Grafos – Parte 2

SCC-603
Algoritmos e Estruturas de Dados II

Profª. Rosane Minghim / 2012

Baseado em material de professores dos anos anteriores

Percorrendo um grafo

Percorrendo um Grafo

- Percorrer um grafo é um problema fundamental
- Deve-se ter uma forma sistemática de visitar as arestas e os vértices
- O algoritmo deve ser suficientemente flexível para adequar-se à diversidade de grafos

Eficiência

Percorrendo um Grafo

- Eficiência
  - Não deve haver repetições (desnecessárias) de visitas a um vértice e/ou aresta (apenas duas visitas a cada aresta)

Correção

Percorrendo um Grafo

- Correção
  - Todas os vértices e/ou arestas devem ser visitados

Solução

Percorrendo um Grafo

- Solução
  - Marcar os vértices com...
    - não visitados
    - visitados
    - processados

Solução

Percorrendo um Grafo

- Solução
  - Manter uma lista de vértices no estado '*visitados*'
  - Há duas possibilidades de implementação:
    - Fila
    - Pilha

Travessia em Largura – BFS

Percorrendo um Grafo

- *BFS – Breadth-First Search*
  - Em grafos não-dirigidos cada aresta é visitada duas vezes
  - Em grafos dirigidos cada aresta é visitada uma única vez

BFS

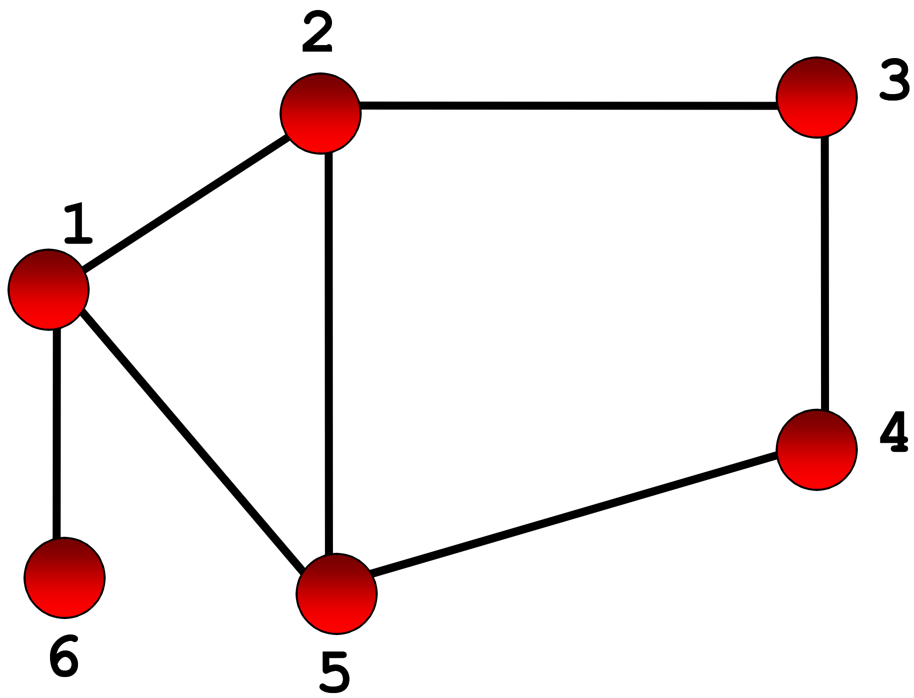
```
{ Percorre um grafo G a partir de um vértice
  inicial s informado. Pode realizar processa-
  mento à medida que visita vértices e arestas }
```

"Descobre" todos os vértices alcançáveis a partir de *s*;
Calcula a distância de *s* a cada vértice alcançável;
Gera uma árvore em largura com raiz em *s* com todos os vértices alcancáveis *v*, tal que o caminho na árvore corresponde ao menor caminho entre *s* e *v*.

```
BFS (G,s)
 for each vertex u ∈ V[G] – {s} do
     color[u] = “WHITE”
     d[u] = INF
     p[u] = NIL
 end-for
 color[s]= GRAY, d[s] = 0, p[s] = NIL
 initialize(Q)
 enqueue(Q,s)
 while (not empty(Q)) do
   u = dequeue[Q]
   processe o vértice u conforme desejado
   for each v ∈ Adj[u] do
      processe a aresta (u,v) conforme desejado
      if color[v] = “WHITE” then
         color[v] = “GRAY”
         d[v] = d[u] + 1
         p[v] = u
         enqueue(Q,v)
      end-if
   end-for
   color[u] = “BLACK”
 end-while
```

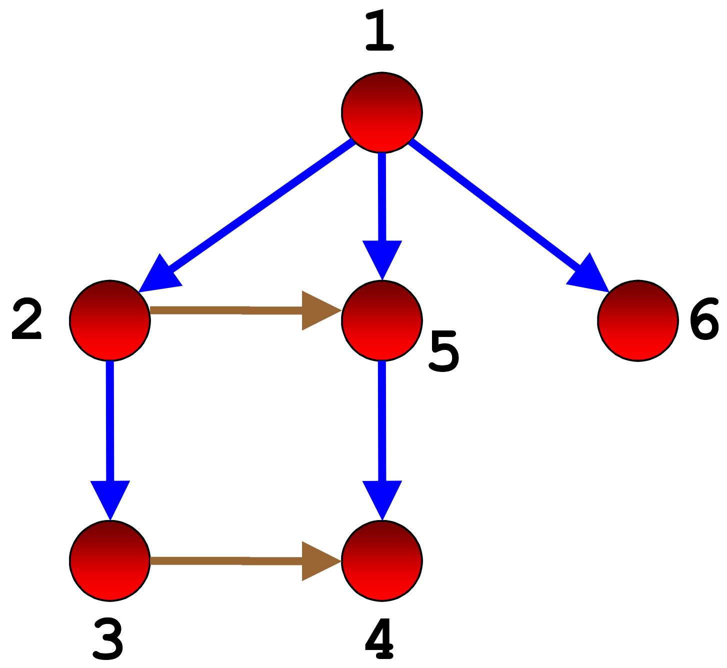
BFS – exemplo

Percorrendo um Grafo: BFS



BFS Tree

Percorrendo um Grafo: BFS Tree



Complexidade do BFS

$O(|V| + |E|)$, ou seja, linear em relação ao tamanho da representação de G por lista de adjacências:

- Todos os vértices são empilhados/desempilhados no máximo uma vez. O custo de cada uma dessas operações é $O(1)$, e elas são executadas $O(|V|)$ vezes.

- A lista de adjacências de cada vértice é percorrida no máximo uma vez (quando o vértice é desempilhado). O tempo total é $O(|E|)$ (soma dos comprimentos de todas as listas, igual ao número de arestas)

- Inicialização é $O(|V|)$

Travessia em Profundidade – DFS

Percorrendo um Grafo

- *DFS – Depth First Search*
  - Recursivo, eliminando assim a necessidade de uma estrutura de lista (fila ou pilha)

DFS

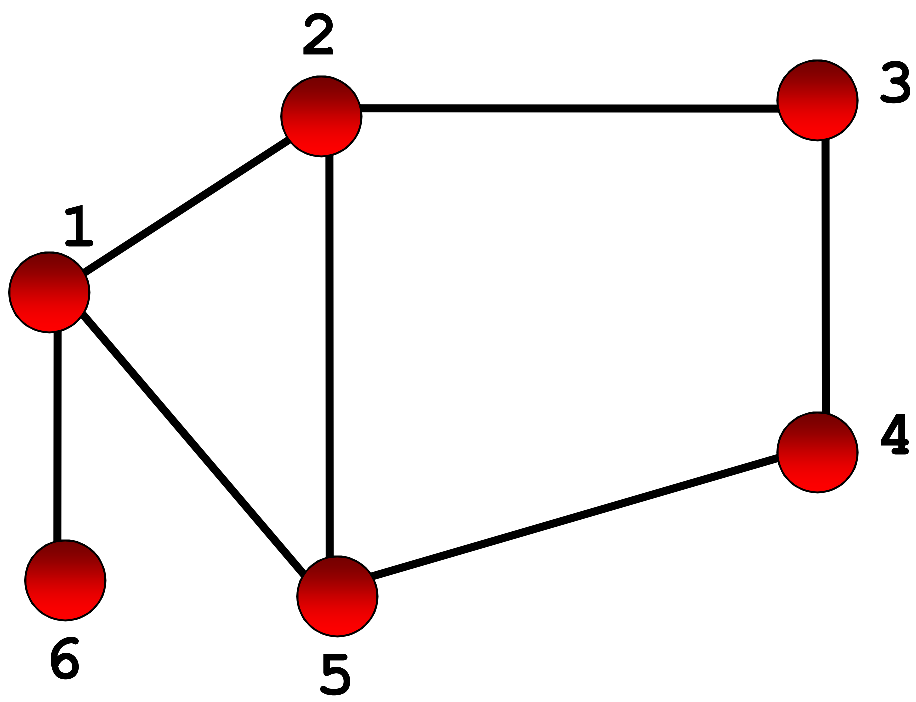
```
{ Percorre um grafo G. Pode realizar processa-
  mento à medida que visita vértices e arestas }

DFS-graph (G)
   for each vertex u ∈ V[G] do
       color[u] = "WHITE"
       p[u] = NIL
   end-for
   time = 0
   for each vertex u ∈ V[G] do
      if color[u] = "WHITE" then
         inicialize um novo componente
         DFS-visit(u)
      end-if
   end-for
```

