Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Ciência da Computação
Curso de Ciência da Computação

Algoritmos em Grafos

Parte 1

Raquel Mini
raquelmini@pucminas.br

Conceitos Básicos

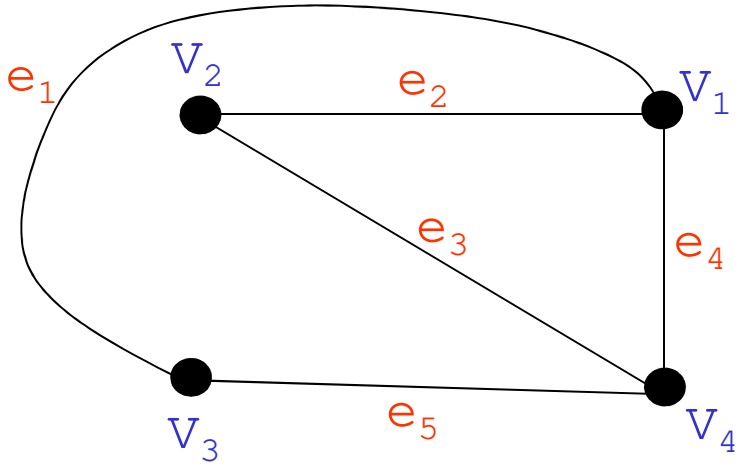
Cormen – páginas 853 até 861

ou

Deo – páginas 1 até 23
(mais didático e detalhado)

Conceitos Básicos

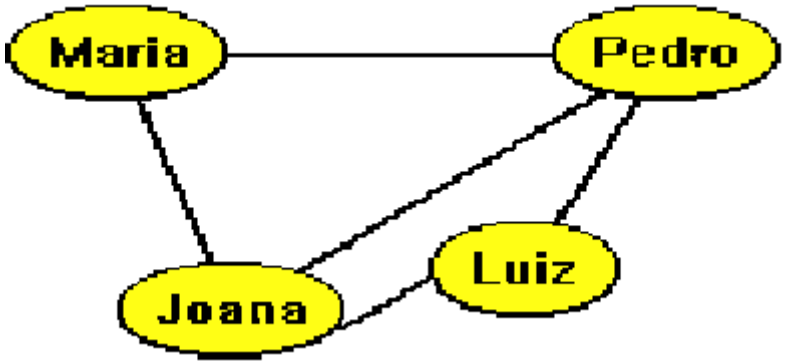
- Grafo é uma coleção de vértices e arestas
- Vértice é um objeto simples que pode ter nomes e outros atributos
- Aresta é uma conexão entre dois vértices



Por definição um grafo deve ter pelo menos 1 vértice

Grafo da Amizade

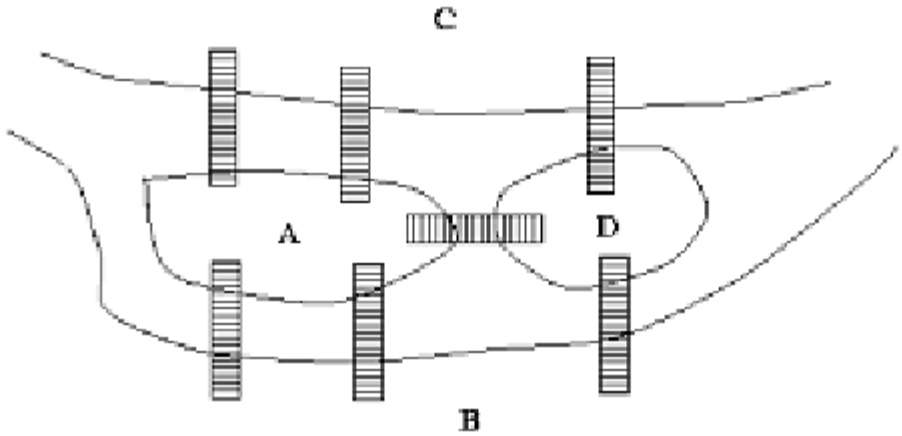
- Modelagem:
  - Vértices: pessoas
  - Arestas: relação de amizade
- **n**: número de vértices
- **e**: número de arestas



n = 4 **e** = 5

Problema das Pontes de Königsberg

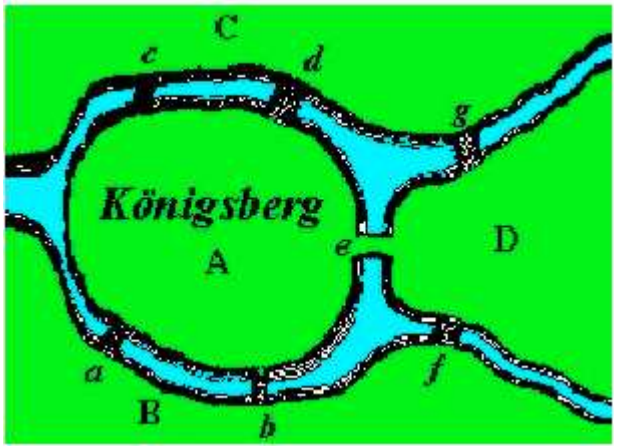
- No século XVIII havia na cidade de Königsberg um conjunto de sete pontes que cruzavam o rio Pregel. Elas conectavam duas ilhas (**A** e **D**) entre si e as ilhas com as margens (**B** e **C**)



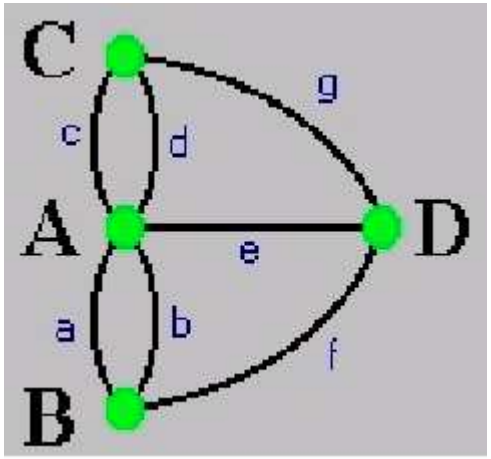
Por muito tempo os habitantes daquela cidade perguntavam-se se era possível **cruzar as sete pontes** numa caminhada contínua **sem passar duas vezes** por qualquer uma delas

Problema das Pontes de Königsberg

- As pontes são identificadas pelas letras de **a** até **g**



- Este problema pode ser representado pelo grafo

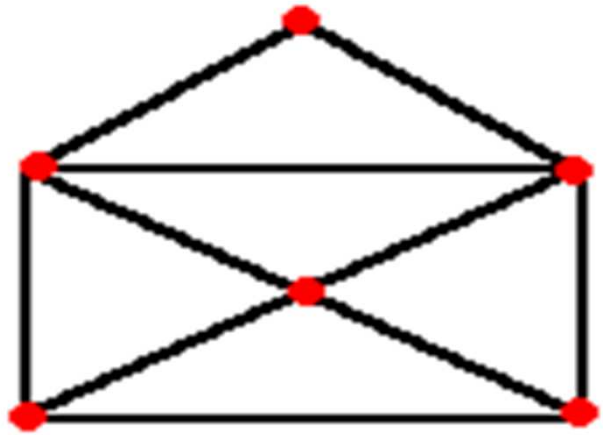


vértices: pontos de terra

aresta: pontes

Problema do Desenho da Casa

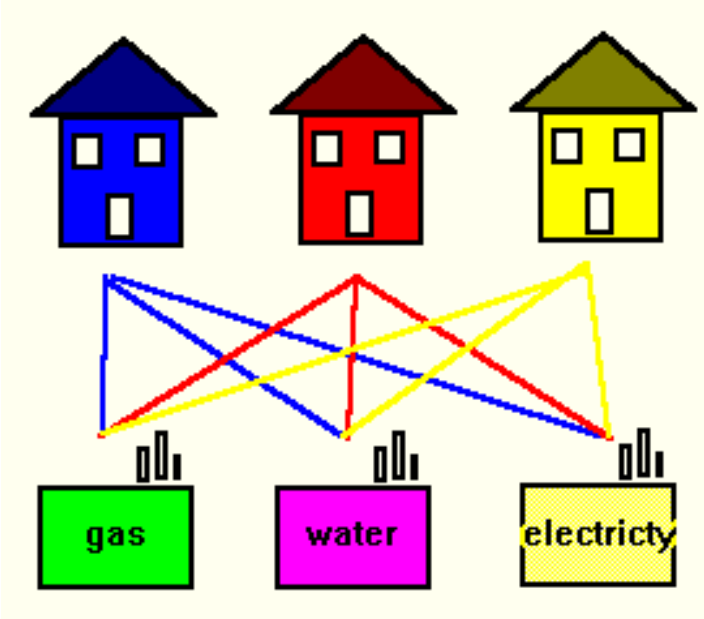
- No desenho abaixo, uma criança diz ter posto a ponta do lápis numa das bolinhas e com movimentos contínuos (sem levantar e sem retroceder o lápis) traçou as linhas que formam o desenho da casa, traçando cada linha uma única vez



A mãe da criança acha que ela trapaceou pois não foi capaz de achar nenhuma sequência que pudesse produzir tal resultado. Você concorda com esta mãe?

O Problema das 3 Casas e 3 Serviços

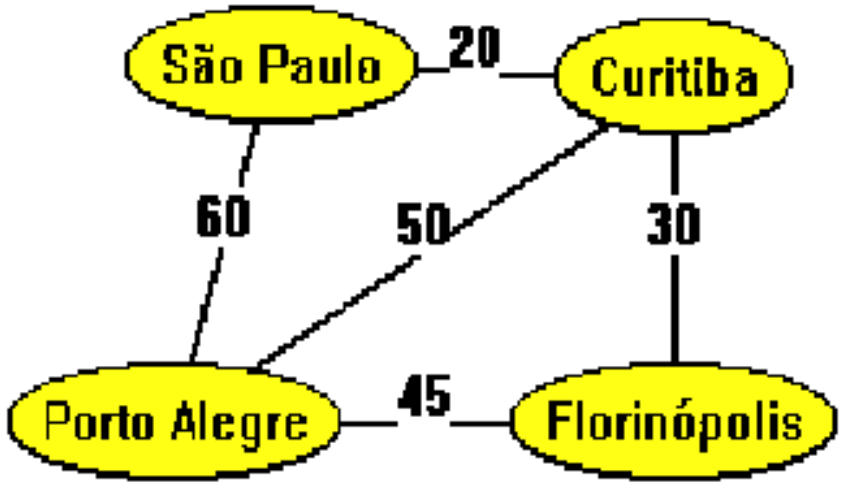
- Suponha que tenhamos três casas e três serviços, a exemplo de:



É possível conectar cada serviço a cada uma das três casas sem que haja cruzamento de tubulações?

Problema de Transporte

- Nos problemas que envolvem transportes de carga ou pessoas, os pontos de parada, embarque e desembarque são os vértices e as estradas entre os pontos são as arestas

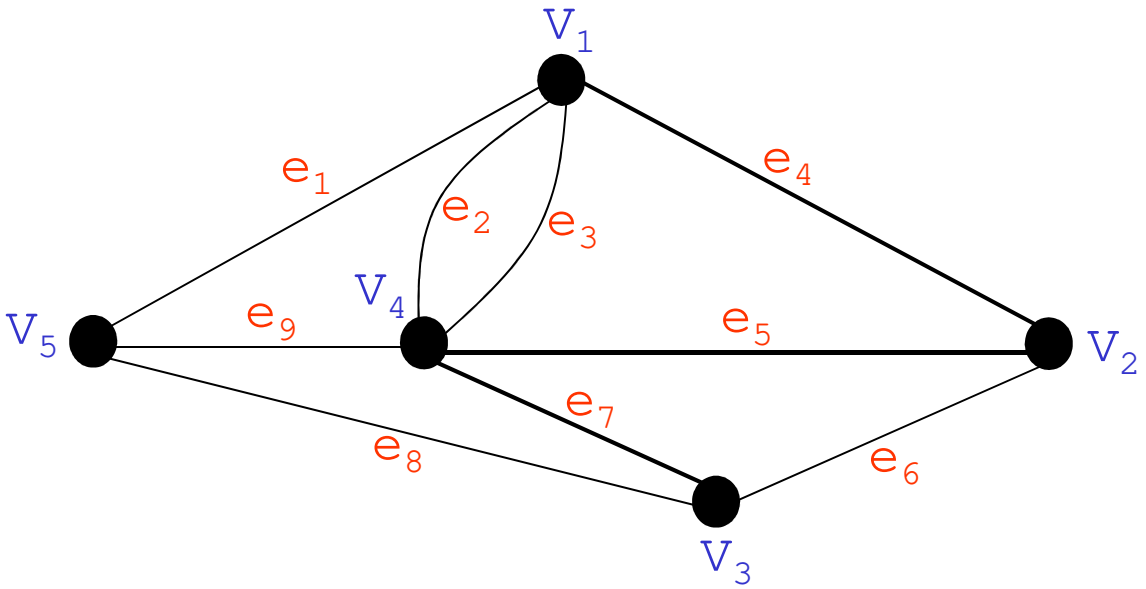


Problema de Transporte

- **Problema de Conectividade**: dadas as direções das vias, é possível ir da cidade **A** até a cidade **B**, sem andar na contramão?
- **Problema de Fluxo Máximo**: dada a capacidade de fluxo em cada via, qual é a quantidade de mercadoria que podemos mandar de uma cidade **A** a uma cidade **B**?
- **Problema de Menor Caminho**: Dados os comprimentos de cada via, qual o percurso mais rápido para sair de uma cidade **A** e chegar a uma cidade **B**?

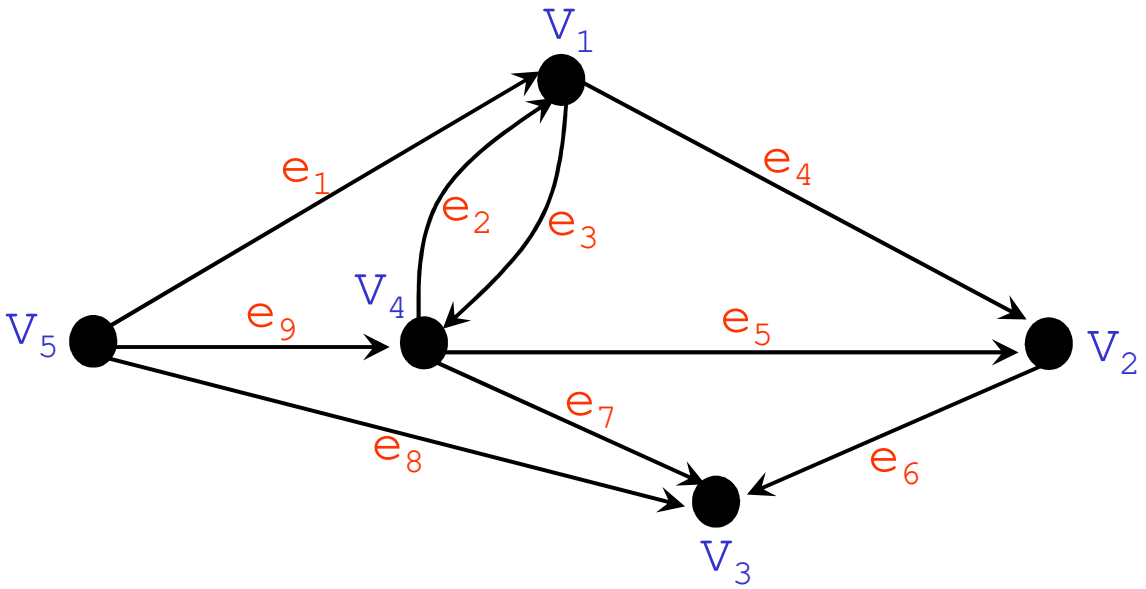
Terminologia

- Grafo não orientado: $G=(V,E)$ é um par onde V é um conjunto finito e o conjunto de arestas E consiste em pares de vértices não orientados. A aresta (v_i,v_j) e (v_j,v_i) são consideradas a mesma aresta.



Terminologia

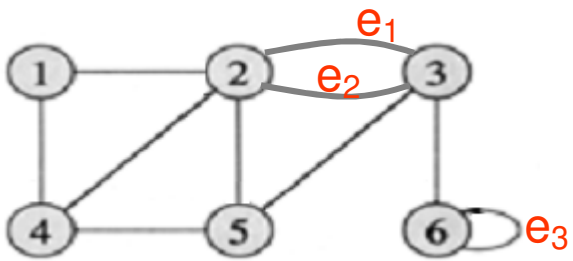
- Grafo orientado: `G` é um par `(V,E)`, onde `V` é um conjunto finito e `E` é uma relação binária em `V`



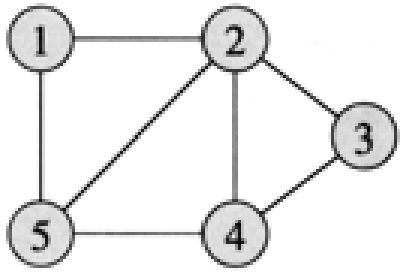
Terminologia

- *Autoloop*: uma aresta associada ao par de vértices (v_i, v_i)
- Arestas paralelas: quando mais de uma aresta está associada ao mesmo par de vértices
- Grafo simples: um grafo que não possui *autoloops* e nem arestas paralelas

e_1 e e_2 são arestas paralelas



e_3 é um autoloop



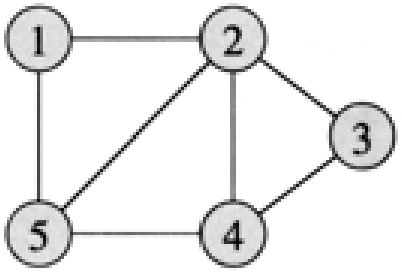
grafo simples

Terminologia

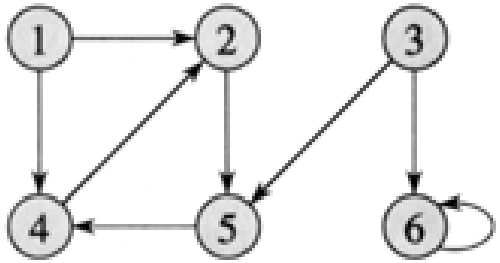
- Se `(u,v)` é uma aresta em um grafo `G=(V,E)`, dizemos que o vértice `v` é adjacente ao vértice `u` (em grafo não orientado, a relação de adjacência é simétrica). Se `(u,v)` é uma aresta de um grafo orientado e `(v,u)` não é uma aresta, dizemos que o vértice `v` é adjacente ao vértice `u`, mas `u` não é adjacente a `v`
- Se `(u,v)` é uma aresta em um grafo orientado `G=(V,E)`, dizemos que a aresta `(u,v)` é incidente do ou sai do vértice `u` e é incidente no ou entra no vértice `v`
- Se `(u,v)` é uma aresta em um grafo não orientado `G=(V,E)` dizemos que a aresta `(u,v)` é incidente nos vértices `u` e `v`

Terminologia

- O grau de um vértice em um grafo não orientado é o número de arestas incidentes nele. Em um grafo orientado, o grau de saída de um vértice é o número de arestas que saem dele, e o grau de entrada de um vértice é o número de arestas que entram nele. O grau de um vértice em um grafo orientado é seu grau de entrada somado a seu grau de saída



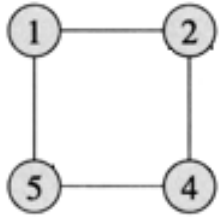
grau do vértice 2 é 4



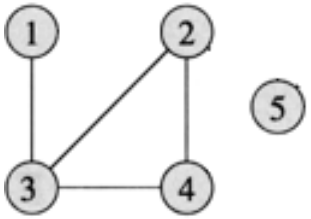
grau de entrada do vértice 5 é 2
grau de saída do vértice 5 é 1

Terminologia

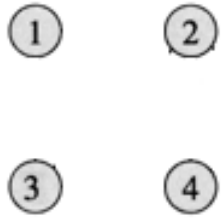
- Um grafo no qual todos os vértices possuem o mesmo grau é chamado de grafo regular
- Um vértice com nenhuma aresta incidente é chamado de vértice isolado
- Um vértice com grau 1 é chamado de vértice pendente
- Um grafo sem nenhuma aresta é chamado de grafo nulo (todos os vértices em um grafo nulo são vértices isolados)



grafo regular



5 é um vértice isolado
1 é um vértice pendente



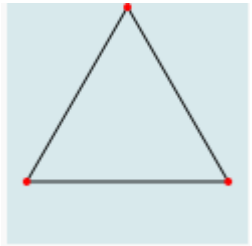
grafo nulo

Terminologia

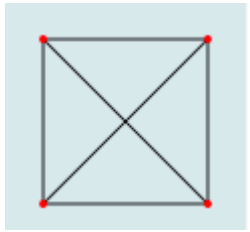
- Um grafo não orientado $G=(V,E)$ é um grafo completo se para cada par de vértices v_i e v_j existe uma aresta entre v_i e v_j. Em um grafo completo quaisquer dois vértices distintos são adjacentes (K_n)

K_1

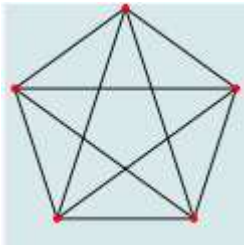
K_2



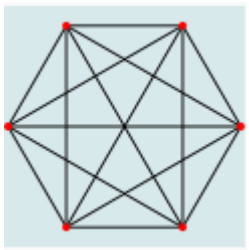
K_3



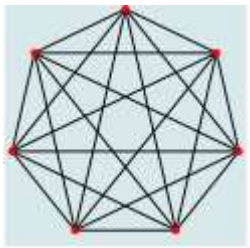
K_4



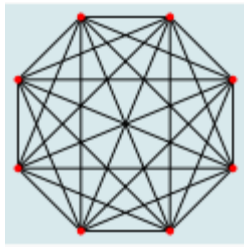
K_5



K_6



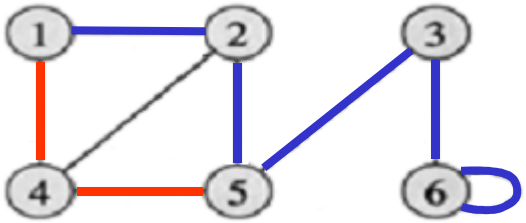
K_7



K_8

Terminologia

- Um caminho de comprimento k de um vértice u até um vértice u' em um grafo $G=(V,E)$ é uma sequência $<V_0, V_1, V_2, \ldots, V_k>$ de vértices tais que $u=V_0$, $u'=V_k$ e $(V_{i-1}, V_i) \in E$ para $i=1,2,\ldots,k$
- O comprimento de um caminho é o número de arestas no caminho
- Se existe um caminho de u até u' dizemos que u' é acessível a partir de u
- Um caminho simples é aquele em que todos os vértices no caminho são distintos

