

# Algoritmos em Grafos\*

Última alteração: 26 de Abril de 2004

\*Transparências elaboradas por Charles Ornelas Almeida e Nivio Ziviani

## Motivação

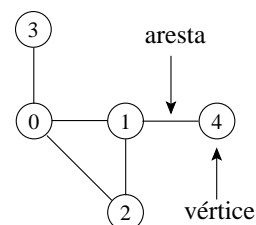
- Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
  - Existe um caminho para ir de um objeto a outro seguindo as conexões?
  - Qual é a menor distância entre um objeto e outro objeto?
  - Quantos outros objetos podem ser alcançados a partir de um determinado objeto?
- Existe um tipo abstrato chamado grafo que é usado para modelar tais situações.

## Aplicações

- Alguns exemplos de problemas práticos que podem ser resolvidos através de uma modelagem em grafos:
  - Ajudar máquinas de busca a localizar informação relevante na Web.
  - Descobrir os melhores casamentos entre posições disponíveis em empresas e pessoas que aplicaram para as posições de interesse.
  - Descobrir qual é o roteiro mais curto para visitar as principais cidades de uma região turística.

## Conceitos Básicos

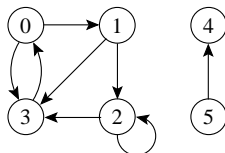
- **Grafo:** conjunto de vértices e arestas.
- **Vértice:** objeto simples que pode ter nome e outros atributos.
- **Aresta:** conexão entre dois vértices.



- **Notação:**  $G = (V, A)$ 
  - $G$ : grafo
  - $V$ : conjunto de vértices
  - $A$ : conjunto de arestas

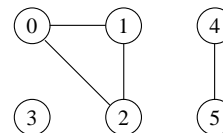
## Grafos Direcionados

- Um grafo direcionado  $G$  é um par  $(V, A)$ , onde  $V$  é um conjunto finito de vértices e  $A$  é uma relação binária em  $V$ .
  - Uma aresta  $(u, v)$  sai do vértice  $u$  e entra no vértice  $v$ . O vértice  $v$  é **adjacente** ao vértice  $u$ .
  - Podem existir arestas de um vértice para ele mesmo, chamadas de *self-loops*.



## Grafos Não Direcionados

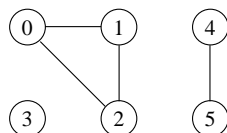
- Um grafo não direcionado  $G$  é um par  $(V, A)$ , onde o conjunto de arestas  $A$  é constituído de pares de vértices não ordenados.
  - As arestas  $(u, v)$  e  $(v, u)$  são consideradas como uma única aresta. A relação de adjacência é simétrica.
  - Self-loops* não são permitidos.



## Grau de um Vértice

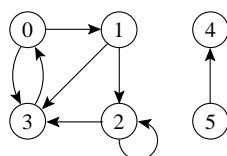
- Em grafos não direcionados:
  - O grau de um vértice é o número de arestas que incidem nele.
  - Um vértice de grau zero é dito **isolado** ou **não conectado**.

Ex.: O vértice 1 tem grau 2 e o vértice 3 é isolado.



- Em grafos direcionados
  - O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).

Ex.: O vértice 2 tem in-degree 2 out-degree 2 e grau 4



## Caminho entre Vértices

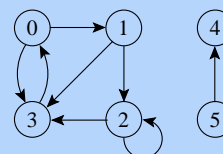
- Um caminho de **comprimento**  $k$  de um vértice  $x$  a um vértice  $y$  em um grafo  $G = (V, A)$  é uma sequência de vértices  $(v_0, v_1, v_2, \dots, v_k)$  tal que  $x = v_0$  e  $y = v_k$ , e  $(v_{i-1}, v_i) \in A$  para  $i = 1, 2, \dots, k$ .

- O comprimento de um caminho é o número de arestas nele, isto é, o caminho contém os vértices  $v_0, v_1, v_2, \dots, v_k$  e as arestas  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ .

- Se existir um caminho  $c$  de  $x$  a  $y$  então  $y$  é **alcançável** a partir de  $x$  via  $c$ .

- Um caminho é **simple** se todos os vértices do caminho são distintos.

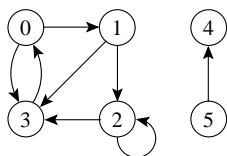
Ex.: O caminho  $(0, 1, 2, 3)$  é simple e tem comprimento 3. O caminho  $(1, 3, 0, 3)$  não é simple.



## Ciclos

- Em um grafo direcionado:
  - Um caminho  $(v_0, v_1, \dots, v_k)$  forma um ciclo se  $v_0 = v_k$  e o caminho contém pelo menos uma aresta.
  - O ciclo é simples se os vértices  $v_1, v_2, \dots, v_k$  são distintos.
  - O self-loop é um ciclo de tamanho 1.
  - Dois caminhos  $(v_0, v_1, \dots, v_k)$  e  $(v'_0, v'_1, \dots, v'_k)$  formam o mesmo ciclo se existir um inteiro  $j$  tal que  $v'_i = v_{(i+j) \bmod k}$  para  $i = 0, 1, \dots, k-1$ .

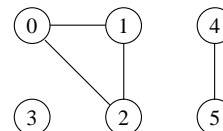
Ex.: O caminho  $(0, 1, 2, 3, 0)$  forma um ciclo. O caminho  $(0, 1, 3, 0)$  forma o mesmo ciclo que os caminhos  $(1, 3, 0, 1)$  e  $(3, 0, 1, 3)$ .



## Ciclos

- Em um grafo não direcionado:
  - Um caminho  $(v_0, v_1, \dots, v_k)$  forma um ciclo se  $v_0 = v_k$  e o caminho contém pelo menos três arestas.
  - O ciclo é simples se os vértices  $v_1, v_2, \dots, v_k$  são distintos.

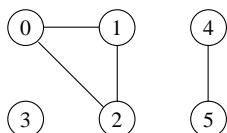
Ex.: O caminho  $(0, 1, 2, 0)$  é um ciclo.



## Componentes Conectados

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.
- Os componentes conectados são as porções conectadas de um grafo.
- Um grafo não direcionado é conectado se ele tem exatamente um componente conectado.

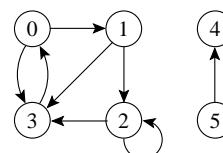
Ex.: Os componentes são:  $\{0, 1, 2\}$ ,  $\{4, 5\}$  e  $\{3\}$ .



## Componentes Fortemente Conectados

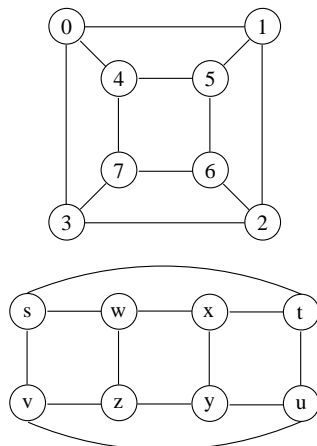
- Um grafo direcionado  $G = (V, A)$  é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado são conjuntos de vértices sob a relação “são mutuamente alcançáveis”.
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.

Ex.:  $\{0, 1, 2, 3\}$ ,  $\{4\}$  e  $\{5\}$  são os componentes fortemente conectados,  $\{4, 5\}$  não o é pois o vértice 5 não é alcançável a partir do vértice 4.



## Grafos Isomorfos

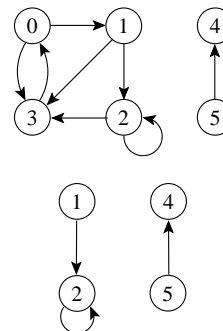
- $G = (V, A)$  e  $G' = (V', A')$  são isomorfos se existir uma bijeção  $f : V \rightarrow V'$  tal que  $(u, v) \in A$  se e somente se  $(f(u), f(v)) \in A'$ .