DFS

```
DFS-visit(u)
   color[u] = "GRAY"
   time = time + 1
   d[u] = time
   processe o vértice u conforme desejado
   for each v ∈ Adj[u] do
        processe a aresta (u,v) conforme desejado
        if color[v] = "WHITE" then
           p[v] = u
           DFS-visit(v)
        end-if
   end-for
   color[u] = "BLACK"
   f[u] = time = time + 1
```

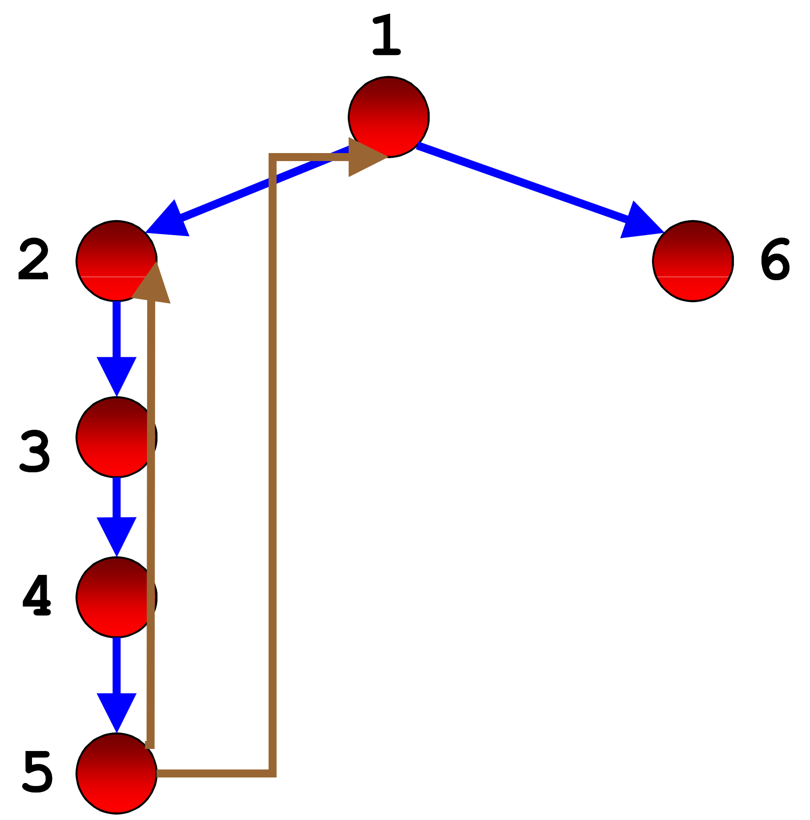
DFS

Percorrendo um Grafo: DFS



DFS Tree

Percorrendo um Grafo: DFS Tree



Complexidade do DFS

$O(|V| + |E|)$

- No algoritmo principal, cada for é $O(|V|)$. O *DFS-visit* é chamado exatamente uma vez para cada vértice de $|V|$ (na pior das hipóteses)

- No *DFS-visit*, o laço é executado $|\, adj[v] \,|$ vezes, i.e., $O(|E|)$ no total

DFS

- Uma aplicação clássica do DFS consiste em decompor um grafo direcionado (dígrafo) em componentes fortemente conexos.
- Um grafo direcionado é fortemente conexo se quaisquer dois vértices são mutuamente alcançáveis entre si.
- Um componente fortemente conexo de um grafo é um sub-conjunto maximal *C* de vértices de *V* tal que qualquer par de vértices de *C* é mutuamente alcançável.
- Algoritmo no Cormen, p. 554; ver também no Ziviani

Tarefas

1. Escrever uma versão não-recursiva do DFS
   - Dica: todo algoritmo recursivo usa uma pilha implicitamente
2. Escreva um algoritmo que verifique se um dado grafo G(V,E) é acíclico.
   - Dica: a solução é uma aplicação do algoritmo DFS. Se na busca em profundidade é encontrada uma aresta (u,v) □ E conectando um vértice u com um seu antecessor v na árvore de busca em profundidade, então o grafo tem ciclo. Igualmente, se G tem ciclo uma aresta desse tipo será encontrada em qualquer busca em profundidade em G
3. Escreva um algoritmo que determina as componentes fortemente conexas de um grafo direcionado G(V,E)
   - Dica: solução também aplica algoritmo de DFS

Caminhos mais curtos

- Em grafos não-orientados e não-valorados o algoritmo BFS(***u***) produz uma *árvore* de caminhos mais curtos entre ***u*** (origem) e todos os demais vértices do grafo alcançáveis a partir dele.

- Assim, o vetor antecessor [ ] é capaz de fornecer o caminho mais curto (menor número de arestas) entre *u* e *v*, para qualquer *v* em *V*, se ele existir.

Caminhos mais curtos (algoritmo)

- Dado o vetor antecessor após BFS(*v*):

```
Imprimir_caminho_mais_curto(origem, v:tipoVértice)
   Se origem = v escreve (origem)
     senão
         Imprimir_caminho_mais_curto(origem, antecessor(v))
         escreve(v)
      fim se
Fim Imprimir_caminho_mais_curto
```

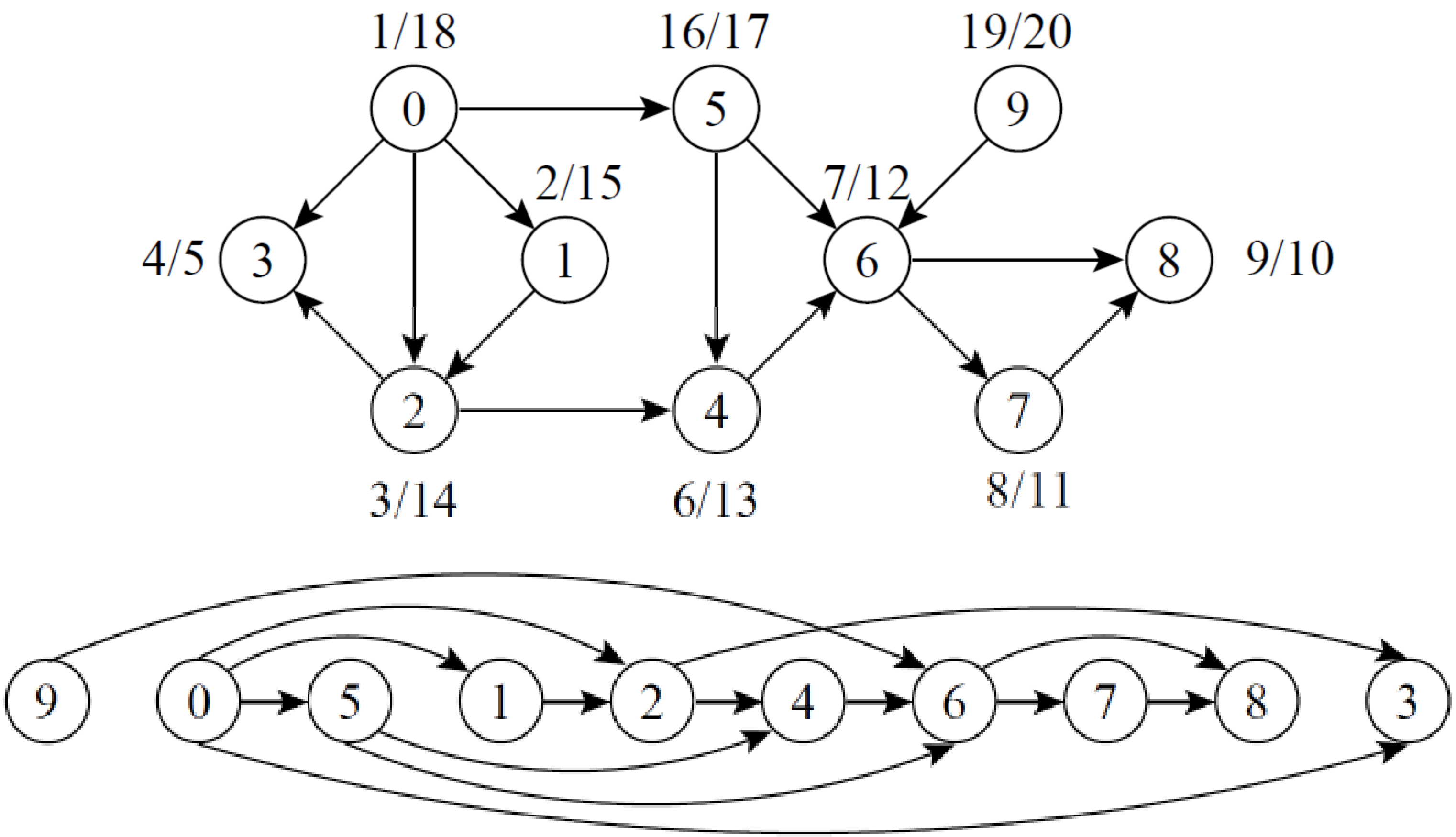
Obs: 'escreve' pode ser qualquer procedimento de armazenamento ou impressão do caminho.