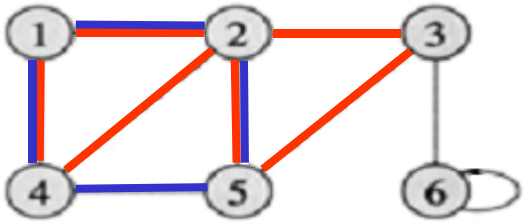


<1,4,5> é um caminho simples

<1,2,5,3,6,6> não é um caminho simples

Terminologia

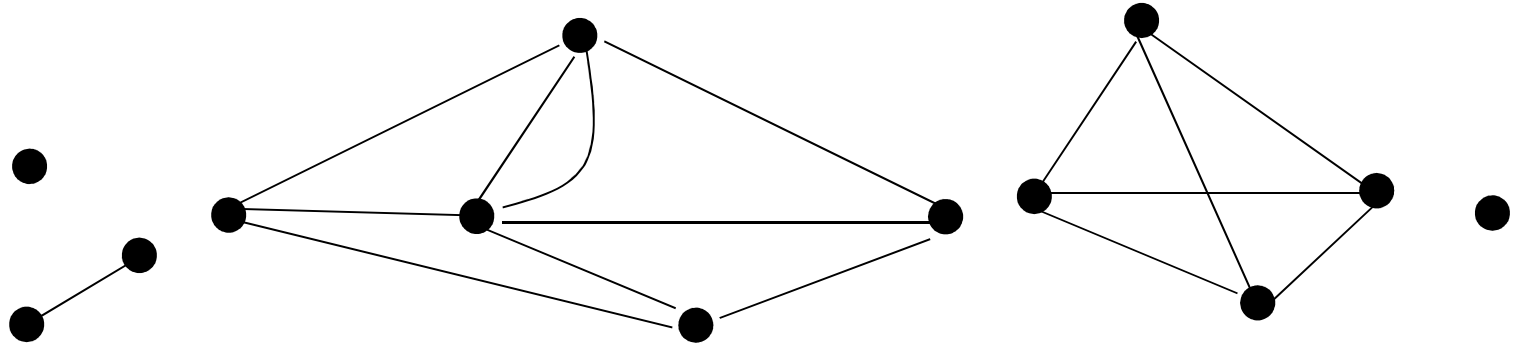
- Um caminho $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos uma aresta
- Ciclo simples é um ciclo no qual os vértices $v_1, v_2, \ldots, v_k$ são distintos
- Um *autoloop* é um ciclo de comprimento 1
- Um grafo sem ciclos é acíclico



<4,1,2,3,5,2,4> é um ciclo
<1,2,5,4,1> é um ciclo simples

Terminologia

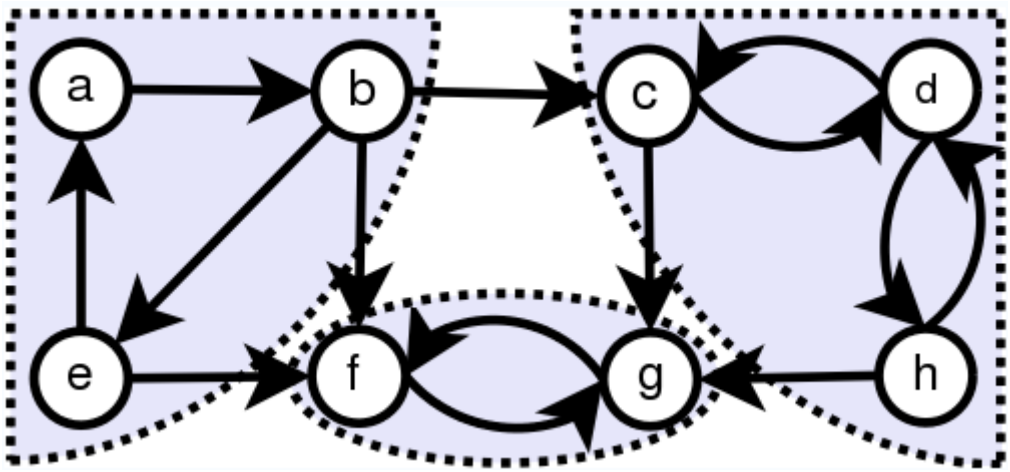
- Um grafo não orientado é conectado se todo par de vértices está conectado por um caminho
- Cada um dos subgrafos conectados é chamado de componente
- Os componentes conexos de um grafo são as classes de equivalência de vértices sob a relação “é acessível a partir de”



grafo não conectado com 5 componentes

Terminologia

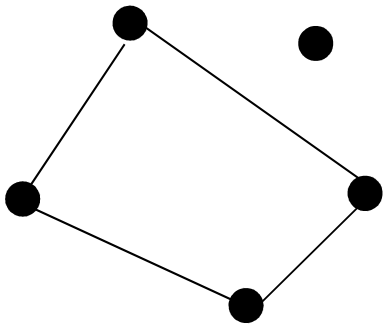
- Um grafo orientado é fortemente conectado se cada um de dois vértices quaisquer é acessível a partir do outro
- Os componentes fortemente conectados de um grafo orientado são as classes de equivalência de vértices sob a relação “são mutuamente acessíveis”



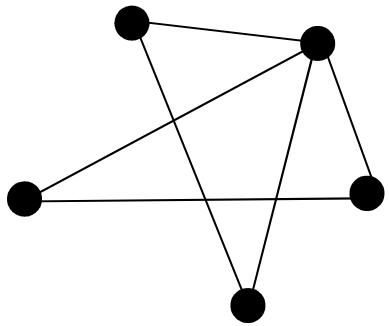
grafo possui 3 componentes fortemente conectados

Terminologia

- Grafo complementar: seja G = (V,E) um grafo simples. O **complemento** de G, $\overline{G}$, é um grafo formado da seguinte maneira:
  - Os vértices de $\overline{G}$ são todos os vértices de G
  - As arestas de $\overline{G}$ são exatamente as arestas que faltam em G para formarmos um grafo completo



G



$\overline{G}$

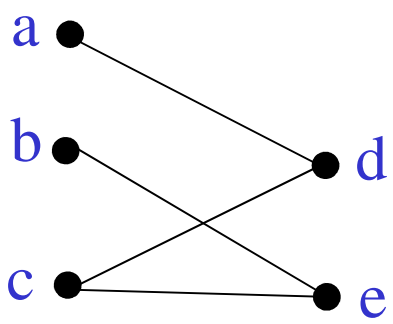
Terminologia

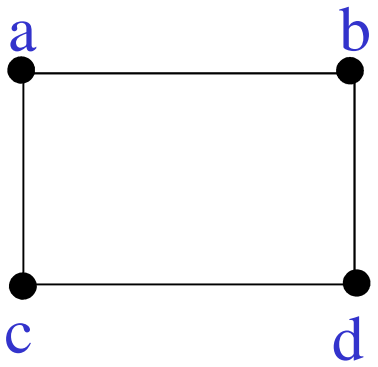
- Um grafo bipartido é um grafo não orientado $G=(V,E)$ em que V pode ser particionado em dois conjuntos $V1$ e $V2$ tais que $(u,v)\in E$ implica que:

  $u\in V1$ e $v\in V2$ ou

  $u\in V2$ e $v\in V1$

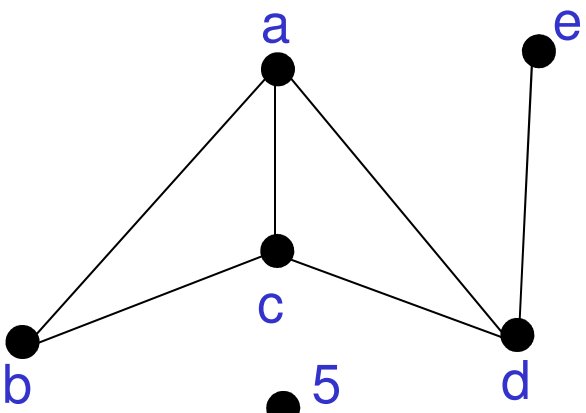
- Em um grafo bipartido, todas as arestas ligam um vértice do conjunto $V1$ com um vértice do conjunto $V2$

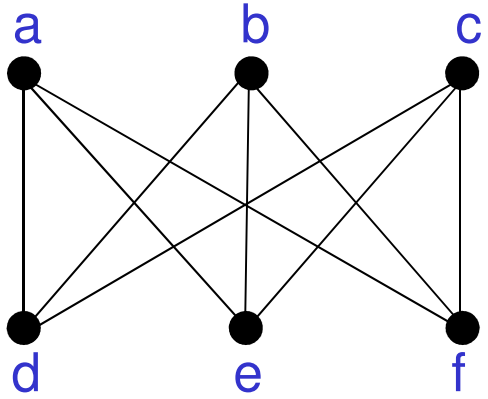


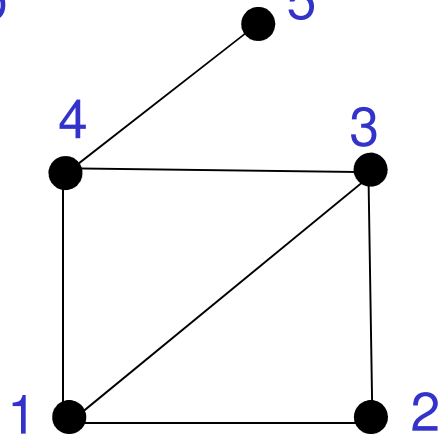


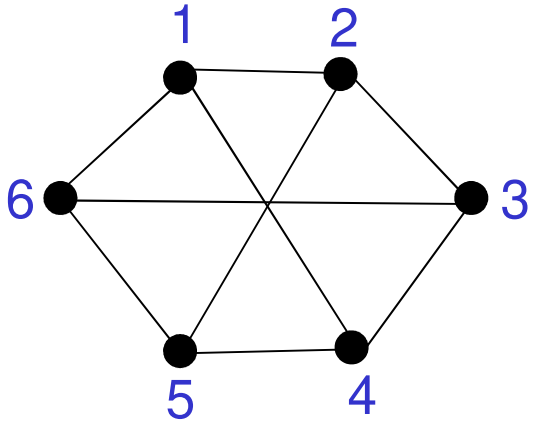
Terminologia

- Dois grafos `G` e `H` são ditos isomorfos se existir uma correspondência um-para-um entre seus vértices e entre suas arestas, de maneira que as relações de incidência são preservadas





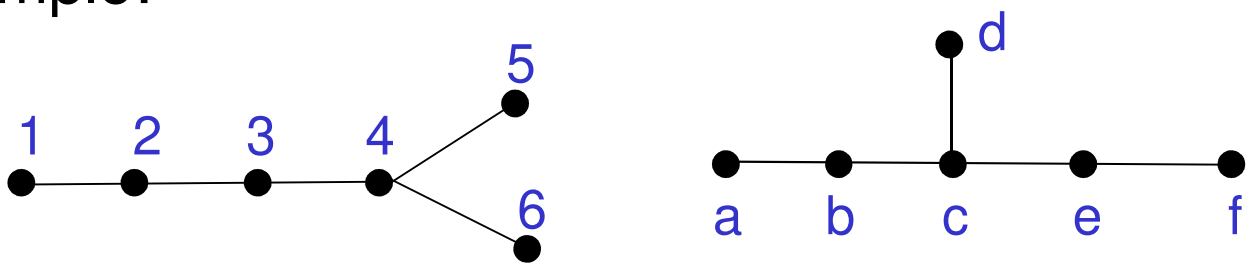




Terminologia

- Condições necessárias **mas não suficientes** para que `G` e `H` sejam isomorfos:
  - mesmo número de vértices
  - mesmo número de arestas
  - mesmo número de componentes
  - mesmo número de vértices com o mesmo grau

- Exemplo:



Não existe um algoritmo eficiente para determinar se dois grafos são isomorfos

Número de Vértices de Grau Ímpar

- A soma dos graus de todos os vértices de um grafo G é duas vezes o número de arestas de G.

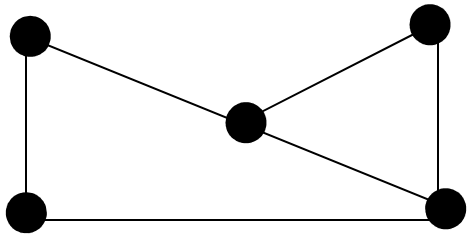
$$\sum_{i=1}^{n} d(v_i) = 2e$$

- O número de vértices de grau ímpar em um grafo é par

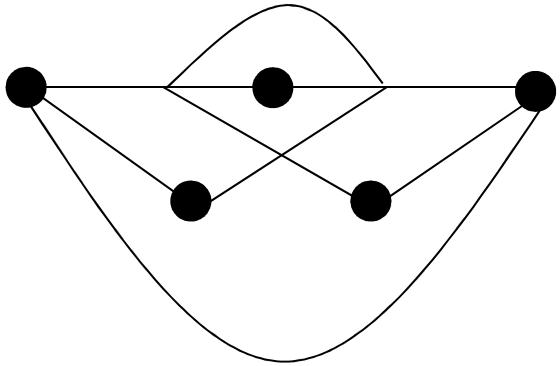
$$\sum_{i=1}^{n} d(v_i) = \sum_{d(v_j)\,par} d(v_j) + \sum_{d(v_k)\,ímpar} d(v_k)$$

Exercícios

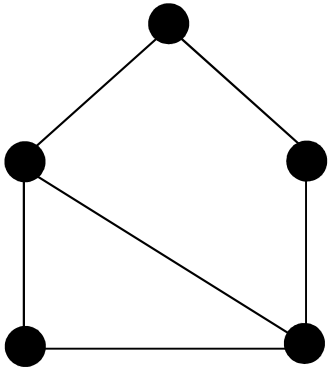
1. Todos os grafos abaixo são isomorfos, exceto:



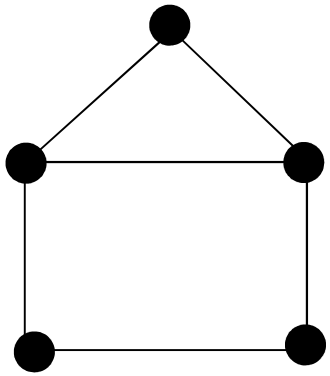
(a)



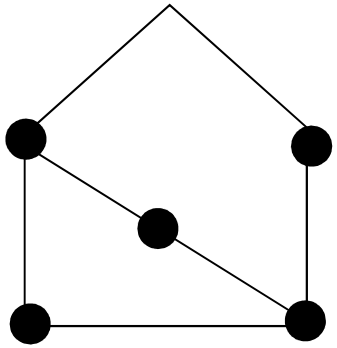
(b)



(c)



(d)



(e)

Exercícios

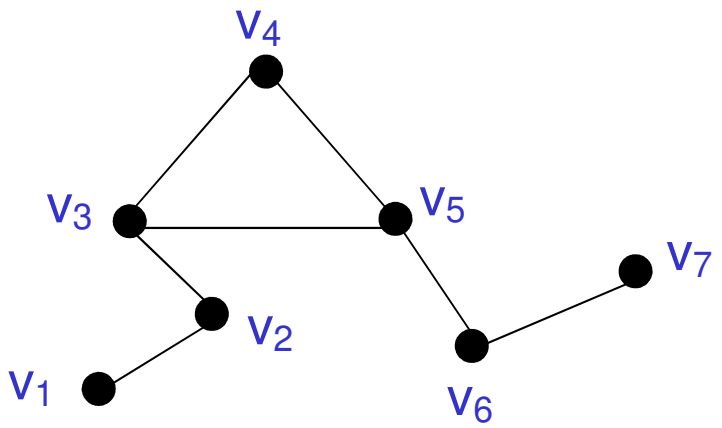
2. Com relação ao grafo completo K_n, responda:
   a) Qual é o grau dos seus vértices?
   b) Quantas arestas ele possui?

3. Encontre um grafo com 5 vértices que seja isomorfo a seu complemento.

4. Qual o número de arestas de um grafo que é isomorfo a seu complemento?

Exercícios

5. Com relação ao grafo abaixo, responda:



a) O grafo é simples?
b) Completo?
c) Regular?
d) Conectado?
e) Encontre 2 caminhos simples entre V_3 e V_6
f) Encontre 1 ciclo
g) Indique uma aresta cuja remoção tornará o grafo não conectado

Número Mínimo e Máximo de Arestas

- O número mínimo de arestas de um grafo simples com `n` vértices e `k` componentes é `n-k`
- O número máximo de arestas de um grafo simples com `n` vértices e `k` é $\frac{(n-k)(n-k+1)}{2}$

Grafo simples e conexo

$$n-1 \leq e \leq \frac{n(n-1)}{2}$$

Grafo simples

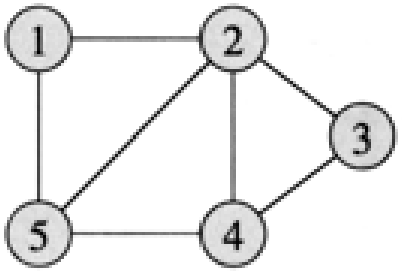
$$n-k \leq e \leq \frac{(n-k)(n-k+1)}{2}$$

Estruturas de Dados

Cormen – páginas 419 até 422

Matriz de Adjacências

- A matriz de adjacências de um grafo simples G com **n** vértices é uma matriz **n** x **n**, definida como:
  - $M_{ij} = 1$, se existe uma aresta entre os vértices i e j
  - $M_{ij} = 0$, caso contrário

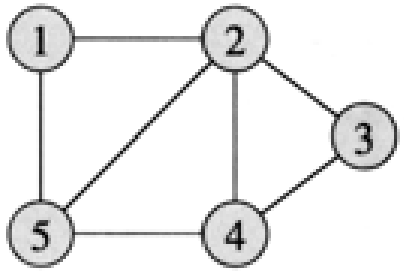


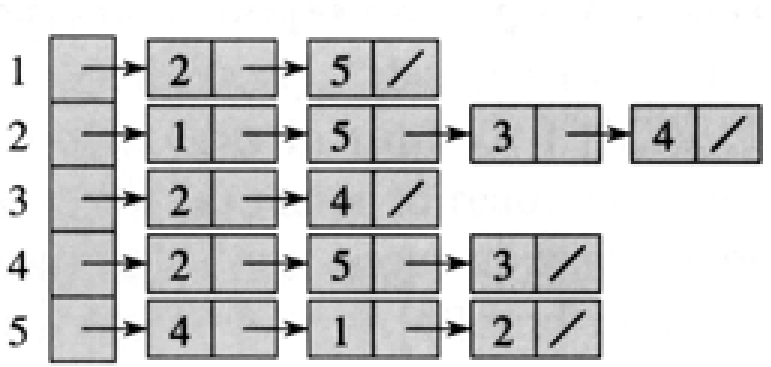
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Vantagem: verificar adjacência é O(1)

Lista de Adjacências

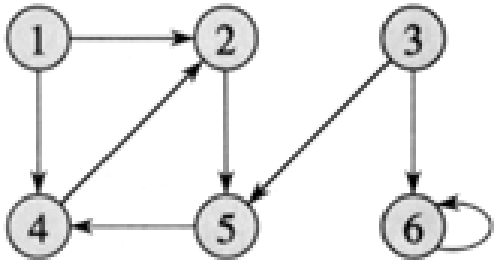
- Consiste de uma lista para cada vértice do grafo contendo todos os vértices adjacentes a ele
- Armazena apenas os elementos diferentes de zero da matriz de adjacências





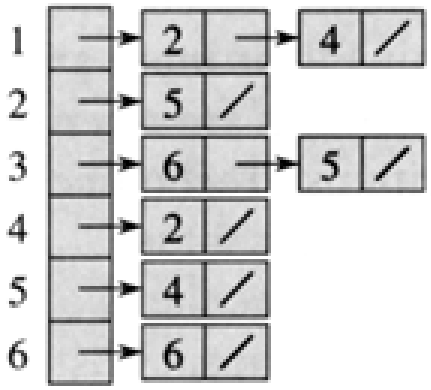
- Desvantagem: para encontrar se um vértice é adjacente a outro devemos percorrer uma lista encadeada
- Adequada quando **e** << $\mathbf{n^2}$

Estruturas de Dados para Dígrafos



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Matriz de Adjacências



Lista de Adjacências

Algoritmos Elementares em Grafos

Cormen – páginas 422 até 444

Algoritmos Elementares em Grafos

- Busca em Largura ou Busca em Amplitude (*Breadth First*)
- Busca em Profundidade (*Depth First*)
- Número de Componentes de um Grafo
- Ordenação Topológica
- Componentes Fortemente Conectados

Busca em Largura

- Dado um grafo `G=(V,E)` e um vértice de origem, `s`, a busca em largura explora sistematicamente as arestas de `G` até descobrir cada vértice acessível a partir de `s`
- Calcula a distância (menor número de arestas) desde `s` até todos os vértices acessíveis a partir de `s`
- Produz uma árvore primeiro na extensão com raiz em `s` que contém todos os vértices acessíveis
- Visita todos os vértices à distância `k` a partir de `s`, antes de visitar quaisquer vértices à distância `k+1`

Busca em Largura

- `d[u]` = distância desde a origem `s` até o vértice `u`
- π`[u]` = predecessor de `u`, se `u` não tem predecessor π`[u]` = NIL
- `Adj[u]` = conjunto de vértices adjacentes a `u`
- `color[u]` = cor do vértice `u`, pode assumir os seguintes valores:
  - branco: vértices não visitados (cor inicial de todos os vértices)
  - cinza: vértices visitados que possuem vizinhos visitados e não visitados
  - preto: vértices visitados que possuem apenas vizinhos visitados (cor final de todos os vértices)