## Subgrafos

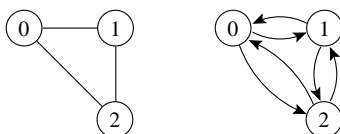
- Um grafo  $G' = (V', A')$  é um subgrafo de  $G = (V, A)$  se  $V' \subseteq V$  e  $A' \subseteq A$ .
- Dado um conjunto  $V' \subseteq V$ , o subgrafo induzido por  $V'$  é o grafo  $G' = (V', A')$ , onde  $A' = \{(u, v) \in A | u, v \in V'\}$ .

Ex.: Subgrafo induzido pelo conjunto de vértices  $\{1, 2, 4, 5\}$ .



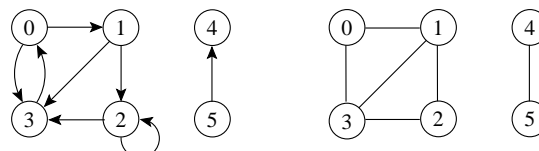
## Versão Direcionada de um Grafo Não Direcionado

- A versão direcionada de um grafo não direcionado  $G = (V, A)$  é um grafo direcionado  $G' = (V', A')$  onde  $(u, v) \in A'$  se e somente se  $(u, v) \in A$ .
- Cada aresta não direcionada  $(u, v)$  em  $G$  é substituída por duas arestas direcionadas  $(u, v)$  e  $(v, u)$ .
- Em um grafo direcionado, um **vizinho** de um vértice  $u$  é qualquer vértice adjacente a  $u$  na versão não direcionada de  $G$ .



## Versão Não Direcionada de um Grafo Direcionado

- A versão não direcionada de um grafo direcionado  $G = (V, A)$  é um grafo não direcionado  $G' = (V', A')$  onde  $(u, v) \in A'$  se e somente se  $u \neq v$  e  $(u, v) \in A$ .
- A versão não direcionada contém as arestas de  $G$  sem a direção e sem os *self-loops*.
- Em um grafo não direcionado,  $u$  e  $v$  são vizinhos se eles são adjacentes.



## Outras Classificações de Grafos

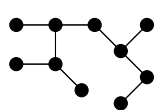
- **Grafo ponderado:** possui pesos associados às arestas.
- **Grafo bipartido:** grafo não direcionado  $G = (V, A)$  no qual  $V$  pode ser particionado em dois conjuntos  $V_1$  e  $V_2$  tal que  $(u, v) \in A$  implica que  $u \in V_1$  e  $v \in V_2$  ou  $u \in V_2$  e  $v \in V_1$  (todas as arestas ligam os dois conjuntos  $V_1$  e  $V_2$ ).
- **Hipergrafo:** grafo não direcionado em que cada aresta conecta um número arbitrário de vértices.

## Grafos Completos

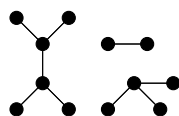
- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui  $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$  arestas, pois do total de  $|V|^2$  pares possíveis de vértices devemos subtrair  $|V|$  *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).
- O número total de **grafos diferentes** com  $|V|$  vértices é  $2^{|V|(|V|-1)/2}$  (número de maneiras diferentes de escolher um subconjunto a partir de  $|V|(|V| - 1)/2$  possíveis arestas).

## Árvores

- **Árvore livre:** grafo não direcionado acíclico e conectado. É comum dizer apenas que o grafo é uma árvore omitindo o “livre”.
- **Floresta:** grafo não direcionado acíclico, podendo ou não ser conectado.
- **Árvore geradora** de um grafo conectado  $G = (V, A)$ : subgrafo que contém todos os vértices de  $G$  e forma uma árvore.
- **Floresta geradora** de um grafo  $G = (V, A)$ : subgrafo que contém todos os vértices de  $G$  e forma uma floresta.



(a)



(b)

## O Tipo Abstratos de Dados Grafo

- Importante considerar os algoritmos em grafos como **tipos abstratos de dados**.
- Conjunto de operações associado a uma estrutura de dados.
- Independência de implementação para as operações.

## Operadores do TAD Grafo

1. *FGVazio(Grafo)*: Cria um grafo vazio.
2. *InseraAresta(V1,V2,Peso, Grafo)*: Insere uma aresta no grafo.
3. *ExisteAresta(V1,V2,Grafo)*: Verifica se existe uma determinada aresta.
4. Obtem a lista de vértices adjacentes a um determinado vértice (tratada a seguir).
5. *RetiraAresta(V1,V2,Peso, Grafo)*: Retira uma aresta do grafo.
6. *LiberaGrafo(Grafo)*: Liberar o espaço ocupado por um grafo.
7. *ImprimeGrafo(Grafo)*: Imprime um grafo.
8. *GrafoTransposto(Grafo,GrafoT)*: Obtém o transposto de um grafo direcionado.
9. *RetiraMin(A)*: Obtém a aresta de menor peso de um grafo com peso nas arestas.

## Implementação da Operação “Obter Lista de Adjacentes”

- É comum encontrar um pseudo comando do tipo:  
**for**  $u \in \text{ListaAdjacentes}(v)$  **do** { faz algo com  $u$  }
- O trecho de programa abaixo apresenta um possível refinamento do pseudo comando acima.

```

if not ListaAdjVazia(v, Grafo)
then begin
    Aux := PrimeiroListaAdj(v, Grafo);
    FimListaAdj := false;
    while not FimListaAdj
    do ProxAdj(v, Grafo, u, Peso, Aux, FimListaAdj);
end;
```

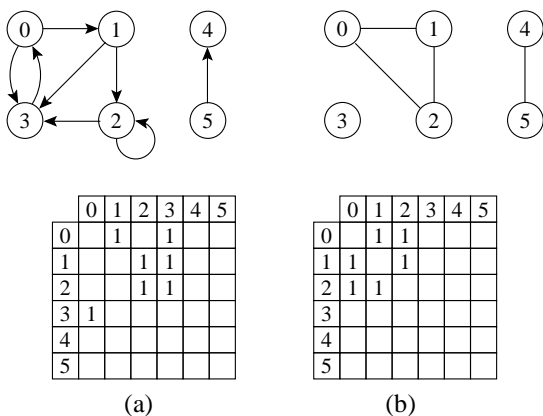
## Operação “Obter Lista de Adjacentes”

1. *ListaAdjVazia(v, Grafo)*: retorna *true* se a lista de adjacentes de  $v$  está vazia.
2. *PrimeiroListaAdj(v, Grafo)*: retorna o endereço do primeiro vértice na lista de adjacentes de  $v$ .
3. *ProxAdj(v, Grafo, u, Peso, Aux, FimListaAdj)*: retorna o vértice  $u$  (apontado por *Aux*) da lista de adjacentes de  $v$ , bem como o peso da aresta  $(v, u)$ . Ao retornar, *Aux* aponta para o próximo vértice da lista de adjacentes de  $v$ , e *FimListaAdj* retorna *true* se o final da lista de adjacentes foi encontrado.

## Matriz de Adjacência

- A matriz de adjacência de um grafo  $G = (V, A)$  contendo  $n$  vértices é uma matriz  $n \times n$  de *bits*, onde  $A[i, j]$  é 1 (ou verdadeiro) se e somente se existe um arco do vértice  $i$  para o vértice  $j$ .
- Para grafos ponderados  $A[i, j]$  contém o rótulo ou peso associado com a aresta  $e$ , neste caso, a matriz não é de *bits*.
- Se não existir uma aresta de  $i$  para  $j$  então é necessário utilizar um valor que não possa ser usado como rótulo ou peso.