Ordenação Topológica

- Define-se Ordenação Topológica para Grafos orientados acíclicos.

- O objetivo da ordenação topológica é alinhar todos os vértices de um grafo em sequência, de forma que se a aresta (u,v) pertence a V, então u está antes de v na sequência

Ordenação Topológica – Exemplo (Ziviani 2004)



Ordenação Topológica (algoritmo)

1. Chame DFS para todos os vértices do grafo *G* (isto é, enquanto existirem vértices ‘brancos’).
2. A cada vértice que é terminado (isto é, que se torna ‘preto’), insira-o na cabeça de uma lista encadeada.
3. Retorna a lista encadeada de vértices do grafo produzida no passo (2)

Ordenação Topológica (algoritmo)

- A implementação da ordenação topológica se dá adicionando um comando:

```
Insere_primeiro(u, L:lista)
```

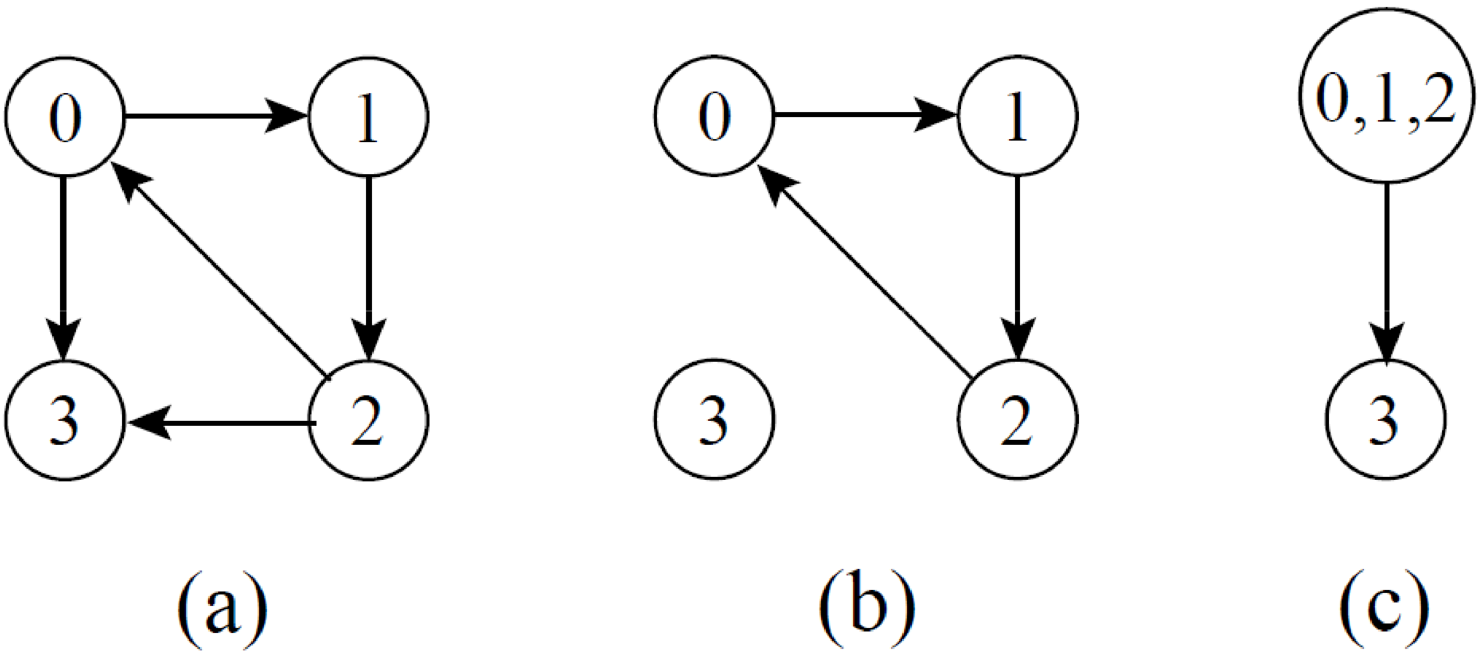
- Inserção na cabeça da lista *L*, na posição do algoritmo DFS logo após a determinação do tempo *t*[*u*] e da finalização do nó, isto é, após o momento em que ele se torna 'preto'.
  - Obs: naturalmente `Inicializa(L)` precisa ser chamada no início do algoritmo que chama DFS para todos os vértices 'brancos'.

Componentes Fortemente Conectados

- Define-se *componentes fortemente conectados* para um grafo orientado.

- Um *Componente Fortemente Conectado* (ou *Fortemente Conexo*) *C* de um grafo *G* é um conjunto de vértices maximal de *G* de forma que para todos os vértices *u* e *v* em *C, u* é alcançável a partir de *v* e *v* é alcançável a partir de *u*.

Componentes Fortemente Conectados (Exemplo – Ziviani 2004)



(a) Grafo original

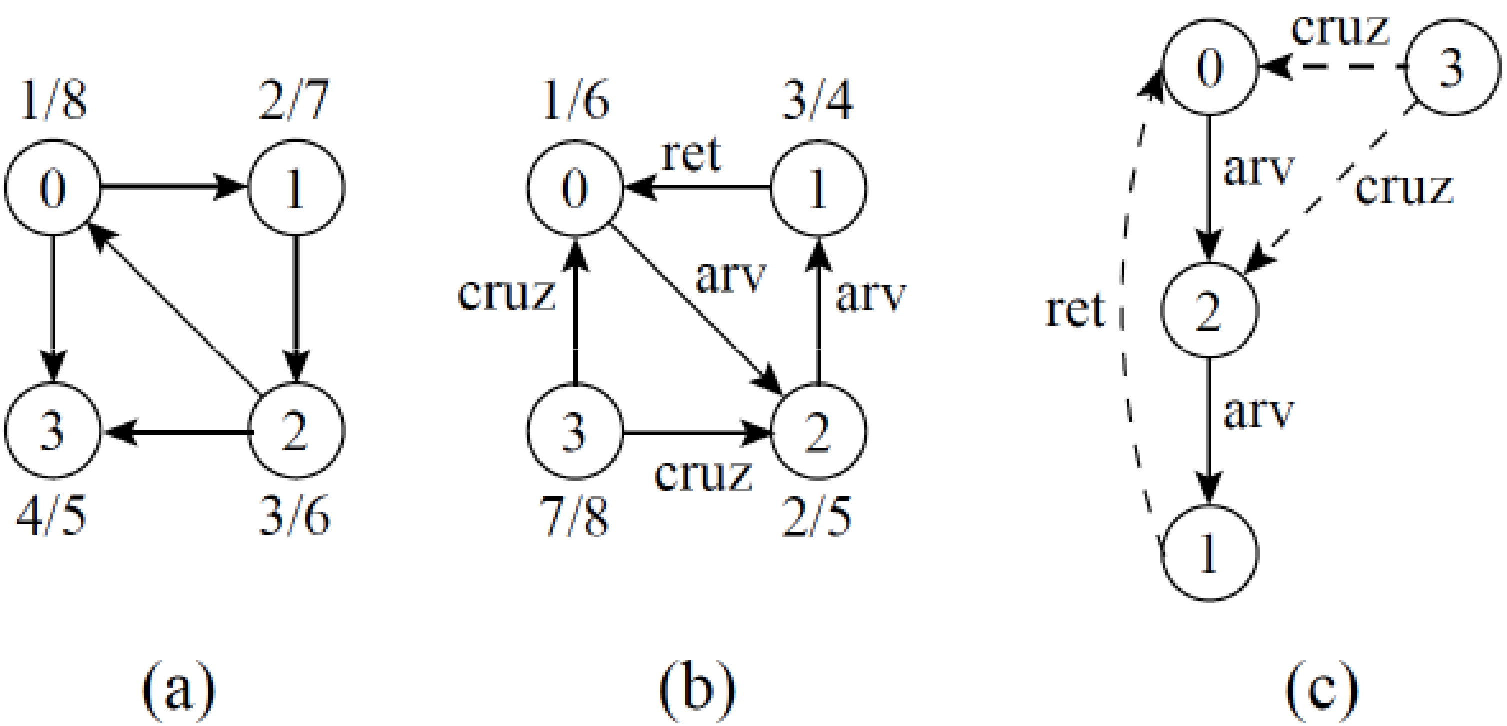
(b) Componentes Conexas

(c) Colapso dos vértices das componentes

Componentes Fortemente Conectados (algoritmo)

1. Chama DFS(G) para obter os tempos de término $t[u]$ para todos os vértices de G, isto é, enquanto existirem vértices 'brancos' em G.
2. Obtém G^T.
3. Chama DFS(G^T) em ordem decrescente de $t[u]$ obtido no passo (1), enquanto existirem vértices u 'brancos' em G^T.
4. Retorne todas as árvores obtidas no passo (3).