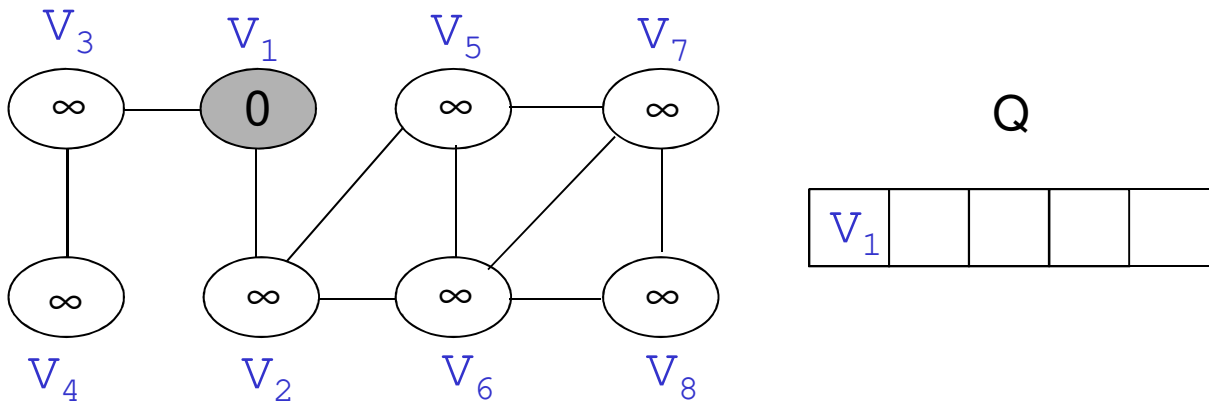
Busca em Largura

```
BFS(G, s)
 1  for each vertex u ∈ V[G] - {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ← Adj[u]
13              do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Mostre **s**

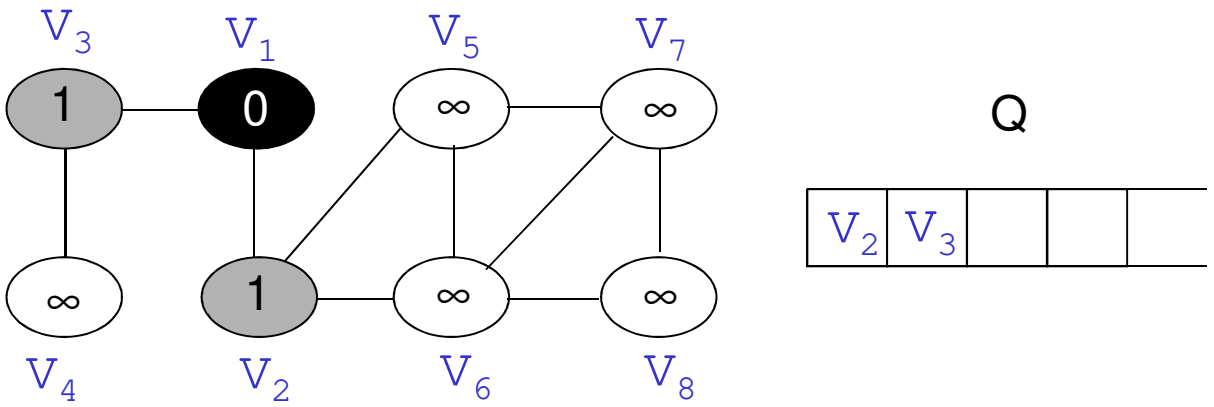
Mostre **v**

Busca em Largura



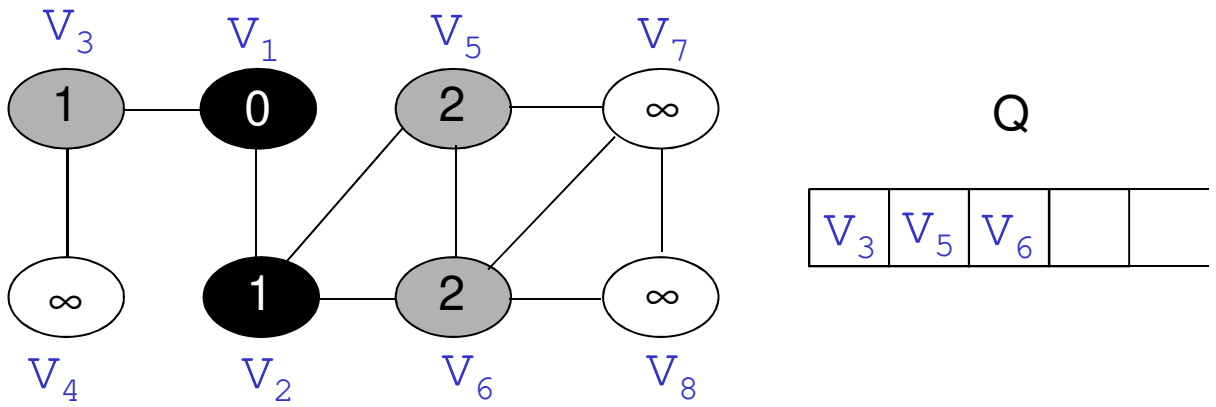
Saída do “Mostre s” e ”Mostre v”: V_1

Busca em Largura



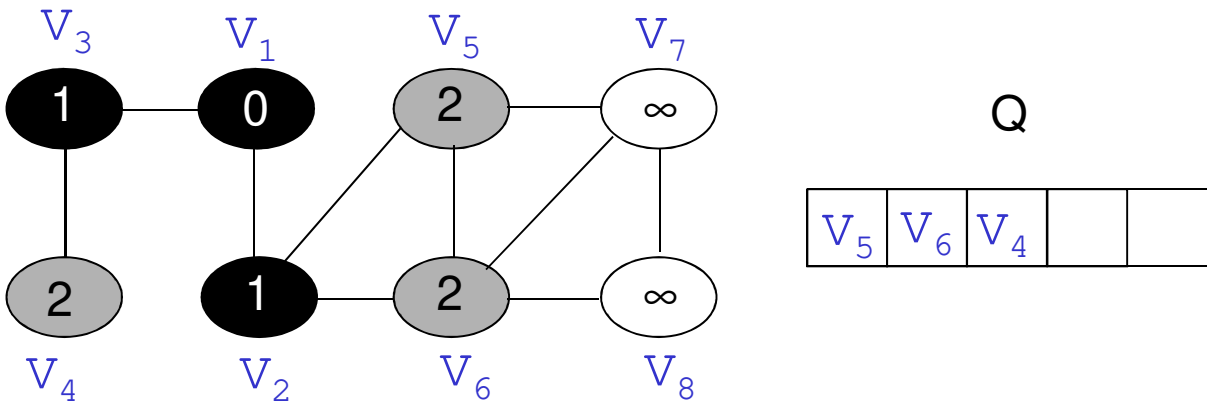
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3

Busca em Largura



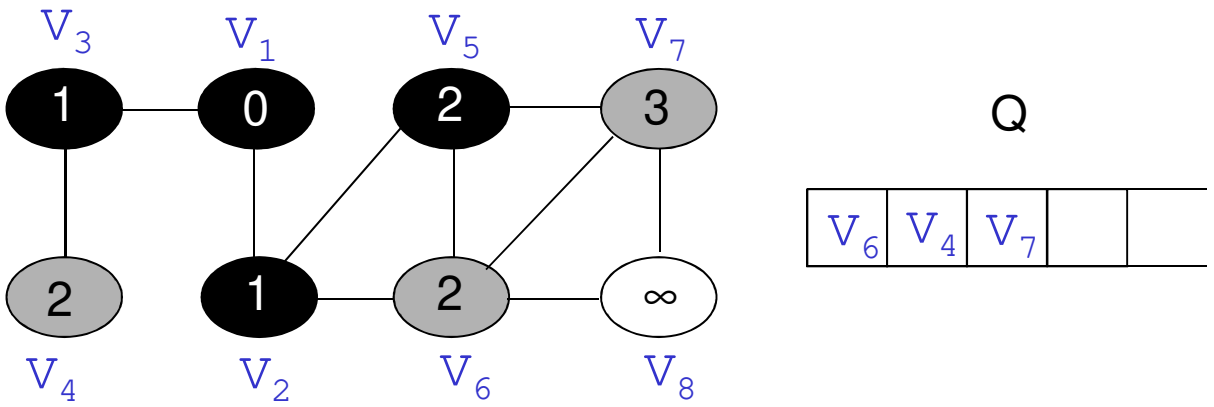
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6

Busca em Largura



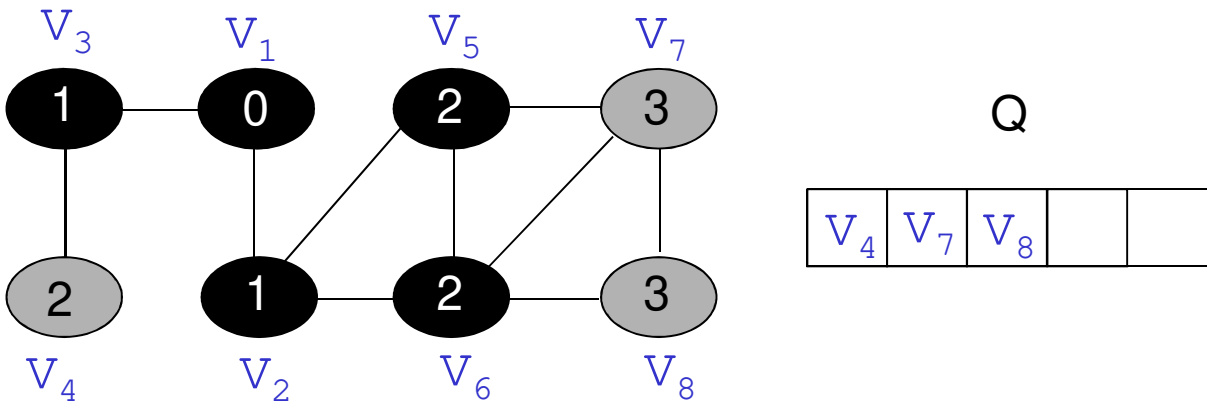
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4

Busca em Largura



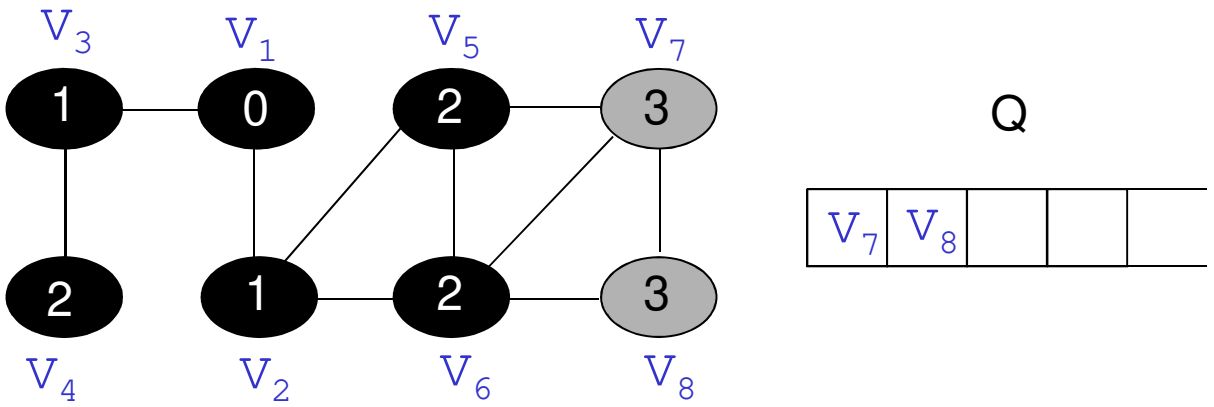
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4 V_7

Busca em Largura



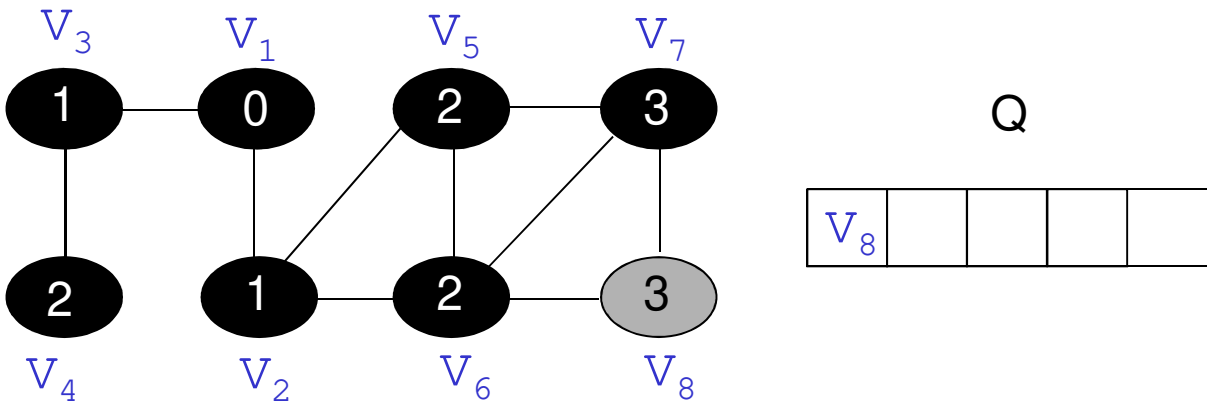
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4 V_7 V_8

Busca em Largura



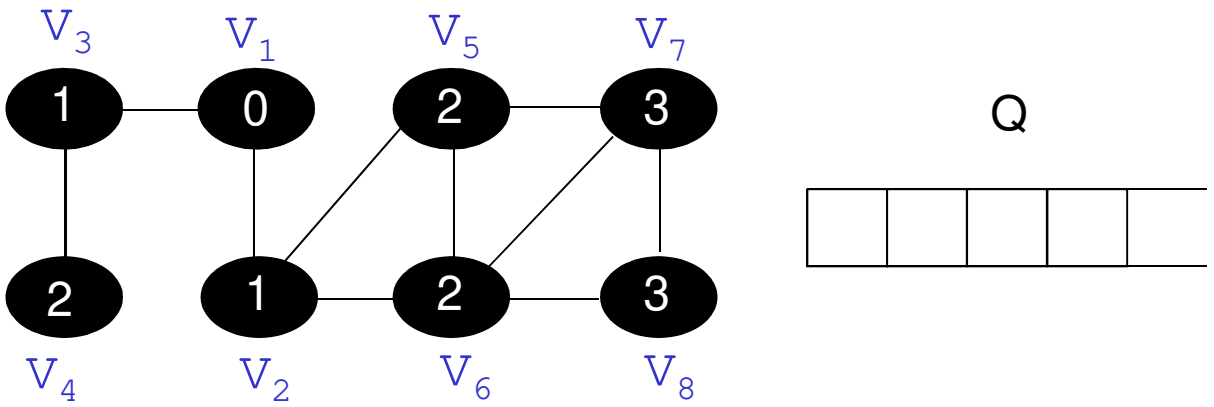
Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4 V_7 V_8

Busca em Largura



Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4 V_7 V_8

Busca em Largura



Saída do “Mostre s” e ”Mostre v”: V_1 V_2 V_3 V_5 V_6 V_4 V_7 V_8

Busca em Largura

- Análise de Complexidade (considerando matriz de adjacências):
  - O *loop* inicial é $O(n)$
  - Os vértices do grafo são colocados e retirados na fila exatamente uma vez
  - As operações de enfileirar e desenfileirar demoram o tempo $O(1)$
  - Portanto, o tempo total dedicado a operações de filas é $O(n)$
  - A lista de adjacências dos vértices (comando `for`) é examinada somente quando o vértice é desenfileirado (executada apenas uma vez) e seu custo é $O(n)$ para cada vértice
  - Portanto, o tempo de execução total é $O(n) + O(n^2) = O(n^2)$

Busca em Profundidade

- Utiliza a estratégia de procurar “mais fundo” no grafo sempre que possível
- As arestas são exploradas a partir do vértice v mais recentemente visitado que ainda tem arestas inexploradas saindo dele
- Quando todas as arestas de v são exploradas, a busca “regressa” para explorar as arestas que deixam o vértice a partir do qual v foi visitado
- Esse processo continua até que visitamos todos os vértices acessíveis a partir do vértice de origem inicial

Busca em Profundidade

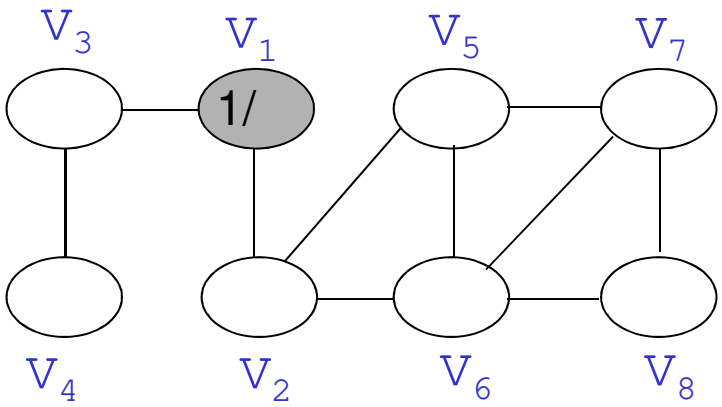
- `d[u]` = registra quando `u` é visitado pela primeira vez
- `f[u]` = registra quando a busca termina de examinar a lista de adjacências de `u`
- π`[u]` = predecessor de `u`, se `u` não tem predecessor π`[u]` = NIL
- `Adj[u]` = conjunto de vértices adjacentes a `u`
- `color[u]` = cor do vértice `u`, pode assumir os seguintes valores:
  - branco: vértices não visitados (cor inicial de todos os vértices)
  - cinza: vértices visitados cuja lista de adjacência não foi completamente examinada
  - preto: vértices visitados cuja lista de adjacência foi completamente examinada (cor final de todos os vértices)

Busca em Profundidade

```
DFS(G)
1  for each vertex u ∈ V [G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
DFS-VISIT(u)

1  color[u] ← GRAY
2  time ← time +1
3  d[u]←time

4  for each v ∈ Adj[u]
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)

8  color[u]←BLACK

9  f [u] ← time ← time +1
```

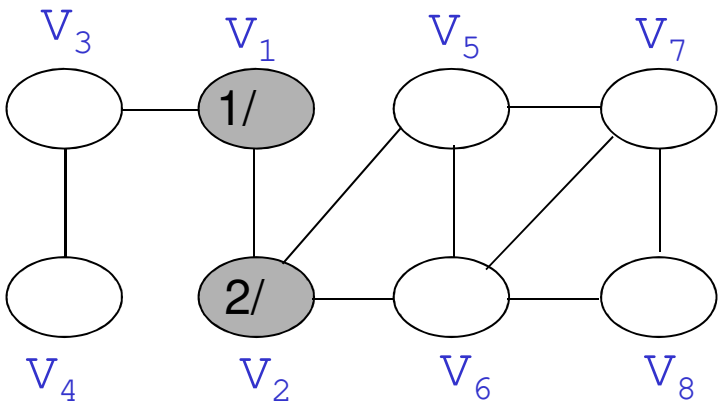
Mostre **u**

Busca em Profundidade



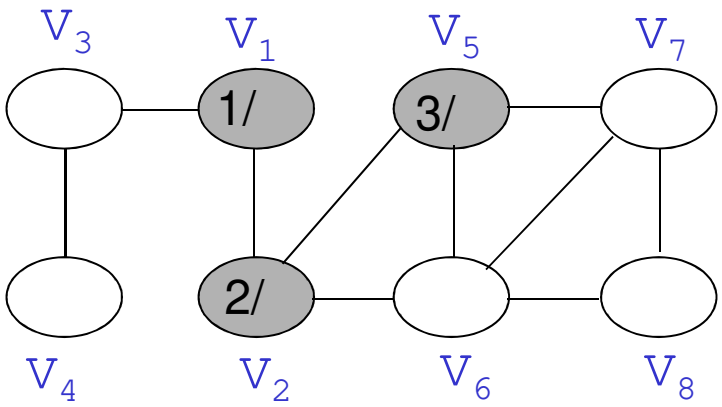
Saída do “Mostre u”: V_1

Busca em Profundidade



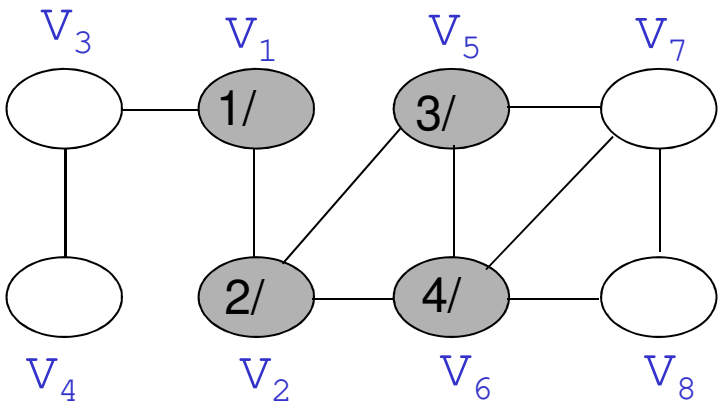
Saída do “Mostre u”: V_1 V_2

Busca em Profundidade



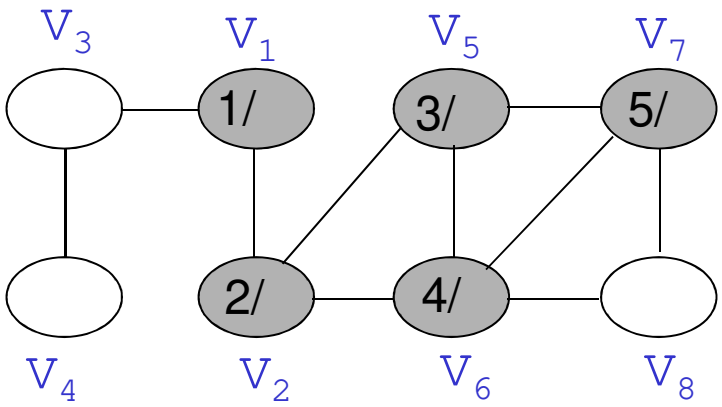
Saída do “Mostre u”: V_1 V_2 V_5

Busca em Profundidade



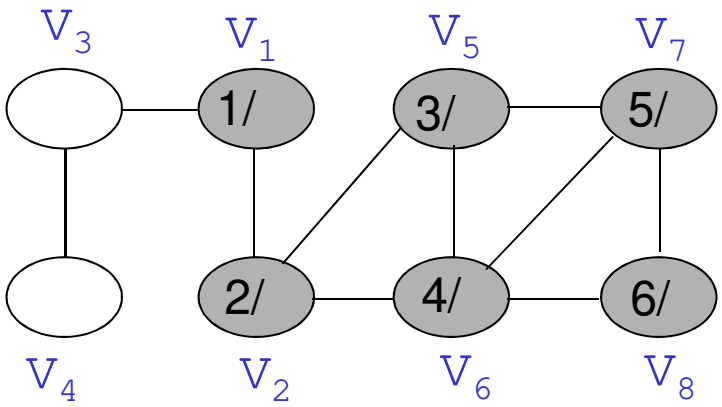
Saída do “Mostre u”: V_1 V_2 V_5 V_6

Busca em Profundidade



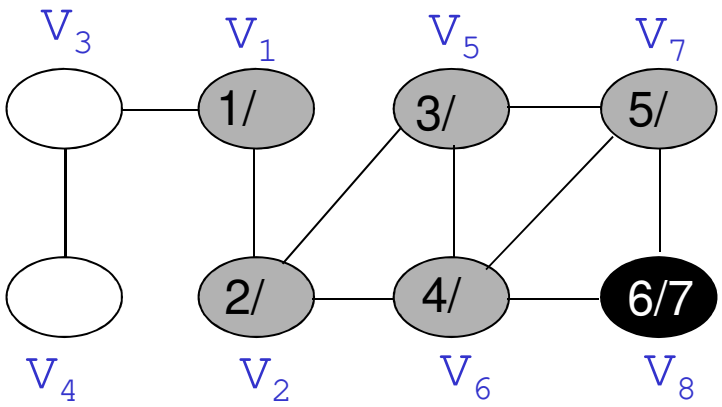
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7

Busca em Profundidade



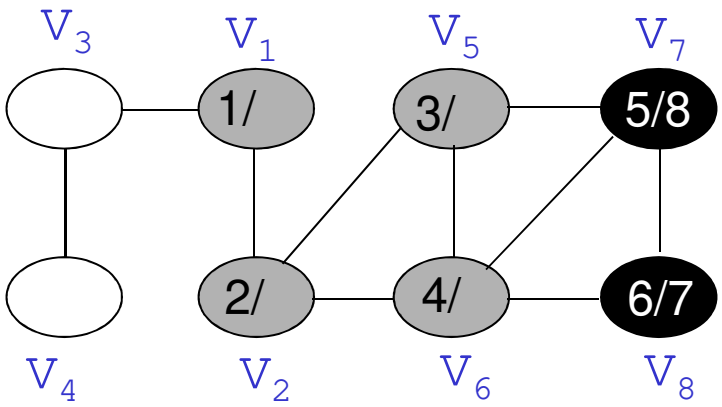
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



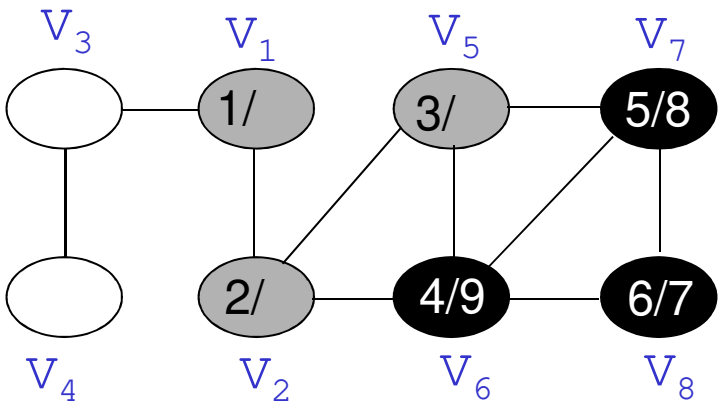
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



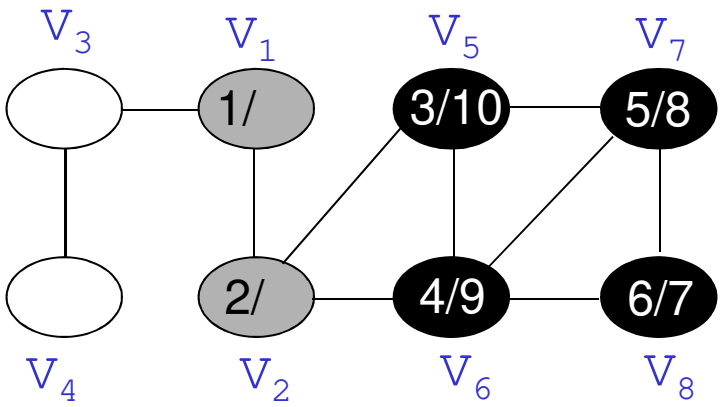
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



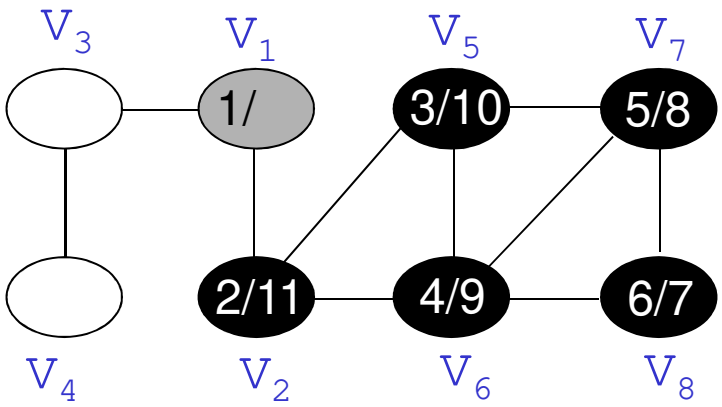
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



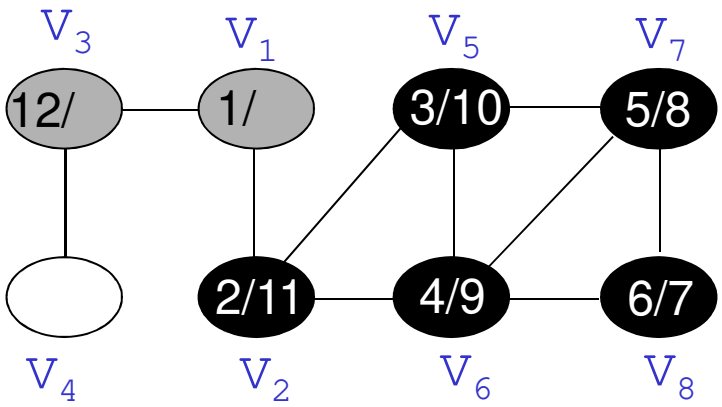
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



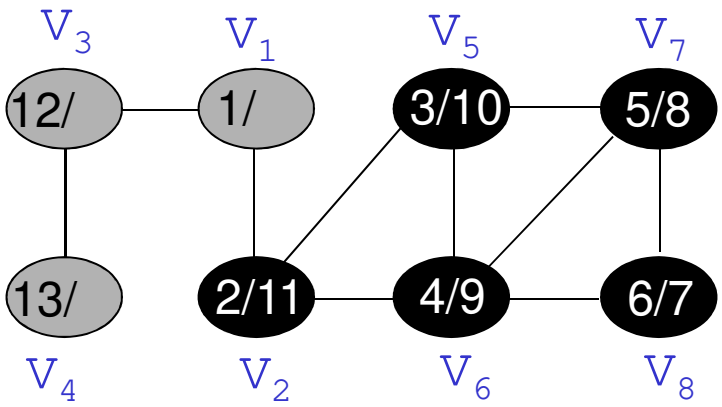
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8

Busca em Profundidade



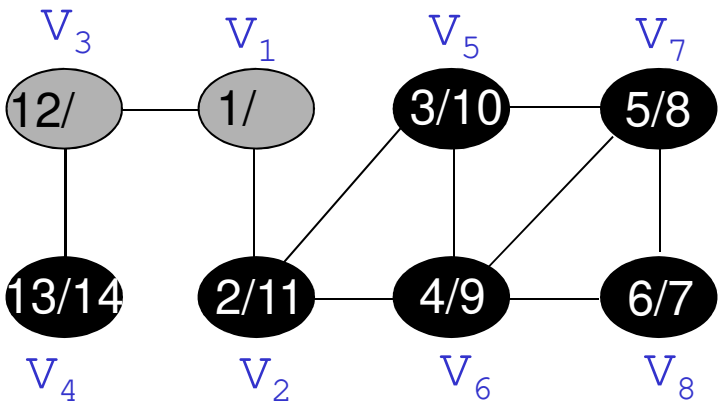
Saída do “Mostre u”: v_1 v_2 v_5 v_6 v_7 v_8 v_3

Busca em Profundidade



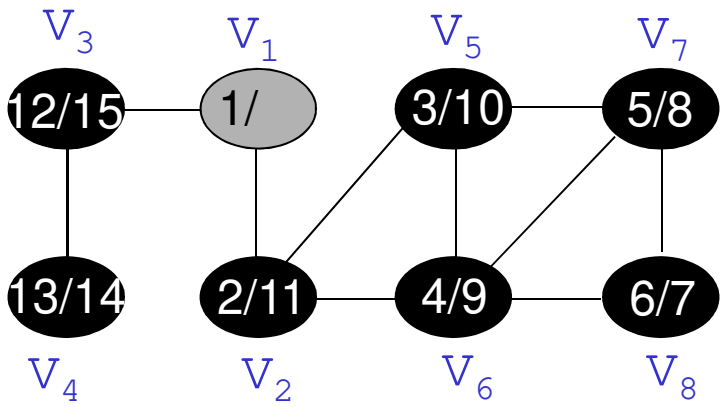
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8 V_3 V_4

Busca em Profundidade



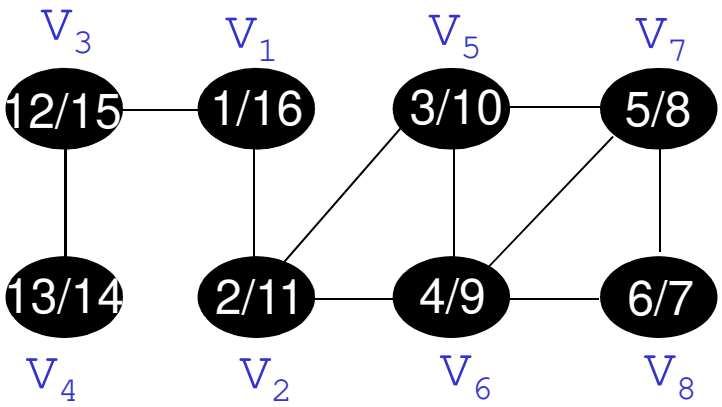
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8 V_3 V_4

Busca em Profundidade



Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8 V_3 V_4

Busca em Profundidade



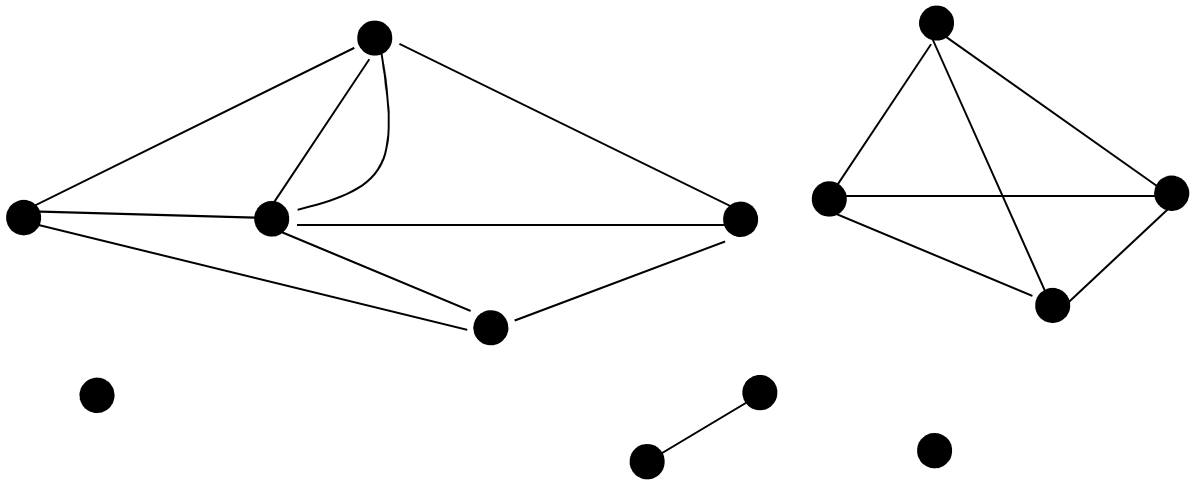
Saída do “Mostre u”: V_1 V_2 V_5 V_6 V_7 V_8 V_3 V_4

Busca em Profundidade

- Análise de Complexidade (considerando matriz de adjacências):
  - O *loop* inicial é $O(n)$
  - O procedimento DFS-VISIT é chamado exatamente uma vez pra cada vértice do grafo, pois ele é invocado somente sobre vértices brancos e a primeira ação é pintar o vértice de cinza
  - Durante a execução de DFS-VISIT o *loop* das linhas 4 a 7 é executado n vezes
  - Portanto, o tempo de execução total é $O(n) + O(n^2) = O(n^2)$

Número de Componentes de um Grafo

- Como encontrar o número de componentes de um grafo?



grafo com 5 componentes

Número de Componentes de um Grafo