## Matriz de Adjacência - Exemplo



## Matriz de Adjacência - Análise

- Deve ser utilizada para grafos **densos**, onde  $|A|$  é próximo de  $|V|^2$ .
- O tempo necessário para acessar um elemento é independente de  $|V|$  ou  $|A|$ .
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.
- A maior desvantagem é que a matriz necessita  $\Omega(|V|^2)$  de espaço. Ler ou examinar a matriz tem complexidade de tempo  $O(|V|^2)$ .

## Matriz de Adjacência - Implementação

- A inserção de um novo vértice ou retirada de um vértice já existente pode ser realizada com custo constante.

```

const MaxNumVertices = 100;
      MaxNumArestas   = 4500;

type
  TipoValorVertice = 0..MaxNumVertices;
  TipoPeso         = integer;
  TipoGrafo = record
    Mat: array[TipoValorVertice, TipoValorVertice]
          of TipoPeso;
    NumVertices: 0..MaxNumVertices;
    NumArestas : 0..MaxNumArestas;
  end;
  Apontador = TipoValorVertice;

procedure FGVazio(var Grafo: TipoGrafo);
var i, j: integer;
begin
  for i := 0 to Grafo.NumVertices do
    for j := 0 to Grafo.NumVertices do Grafo.mat[i, j] := 0;
  end;
end;
```

## Matriz de Adjacência - Implementação

```

procedure InsereAresta (var V1, V2: TipoValorVertice;
                        var Peso : TipoPeso;
                        var Grafo : TipoGrafo);
begin
  Grafo.Mat[V1, V2] := peso;
end;

function ExisteAresta (Vertice1, Vertice2: TipoValorVertice;
                      var Grafo: TipoGrafo): boolean;
begin
  ExisteAresta := Grafo.Mat[Vertice1, Vertice2] > 0;
end; { ExisteAresta }

{--- Operador para obter a lista de adjacentes---}
function ListaAdjVazia (var Vertice: TipoValorVertice;
                       var Grafo: TipoGrafo): boolean;
var Aux: Apontador; ListaVazia: boolean;
begin
  ListaVazia := true; Aux := 0;
  while (Aux < Grafo.NumVertices) and ListaVazia do
    if Grafo.Mat[Vertice, Aux] > 0
    then ListaVazia := false
    else Aux := Aux + 1;
  ListaAdjVazia := ListaVazia = true;
end; { ListaAdjVazia }
```

## Matriz de Adjacência - Implementação

{— Operador para obter a lista de adjacentes—}

```
function PrimeiroListaAdj (var Vertice: TipoValorVertice;
    var Grafo: TipoGrafo): Apontador;
var Aux: Apontador; Listavazia: boolean;
begin
    ListaVazia := true; Aux := 0;
    while (Aux < Grafo.NumVertices) and ListaVazia do
        if Grafo.Mat[Vertice, Aux] > 0
            then begin PrimeiroListaAdj := Aux; ListaVazia := false;
                end
            else Aux := Aux + 1;
    if Aux = Grafo.NumVertices
        then writeln('Erro: Lista adjacencia vazia (PrimeiroListaAdj)');
end; { PrimeiroListaAdj }
```

{— Operador para obter a lista de adjacentes—}

```
procedure ProxAdj (var Vertice: TipoValorVertice; var Grafo: TipoGrafo;
    var Adj: TipoValorVertice; var Peso: TipoPeso;
    var Prox: Apontador; var FimListaAdj : boolean);
{—Retorna Adj apontado por Prox—}
begin
    Adj := Prox; Peso := Grafo.Mat[Vertice, Prox]; Prox := Prox + 1;
    while (Prox < Grafo.NumVertices) and
        (Grafo.Mat[Vertice, Prox] = 0) do Prox := Prox + 1;
    if Prox = Grafo.NumVertices then FimListaAdj := true;
end; { ProxAdj }
```

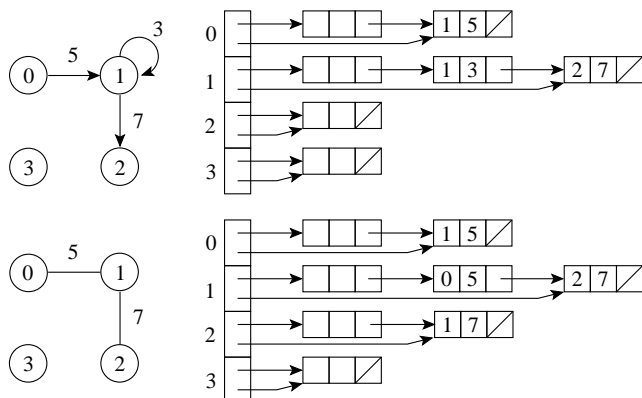
## Matriz de Adjacência - Implementação

```
procedure RetiraAresta (var V1, V2: TipoValorVertice;
    var Peso : TipoPeso;
    var Grafo : TipoGrafo);
begin
    if Grafo.Mat[V1, V2] = 0
        then writeln('Aresta nao existe')
        else begin Peso := Grafo.Mat[V1, V2]; Grafo.Mat[V1, V2] := 0;
            end;
end; { RetiraAresta }
```

```
procedure LiberaGrafo (var Grafo: TipoGrafo);
begin { Não faz nada no caso de matrizes de adjacencia }
end; { LiberaGrafo }
```

```
procedure ImprimeGrafo (var Grafo : TipoGrafo);
var i, j: integer;
begin
    write(' ');
    for i := 0 to Grafo.NumVertices-1 do write(i:3); writeln;
    for i := 0 to Grafo.NumVertices-1 do
        begin
            write(i:3);
            for j := 0 to Grafo.NumVertices-1 do write(Grafo.mat[i,j]:3);
                writeln;
            end;
        end;
end; { ImprimeGrafo }
```

## Listas de Adjacência usando Apontadores



- Um arranjo  $Adj$  de  $|V|$  listas, uma para cada vértice em  $V$ .
- Para cada  $u \in V$ ,  $Adj[u]$  contém todos os vértices adjacentes a  $u$  em  $G$ .

## Listas de adjacência - Análise

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- Possui uma complexidade de espaço  $O(|V| + |A|)$
- Indicada para grafos **esparcos**, onde  $|A|$  é muito menor do que  $|V|^2$ .
- É compacta e usualmente utilizada na maioria das aplicações.
- A principal desvantagem é que ela pode ter tempo  $O(|V|)$  para determinar se existe uma aresta entre o vértice  $i$  e o vértice  $j$ , pois podem existir  $O(|V|)$  vértices na lista de adjacentes do vértice  $i$ .



## Listas de Adjacência usando Apontadores - Implementação

- No uso de apontadores a lista é constituída de células, onde cada célula contém um item da lista e um apontador para a célula seguinte.

```

const MaxNumVertices = 100;
      MaxNumArestas   = 4500;

type
  TipoValorVertice = 0..MaxNumVertices;
  TipoPeso         = integer;
  TipoItem         = record
    Vertice: TipoValorVertice;
    Peso   : TipoPeso;
  end;
  Apontador       = ^Celula;
  Celula          = record
    Item: TipoItem;
    Prox: Apontador;
  end;
  TipoLista       = record
    Primeiro: Apontador;
    Ultimo: Apontador;
  end;
  TipoGrafo = record
    Adj: array[TipoValorVertice] of TipoLista;
    NumVertices: TipoValorVertice;
    NumArestas: 0..MaxNumArestas;
  end;