Componentes Fortemente Conectados (Exemplo – Ziviani 2004)



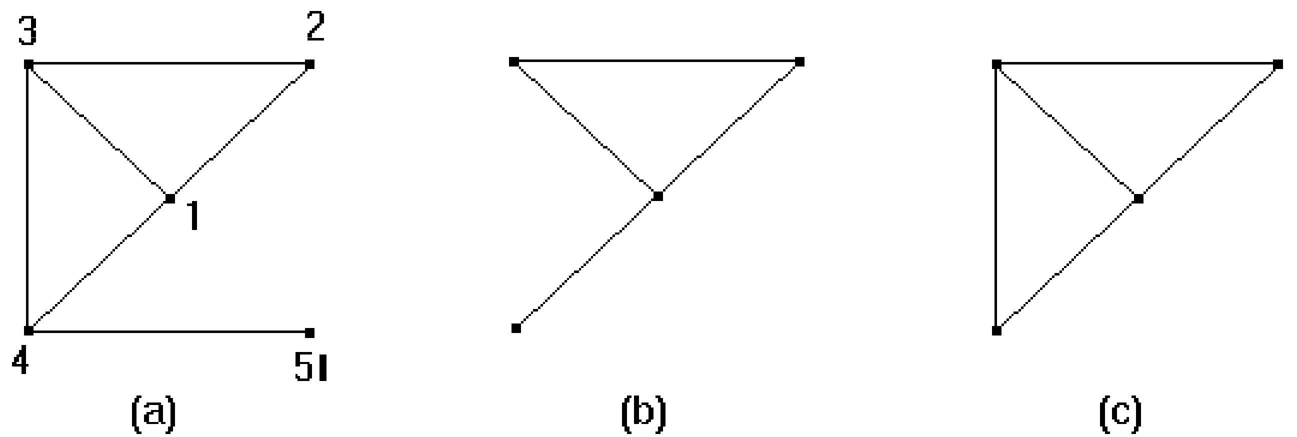
(a) Grafo original com resultado da BFs

(b) Grafo transposto com resultado da BFs

(c) árvores encontradas

Sub-grafo

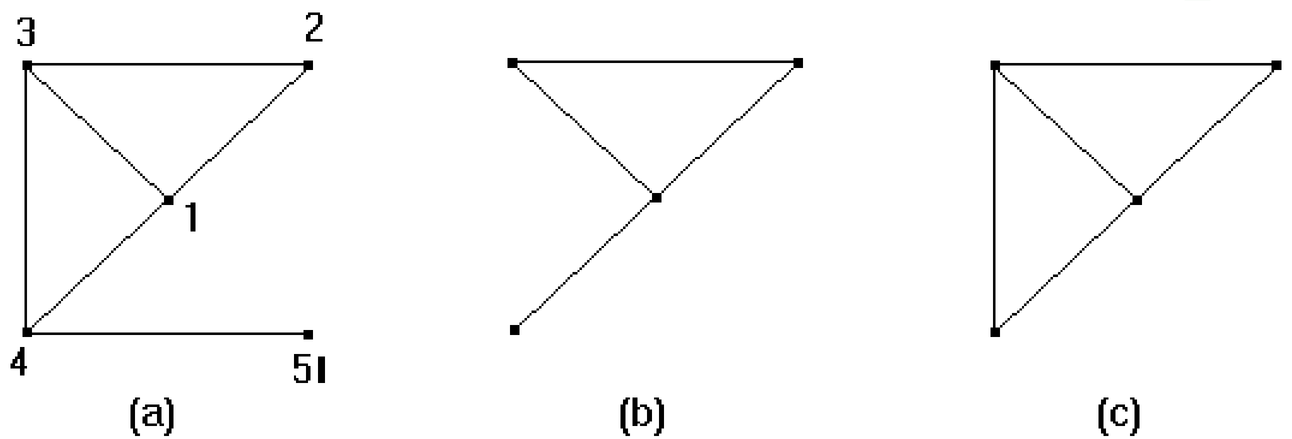
Um sub-grafo $G_2(V_2,E_2)$ de um grafo $G_1(V_1,E_1)$ é um grafo tal que V_2 está contido em V_1 e E_2 está contido em E_1



b e c são subgrafos de a

Sub-grafo induzido

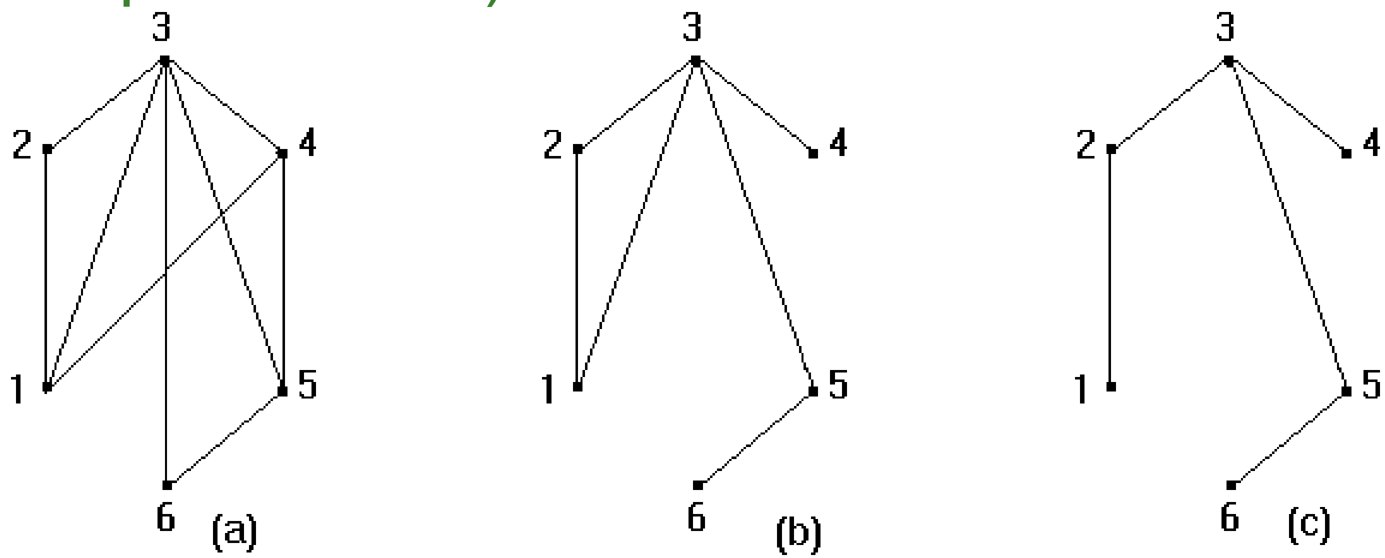
Se o sub-grafo G_2 de G_1 satisfaz: para quaisquer ***v***, ***w*** pertencentes a $\mathbf{V_2}$, se **(*v*,*w*)** pertence a $\mathbf{E_1}$, então **(*v*,*w*)** também pertence a $\mathbf{E_2}$. Dessa forma, $\mathbf{G_2}$ é dito sub-grafo induzido pelo conjunto de vértices $\mathbf{V_2}$



b e c são sub-grafos de a, <u>mas apenas</u> c é sub-grafo induzido

Sub-grafo gerador

Sub-grafo Gerador ou sub-grafo de espalhamento de um grafo $G_1(V_1,E_1)$ é um sub-grafo $G_2(V_2,E_2)$ de G_1 tal que $V_1=V_2$. Quando o sub-grafo gerador é uma árvore, ele recebe o nome de árvore geradora (ou de espalhamento).



b e c são sub-grafos geradores de a; c é árvore geradora de a e b

Sub-grafo gerador de custo mínimo

- Formalmente...
- Dado um grafo não-orientado *G*(*V*,*E*)
  - onde *w*: E$\rightarrow\Re^+$ define os custos das arestas
  - queremos encontrar um sub-grafo gerador conexo *T* de *G* tal que, para todo sub-grafo gerador conexo *T'* de *G*

$$\sum_{e \in T} w(e) \leq \sum_{e \in T'} w(e)$$

Árvore geradora mínima (MST)

- Claramente, o problema só tem solução se *G* é conexo
- A partir de agora, assumimos *G* conexo
- Também não é difícil ver que a solução para esse problema será sempre uma árvore
  - Basta notar que *T* não terá ciclos pois, poderíamos obter um outro sub-grafo *T'*, ainda conexo e com custo menor que o de *T*, removendo o ciclo!