```
DFS(G)
1  for each vertex u ∈ V [G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
DFS-VISIT(u)
1  color[u] ← GRAY
2  time ← time +1
3  d[u]←time
4  for each v ∈ Adj[u]   ▷E:
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u]←BLACK
9  f [u] ← time ← time +1
```

k ← 0

k ← k+1

Mostre k

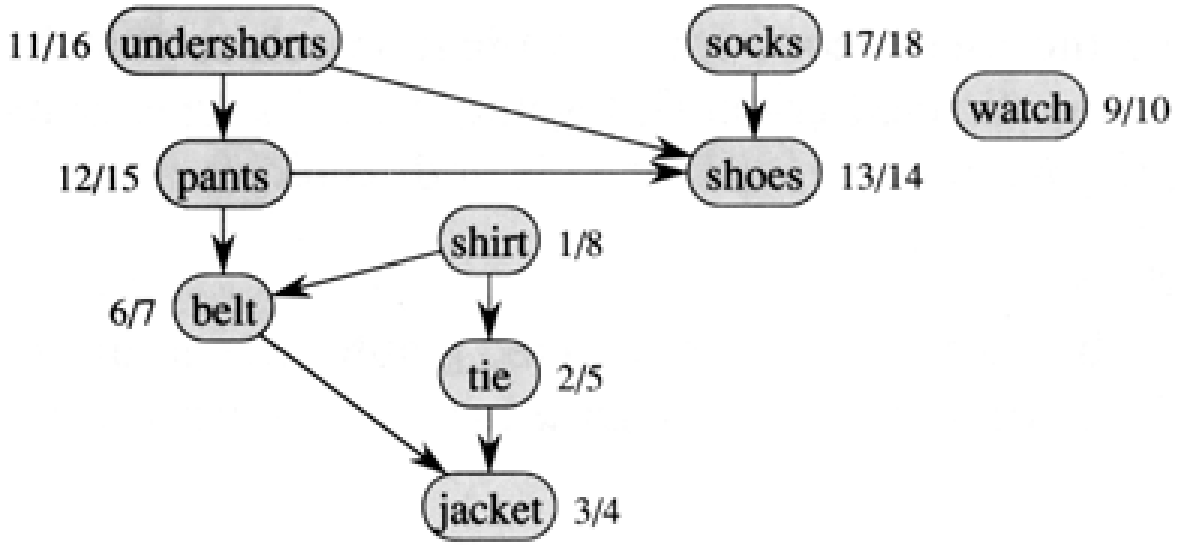
A variável k armazenará o número de componentes do grafo G

Ordenação Topológica

- A busca em profundidade pode ser usada para executar ordenações topológicas em **grafos acíclicos orientados** (gaos)
- Uma ordenação topológica de um gaos `G=(V,E)` é uma ordenação linear de todos os seus vértices, tal que se `G` contém uma aresta `(u,v)`, então `u` aparece antes de `v` na ordenação
- Se o grafo não é acíclico, então não é possível nenhuma ordenação linear
- Uma ordenação topológica de um grafo pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal, de tal forma que todas as arestas orientadas sigam da esquerda para a direita

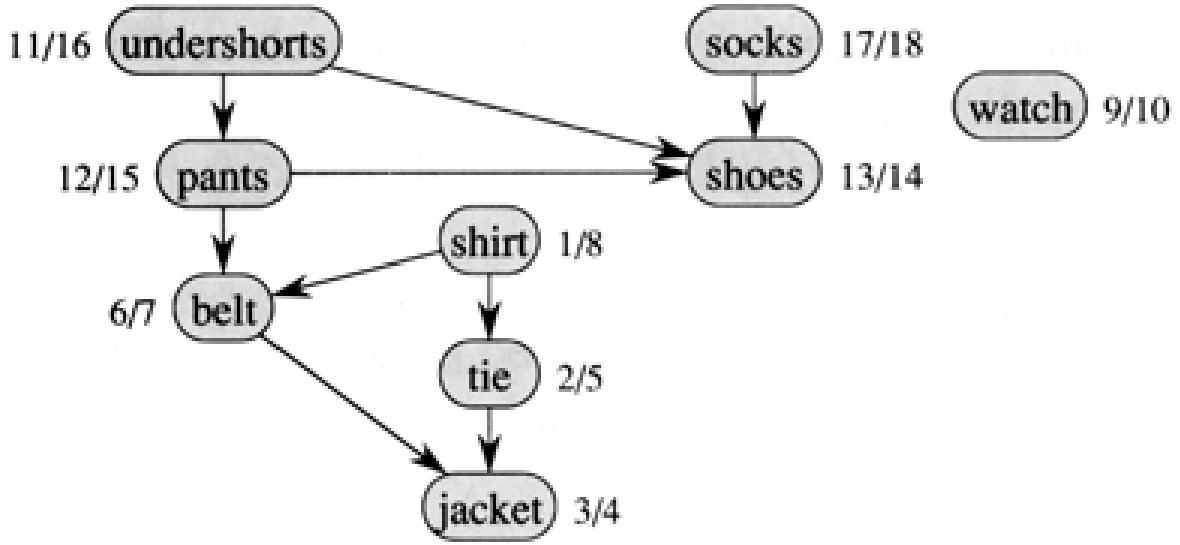
Ordenação Topológica

- Grafos acíclicos orientados são usados em muitas aplicações para indicar precedência entre eventos
- Como exemplo, considere o grafo que surge quando o professor Bumstead se veste pela manhã



Ordenação Topológica

- Uma aresta orientada `(u,v)` no gao indica que a peça de roupa `u` deve ser vestida antes da peça `v`
- Uma ordenação topológica desse gao fornece uma ordem para o processo de se vestir

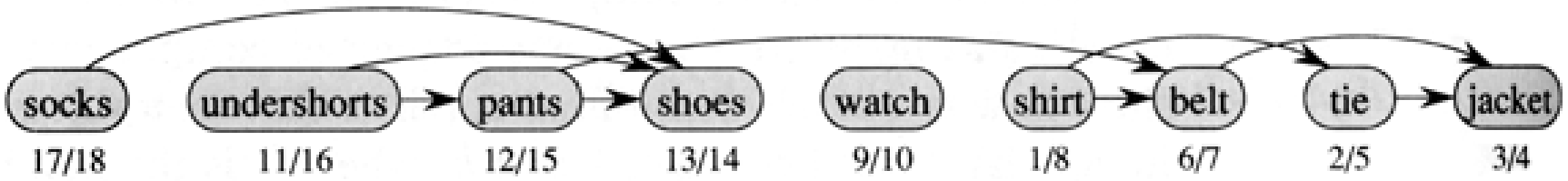


Ordenação Topológica

- Algoritmo a seguir ordena topologicamente um gao

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

- Os vértices topologicamente ordenados aparecem na ordem inversa de seus tempos de término

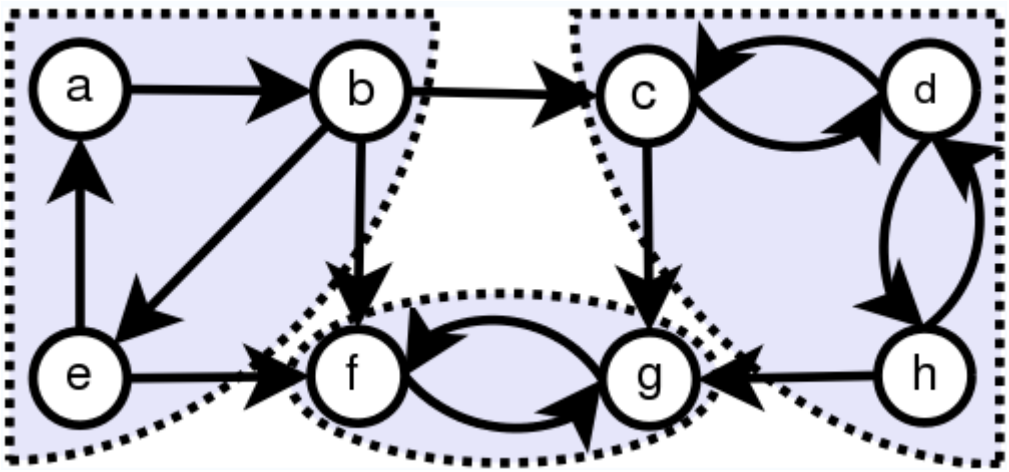


Ordenação Topológica

- Análise de complexidade:
  - A busca em profundidade é $O(n^2)$ e leva o tempo $O(1)$ para inserir cada um dos n vértices à frente da lista encadeada
  - Portanto, a ordenação topológica tem complexidade de tempo de $O(n^2)$

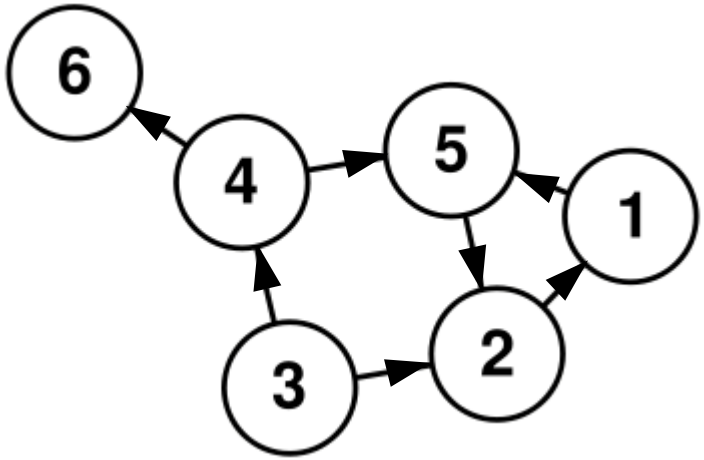
Componentes Fortemente Conectados

- Um **grafo orientado** é **fortemente conectado** se cada um de dois vértices quaisquer é acessível a partir do outro
- Um componente fortemente conectado de um grafo orientado `G=(V,E)` é um conjunto máximo de vértices $C \subseteq V$ tal que, para todo par de vértices `u` e `v` em `C`, temos que os vértices `u` e `v` são acessíveis um a partir do outro

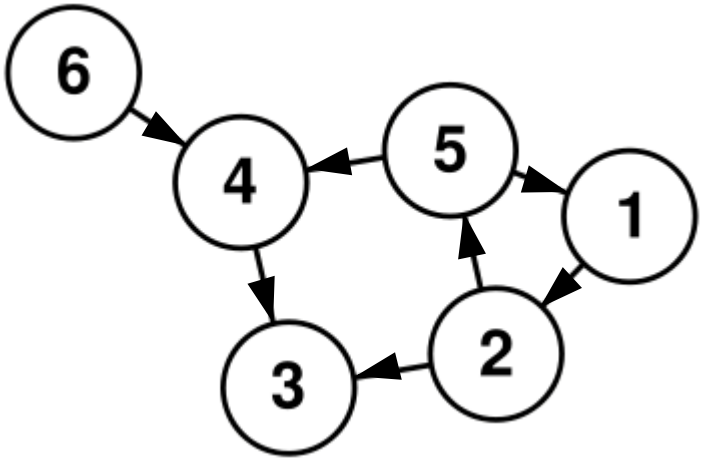


Componentes Fortemente Conectados

- A busca em profundidade pode ser utilizada também para realizar a decomposição de um grafo orientado em seus componentes fortemente conectados
- A transposta de um grafo $G=(V,E)$ é um grafo $G^T=(V,E^T)$, onde E^T consiste das arestas de G com seus sentidos invertidos



G



G^T

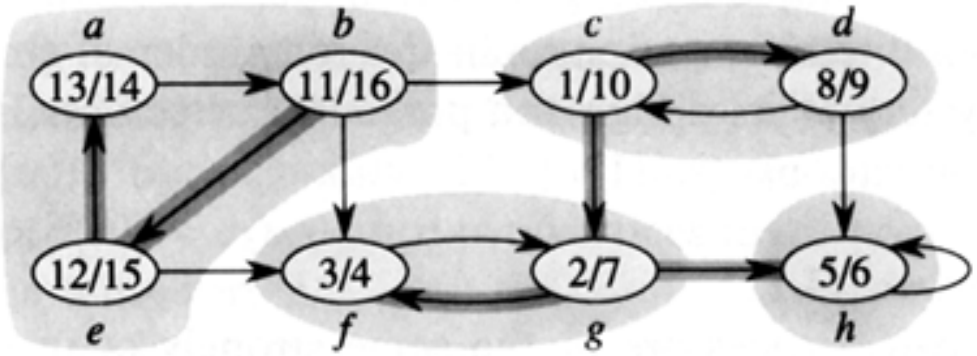
Componentes Fortemente Conectados

- Algoritmo a seguir calcula os componentes fortemente conectados de um grafo orientado $G=(V,E)$ usando duas pesquisas primeiro na profundidade, uma sobre G e uma sobre G^T

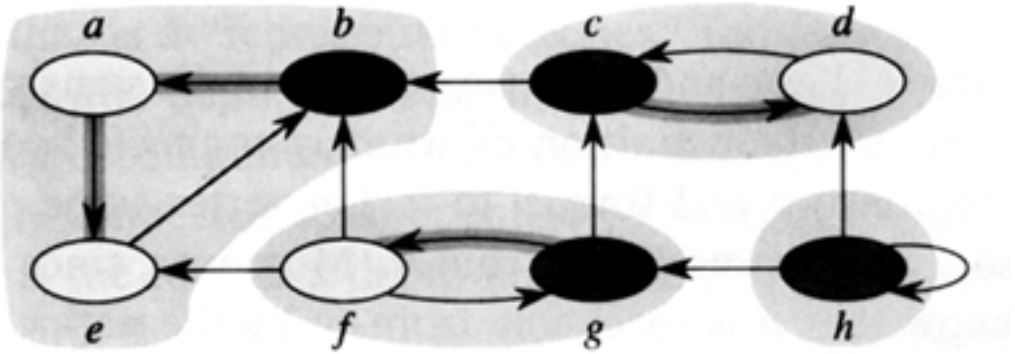
```
STRONGLY-CONNECTED-COMPONENTS (G)
1  call DFS (G) to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS (G^T), but in the main loop of DFS, consider the vertices
         in order of decreasing f[u] (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in
line 3 as a separate strongly connected component
```

Componentes Fortemente Conectados

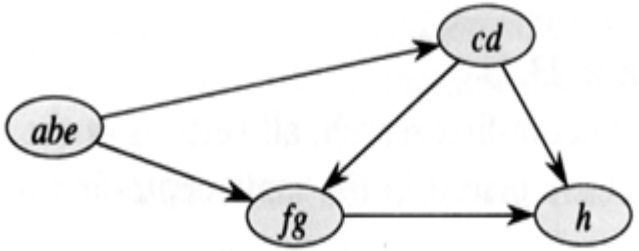
Grafo orientado G



Grafo G^T



Grafo de componentes acíclicos obtido pela condensação de cada componente fortemente conectado de G, de modo que apenas um único vértice permaneça em cada componente



Componentes Fortemente Conectados

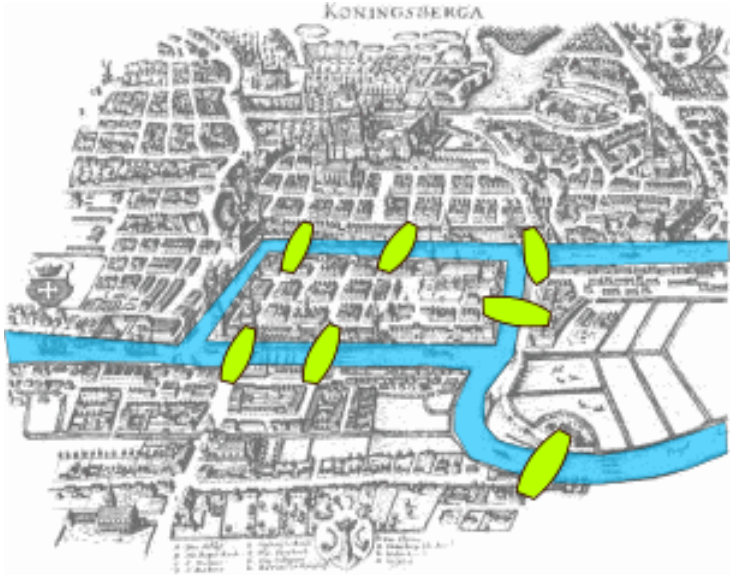
- Análise de Complexidade:
  - Busca em profundidade sobre G: $O(n^2)$
  - Cálculo de G^T: $O(n^2)$
  - Busca em profundidade sobre G^T : $O(n^2)$
  - Portanto, a complexidade de tempo é $O(n^2) + O(n^2) + O(n^2) = O(n^2)$

Grafos Eulerianos
Grafos Unicursais
Problema do Carteiro Chinês

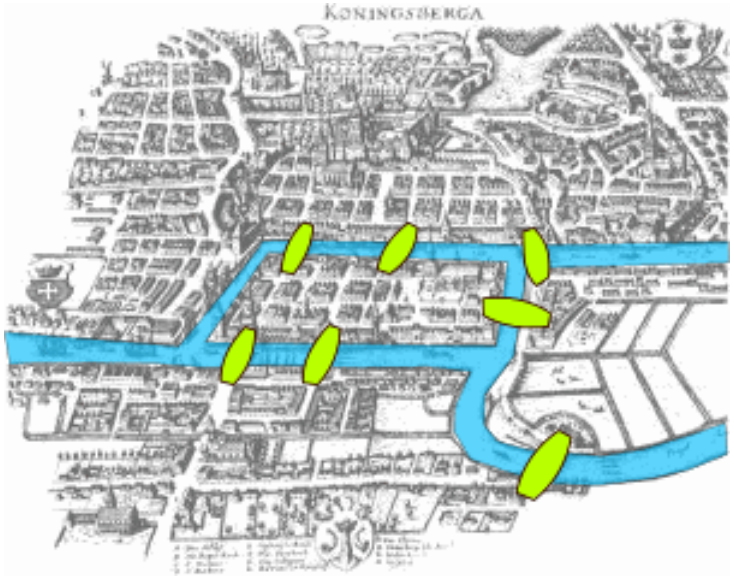
Deo – páginas 23 até 30

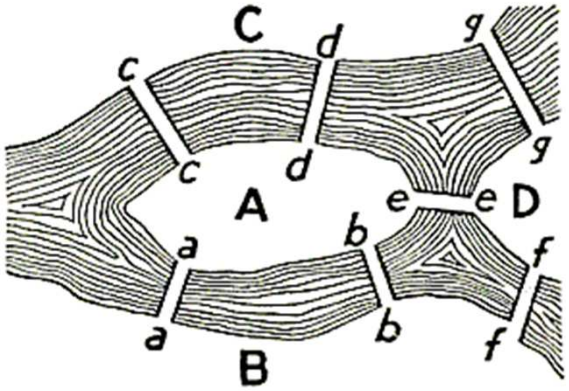
Problema das Pontes de Königsberg

- No século XVIII, Königsberg era a capital da Prússia Oriental
- A cidade foi construída à volta do rio Pregel e para unir todas as partes da cidade foram construídas 7 pontes
- Os habitantes da cidade gostavam de passear pelas pontes e tentavam encontrar uma forma de atravessar todas as pontes exatamente uma vez e retornar ao ponto inicial



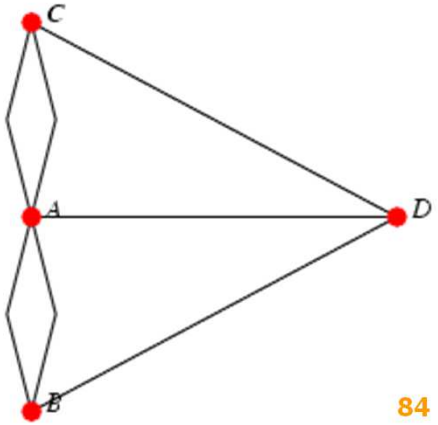
Problema das Pontes de Königsberg





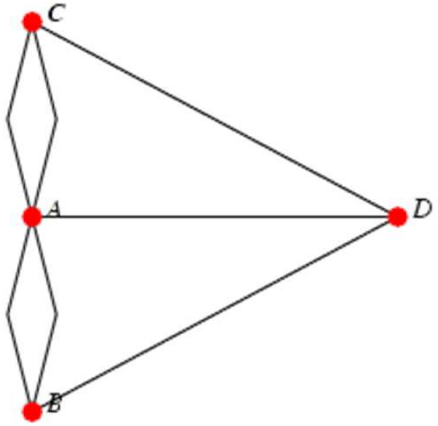
vértices: pontos de terra

aresta: pontes



Problema das Pontes de Königsberg

- Em 1736, Euler mostrou que existe um caminho com ponto de início em qualquer vértice que passa através de cada aresta exatamente uma vez e termina no vértice inicial **se e somente se** todos os vértices tiverem grau par
- O grafo que não cumprir com essas condições não é Euleriano
- No exemplo da ponte, todos os quatro vértices têm grau ímpar, logo não é possível atravessar todas as pontes exatamente uma vez e retornar ao ponto inicial

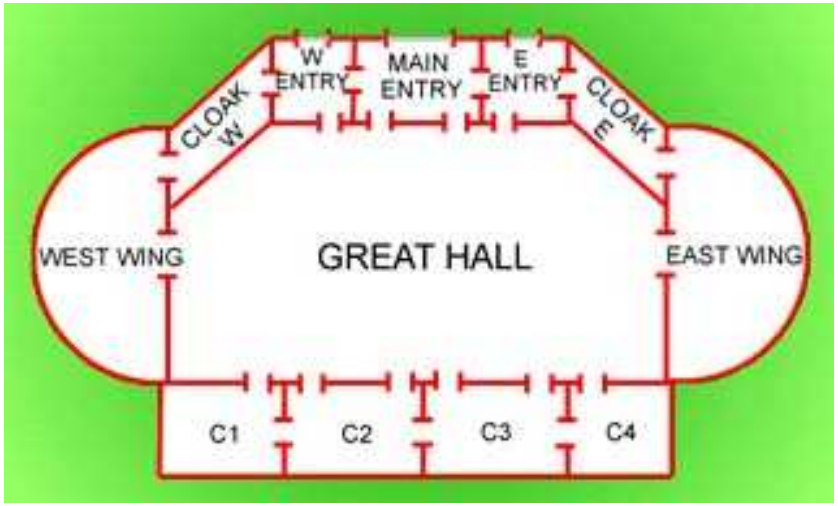


