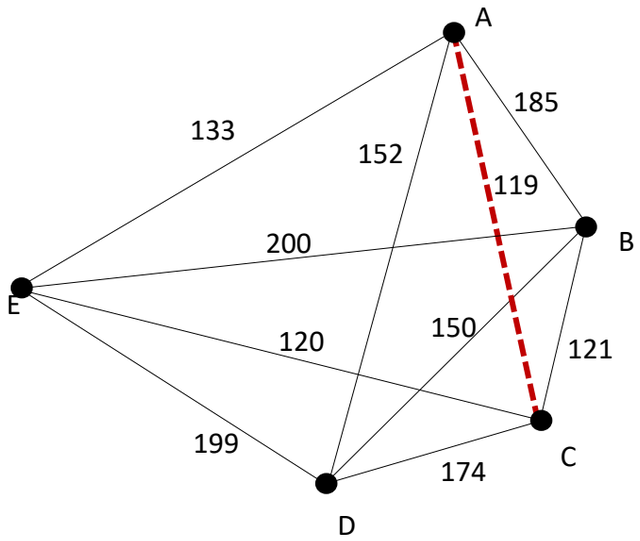
Circuito	Custo
A,C,E,D,B,A	773
B,C,A,E,D,B	722
C,A,E,D,B,C	722
D,B,C,A,E,D	722
E,C,A,D,B,E	741