Árvore geradora mínima

- Árvore Geradora (*Spanning Tree*) de um grafo *G* é um sub-grafo de *G* que contém todos os seus vértices e, ainda, é uma árvore

- Árvore Geradora Mínima (*Minimum Spanning Tree* – MST) é a árvore geradora de um grafo valorado cuja soma dos pesos associados às arestas é mínimo, i.e., é uma árvore geradora de custo mínimo

Porque é um problema interessante?

- Suponha que queremos construir estradas para interligar *n* cidades
  - Cada estrada direta entre as cidades *i* e *j* tem um custo associado
  - Nem todas as cidades precisam ser ligadas diretamente, desde que todas sejam acessíveis
- Como determinar eficientemente quais estradas devem ser construídas de forma a minimizar o custo total de interligação das cidades?

Árvore geradora mínima (MST)

Como encontrar a árvore geradora mínima de um grafo *G*?

- Algoritmo Genérico
- Algoritmo de Prim
- Algoritmo de Kruskal

Árvore geradora mínima

Algoritmo Genérico

```
Generic-MST(G)
A = ∅
While A não define uma spanning tree
   encontre uma aresta (u,v) segura para A
   A = A ∪ {(u,v)}
Return A
```

- A = conjunto de arestas
- G conexo, não direcionado, ponderado
- Abordagem 'gulosa', MST cresce uma aresta por vez
- Aresta é 'segura' se mantém a condição: antes de cada iteração, A é um sub-conjunto de alguma MST

Algoritmo de Prim

```
{ Gera uma Minimum Spanning Tree do
  grafo ponderado G - Algoritmo de Prim}

Prim-MST (G)
   Escolha um vértice s para iniciar a árvore
   enquanto "Há vértices que não estão na árvore"
      Selecione a aresta com menor peso adjacente
      a um vértice pertencente à árvore e a outro
      não pertencente à árvore

      Insira a aresta selecionada e o respectivo
      vértice na árvore
   fim-enquanto
```

Algoritmo de Prim

Inicia em um determinado vértice e gera a árvore, uma aresta por vez

- Complexidade (tempo): $O(n.m)$

  n: número de vértices

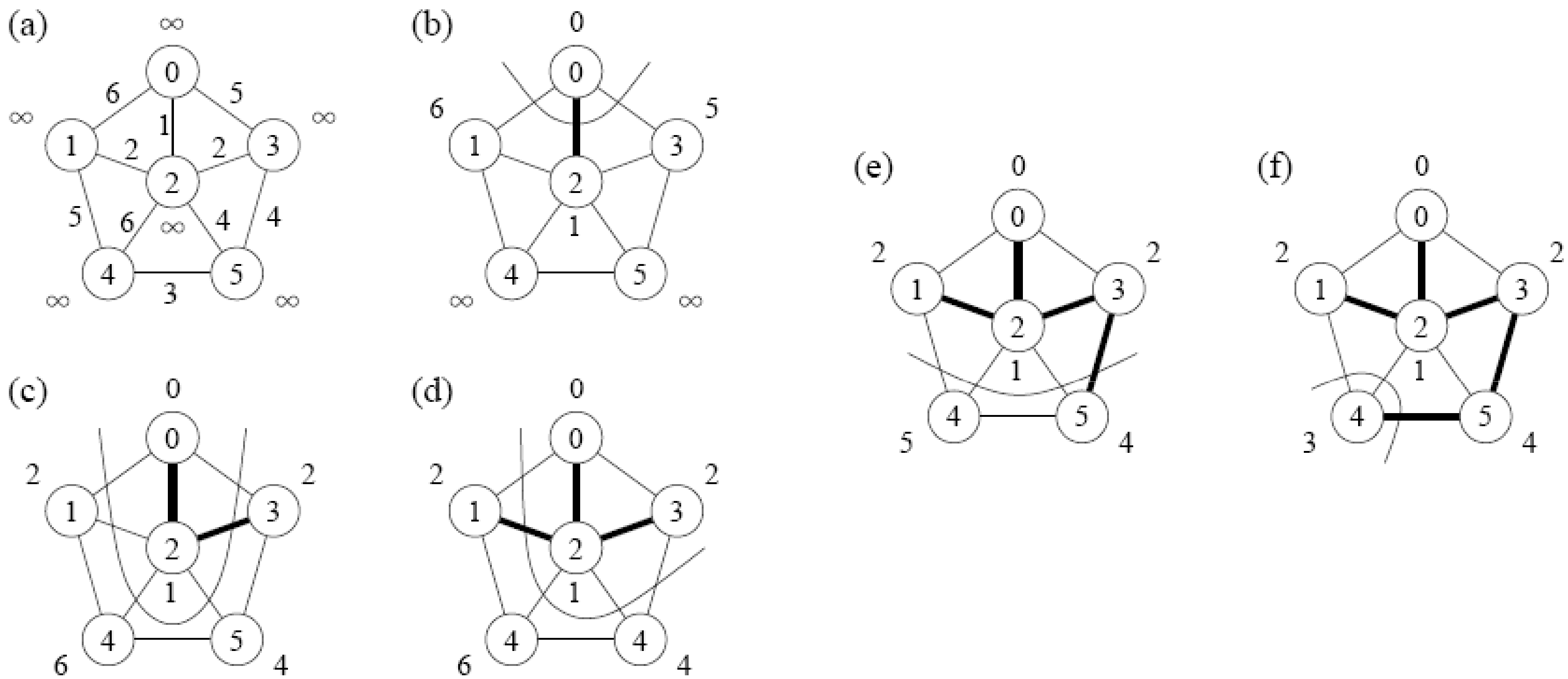
  m: número de arestas

Algoritmo de Prim

- Maneira mais eficiente de determinar a aresta de menor peso a partir de um dado vértice
  - manter todas as arestas que ainda não estão na árvore em uma fila de prioridade (*heap*)
  - prioridade é dada à aresta de menor peso adjacente a um vértice na árvore e outro fora dela

Complexidade (tempo): $O(m.\log(n))$

Algoritmo de Prim



Algoritmo de Prim
s:origem

```
Inicialize a fila de Prioridades fp com todos o nó s
Inicializa  peso(v) como INFINITO para todo v, exceto s
Inicializa  peso(s) como 0
Inicialize antecessor(v) como -1 para todo v
Enquanto não vazia (fp)
   v <- primeiro (fp)
   elimina (v,fp)
   Enquanto u <- prox_adj(v) não nulo
              Se na_fila(fp,v) e w(u,v) <= peso(v) então
                 antecessor[v] = u
                 peso(v) = w(u,v)
              fim-se
   fim-enquanto
fim-enquanto
```

Prim

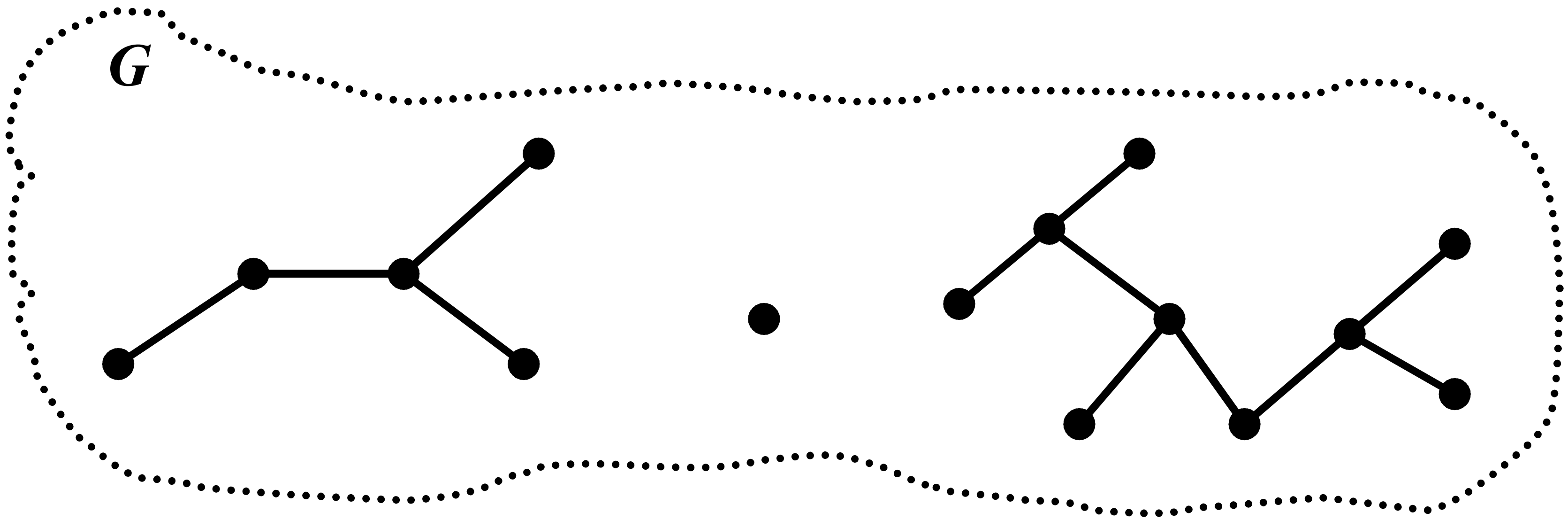
Implementação Ziviani

```
begin { AgmPrim }
  for u := 0 to Grafo.NumVertices do
    begin {Constroi o heap com todos os valores igual a Infinito}
    Antecessor[u] := -1;  p[u] := Infinito;
    A[u+1].Chave := u; {Heap a ser construido}
    ItensHeap[u] := true;  Pos[u] := u+1;
    end;
  n := Grafo.NumVertices;
  p[Raiz] := 0;
  Constroi(A);
  while n >= 1 do {enquanto heap nao vazio}
    begin
    u := RetiraMin(A).Chave;
    if (u <> Raiz)
    then write('Aresta de arvore: v[',u,'] v[',Antecessor[u],']');readln;
    ItensHeap[u] := false;
    if not ListaAdjVazia(u,Grafo)
      then begin
           Aux := PrimeiroListaAdj(u,Grafo);  FimListaAdj := false;
           while not FimListaAdj do
             begin
             ProxAdj(u, Grafo, v, Peso, Aux, FimListaAdj);
             if ItensHeap[v] and (Peso < p[v])
             then begin
                  Antecessor[v] := u; DiminuiChave(Pos[v],Peso,A);
                  end
             end;
           end;
    end;
end; { AgmPrim }
```