Grafos Eulerianos

- Problema: encontrar um ciclo que passe por todas as arestas uma única vez
- Se é possível encontrar um ciclo que passe por todas as arestas uma única vez, dizemos que G é um grafo euleriano
- TEOREMA: Um grafo é euleriano se, e somente se, todos os seus vértices tiverem grau par

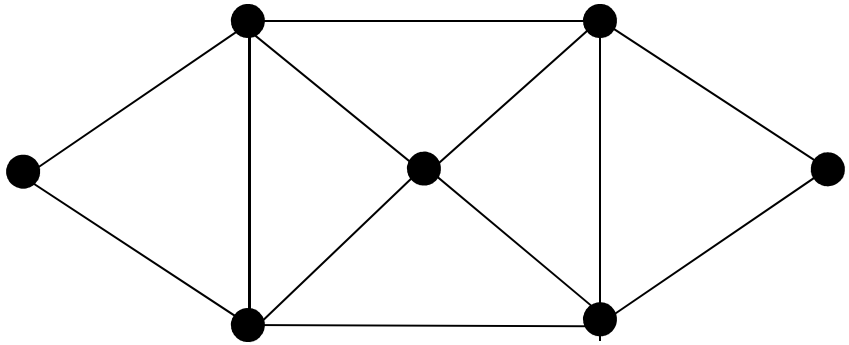
Mapa do Departamento de Matemática

- A figura abaixo ilustra o mapa do Departamento de Matemática de uma importante Universidade. A entrada principal está na parte norte do Departamento. Determine se é possível que uma pessoa possa andar pelo Departamento passando através de cada porta exatamente uma vez e terminando onde começou.



O Problema do Explorador

- Um explorador deseja explorar todas as estradas entre um número de cidades. É possível encontrar um roteiro que passe por cada estrada apenas uma vez e volte a cidade inicial?

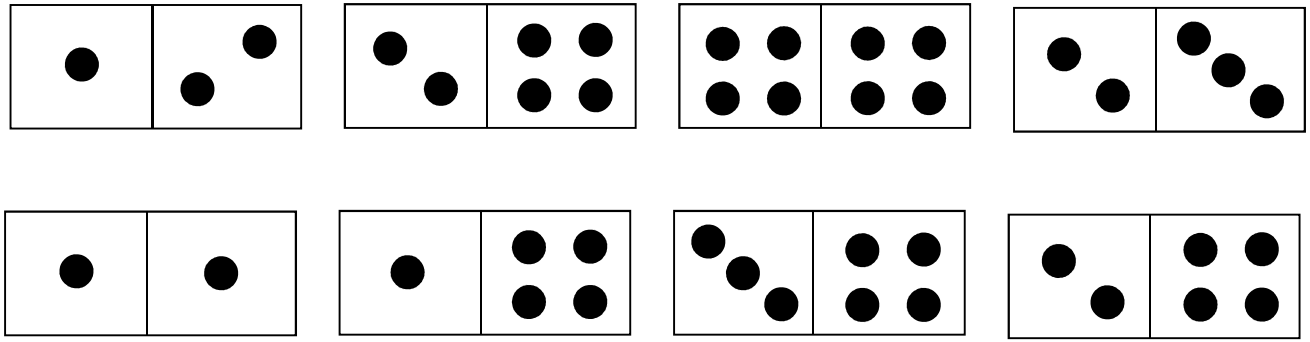


vértices: cidades

aresta: estradas

Problema do Dominó

- É possível arranjar todas as peças de um dominó em um caminho fechado?



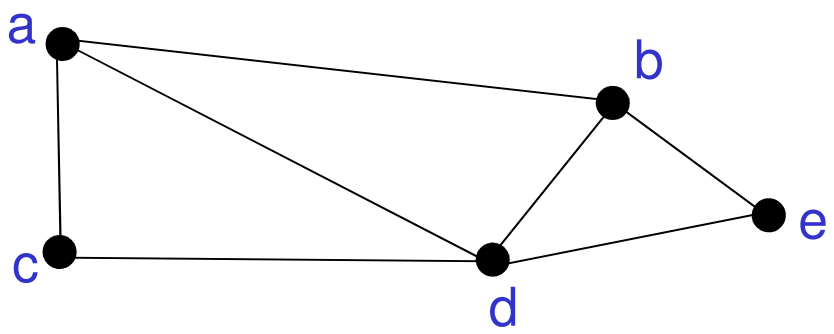
Grafos Eulerianos

- Análise de Complexidade:
  - Para verificar se um grafo é euleriano basta verificar o grau de todos os vértices do grafo
  - Para verificar o grau de um vértice temos que percorrer uma linha da matriz de adjacências que tem tamanho n
  - Como são n vértices, a complexidade é $O(n^2)$

Grafos Unicursais

- Um grafo G é dito unicursal se ele possuir um caminho aberto de Euler, ou seja, se é possível percorrer todas as arestas de G apenas uma vez sem retornar ao vértice inicial.

  Caminho de Euler: a c d a b d e b



- Se adicionarmos uma aresta entre os vértices inicial e final do caminho aberto de Euler, esse grafo passa a ser um grafo euleriano

Grafos Unicursais

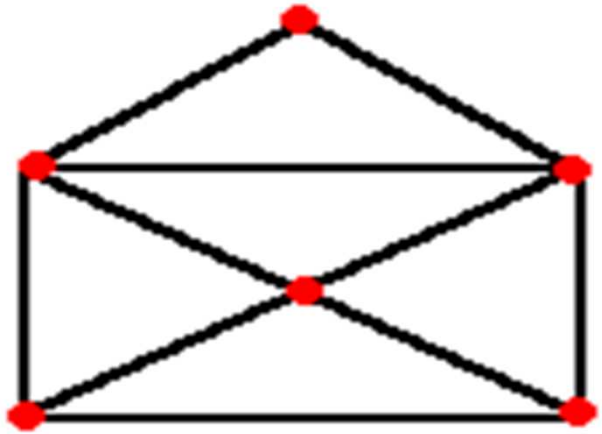
- Um grafo conexo é unicursal se, e somente se, ele possuir exatamente 2 vértices de grau ímpar

- TEOREMA: Em um grafo conexo G com exatamente $2K$ vértices de grau ímpar, existem K subgrafos disjuntos de arestas, todos eles unicursais, de maneira que juntos eles contêm todas as arestas de G

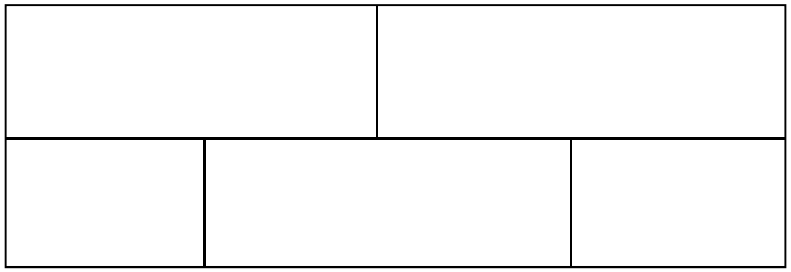
- Casos:
  - Grafo euleriano: todos os vértices de grau par
  - Grafo unicursal: dois vértices de grau ímpar
  - Grafo qualquer: $2K$ vértices de grau ímpar (k-traçável)

Grafos Unicursais

- É possível fazer o desenho abaixo sem retirar o lápis do papel e sem retroceder?

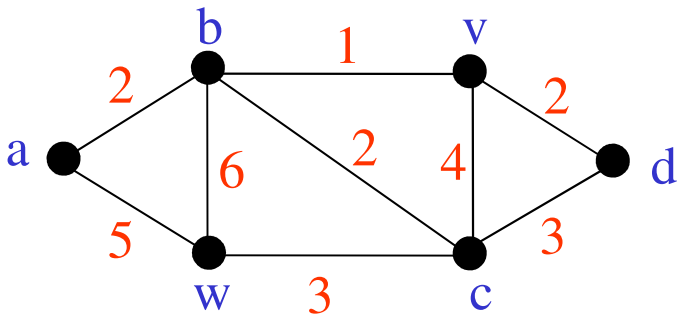


- Quantos traços são necessários para traçar o diagrama abaixo?



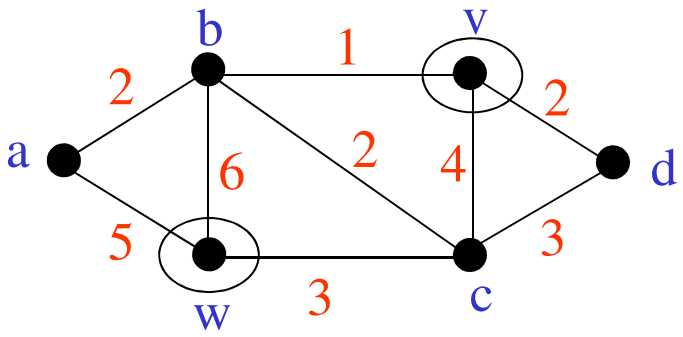
Carteiro Chinês

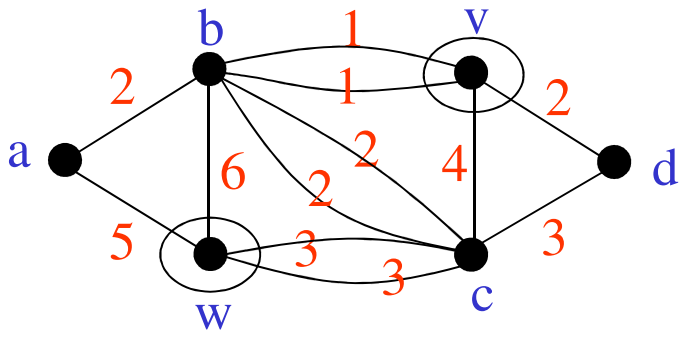
- Um carteiro deseja entregar cartas ao longo de todas as ruas de uma cidade, e retornar ao ponto inicial. Como ele pode planejar as rotas de forma a percorrer a menor distância possível?
  - Se o grafo for euleriano, basta percorrer o ciclo de Euler
  - Caso contrário, algumas arestas serão percorridas mais de uma vez



Carteiro Chinês

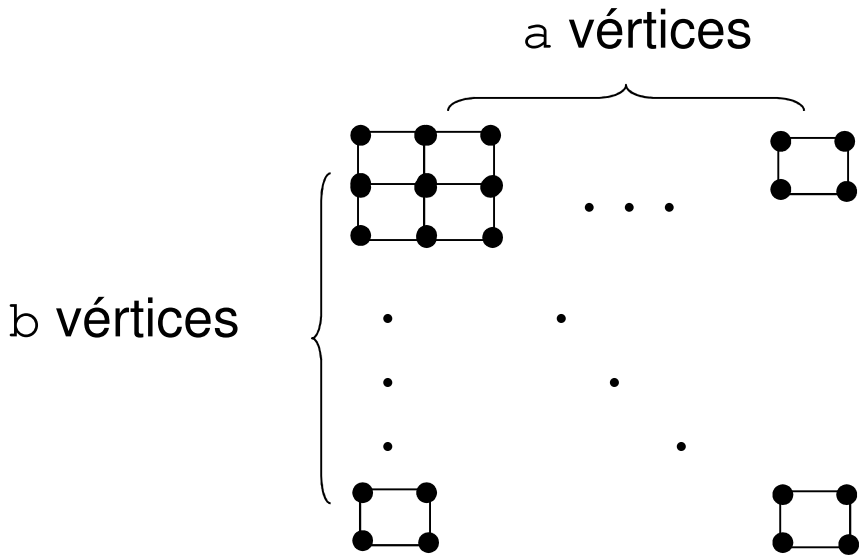
PASSO 1: Identifique os `m` nós de grau ímpar de `G(N,A)` (`m` é sempre par)
PASSO 2: Encontre o "casamento de pares com a mínima distância" (*minimum-length pairwise matching*) desses `m` nós e identifique os `m/2` caminhos mínimos deste "casamento" ótimo.
PASSO 3: Adicione estes `m/2` caminhos mínimos como arcos ligando os nós do "casamento" ótimo. O novo grafo `G(N,A)` contém zero vértices de grau ímpar.
PASSO 4: Encontre um ciclo euleriano em `G(N,A)`. Este ciclo é a solução ótima do problema no grafo original `G(N,A)` e o seu comprimento é igual ao comprimento total das arestas do grafo original mais o comprimento total dos `m/2` caminhos mínimos.





Exercícios

6. Para quais valores de `a` e `b` o grafo abaixo é euleriano?

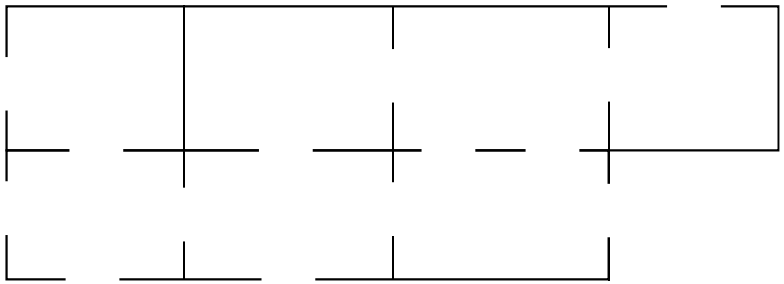


7. Determine os valores de `n` para os quais o grafo completo K_n é euleriano. Para quais valores de `n`, o K_n é unicursal? Justifique.

Exercícios

8. Para o grafo do problema das pontes de Königsberg, qual é o menor número de pontes que devem ser removidas para que o grafo resultante seja unicursal? Quais pontes? Idem para Euleriano.

9. É possível visitar todas as salas passando por todas as portas exatamente uma vez e retornando ao ponto inicial?

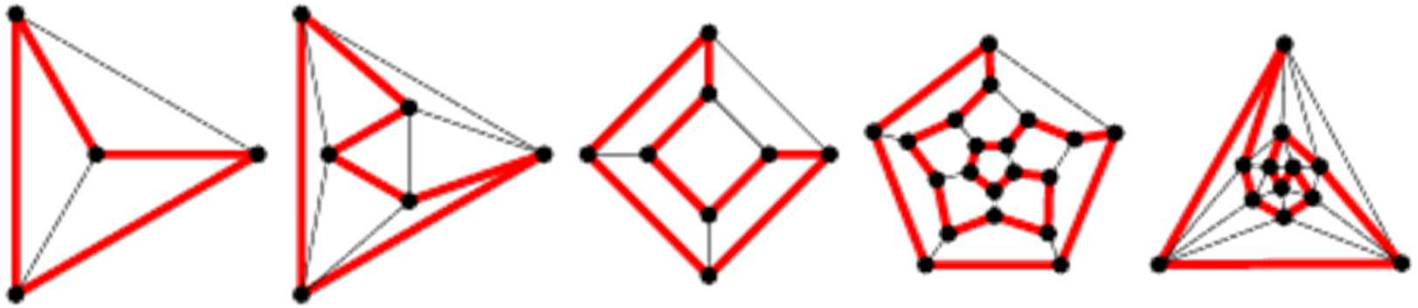


Grafos Hamiltonianos
Passeio do Cavalo
Problema do Caixeiro Viajante

Deo – páginas 30 até 35

Grafos Hamiltonianos

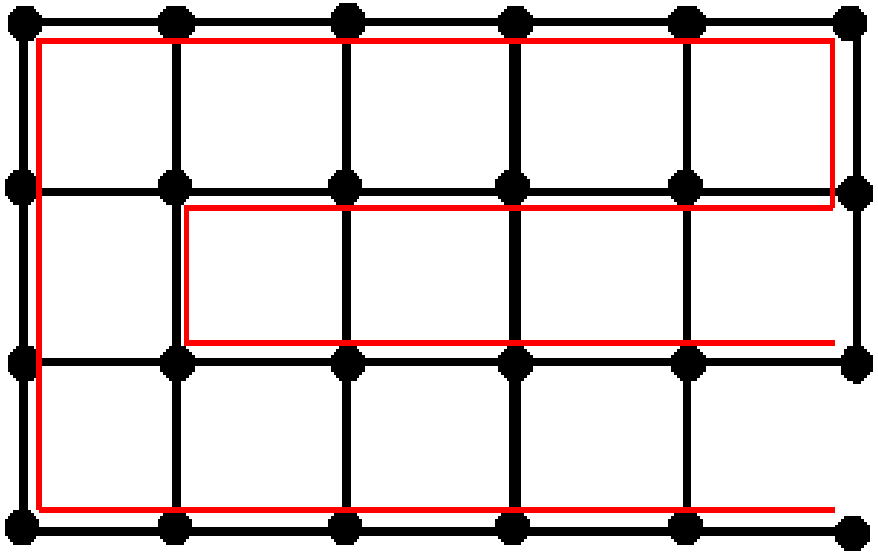
- Um ciclo de Hamilton em um grafo conexo é um ciclo simples que passa por todos os vértices do grafo uma única vez
- Todo grafo que possui um ciclo de hamilton é chamado de grafo hamiltoniano



- O ciclo de hamilton de um grafo com `n` vértices contém `n` arestas

Grafos Hamiltonianos

- Um caminho de Hamilton em um grafo conexo é um caminho que passa por todos os vértices do grafo exatamente uma vez



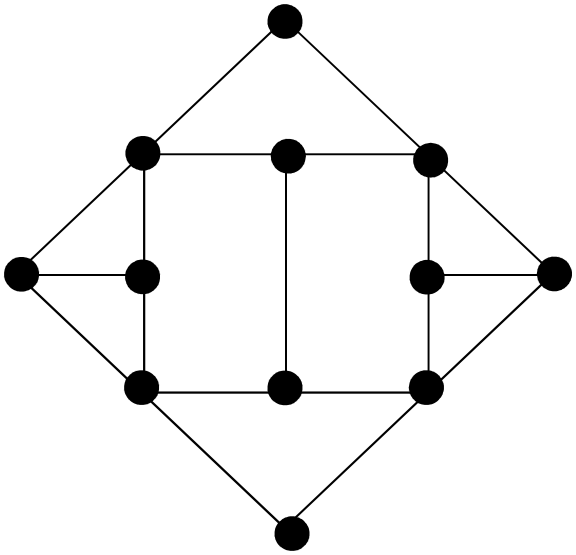
Grafos Hamiltonianos

- Considerações sobre grafos Hamiltonianos:
  - O grafo deve ser conexo
  - *Autoloops* e arestas paralelas podem ser desconsideradas
  - Se um grafo é hamiltoniano, então a inclusão de qualquer aresta não atrapalha esta condição

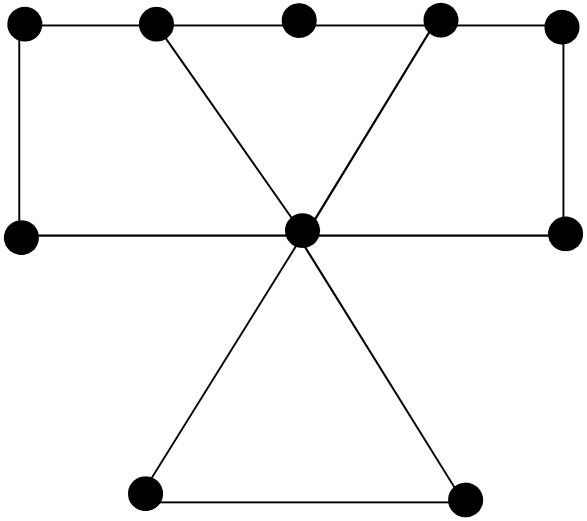
- Análise de Complexidade