```

## Listas de Adjacência usando Apontadores - Implementação

{— Entram aqui os operadores FLVazia, Vazia, Insere, Retira e Imprime do TAD Lista de Apontadores—}

```

procedure FGVazio(var Grafo: TipoGrafo);
var i: integer;
begin
  for i := 0 to Grafo.NumVertices-1 do FLVazia(Grafo.Adj[i]);
end; { FGVazio }

procedure InsereAresta(var V1, V2: TipoValorVertice; var Peso: TipoPeso;
                       var Grafo: TipoGrafo);
var x: TipoItem;
begin
  x.Vertice := V2; x.Peso := Peso;
  Insere(x, Grafo.Adj[V1]);
end; { InsereAresta }

function ExisteAresta (Vertice1, Vertice2: TipoValorVertice;
                      var Grafo: TipoGrafo): boolean;
var Aux: Apontador;
    EncontrouAresta: boolean;
begin
  Aux := Grafo.Adj[Vertice1].Primeiro^.Prox;
  EncontrouAresta := false;
  while (Aux <> nil) and (EncontrouAresta = false) do
    begin
      if Vertice2 = Aux^.Item.Vertice then EncontrouAresta := true;
      Aux := Aux^.Prox;
    end;
  ExisteAresta := EncontrouAresta;
end; { ExisteAresta }

```

## Listas de Adjacência usando Apontadores - Implementação

```

{— Operador para obter a lista de adjacentes—}
function ListaAdjVazia (var Vertice: TipoValorVertice;
                       var Grafo: TipoGrafo): boolean;
begin
  ListaAdjVazia := Grafo.Adj[Vertice].Primeiro =
    Grafo.Adj[Vertice].Ultimo;
end; { ListaAdjVazia }

{— Operador para obter a lista de adjacentes—}
function PrimeiroListaAdj (var Vertice: TipoValorVertice;
                          var Grafo: TipoGrafo): Apontador;
begin
  PrimeiroListaAdj := Grafo.Adj[Vertice].Primeiro^.Prox;
end; { PrimeiroListaAdj }

{— Operador para obter a lista de adjacentes—}
procedure ProxAdj (var Vertice   : TipoValorVertice;
                  var Grafo     : TipoGrafo;
                  var Adj       : TipoValorVertice;
                  var Peso      : TipoPeso;
                  var Prox      : Apontador;
                  var FimListaAdj : boolean);
{—Retorna Adj e Peso do Item apontado por Prox—}
begin
  Adj := Prox^.Item.Vertice;
  Peso := Prox^.Item.Peso;
  Prox := Prox^.Prox;
  if Prox = nil then FimListaAdj := true;
end; { ProxAdj }

```

## Listas de Adjacência usando Apontadores - Implementação

```

procedure RetiraAresta (var V1, V2: TipoValorVertice;
                       var Peso   : TipoPeso;
                       var Grafo  : TipoGrafo);
var AuxAnterior, Aux: Apontador;
    EncontrouAresta: boolean;
    x: TipoItem;
begin
  AuxAnterior := Grafo.Adj[V1].Primeiro;
  Aux := Grafo.Adj[V1].Primeiro^.Prox;
  EncontrouAresta := false;
  while (Aux <> nil) and (EncontrouAresta = false) do
    begin
      if V2 = Aux^.Item.Vertice
      then begin
        Retira(AuxAnterior, Grafo.Adj[V1], x);
        Grafo.NumArestas := Grafo.NumArestas - 1;
        EncontrouAresta := true;
      end;
      AuxAnterior := Aux; Aux := Aux^.Prox;
    end;
  end; { RetiraAresta }

```

## Listas de Adjacência usando Apontadores - Implementação

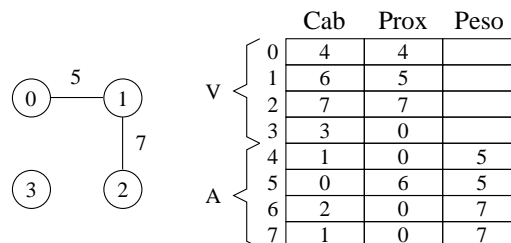
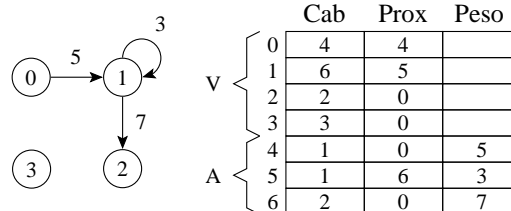
```

procedure LiberaGrafo (var Grafo: TipoGrafo);
var AuxAnterior, Aux: Apontador;
begin
  for i := 0 to Grafo.NumVertices-1 do
    begin
      Aux := Grafo.Adj[i].Primeiro^.Prox;
      dispose(Grafo.Adj[i].Primeiro); {Libera celula cabeca}
      while Aux <> nil do
        begin AuxAnterior := Aux; Aux := Aux^.Prox; dispose(AuxAnterior);
        end;
    end;
end; { LiberaGrafo }

procedure ImprimeGrafo (var Grafo : TipoGrafo);
var i: integer; Aux: Apontador;
begin
  for i := 0 to Grafo.NumVertices-1 do
    begin
      write('Vertice', i:2, ': ');
      if not Vazia(Grafo.Adj[i])
      then begin
        Aux := Grafo.Adj[i].Primeiro^.Prox;
        while Aux <> nil do
          begin
            write(Aux^.Item.Vertice:3, ' (' , Aux^.Item.Peso, ') ');
            Aux := Aux^.Prox;
          end;
        end;
      writeln;
    end;
end; { ImprimeGrafo }

```

## Listas de Adjacência usando Arranjos



- **Cab**: endereços do último item da lista de adjacentes de cada vértice (nas  $|V|$  primeiras posições) e os vértices propriamente ditos (nas  $|A|$  últimas posições)
- **Prox**: endereço do próximo item da lista de adjacentes.
- **Peso**: valor do peso de cada aresta do grafo (nas últimas  $|A|$  posições).

## Listas de Adjacência usando Arranjos - Implementação

```

const MaxNumVertices = 100;
      MaxNumArestas   = 4500;
      MaxTam          = MaxNumVertices+2*MaxNumArestas;

type
  TipoValorVertice = 0..MaxNumVertices;
  TipoPeso         = integer;
  TipoTam          = 0..MaxTam;
  TipoGrafo = record
    Cab      : array[TipoTam] of TipoTam;
    Prox     : array[TipoTam] of TipoTam;
    Peso     : array[TipoTam] of TipoTam;
    ProxDisponivel: TipoTam;
    NumVertices : 0..MaxNumVertices;
    NumArestas  : 0..MaxNumArestas;
  end;
  Apontador = TipoTam;

procedure FGVazio(var Grafo: TipoGrafo);
var i: integer;
begin
  for i := 0 to Grafo.NumVertices do
    begin
      Grafo.Prox[i] := 0; Grafo.Cab[i] := i;
      Grafo.ProxDisponivel := Grafo.NumVertices;
    end;
end;

```

## Listas de Adjacência usando Arranjos - Implementação

```

procedure InsereAresta (var V1, V2: TipoValorVertice;
                        var Peso : TipoPeso;
                        var Grafo : TipoGrafo);
var Pos: integer;
begin
  Pos := Grafo.ProxDisponivel;
  if Grafo.ProxDisponivel = MaxTam
  then writeln('nao ha espaco disponivel para a aresta')
  else begin
    Grafo.ProxDisponivel := Grafo.ProxDisponivel + 1;
    Grafo.Prox[Grafo.Cab[V1]] := Pos;
    Grafo.Cab[Pos] := V2; Grafo.Cab[V1] := Pos;
    Grafo.Prox[Pos] := 0; Grafo.Peso[Pos] := Peso;
  end;
end; { InsereAresta }

function ExisteAresta (Vertice1, Vertice2: TipoValorVertice;
                      var Grafo: TipoGrafo): boolean;
var Aux: Apontador; EncontrouAresta: boolean;
begin
  Aux := Grafo.Prox[Vertice1]; EncontrouAresta := false;
  while (Aux <> 0) and (EncontrouAresta = false) do
    begin
      if Vertice2 = Grafo.Cab[Aux] then EncontrouAresta := true;
      Aux := Grafo.Prox[Aux];
    end;
  ExisteAresta := EncontrouAresta;
end; { ExisteAresta }