Kruskal

Floresta

- Uma Floresta é um conjunto de árvores.



Algoritmo de Kruskal

- Mais eficiente que Prim em grafos esparsos
- Não inicia em nenhum vértice em particular
  - Considera se cada aresta individualmente pode ou não pertencer à árvore geradora mínima, analisando-as em ordem crescente de custo
  - As árvores que compõem a floresta são identificadas pelos conjuntos S_i, que contém os vértices que a compõem
  - Ao final do processo, o conjunto E_T contém a solução do problema, i.e., a MST
  - Complexidade: O(m.log(m))
    - Se o teste $\mathbf{S_p} \cap \mathbf{S_q} = \emptyset$ for bem implementado; esse teste garante que a inclusão de e em E_T não introduz um ciclo

Algoritmo de Kruskal

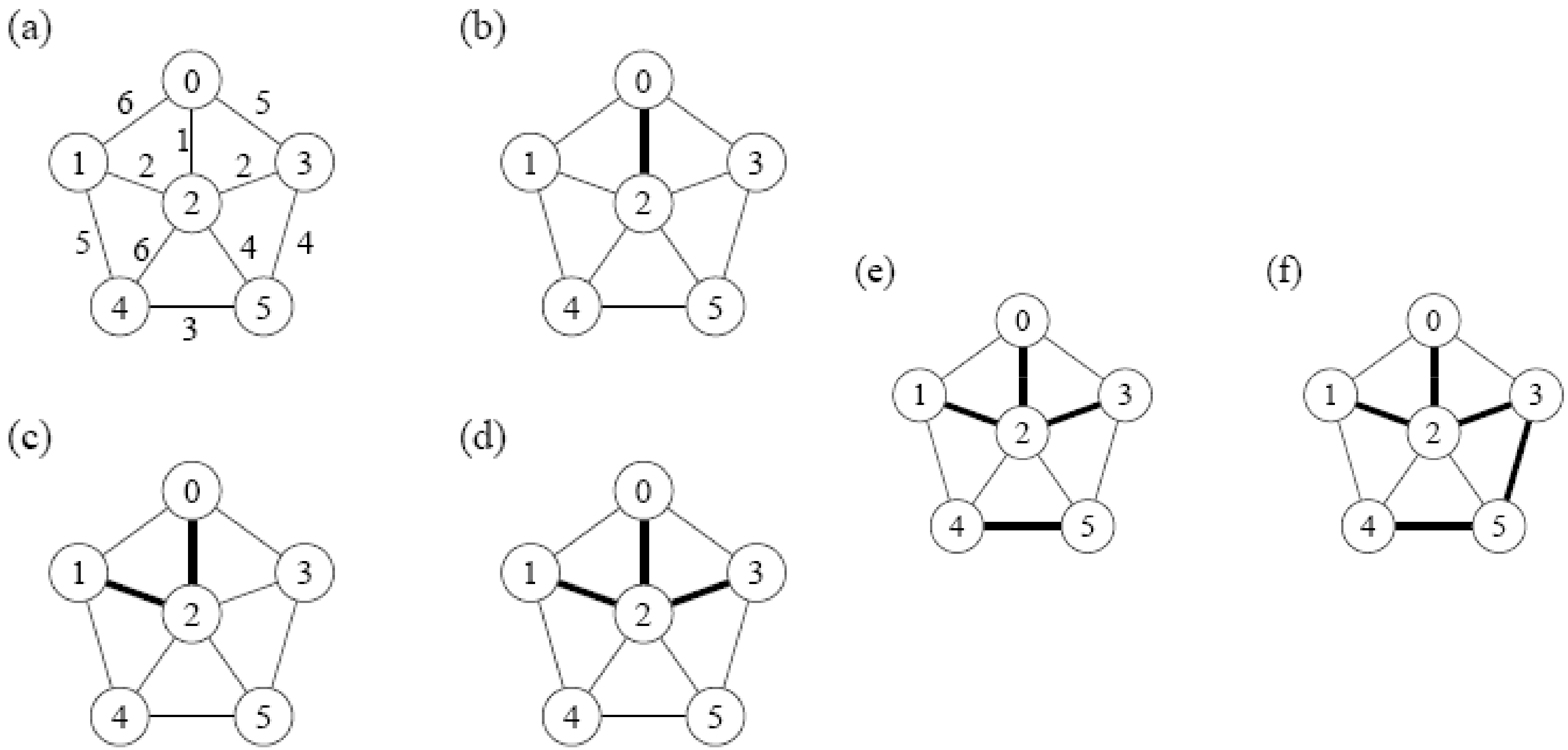
Basicamente, o algoritmo consiste em:

"Incluir em E_T todas as arestas de E em ordem crescente de peso, rejeitando, contudo, cada uma que forma ciclos com as arestas já em E_T."

- Pode ser interpretado como sendo a construção de uma árvore geradora a partir de uma floresta.

- Estado inicial: corresponde à floresta formada por *n* árvores triviais (um só vértice cada), i.e., $E_T = \emptyset$

Algoritmo de Kruskal – exemplo Ziviani



Algoritmo de Kruskal - Ziviani

- Sejam C1 e C2 duas árvores conectadas por (u, v):
  - Como (u, v) tem de ser uma aresta leve conectando C1 com alguma outra árvore, (u, v) é uma aresta segura para C1.
  - É guloso porque, a cada passo, ele adiciona à floresta uma aresta de menor peso.
  - Obtém uma AGM adicionando uma aresta de cada vez à floresta e, a cada passo, usa a aresta de menor peso que não forma ciclo.
  - Inicia com uma floresta de |V| árvores de um vértice: em |V| passos, une duas árvores até que exista apenas uma árvore na floresta.

Algoritmo de Kruskal - Ziviani

- Usa fila de prioridades para obter arestas em ordem crescente de pesos.

- Testa se uma dada aresta adicionada ao conjunto solução *S* forma um ciclo.

- Tratar **conjuntos disjuntos**: maneira eficiente de verificar se uma dada aresta forma um ciclo. Utiliza estruturas dinâmicas. Sempre unindo árvores disjuntas, árvores são obtidas.