  - Para verificar se existe um ciclo de hamilton em um grafo conexo devemos tentar todas as possibilidades, ou seja, devemos gerar todas as permutações nos `n` vértices
  - Portanto, a complexidade do problema é `O(n!)`

Exercícios

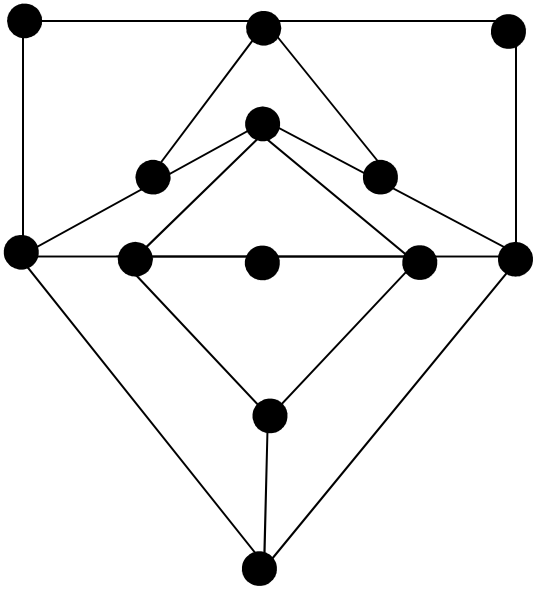
10. Os seguintes grafos são hamiltonianos?



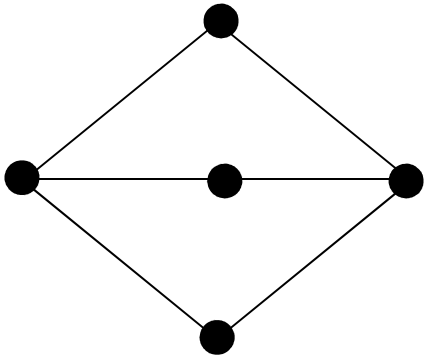
(a)



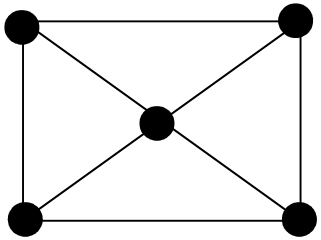
(b)



(c)



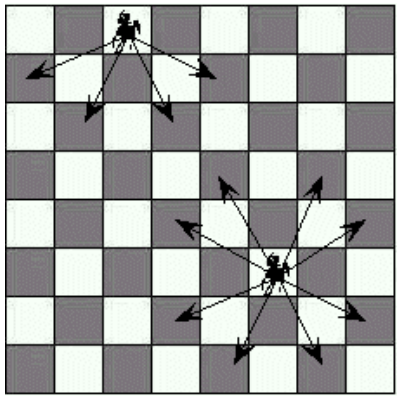
(d)

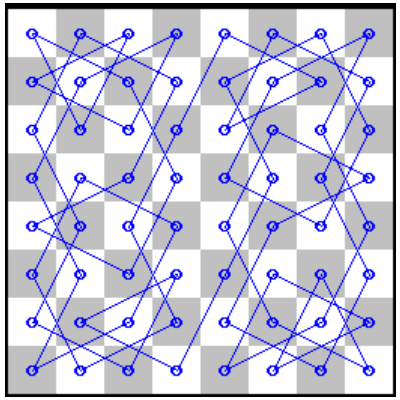


(e)

Passeio do Cavalo

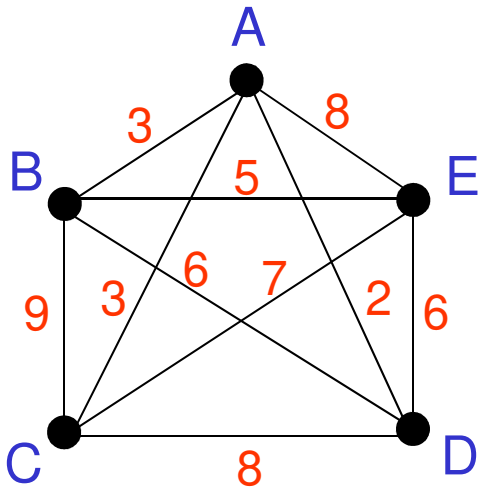
- Um cavalo do xadrez deve começar em alguma posição, visitar todas as posições exatamente uma vez e retornar à posição inicial.
- Para qual tamanho do tabuleiro `nxn` existe esse ciclo?
- Para qual `nxn` o grafo é hamiltoniano?





Problema do Caixeiro Viajante

- Um caixeiro viajante deseja visitar um número de cidades e voltar ao ponto de origem de maneira que ele visite todas as cidades e percorra a menor distância possível. Como escolher sua rota?



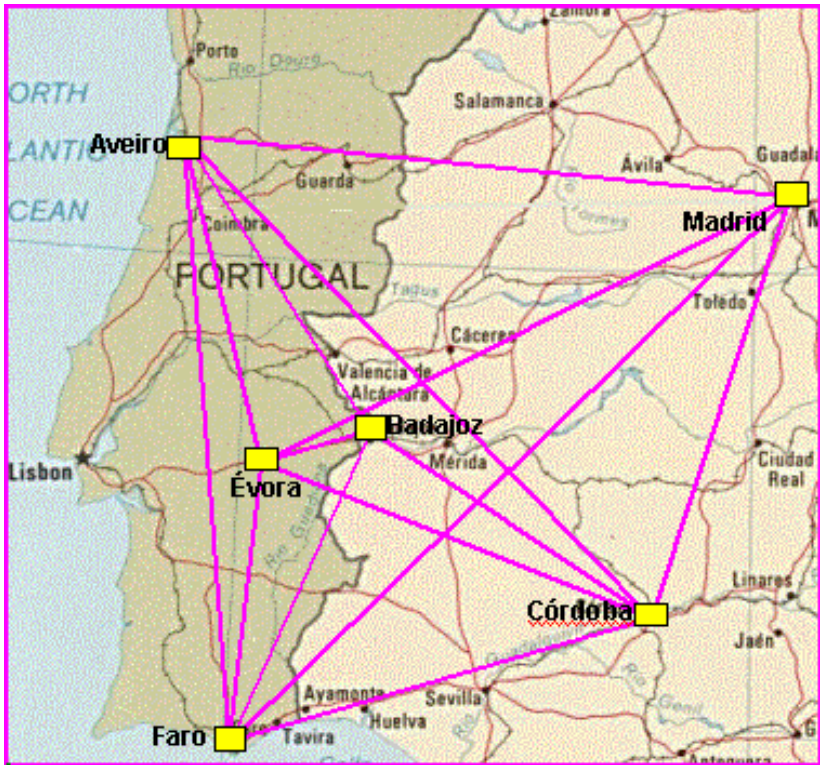
grafo com peso nas arestas

- Vértices: cidades
- Arestas: estradas

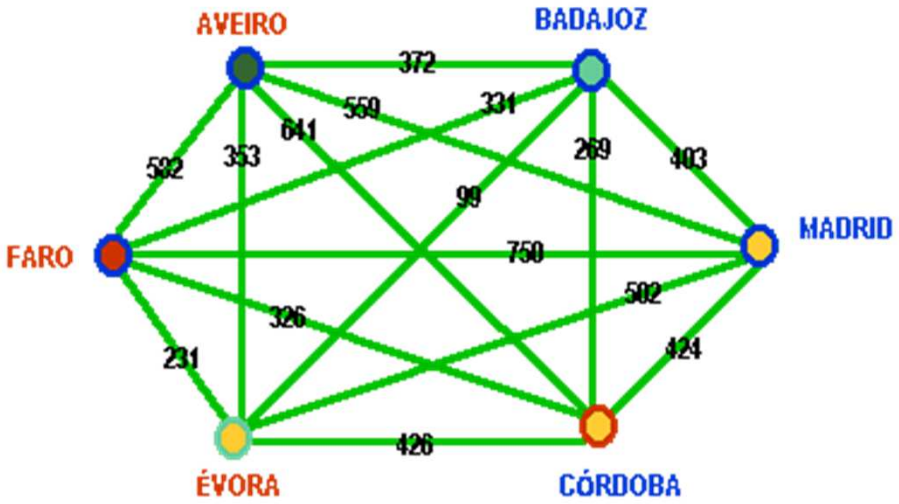
Encontrar um ciclo de hamilton de peso mínimo

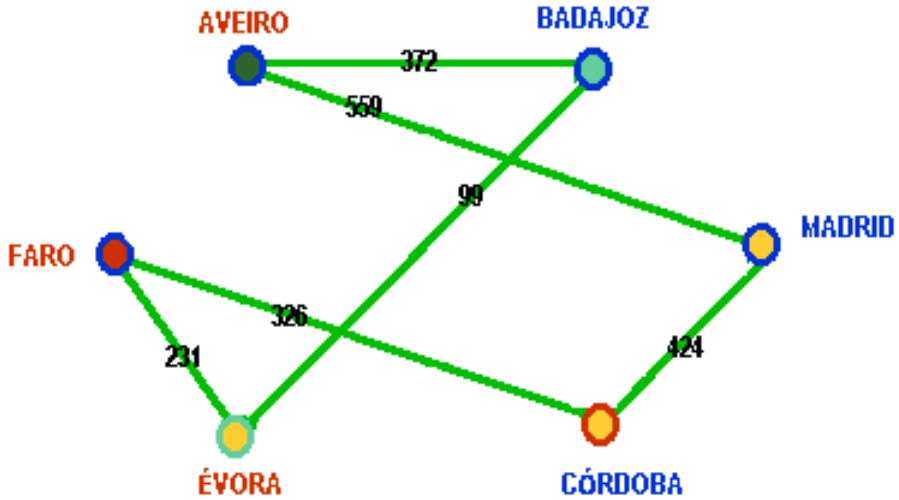
Problema do Caixeiro Viajante

- Um viajante deve visitar clientes instalados em 6 cidades da Península Ibérica. Procura-se determinar qual o menor percurso de forma que o visitante comece em uma cidade, passe por todas as cidades e termine na cidade de origem



Problema do Caixeiro Viajante





Problema do Caixeiro Viajante

- Análise de Complexidade
  - A solução mais direta é gerar todas as permutações e verificar qual é a que possui o menor comprimento. Como o número de permutações é $n!$, esta solução tem complexidade $O(n!)$
  - Usando técnicas de programação dinâmica, é possível resolver o problema com complexidade $O(2^n)$

Exercícios

11. Dê um exemplo de um grafo que seja euleriano mas não seja hamiltoniano.

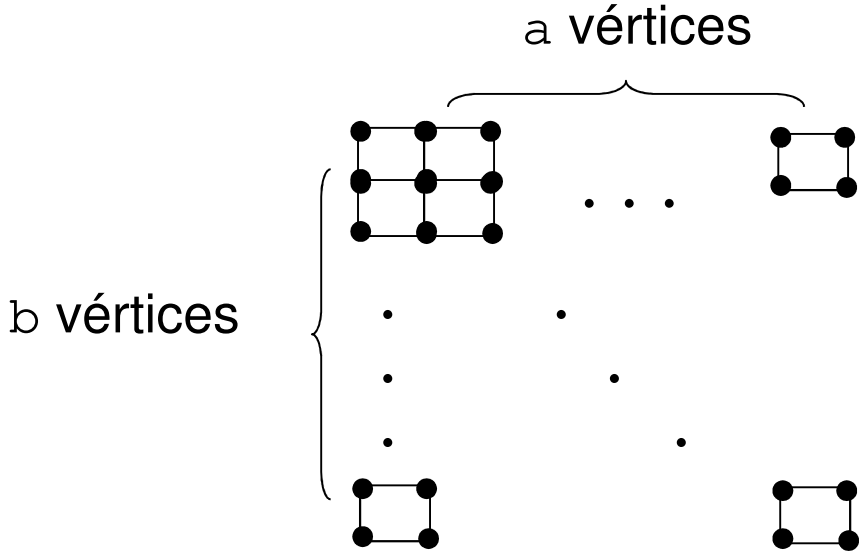
12. Dê um exemplo de um grafo que seja euleriano e hamiltoniano, mas que o ciclo de euler seja diferente do ciclo de hamilton.

13. Desenhe um grafo no qual o ciclo de Euler seja também o ciclo de hamilton. O que podemos dizer desses grafos em geral?

14. Para quais valores de `a` e `b` o grafo bipartido completo $K_{a,b}$ é hamiltoniano? Justifique.

Exercícios

15. Para quais valores de `a` e `b` o grafo abaixo é hamiltoniano?

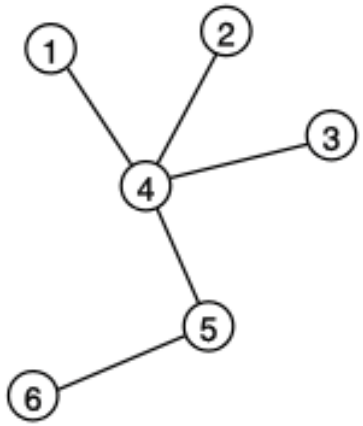


Árvores
Árvores Geradoras Mínimas

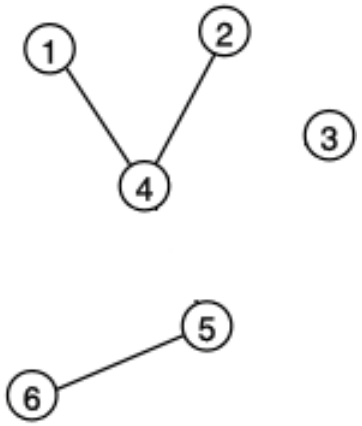
Cormen – páginas 445 até 458
e
Deo – páginas 39 até 44 e 55 até 57 e 61 até 64

Árvore

- Uma árvore é um grafo conexo (existe caminho entre quaisquer dois de seus vértices) e acíclico (não possui ciclos)
- Grafo acíclico mas não conexo é chamado de floresta
- Uma floresta também é definida como uma união disjunta de árvores



árvore com 6 vértices e 5 arestas



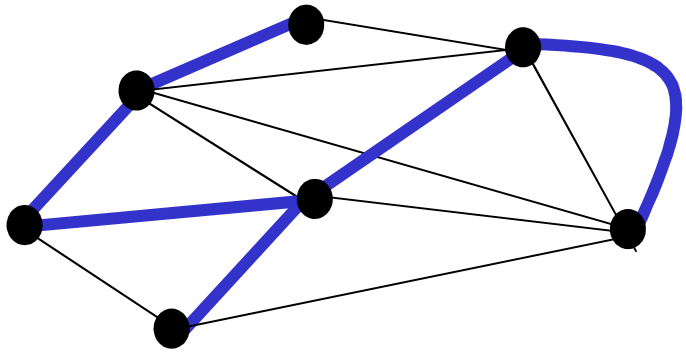
floresta 6 vértices e 3 arestas

Árvore

- Propriedades:
  - Em uma árvore existe um e apenas um caminho entre cada par de vértices.
  - Uma árvore com `n` vértices tem `n-1` arestas
  - Um floresta com `n` vértices e `k` componentes possui `n-k` arestas
  - Um grafo `G` é uma árvore se, e somente se, a remoção de qualquer aresta o desconectar (grafo minimamente conectado)
  - Um grafo `G` com `n` vértices, `n-1` arestas e nenhum ciclo é conexo
  - Toda árvore é um grafo bipartido e planar
  - Em qualquer árvore com pelo menos 2 vértices existem pelo menos 2 vértices pendentes

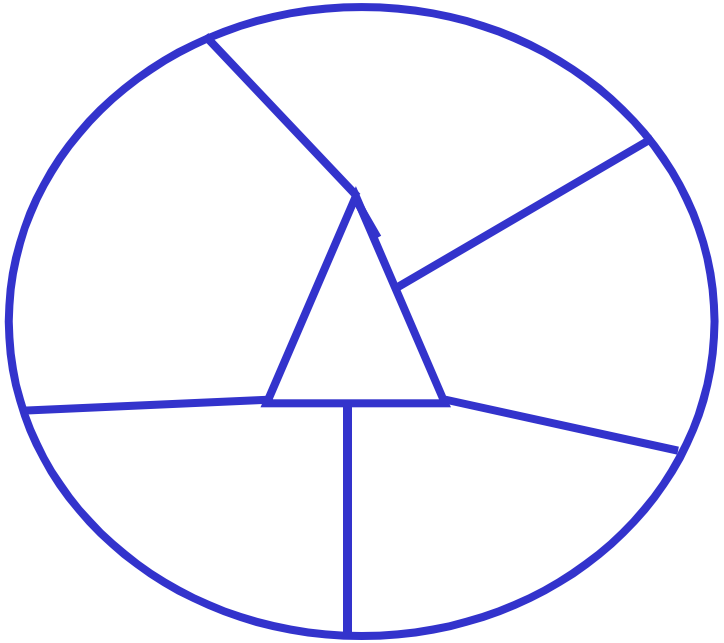
Árvore Geradora (Árvore Espalhada ou Árvore de Extensão)

- Uma árvore geradora de um grafo conexo G é um subgrafo de G que contém todos os vértices de G e é uma árvore
- Árvore geradora só é definida para grafos conexo porque toda árvore é conexa. Grafos não conexo possuem florestas geradoras



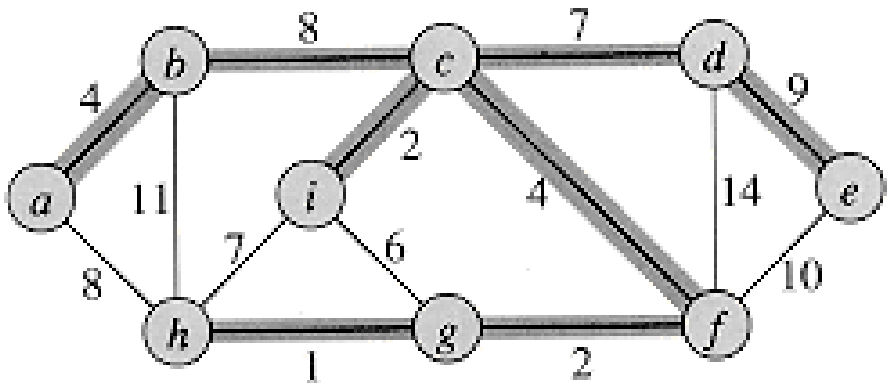
Árvore Geradora

- Um fazendeiro possui 6 lotes murados como mostrado na figura abaixo. Os lotes estão cheios de água. Quantos muros devem ser removidos para que toda a água seja liberada?



Árvore Geradora Mínima

- Árvore Geradora Mínima é a árvore geradora de menor peso de G
- Dado um grafo G com pesos associados às arestas, encontrar uma árvore geradora mínima de G



Árvore Geradora Mínima

- A partir de um grafo conectado não orientado $G=(V,E)$ com uma função peso $w:E\rightarrow\mathbf{R}$, desejamos encontrar uma árvore geradora mínima correspondente a G
- Algoritmos:
  - Algoritmo de Kruskal
  - Algoritmo de Prim

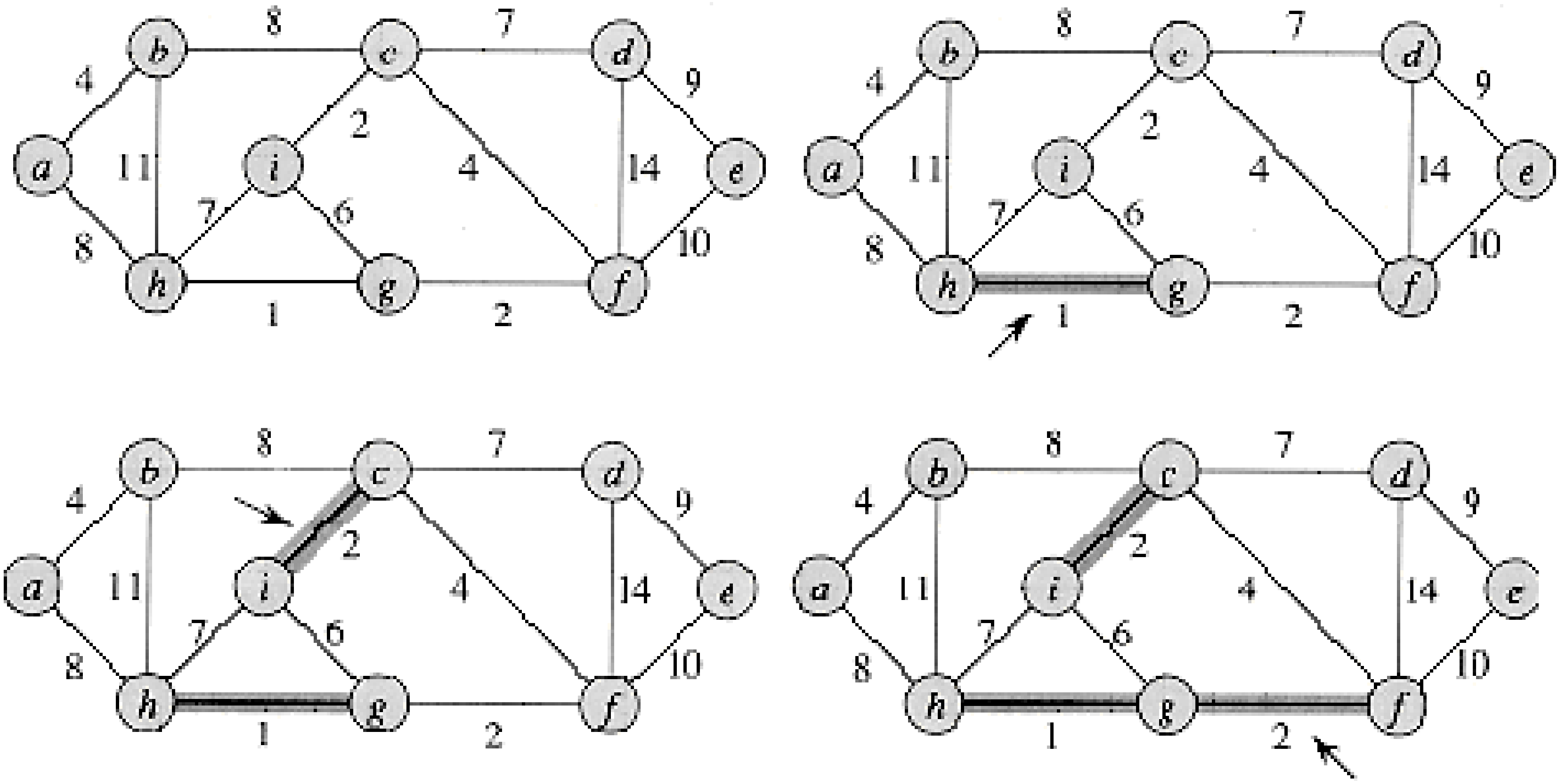
Algoritmo de Kruskal

- Encontre uma aresta segura para adicionar à floresta crescente encontrando, de todas as arestas que conectam duas árvores quaisquer na floresta, uma aresta `(u,v)` de peso mínimo
- Utiliza uma estrutura de dados de conjuntos disjuntos para manter vários conjuntos disjuntos de elementos
- Cada conjunto contém os vértices em uma árvore da floresta atual
- A operação `FIND-SET(u)` retorna um elemento representativo do conjunto que contém `u`
- A combinação de árvores é realizada pelo procedimento `UNION`

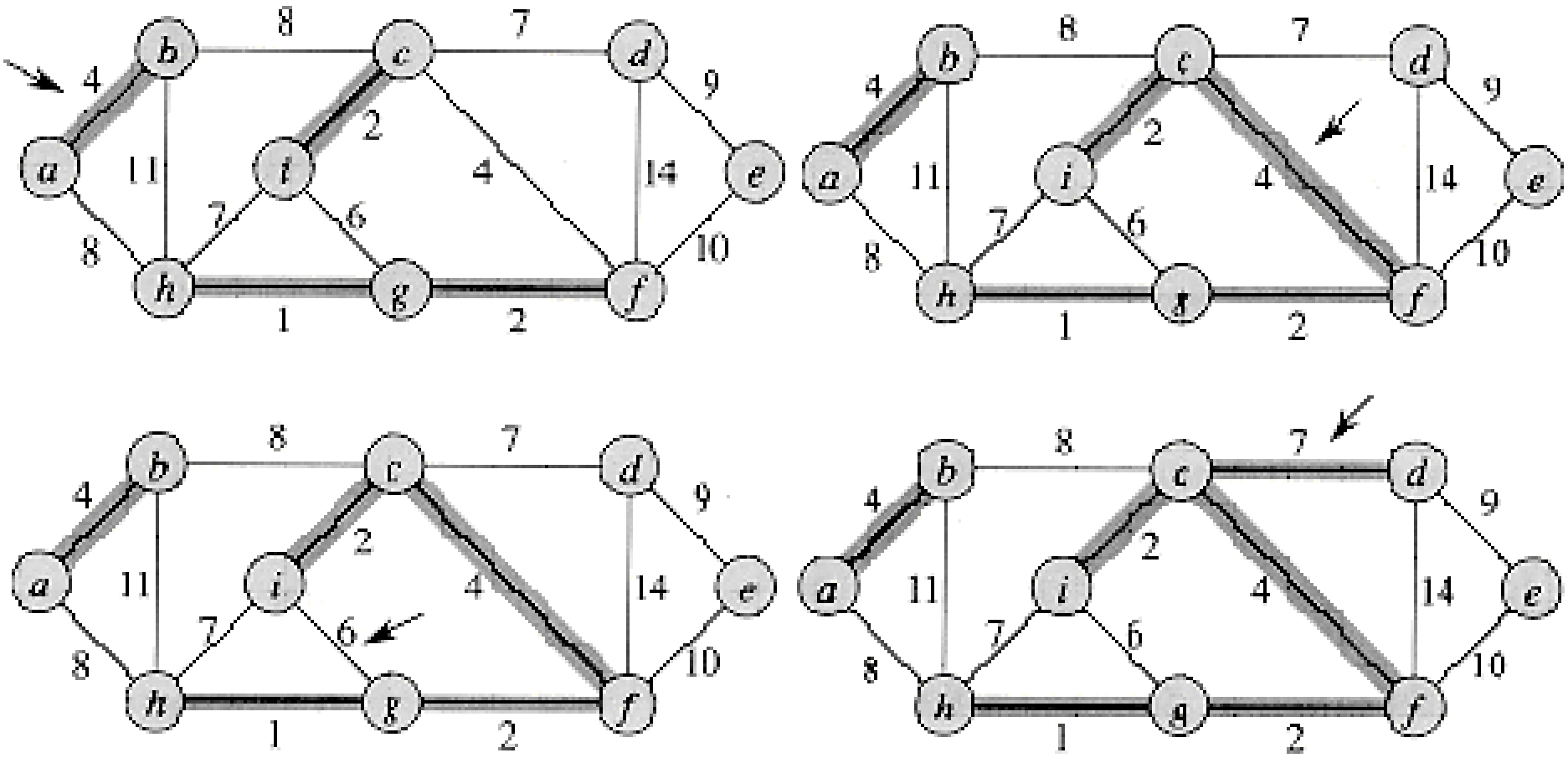
Algoritmo de Kruskal