```

## Listas de Adjacência usando Arranjos - Implementação

```
{— Operador para obter a lista de adjacentes—}
function ListaAdjVazia(var Vertice: TipoValorVertice;
                     var Grafo: TipoGrafo): boolean;

begin
    ListaAdjVazia := Grafo.Prox[Vertice] = 0;
end; { ListaAdjVazia }

{— Operador para obter a lista de adjacentes—}
function PrimeiroListaAdj(var Vertice: TipoValorVertice;
                        var Grafo: TipoGrafo): Apontador;

begin
    PrimeiroListaAdj := Grafo.Prox[Vertice];
end; { PrimeiroListaAdj }

{— Retorna Adj apontado por Prox—}
procedure ProxAdj (var Vertice      : TipoValorVertice;
                   var Grafo        : TipoGrafo;
                   var Adj          : TipoValorVertice;
                   var Peso         : TipoPeso;
                   var Prox         : Apontador;
                   var FimListaAdj  : boolean);

begin
    Adj := Grafo.Cab[Prox]; Peso := Grafo.Peso[Prox];
    Prox := Grafo.Prox[Prox];
    if Prox = 0 then FimListaAdj := true;
end; { ProxAdj }
```

## Busca em Profundidade

- A busca em profundidade, do inglês *depth-first search*), é um algoritmo para caminhar no grafo.
- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice  $v$  mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.
- Quando todas as arestas adjacentes a  $v$  tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual  $v$  foi descoberto.
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

## Listas de Adjacência usando Arranjos - Implementação

```
procedure RetiraAresta (var V1, V2: TipoValorVertice;
                      var Peso: TipoPeso; var Grafo: TipoGrafo);
var Aux, AuxAnterior: Apontador; EncontrouAresta: boolean;
begin
    AuxAnterior := V1; Aux := Grafo.Prox[V1];
    EncontrouAresta := false;
    while (Aux <> 0) and (EncontrouAresta = false) do
        begin
            if V2 = Grafo.Cab[Aux]
            then EncontrouAresta := true
            else begin AuxAnterior := Aux; Aux := Grafo.Prox[Aux]; end;
        end;
    if EncontrouAresta
    then Grafo.Cab[Aux] := MaxNumVertices+2*MaxNumArestas
        {— Apenas marca como retirado —}
    else writeln('Aresta nao existe');
end; { RetiraAresta }

procedure LiberaGrafo (var Grafo: TipoGrafo);
begin {Nada no caso de posicoes contiguas} end; { LiberaGrafo }

procedure ImprimeGrafo (var Grafo : TipoGrafo);
var i: integer;
begin
    writeln('    Cab Prox Peso');
    for i := 0 to Grafo.NumVertices+2*Grafo.NumArestas-1 do
        writeln(i:2, Grafo.Cab[i]:4, Grafo.Prox[i]:4, Grafo.Peso[i]:4);
end; { ImprimeGrafo }
```

## Busca em Profundidade

- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.
- $d[v]$ : tempo de descoberta
- $t[v]$ : tempo de término do exame da lista de adjacentes de  $v$ .
- Estes registros são inteiros entre 1 e  $2|V|$  pois existe um evento de descoberta e um evento de término para cada um dos  $|V|$  vértices.

## Busca em Profundidade - Implementação

```

procedure BuscaEmProfundidade (var Grafo: TipoGrafo);
var Tempo      : TipoValorTempo;
    x           : TipoValorVertice;
    d, t        : array[TipoValorVertice] of TipoValorTempo;
    Cor         : array[TipoValorVertice] of TipoCor;
    Antecessor  : array[TipoValorVertice] of integer;

```

{— *Entra aqui o procedimento VisitaDFS (a seguir)* —}

```

begin
    Tempo := 0;
    for x := 0 to Grafo.NumVertices-1 do
        begin Cor[x] := branco; Antecessor[x] := -1; end;
    for x := 0 to Grafo.NumVertices-1 do
        if Cor[x] = branco then VisitaDfs(x);
end; { BuscaEmProfundidade }

```

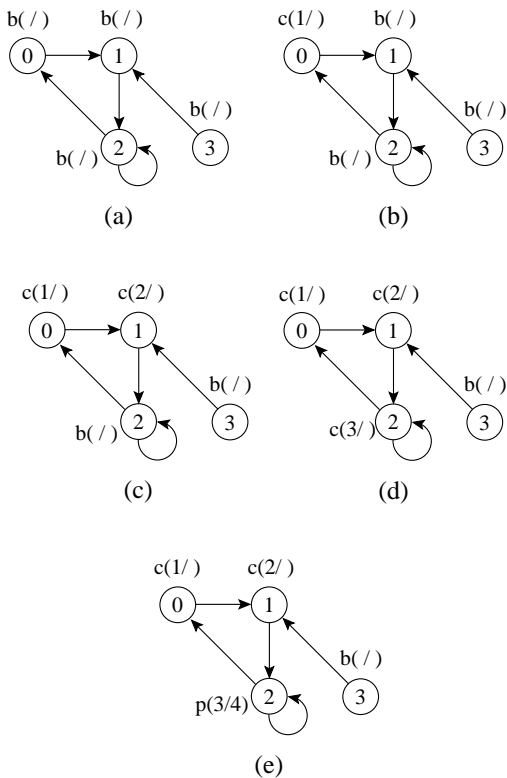
## Busca em Profundidade - Implementação

```

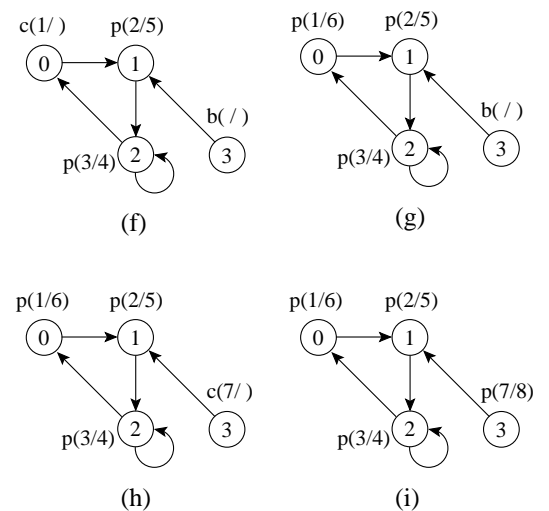
procedure VisitaDfs (u:TipoValorVertice);
var FimListaAdj: boolean;
    Peso        : TipoPeso;
    Aux         : Apontador;
    v           : TipoValorVertice;
begin
    Cor[u] := cinza; Tempo := Tempo + 1; d[u] := Tempo;
    writeln('Visita',u:2,' Tempo descoberta:',d[u]:2,' cinza'); readln;
    if 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 Cor[v] = branco
                then begin Antecessor[v] := u; VisitaDfs(v); end;
            end;
        end;
    Cor[u] := preto; Tempo := Tempo + 1; t[u] := Tempo;
    writeln('Visita',u:2,' Tempo termino:',t[u]:2,' preto'); readln;
end; { VisitaDfs }

```

## Busca em Profundidade - Exemplo



## Busca em Profundidade - Exemplo



## Busca em Profundidade - Análise

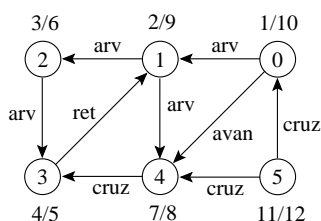
- Os dois anéis da *BuscaEmProfundidade* têm custo  $O(|V|)$  cada um, a menos da chamada do procedimento  $VisitaDfs(u)$  no segundo anel.
- O procedimento  $VisitaDfs$  é chamado exatamente uma vez para cada vértice  $u \in V$ , desde que  $VisitaDfs$  é chamado apenas para vértices brancos e a primeira ação é pintar o vértice de cinza.
- Durante a execução de  $VisitaDfs(u)$  o anel principal é executado  $|Adj[u]|$  vezes.
- Desde que  $\sum_{u \in V} |Adj[u]| = O(|A|)$ , o tempo total de execução de  $VisitaDfs$  é  $O(|A|)$ .
- Logo, a complexidade total da *BuscaEmProfundidade* é  $O(|V| + |A|)$ .