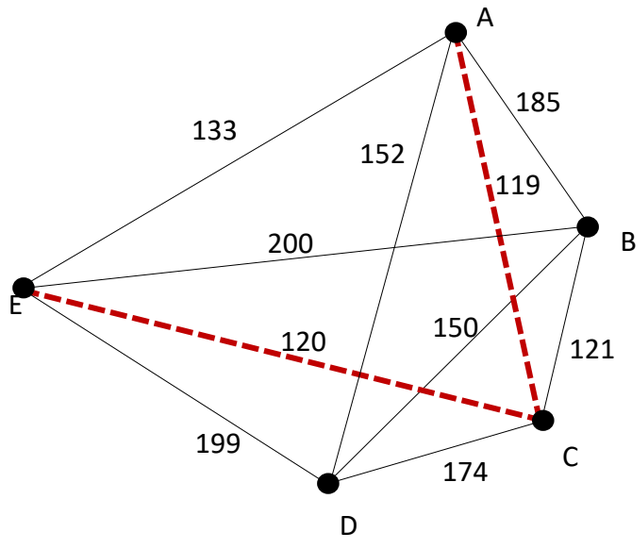
Algoritmo da ligação mais econômica

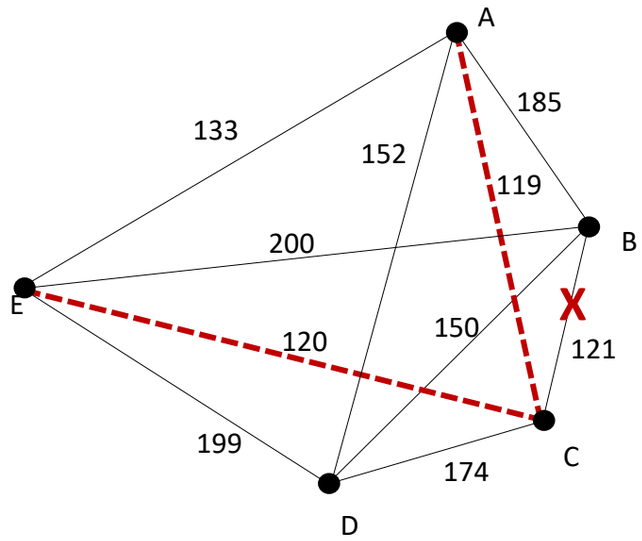
Ideia

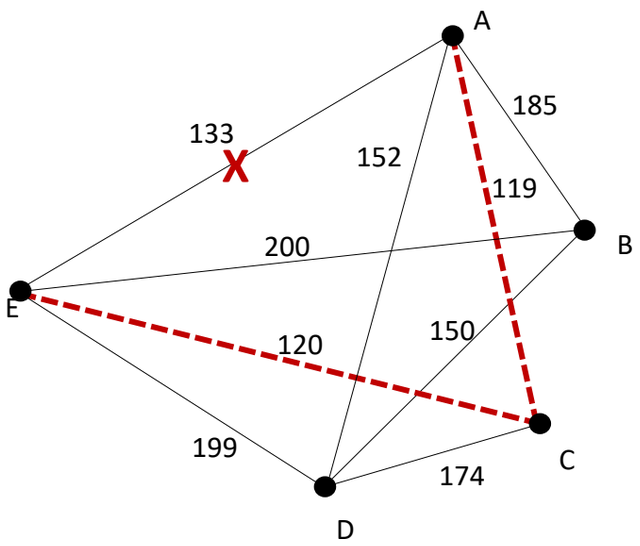
- Escolha a aresta com o menor custo. Marque a aresta correspondente (assinalando-a, por exemplo, a vermelho)
- Escolha a ligação mais econômica seguinte e assinale a aresta correspondente a vermelho.
- Continue a escolher a ligação mais econômica disponível. Marque a aresta correspondente a vermelho **exceto quando**:
  a. esta aresta fecha um circuito (havendo ainda vértices por visitar)
  b. esta aresta parte de um vértice ao qual já estão ligadas duas outras arestas anteriormente escolhidas (pois um circuito de **Hamilton usa exatamente 2 arestas** por vértice).
- Em qualquer um destes casos retire a aresta correspondente
- Quando não existirem mais vértices para ligar, fecha-se o circuito que temos vindo a assinalar a vermelho.

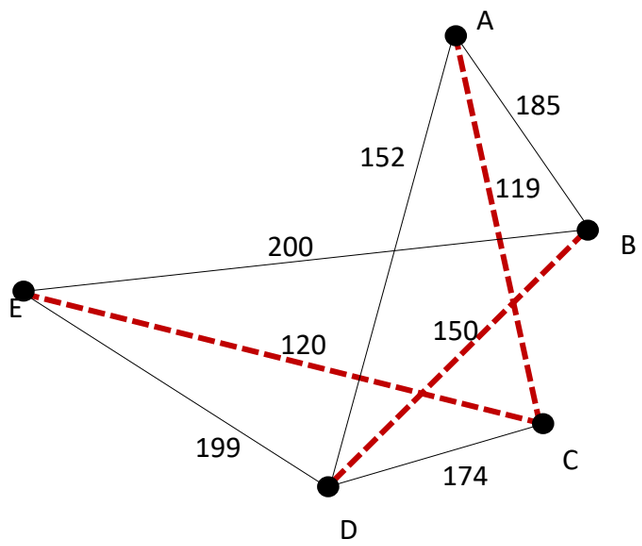
Algoritmo da ligação mais econômica

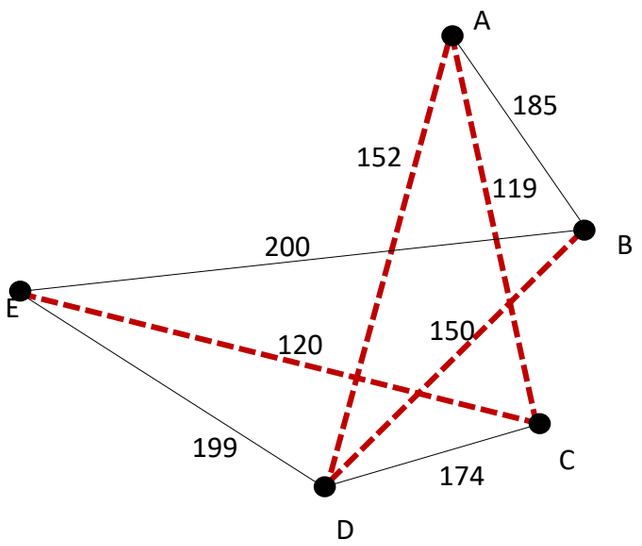












Algoritmo da ligação mais econômica

Assim o circuito de Hamilton obtido por este método será: A, C, E, B, D, A com um custo total de **741** (119 + 120 + 200 + 150 +152).