```
MST-KRUSKAL(G,w)
A ← ∅
for cada vértice v ∈ V[G]
    do MAKE-SET(v)
ordenar as arestas de E por peso w não decrescente
for cada aresta (u,v)∈E, em ordem de peso não decrescente do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A ← A ∪ {(u,v)}
        UNION(u,v)
return A
```

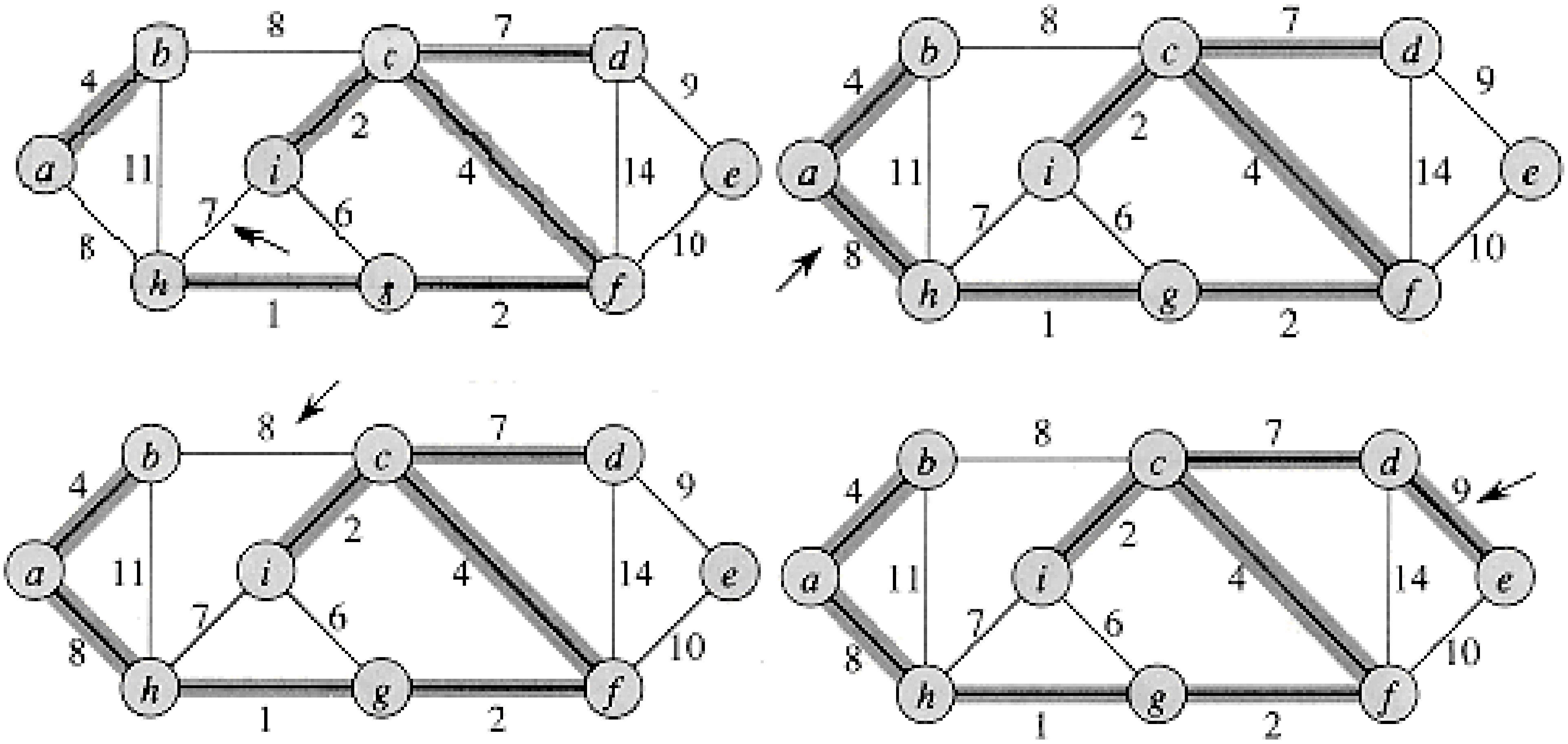
Algoritmo de Kruskal



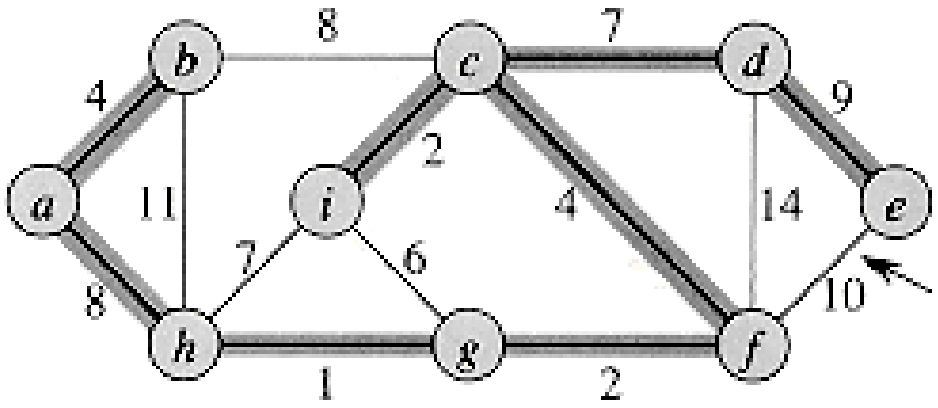
Algoritmo de Kruskal

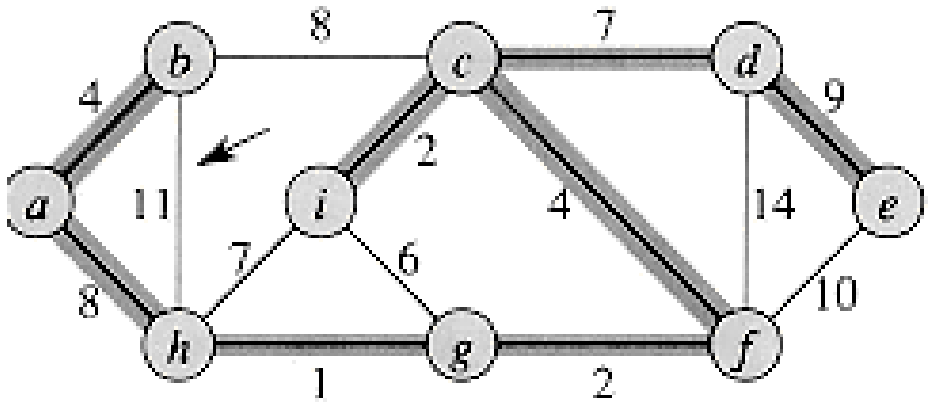


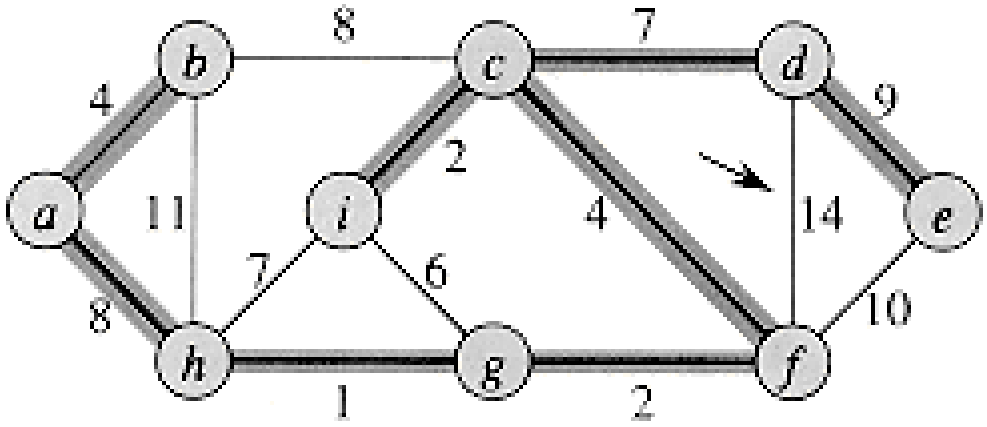
Algoritmo de Kruskal



Algoritmo de Kruskal







Algoritmo de Kruskal

- Análise de Complexidade (considerando que a implementação da floresta de conjuntos disjuntos seja feita com as heurísticas de união por ordenação e compressão de caminho):
  - Tempo para ordenar as arestas é $O(e \log e)$
  - O segundo `for` executa $O(e)$ operações FIND-SET e UNION sobre a floresta de conjuntos disjuntos que juntamente com as n operações MAKE-SET, demoram ao todo o tempo $O((n+e)\alpha(n))$, onde α é a função de crescimento muito lento
  - Como G é supostamente conectado, $e \geq n-1$, e assim as operações de conjuntos disjuntos demoram o tempo $O(e\ \alpha(n))$
  - Como $\alpha(n)=O(\log n)=O(\log e)$, o tempo total é $O(e \log e)$
  - Considerando que $e \leq n^2$, $\log e = O(\log n)$, e portanto podemos redefinir o tempo de execução do algoritmo de Kruskal como $O(e \log n)$

Algoritmo de Prim

- As arestas no conjunto `A` sempre formam uma árvore única
- A árvore começa a partir de um vértice de raiz arbitrária `r` e aumenta até a árvore alcançar todos os vértices em `V`
- Em cada etapa, uma aresta leve conectando um vértice de `A` a um vértice em `V–A` é adicionada à árvore
- Quando o algoritmo termina, as arestas em `A` formam uma árvore geradora mínima
- Durante a execução do algoritmo, todos os vértices que não estão na árvore residem em uma fila de prioridade mínima `Q` baseada em um campo `chave`
- Para cada vértice `v`, `chave[v]` é o peso mínimo de qualquer aresta que conecta `v` a um vértice na árvore
- π`[v]` é o pai de `v` na árvore

Algoritmo de Prim

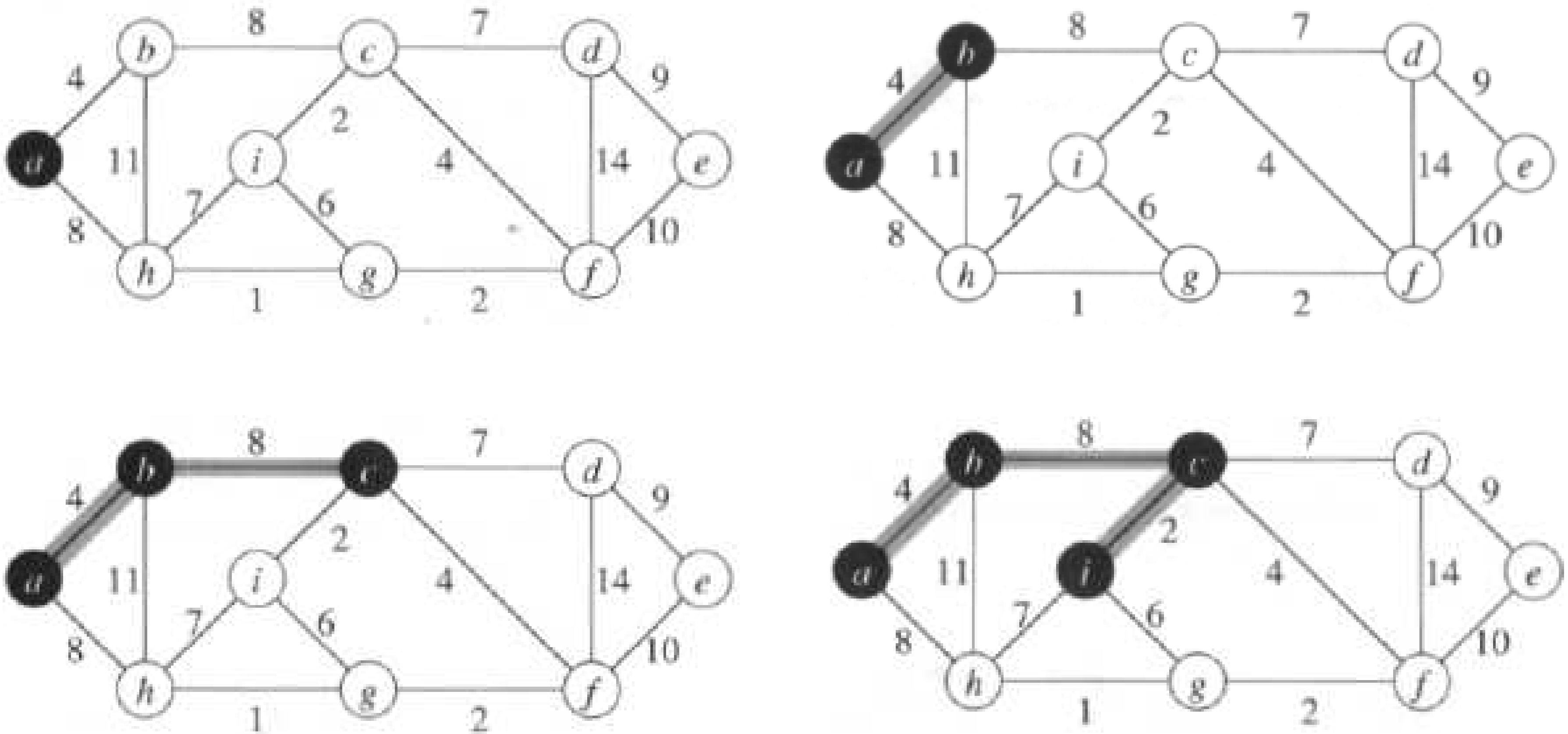
```
MST-PRIM (G,w,r)
for cada u ∈ V[G] do
    chave[u] ← ∞
    π[u] ← NIL
chave[r] ← 0
Q ← V[G]
while Q ≠ 0 do
    u ← EXTRACT-MIN(Q)
    for cada v ∈ Adj[u] do
        if v ∈ Q e w(u,v) < chave[v] then
            π[v] ← u
            chave[v] ← w(u,v)
```

BUILD-MIN-HEAP

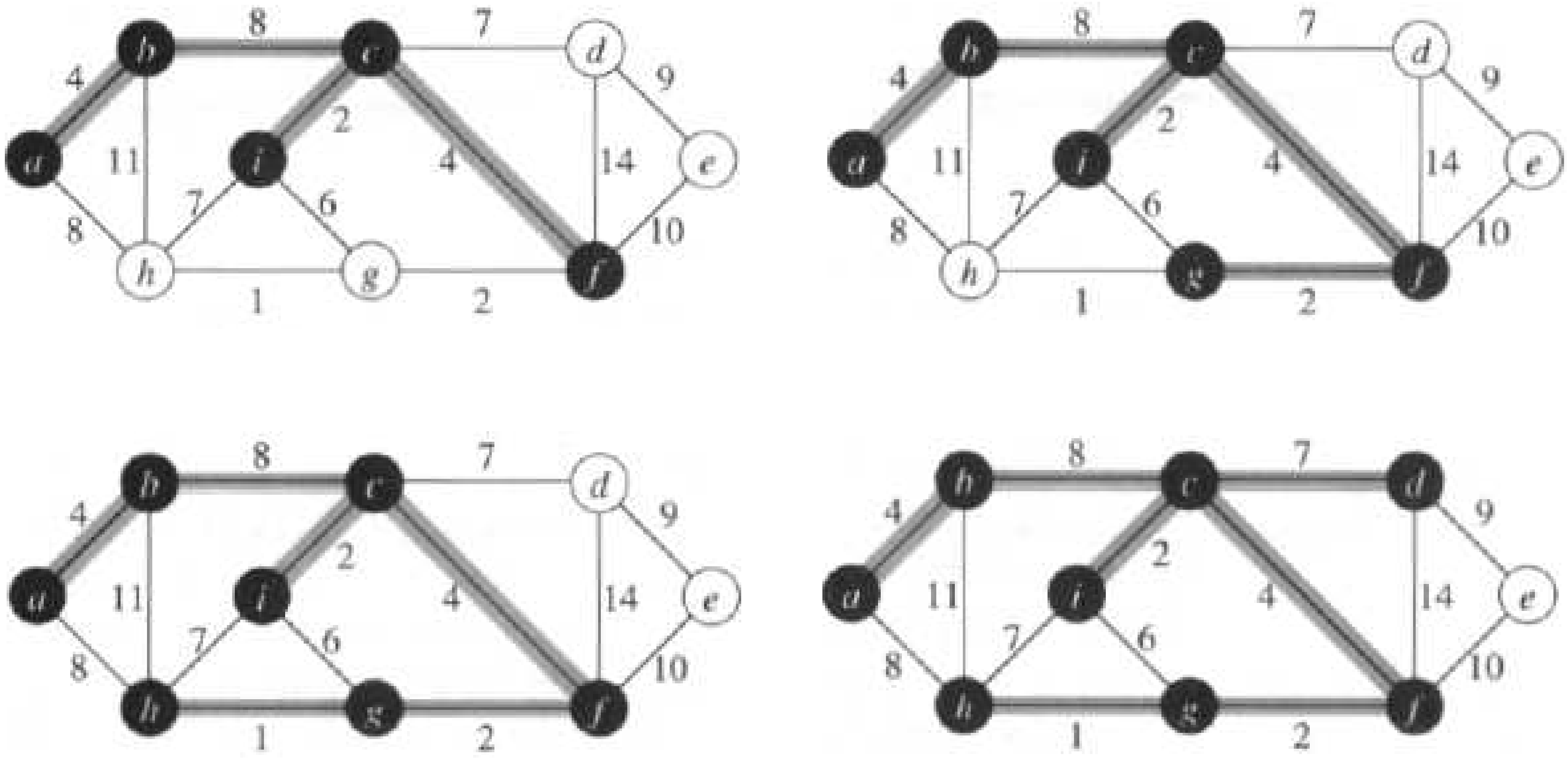
INSERE NA AGM

DECREASE-KEY

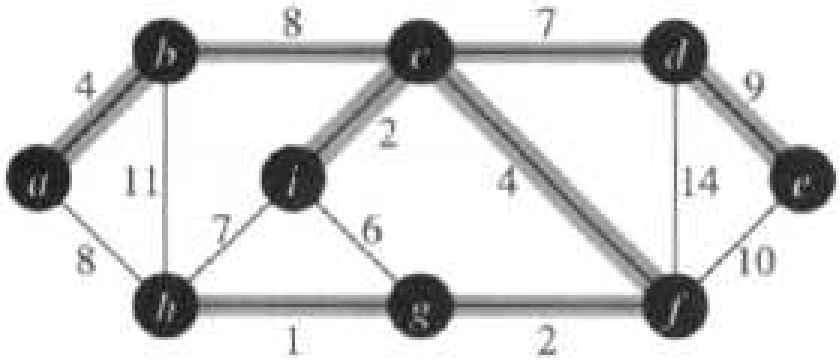
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim

- Análise de Complexidade
  - BUILD-MIN-HEAP é executado em tempo `O(n)`
  - O corpo do **while** é executado `n` vezes, e como cada operação de EXTRACT-MIN demora `O(log n)`, o tempo total para todas as chamadas a EXTRACT-MIN é `O(n log n)`
  - O loop **for** é executado completamente `O(e)` vezes, pois a soma dos comprimentos de todas as listas de adjacências é `2*e`
  - Dentro do *loop* **for**, o teste de pertinência a `Q` pode ser implementado em tempo constante, mantendo-se um bit para cada vértice que informa se ele está ou não em `Q`, e atualizando-se o bit quando o vértice é removido de `Q`
  - A operação DECREASE-KEY sobre o heap mínimo pode ser implementada em tempo `O(log n)`
  - Portanto, o tempo total é `O(n log n + e log n)` = `O(e log n)`

Exercícios

16. Seja G um grafo conexo. O que podemos dizer sobre:
    a) Uma aresta de G que aparece em todas as árvores geradoras?
    b) Uma aresta de G que não aparece em nenhuma árvore geradora?

17. Em um dado grafo simples e conexo G, existe uma aresta e_s cujo peso é menor do que o peso de qualquer outra aresta de G. Podemos dizer toda AGM de G conterá e_s? Justifique.

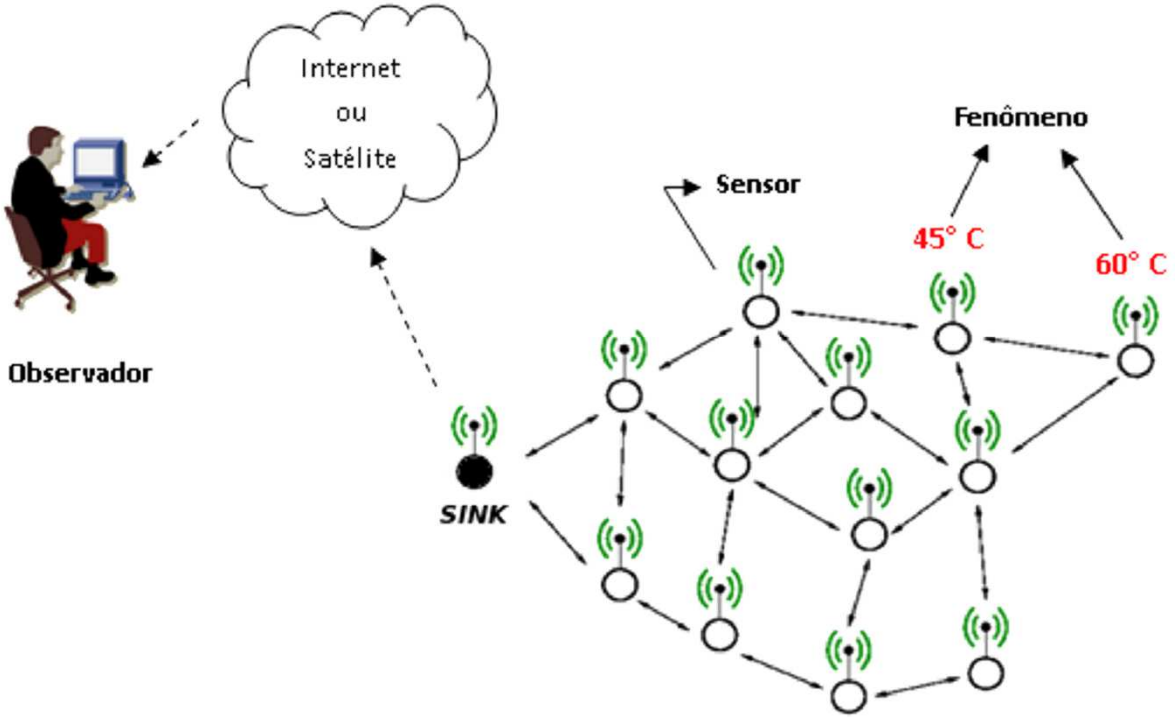
18. Em um dado grafo simples e conexo G, existe uma aresta e_b cujo peso é maior do que o peso de qualquer outra aresta de G. Podemos dizer que nenhuma AGM conterá e_b? Justifique.

Árvore de Steiner

Árvore de Steiner

Problema:

Dada uma rede representada por um grafo $G = (V, E)$, onde $V = v_1, \cdots, v_n$ é o conjunto de nós sensores, e E são as arestas que representam as conexões entre eles com um custo associado, o problema está em como construir a árvore de custo mínimo conectando todos os nós sensores $S = s_1, s_2, \cdots, s_m$, $S \subseteq V$ ao nó sink.



Árvore de Steiner

Problema:

- É um problema em otimização combinatória.
  – Consiste em achar a menor árvore que conecta um conjunto de pontos dados.
- Semelhante ao problema da árvore geradora mínima:
  – Dado um conjunto V de pontos (vértices), interligá-las por um grafo de menor tamanho, que é a soma dos tamanhos de todas as arestas.