Algoritmo de Kruskal

```
{ Gera uma Minimum Spanning Tree do grafo
ponderado G(V,E), conexo - Algoritmo de
Kruskal}
Kruskal-MST (G)

  Definir conjuntos Sj:{vj}, 1 ≤ j ≤ n, e ET = ∅
  Insira as arestas de E em uma fila de
prioridade Q, segundo o peso (ordem crescente)
  Enquanto houver arestas na fila faça
     e = unqueue(Q)
     Seja (v,w) o par de vértices extremos de e
     Se v ∈ Sp e w ∈ Sq, Sp ∩ Sq = ∅ então
         Sp = Sp ∪ {Sq}
         eliminar Sq
         ET = ET ∪ {e}
  Fim Enquanto
```

Caminho mínimo

- **Problema: encontrar o caminho de menor custo (ou o menor caminho) entre dois vértices em um grafo valorado**
  - ***Algoritmo de Djikstra***
  - ***Algoritmo de Floyd-Warshall***

Caminho mínimo

- Grafo dirigido *G*(*V*,*E*) com função peso *w*: E$\rightarrow\Re$ que mapeia as arestas em pesos
  - Peso (custo) do caminho p = <v_0, v_1, ..., v_k>

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

  - Caminho de menor peso entre *u* e *v*:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\Rightarrow} v\} \; se \, \exists \; rota \; de \; u \;\; p/v \\ \infty \; cc \end{cases}$$

Caminho mínimo

- Menor caminho entre os vértices *u* e *v* definido como qualquer rota *p* com um peso:

$$w(p) = \delta(u,v)$$

Dijkstra(L,v_p,v_q)

-T={v_p} ; PL={0}; P={0}; W=V-T=V-{v_p}   //inicialização
-TL={∞ qq w_i pertencente W}
-Enquanto $v_q \notin$ T OU W $\neq$ Ø
-Determine v_i tal que $v_i \in W$, $v_k \in T$ e $(v_k,v_i) \in A$
-Atribua a cada v_i um rótulo temporário igual a dist(v_p,v_i)
-Se existe mais de uma distância para v_i então
- rótulo temporário de v_i = min(PL(v_k)+l_{ki}), para todo $v_k \in T$
-Seja v o v_i com menor rótulo:
-Faça v vértice permanente transferindo-o de W para T
-Armazene em PL o rótulo de v (PL=PL+{TL(v)})
-Armazene em P o vértice antecessor de v (P=P+{v_k:(v_k,v) $\in$ A})
-TL={∞ qq w_i pertencente W} (lembre-se v$\notin$W)
-Fim do enquanto
-Se $v_q \in$ T então
-A distância do menor caminho de v_p a v_q é dada por PL(v_q)
-Para encontrar o menor caminho propriamente dito,
basta encontrar v_q em T (ele é o último).
A partir dele, encontre o vértice correspondente em P (chame-o v_m). Ache v_m em T. Prossiga achando correspondentes aos v_m em P e em T até chegar a v_p.
-Senão
-Não existe um caminho entre v_p e v_q
-Fim

- L: matriz de distâncias
- V: conjunto de vértices do dígrafo
- T: vetor com os vértices permanentes
- PL: vetor com os labels permanentes
- W: vetor com os vértices ainda não-permanentes
- TL: vetor com rótulos temporários
- P: vetor com vértices 'antecedentes'
- v_p vértice inicial
- v_q vértice final
- l_{ij} distância entre os vértices i e j

Floyd-Warshall

SE $i \neq j$ E $(i,j) \in A$ então $B_0[i,j]=C[i,j]$

SE $(i,j) \notin A$ então $B_0[i,j]=\infty$

SE $i=j$ então $B_0[i,j]=0$

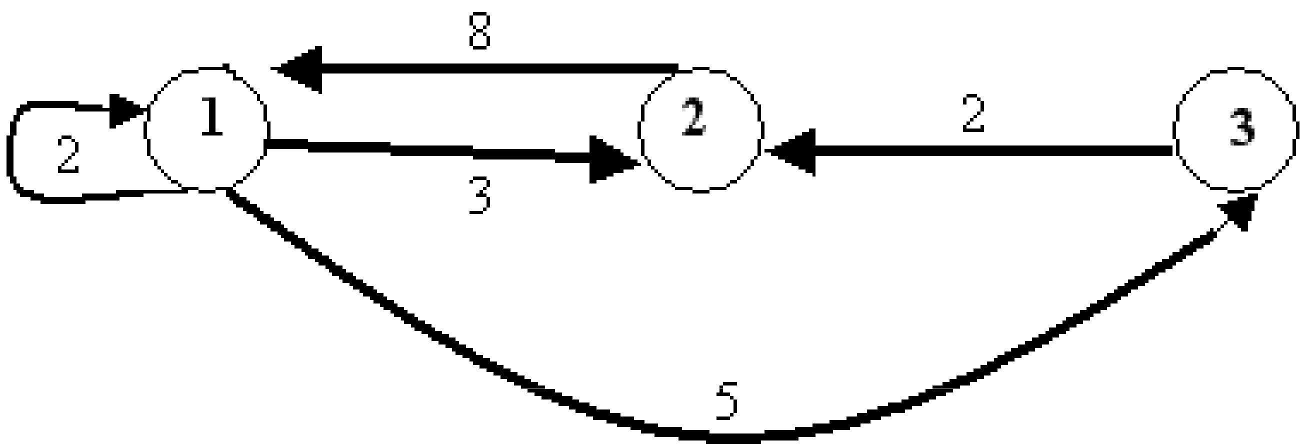
Para $k=1$ até N faça

$B_k[i,j]=\min(B_{k-1}[i,j], B_{k-i}[i,k]+B_{k-1}[k,j])$

B_n contém a distância dos caminhos mínimos de todos os pares de vértices

C[i,j]: custo para ir de i a j
A: conjunto de arestas do grafo
N: número de vértices do grafo

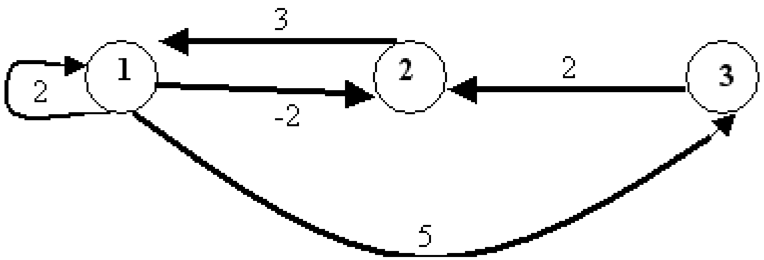
Exemplo de Floyd-Warshall



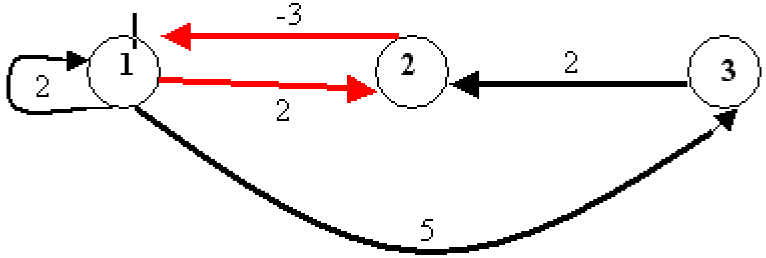
$$A_0 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \quad A_1 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \quad A_3 = \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Floyd-Warshall

- O algoritmo de Floyd-Warshall determina as distâncias dos menores caminhos entre todos os pares de vértices de um grafo
- Trabalha com arestas com pesos negativos
- Mas não funciona quando existem ciclos negativos no grafo



Ok! Grafo sem ciclo negativo



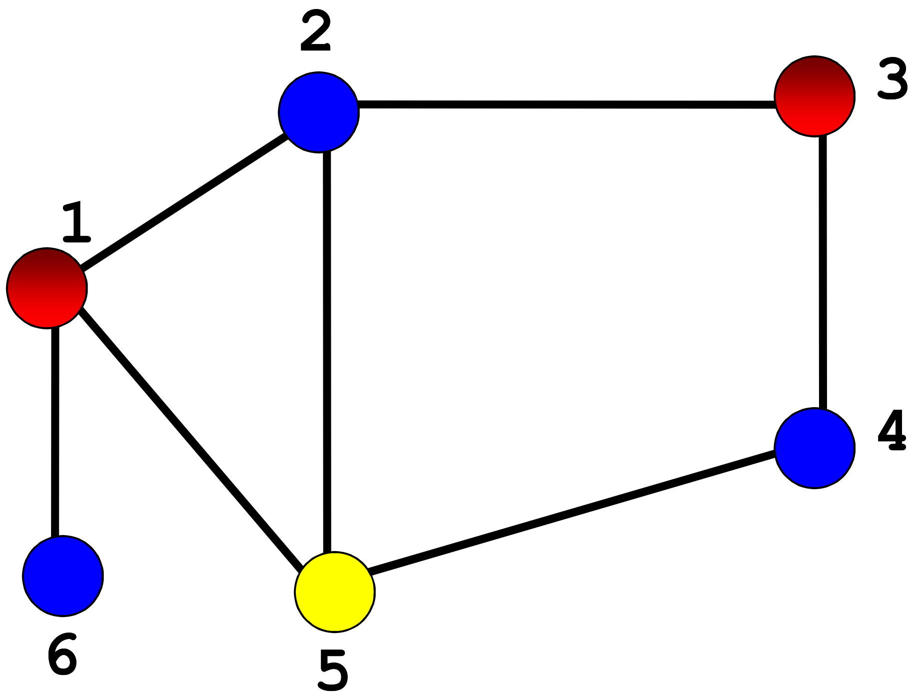
Nada feito. Grafo com ciclo negativo (arestas vermelhas)