## Classificação de Arestas

- Existem:
- Arestas de árvore:** são arestas de uma árvore de busca em profundidade. A aresta  $(u, v)$  é uma aresta de árvore se  $v$  foi descoberto pela primeira vez ao percorrer a aresta  $(u, v)$ .
  - Arestas de retorno:** conectam um vértice  $u$  com um antecessor  $v$  em uma árvore de busca em profundidade (inclui *self-loops*).
  - Arestas de avanço:** não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.
  - Arestas de cruzamento:** podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

## Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica uma aresta de avanço quando  $u$  é descoberto antes de  $v$  ou uma aresta de cruzamento caso contrário.



## Teste para Verificar se Grafo é Acíclico

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em  $G$ , então o grafo tem ciclo.
- Um grafo direcionado  $G$  é acíclico se e somente se a busca em profundidade em  $G$  não apresentar arestas de retorno.

## Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância  $k$  do vértice origem antes de descobrir qualquer vértice a uma distância  $k + 1$ .
- O grafo  $G(V, A)$  pode ser direcionado ou não direcionado.

## Busca em Largura - Implementação

{— Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do—}  
 {— TAD Filas com arranjos ou apontadores, dependendo da implementação—}  
 {— da busca em largura usar arranjos ou apontadores, respectivamente—}

```
procedure BuscaEmLargura (var Grafo: TipoGrafo);
var x      : TipoValorVertice;
    Dist   : array[TipoValorVertice] of integer;
    Cor     : array[TipoValorVertice] of TipoCor;
    Antecessor : array[TipoValorVertice] of integer;
```

{— Entra aqui o procedimento VisitaBfs (a seguir)—}

```
begin
  for x := 0 to Grafo.NumVertices-1 do
    begin
      Cor[x] := branco; Dist[x] := infinito; Antecessor[x] := -1;
    end;
  for x := 0 to Grafo.NumVertices-1 do
    if Cor[x] = branco then VisitaBfs(x);
  end; { BuscaEmLargura }
```

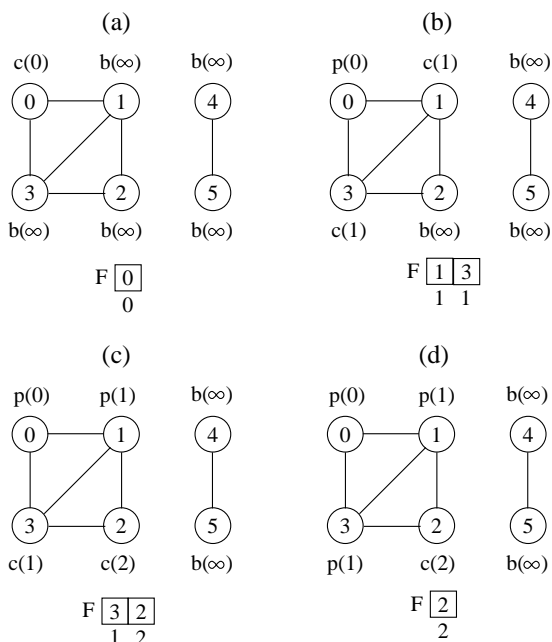
## Busca em Largura

- Cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é descoberto pela primeira vez ele torna-se cinza.
- Vértices cinza e preto já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura.
- Se  $(u, v) \in A$  e o vértice  $u$  é preto, então o vértice  $v$  tem que ser cinza ou preto.
- Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices descobertos e não descobertos.

## Busca em Largura - Implementação

```
procedure VisitaBfs (u:TipoValorVertice);
var v: TipoValorVertice; Aux: Apontador; FimListaAdj: boolean;
    Peso: TipoPeso; Item: TipoItem; Fila: TipoFila;
begin
  Cor[u] := cinza; Dist[u] := 0;
  FFVazia (Fila); Item.Vertice := u;
  Enfileira (Item, Fila);
  write('Visita origem',u:2,' cor: cinza F:');
  ImprimeFila (Fila); readln;
  while not FilaVazia (Fila) do
    begin
      Desenfileira (Fila, Item); u := Item.vertice;
      if not ListaAdjVazia(u, Grafo)
      then begin
        Aux := PrimeiroListaAdj(u,Grafo); FimListaAdj := false;
        while FimListaAdj = false do
          begin
            ProxAdj(u, v, Peso, Aux, FimListaAdj);
            if Cor[v] = branco
            then begin
              Cor[v] := cinza; Dist[v] := Dist[u] + 1;
              Antecessor[v] := u;
              Item.Vertice := v; Item.Peso := Peso;
              Enfileira (Item, Fila);
            end;
          end;
        end;
        Cor[u] := preto;
        write('Visita ', u:2,' Dist',Dist[u]:2,' cor: preto F:');
        ImprimeFila (Fila); readln;
      end;
    end; { VisitaBfs }
```

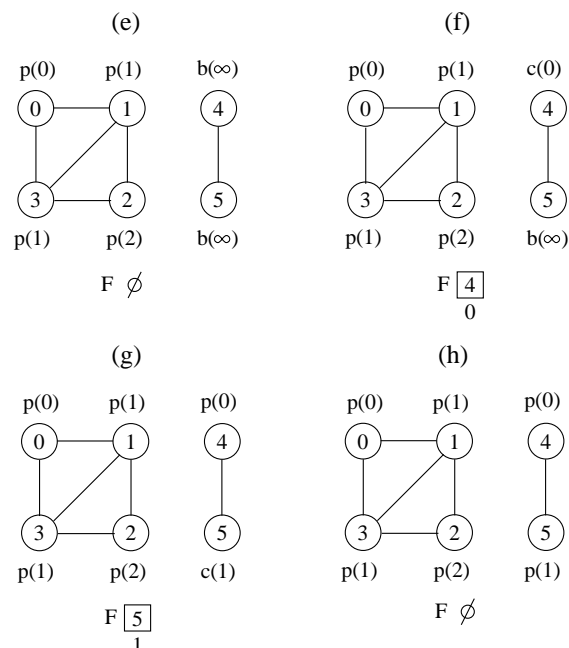
## Busca em Largura - Exemplo



## Busca em Largura - Análise (para listas de adjacência)

- O custo de inicialização do primeiro anel em *BuscaEmLargura* é  $O(|V|)$  cada um.
- O custo do segundo anel é também  $O(|V|)$ .
- *VisitaBfs*: enfileirar e desenfileirar têm custo  $O(1)$ , logo, o custo total com a fila é  $O(|V|)$ .
- Cada lista de adjacentes é percorrida no máximo uma vez, quando o vértice é desenfileirado.
- Desde que a soma de todas as listas de adjacentes é  $O(|A|)$ , o tempo total gasto com as listas de adjacentes é  $O(|A|)$ .
- Complexidade total: é  $O(|V| + |A|)$ .