  – As AGMs conectam um conjunto de pontos dados e possuem custo mínimo, mas requerem que todas as conexões sejam entre estes pontos.
- Diferença para o problema da AGM: vértices intermediários e arestas extras podem ser adicionados ao grafo, a fim de reduzir o comprimento da árvore de expansão.
  – Isso serve para diminuir o comprimento total da conexão (pontos ou vértices de Steiner).
  – Podem-se utilizar pontos extras, de modo que o custo da árvore gerada seja ainda menor do que o custo da AGM.
- Resultado: uma árvore, conhecida como a árvore Steiner.
  – Pode haver várias árvores Steiner para um determinado conjunto de vértices inicial.

Árvore de Steiner

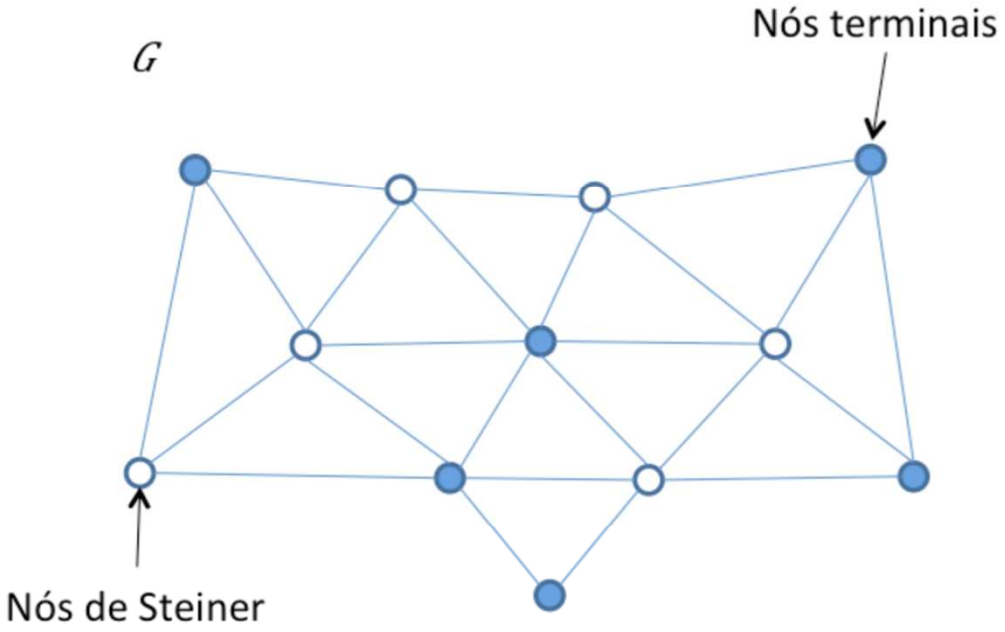
- Considere um grafo $G(V, E)$, não dirigido
- Considere também um subconjunto $T \in V$ (de nós terminais) e o subconjunto $\in$ (de nós de Steiner)
- Tais que $T = V$ e $T \cap S = \varnothing$
- A Árvore de Steiner é uma árvore em G que une todos os vértices em T e, opcionalmente, alguns vértices de S

Importante!

- Não é necessário utilizar todos os vértices de V.
- Vértices em S são utilizados para manter a conectividade.
- Problema NP-Completo

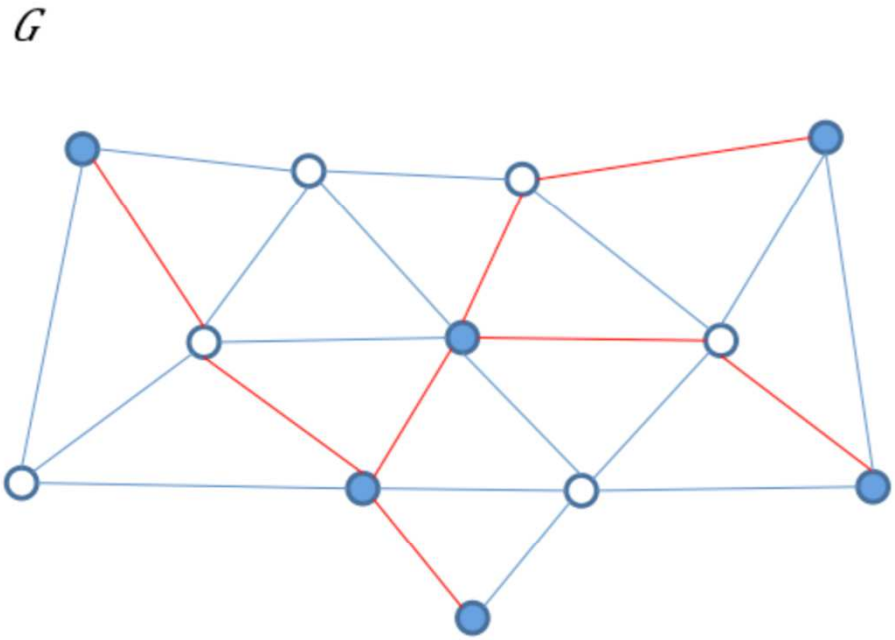
Árvore de Steiner

- A partir de um grafo com nós terminais e nós de Steiner, o objetivo do problema de Steiner é encontrar uma AGM que utilize o menor número de nós de Steiner possível.
- Os nós de Steiner são adicionados no grafo.
  – Assim, eles mantém o grafo resultante conexo.



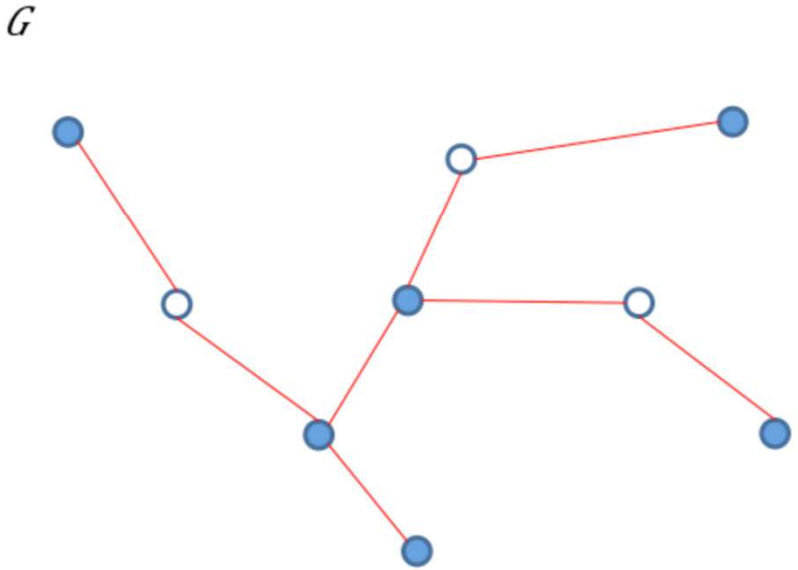
Árvore de Steiner

- A partir de um grafo com nós terminais e nós de Steiner, o objetivo do problema de Steiner é encontrar uma AGM que utilize o menor número de nós de Steiner possível.
- Os nós de Steiner são adicionados no grafo.
  – Assim, eles mantém o grafo resultante conexo.



Árvore de Steiner

- A partir de um grafo com nós terminais e nós de Steiner, o objetivo do problema de Steiner é encontrar uma AGM que utilize o menor número de nós de Steiner possível.
- Os nós de Steiner são adicionados no grafo.
  – Assim, eles mantém o grafo resultante conexo.

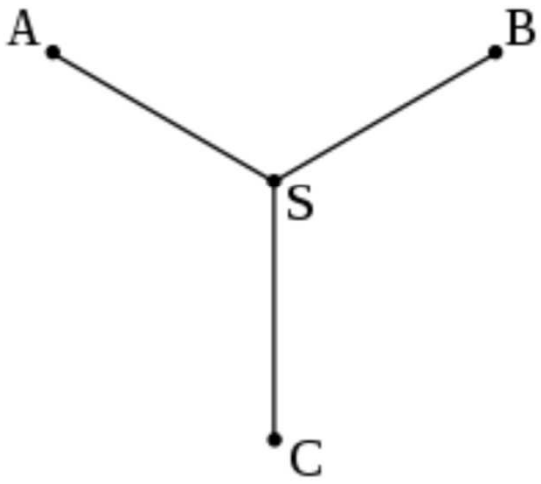


Os nós de Steiner mantém o grafo resultante conexo.

Árvore de Steiner

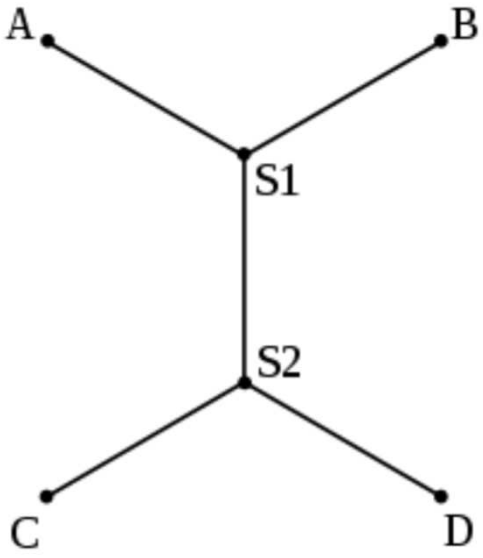
Três pontos:

Árvore de Steiner para três pontos A, B, e C (note que existem conexões diretas entre A, B, C). O vértice de steiner é o S.



Quatro pontos:

Solução para quatro pontos (note que existem dois vértices de steiner – S1 e S2).



Árvore de Steiner

Aplicações:

- Layout de circuito.
- Projeto de rede.

Importante!

- Apesar de ser NP-completo, alguns casos restritos podem ser resolvidos em tempo polinomial.
  – Na prática, heurísticas são utilizadas.

Algoritmos para o Menor Caminho

Cormen – páginas 470 até 475

Algoritmos para o Menor Caminho

- Um motorista deseja encontrar a rota mais curta possível do Rio de Janeiro a São Paulo
- Dado um mapa rodoviário do Brasil no qual a distância entre cada par de interseções adjacentes esteja marcada, como podemos determinar essa rota mais curta?

Algoritmos para o Menor Caminho

- Para resolver de forma eficiente problemas de menor caminho, utilizamos um grafo ponderado $G=(V,E)$, com função de peso $w:E\rightarrow\mathbf{R}$ que mapeia arestas para pesos de valores reais
- O peso do caminho $p=\langle v_0, v_1, \ldots, v_k \rangle$ é o somatório dos pesos de suas arestas constituintes

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

Algoritmo de Dijkstra

- O algoritmo de Dijkstra resolve o problema de caminhos mais curtos de única origem em um grafo ponderado $G=(V,E)$ para o caso no qual os pesos de arestas são não negativos
- Portanto, $w(u, v) \geq 0$ para cada aresta $(u, v) \in E$

Algoritmo de Dijkstra

- Mantém um conjunto `S` de vértices cujos pesos finais de caminhos mais curtos desde a origem `s` já foram determinados
- O algoritmo seleciona repetidamente o vértice `u ∈ V−S` com a estimativa mínima de caminhos mais curtos, adiciona `u` a `S` e relaxa todas as arestas que saem de `u`
- A fila de prioridades mínima `Q` de vértices possui os valores `d` como chave

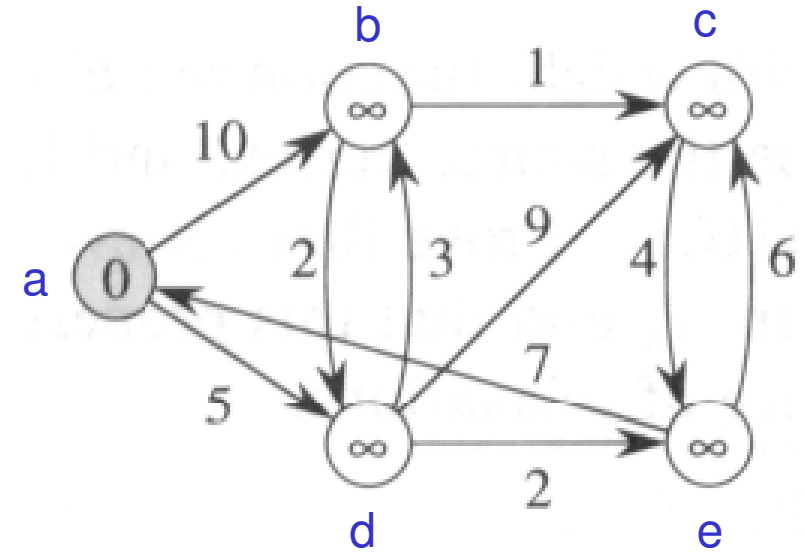
Algoritmo de Dijkstra

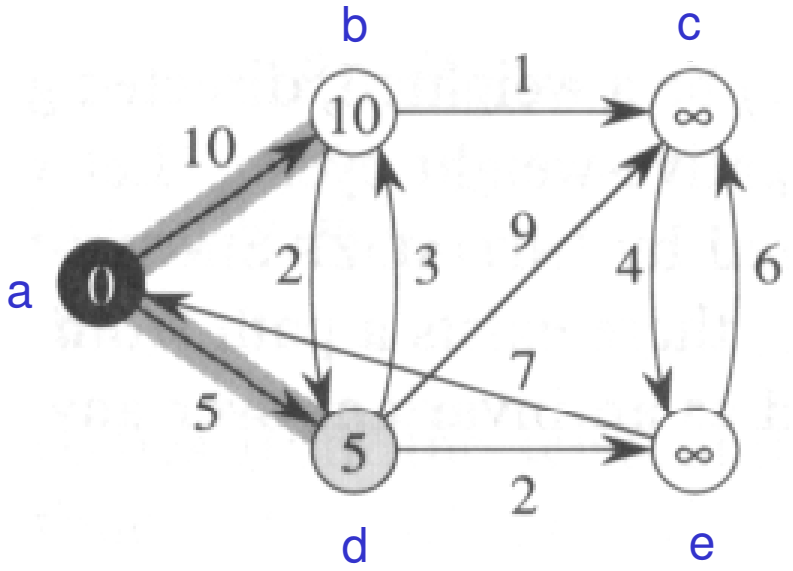
```
DIJKSTRA(G,w,s)
for cada vértice v ∈ V[G] do
    d[v] ← ∞
    π[v] ← NIL
d[s] ← 0
S ← ∅
Q ← V[G]
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)
    S ← S ∪ {u}
    for cada vértice v ∈ Adj[u] do
        if d[v] > d[u] + w(u,v) then
            d[v] ← d[u] + w(u,v)
            π[v] ← u
```

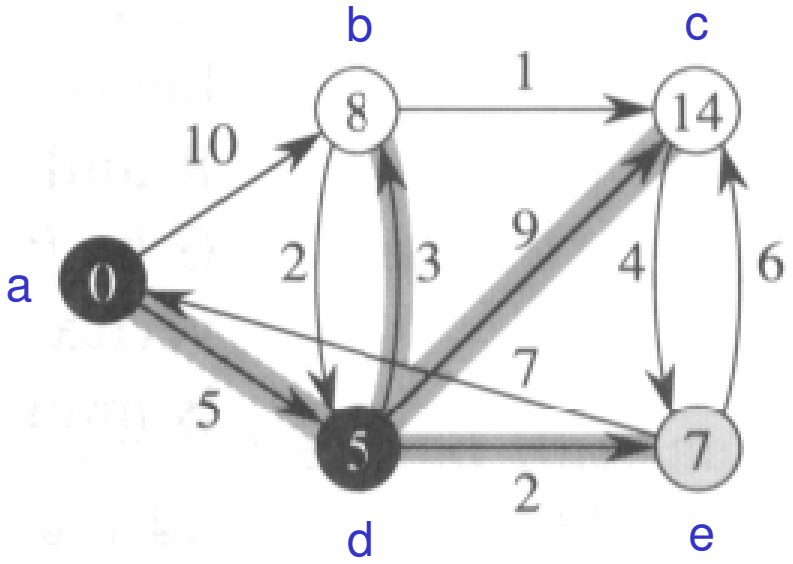
BUILD-MIN-HEAP

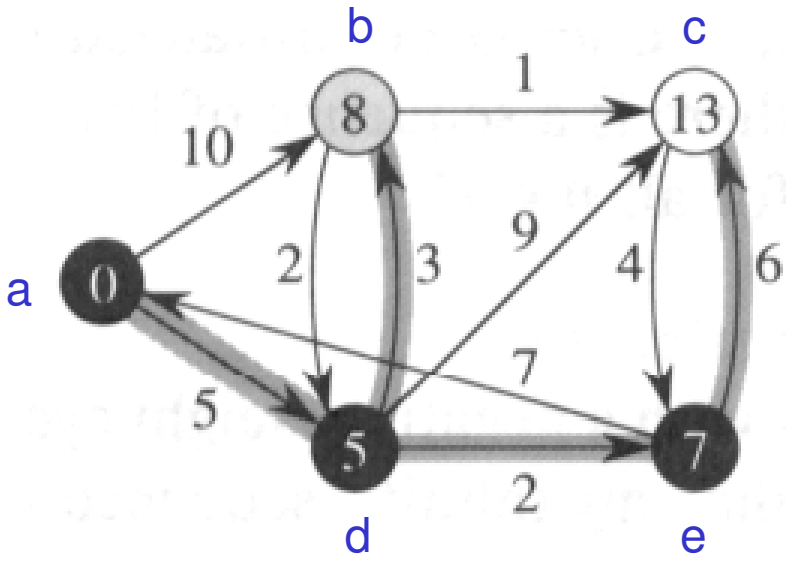
DECREASE-KEY

Algoritmo de Dijkstra









Algoritmo de Dijkstra