Aplicações

- **Coloração de Grafos**
  - **Coloração de Vértices é a busca pela associação de uma cor para cada vértice de forma que:**
    - **nenhuma aresta ligue dois vértices de mesma cor**
    - **utiliza-se menor número possível de cores**

Aplicações

- **Aplicações: Coloração de Grafos**



Aplicações

- **Aplicações: Ordenação Topológica**

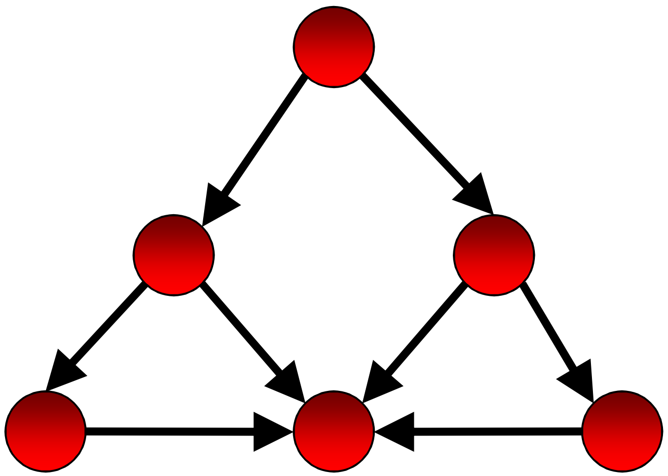
  - ***DAG – Directed-Acyclic Graphs*** **são mais complexos que as árvores**

    - **DFS pode ser utilizado para verificar se um grafo é um DAG**

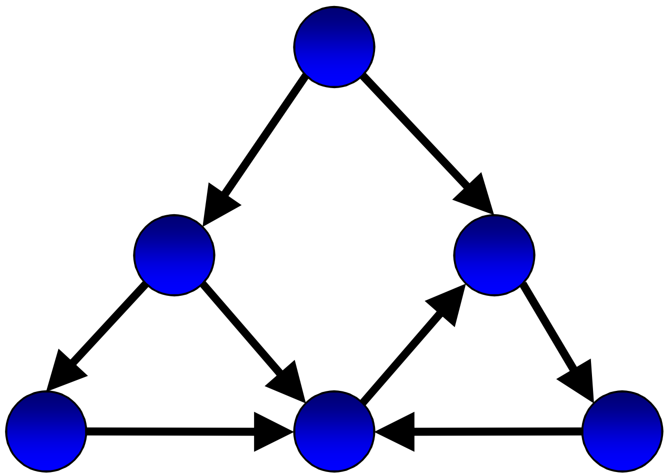
    - **Caso DFS não encontre nenhuma aresta de retorno durante o percurso, o grafo é um DAG**

Aplicações

- **Aplicações: Ordenação Topológica**



DAG

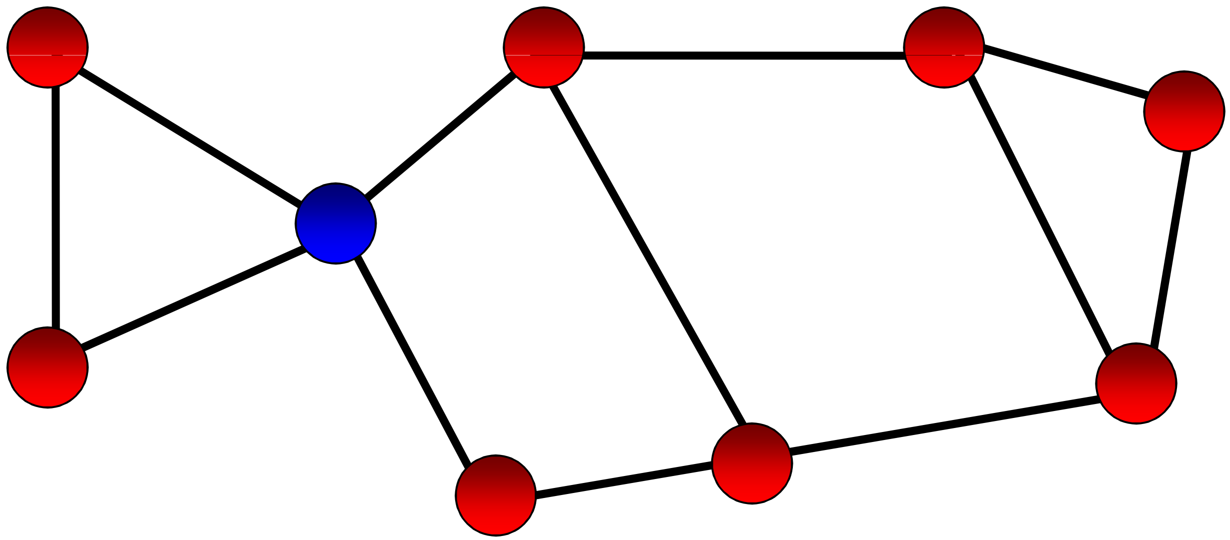


Aplicações

- **Aplicações: Vértices de Articulação**
  - Um **vértice de articulação** é um vértice cuja exclusão desconecta o grafo
  - Grafos com este tipo de vértice são frágeis
  - **Conectividade** de um grafo é o menor número de vértices cuja exclusão desconecta o grafo

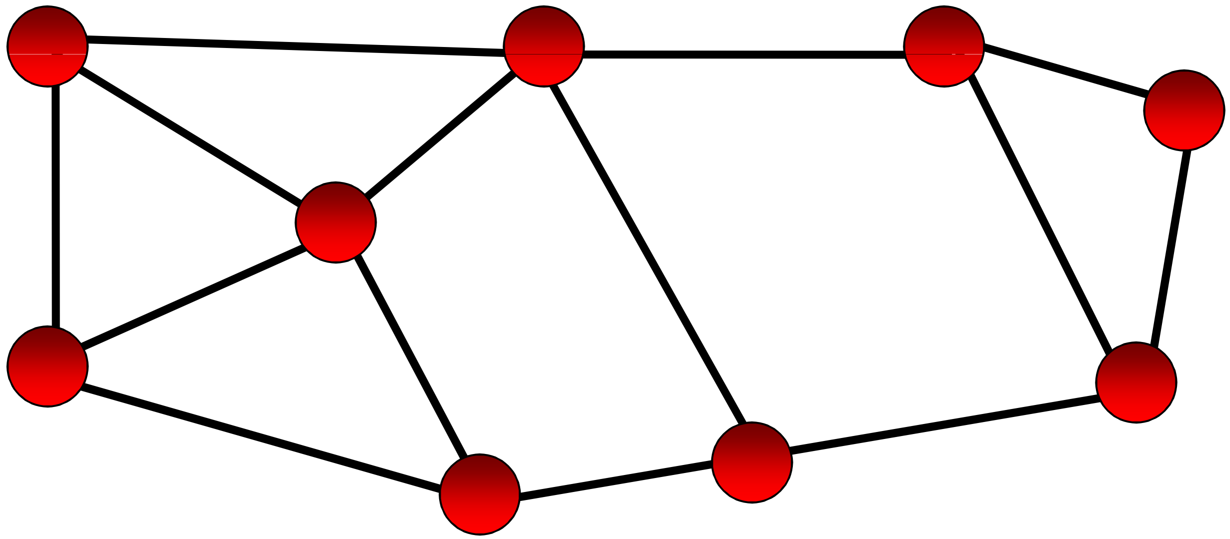
Aplicações

- **Aplicações: Vértices de Articulação**



Aplicações

- **Aplicações: Vértices de Articulação**



Modelando Problemas por Grafos

- **Caminhos em *videogames***
  - Procuro um algoritmo para descobrir rotas mínimas para personagens de um *videogame* movimentarem-se entre dois pontos do cenário.
  - Como poderia resolver isto?

Modelando Problemas por Grafos

- **DNA**
  - Na ordenação de fragmentos de DNA, para cada fragmento f, temos certos outros que são forçados a ligarem-se a f pelo seu lado direito, outros pelo seu lado esquerdo e ainda outros que podem se ligar a qualquer lado
  - Como encontrar uma ordenação consistente para todos os fragmentos?

Modelando Problemas por Grafos

- **DNA (cont.)**
  - Usando um grafo dirigido no qual cada fragmento é representado por um vértice e o algoritmo de ordenação topológica de grafos dirigidos

Modelando Problemas por Grafos

- **Organização em Grupos**
  - Dado um conjunto arbitrário de retângulos num plano, como posso organizá-los em um número mínimo de grupos, de forma que nenhum retângulo sobreponha-se a outro em seu respectivo grupo?

Modelando Problemas por Grafos

- **Organização em Grupos (cont.)**
  - Tratar como um problema de coloração de grafos, em que cada retângulo é representado por um vértice
  - Dois vértices são ligados por uma aresta se os retângulos correspondentes se sobrepõem