## Busca em Largura - Exemplo



## Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de  $u$  até  $v$ .
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.
- O programa abaixo imprime os vértices do caminho mais curto entre o vértice origem e outro vértice qualquer do grafo, a partir do vetor *Antecessor* obtido na busca em largura.

```

procedure ImprimeCaminho (Origem, v: TipovalorVertice);
begin
  if Origem = v
  then write(Origem:3)
  else if Antecessor[v] = -1
  then write('Nao existe caminho de ',Origem:3,' ate ',v:3)
  else begin
    ImprimeCaminho(Origem, Antecessor[v]);
    write(v:3);
  end;
end; { ImprimeCaminho }

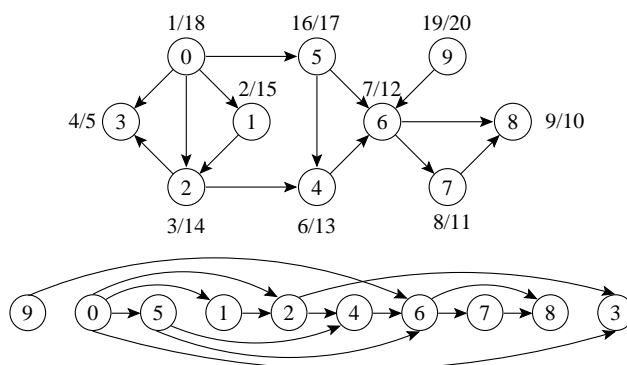
```

## Ordenação Topológica

- Ordenação linear de todos os vértices, tal que se  $G$  contém uma aresta  $(u, v)$  então  $u$  aparece antes de  $v$ .
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.
- Pode ser feita usando a busca em profundidade.

## Ordenação Topológica

- Os grafos direcionados acíclicos são usados para indicar precedências entre eventos.
- Uma aresta direcionada  $(u, v)$  indica que a atividade  $u$  tem que ser realizada antes da atividade  $v$ .



## Ordenação Topológica

- Algoritmo para ordenar topologicamente um grafo direcionado acíclico  $G = (V, A)$ :
  1. Chama *BuscaEmProfundidade*( $G$ ) para obter os tempos de término  $t[u]$  para cada vértice  $u$ .
  2. Ao término de cada vértice insira-o na frente de uma lista linear encadeada.
  3. Retorna a lista encadeada de vértices.
- A Custo  $O(|V| + |A|)$ , uma vez que a busca em profundidade tem complexidade de tempo  $O(|V| + |A|)$  e o custo para inserir cada um dos  $|V|$  vértices na frente da lista linear encadeada custa  $O(1)$ .

## Ordenação Topológica - Implementação

- Basta inserir uma chamada para o procedimento *InsLista* no procedimento *BuscaDfs*, logo após o momento em que o tempo de término  $t[u]$  é obtido e o vértice é pintado de preto.
- Ao final, basta retornar a lista obtida (ou imprimí-la).

```

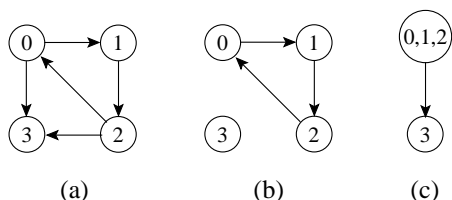
procedure InsLista (var Item: TipoItem; var Lista: TipoLista);
  {— Insere antes do primeiro item da lista —}
  var Aux: Apontador;
  begin
    Aux := Lista.Primeiro^.Prox;
    new(Lista.Primeiro^.Prox);
    Lista.Primeiro^.Prox^.Item := Item;
    Lista.Primeiro^.Prox^.Prox := Aux;
  end; { Insere }

```



## Componentes Fortemente Conectados

- Um componente fortemente conectado de  $G = (V, A)$  é um conjunto maximal de vértices  $C \subseteq V$  tal que para todo par de vértices  $u$  e  $v$  em  $C$ ,  $u$  e  $v$  são mutuamente alcançáveis
- Podemos particionar  $V$  em conjuntos  $V_i$ ,  $1 \leq i \leq r$ , tal que vértices  $u$  e  $v$  são equivalentes se e somente se existe um caminho de  $u$  a  $v$  e um caminho de  $v$  a  $u$ .



## Componentes Fortemente Conectados - Algoritmo

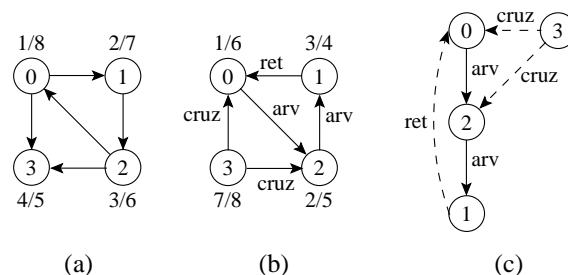
- Usa o **transposto** de  $G$ , definido  $G^T = (V, A^T)$ , onde  $A^T = \{(u, v) : (v, u) \in A\}$ , isto é,  $A^T$  consiste das arestas de  $G$  com suas direções invertidas.
- $G$  e  $G^T$  possuem os mesmos componentes fortemente conectados, isto é,  $u$  e  $v$  são mutuamente alcançáveis a partir de cada um em  $G$  se e somente se  $u$  e  $v$  são mutuamente alcançáveis a partir de cada um em  $G^T$ .

## Componentes Fortemente Conectados - Algoritmo

- Chama  $\text{BuscaEmProfundidade}(G)$  para obter os tempos de término  $t[u]$  para cada vértice  $u$ .
- Obtem  $G^T$ .
- Chama  $\text{BuscaEmProfundidade}(G^T)$ , realizando a busca a partir do vértice de maior  $t[u]$  obtido na linha 1. Inicie uma nova busca em profundidade a partir do vértice de maior  $t[u]$  dentre os vértices restantes se houver.
- Retorne os vértices de cada árvore da floresta obtida como um componente fortemente conectado separado.

## Componentes Fortemente Conectados - Exemplo

- A parte (b) apresenta o resultado da busca em profundidade sobre o grafo transposto obtido, mostrando os tempos de término e a classificação das arestas.
- A busca em profundidade em  $G^T$  resulta na floresta de árvores mostrada na parte (c).



## Componentes Fortemente Conectados - Implementação

```

procedure GrafoTransposto (var Grafo : TipoGrafo; var GrafoT: TipoGrafo);
var v, u: TipoValorVertice;
    Peso: TipoPeso;
    Aux : Apontador;
begin
    FGVazio(GrafoT);
    GrafoT.NumVertices := Grafo.NumVertices;
    GrafoT.NumArestas := Grafo.NumArestas;
    for v := 0 to Grafo.NumVertices-1 do
    if not ListaAdjVazia(v, Grafo)
    then begin
        Aux := PrimeiroListaAdj(v, Grafo);
        FimListaAdj := false;
        while not FimListaAdj do
            begin
                ProxAdj(v, Grafo, u, Peso, Aux, FimListaAdj);
                InsereAresta(u, v, Peso, GrafoT);
            end;
        end;
    end; { GrafoTransposto }

```

## Componentes Fortemente Conectados - Implementação

```

procedure BuscaEmProfundidadeCfc (var Grafo: TipoGrafo;
                                var TT: TipoTempoTermino);
var
    Tempo      : TipoValorTempo;
    x, VRaiz   : TipoValorVertice;
    d, t       : array[TipoValorVertice] of TipoValorTempo;
    Cor        : array[TipoValorVertice] of TipoCor;
    Antecessor : array[TipoValorVertice] of integer;
    {— Entra aqui o procedimento VisitaDFS (a seguir) —}
begin
    Tempo := 0;
    for x := 0 to Grafo.NumVertices-1 do
        begin Cor[x] := branco; Antecessor[x] := -1; end;
    TT.NumRestantes := Grafo.NumVertices;
    for x := 0 to Grafo.NumVertices-1 do
        TT.Restantes[x] := true;
    while TT.NumRestantes > 0 do
        begin
            VRaiz := MaxTT (TT);
            writeln('Raiz da proxima arvore: ', VRaiz:2);
            VisitaDfs (VRaiz);
        end;
    end; { BuscaEmProfundidadeCfc }

```

## Componentes Fortemente Conectados - Implementação

- O Programa BuscaEmProfundidadeCfc utiliza a função *MaxTT* para obter o vértice de maior  $t[u]$  dentre os vértices restantes  $u$  ainda não visitados por VisitaDFS.

```

type
    TipoTempoTermino = record
        t: array[TipoValorVertice] of TipoValorTempo;
        Restantes: array[TipoValorVertice] of boolean;
        NumRestantes: TipoValorVertice;
    end;

Function MaxTT (var TT: TipoTempoTermino): TipoValorVertice;
var i, Temp: integer;
begin
    i:=0;
    while not TT.Restantes[i] do i := i + 1;
    Temp := TT.t[i]; MaxTT := i;
    for i := 0 to Grafo.NumVertices-1 do
        if TT.Restantes[i]
        then if Temp < TT.t[i]
            then begin Temp := TT.t[i]; MaxTT := i; end;
    end; { MaxTT }

```

## Componentes Fortemente Conectados - Implementação

```

procedure VisitaDfs (u:TipoValorVertice);
var FimListaAdj: boolean;
    Peso      : TipoPeso;
    Aux       : Apontador;
    v         : TipoValorVertice;
begin
    Cor[u] := cinza; Tempo := Tempo + 1; d[u] := Tempo;
    TT.Restantes[u] := false; TT.NumRestantes := TT.NumRestantes-1;
    writeln('Visita ',u:2,' Tempo descoberta: ',d[u]:2,' cinza'); readln;
    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 Cor[v] = branco
                then begin
                    Antecessor[v] := u;
                    VisitaDfs(v);
                end;
            end;
        end;
    end;
    Cor[u] := preto; Tempo := Tempo + 1; t[u] := Tempo;
    writeln('Visita ',u:2,' Tempo termino: ',t[u]:2,' preto'); readln;
end; { VisitaDfs }

```

## Componentes Fortemente Conectados - Análise

- Utiliza o algoritmo para busca em profundidade duas vezes, uma em  $G$  e outra em  $G^T$ . Logo, a complexidade total é  $O(|V| + |A|)$ .

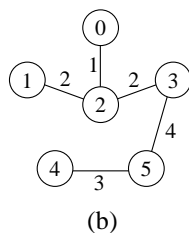
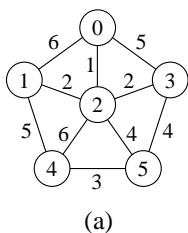
## Árvore Geradora Mínima - Aplicação

- Projeto de redes de comunicações conectando  $n$  localidades.
- Arranjo de  $n - 1$  conexões, conectando duas localidades cada.
- Objetivo: dentre as possibilidades de conexões, achar a que usa menor quantidade de cabos.
- Modelagem:
  - $G = (V, A)$ : grafo conectado, não direcionado.
  - $V$ : conjunto de cidades.
  - $A$ : conjunto de possíveis conexões
  - $p(u, v)$ : peso da aresta  $(u, v) \in A$ , custo total de cabo para conectar  $u$  a  $v$ .
- Solução: encontrar um subconjunto  $T \subseteq A$ , acíclico, que conecta todos os vértices de  $G$  e cujo peso total  $p(T) = \sum_{(u,v) \in T} p(u, v)$  é minimizado.

## Árvore Geradora Mínima (AGM)

- Como  $G' = (V, T)$  é acíclico e conecta todos os vértices,  $T$  forma uma árvore chamada **árvore geradora** de  $G$ .
- O problema de obter a árvore  $T$  é conhecido como **árvore geradora mínima** (AGM).

Ex.: Árvore geradora mínima  $T$  cujo peso total é 12.  $T$  não é única, pode-se substituir a aresta  $(3, 5)$  pela aresta  $(2, 5)$  obtendo outra árvore geradora de custo 12.



## AGM - Algoritmo Genérico

- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração,  $S$  é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a  $S$  uma aresta  $(u, v)$  que não viola o invariante.  $(u, v)$  é chamada de uma **aresta segura**.

**procedure** GenéricoAGM;

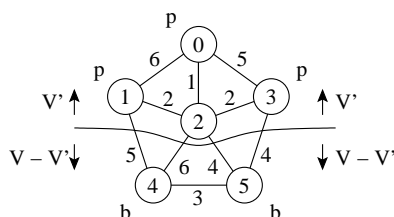
```

1   $S := \emptyset$ ;
2  while  $S$  não constitui uma árvore geradora mínima do
3    Encontre uma aresta  $(u, v)$  que é segura para  $S$ ;
4     $S := S + \{(u, v)\}$ 
5  return  $S$ ;
```

- Dentro do **while**,  $S$  tem que ser um subconjunto próprio da AGM  $T$ , e assim tem que existir uma aresta  $(u, v) \in T$  tal que  $(u, v) \notin S$  e  $(u, v)$  é seguro para  $S$ .

## AGM - Definição de Corte

- Um **corte**  $(V', V - V')$  de um grafo não direcionado  $G = (V, A)$  é uma partição de  $V$ .
- Uma aresta  $(u, v) \in A$  **cruza** o corte  $(V', V - V')$  se um de seus vértices pertence a  $V'$  e o outro vértice pertence a  $V - V'$ .
- Um corte **respeita** um conjunto  $S$  de arestas se não existirem arestas em  $S$  que o cruzem.
- Uma aresta cruzando o corte que tenha custo mínimo sobre todas as arestas cruzando o corte é uma **aresta leve**.



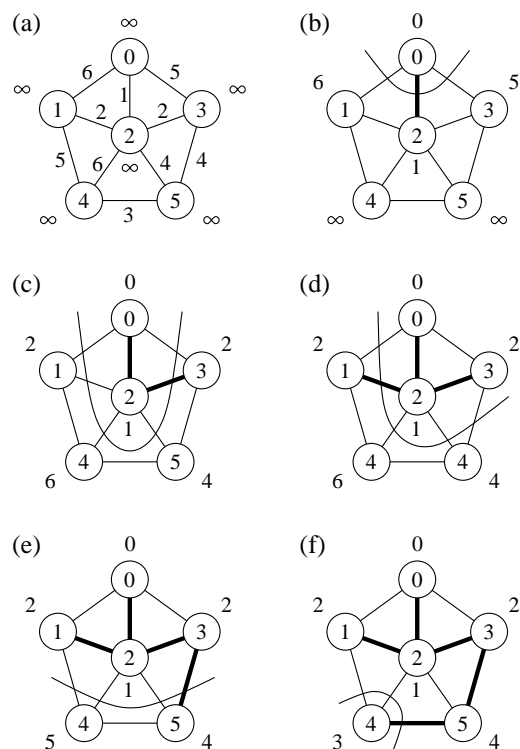
## AGM - Teorema para reconhecer arestas seguras

- Seja  $G = (V, A)$  um grafo conectado, não direcionado, com pesos  $p$  sobre as arestas  $V$ .
- seja  $S$  um subconjunto de  $V$  que está incluído em alguma AGM para  $G$ .
- Seja  $(V', V - V')$  um corte qualquer que respeita  $S$ .
- Seja  $(u, v)$  uma aresta leve cruzando  $(V', V - V')$ .
- Satisfeitas essas condições, a aresta  $(u, v)$  é uma aresta segura para  $S$ .

## AGM - Algoritmo de Prim

- O algoritmo de Prim para obter uma AGM pode ser derivado do algoritmo genérico.
- O subconjunto  $S$  forma uma única árvore, e a aresta segura adicionada a  $S$  é sempre uma aresta de peso mínimo conectando a árvore a um vértice que não esteja na árvore.
- A árvore começa por um vértice qualquer (no caso 0) e cresce até que “gere” todos os vértices em  $V$ .
- A cada passo, uma aresta leve é adicionada à árvore  $S$ , conectando  $S$  a um vértice de  $G_S = (V, S)$ .
- De acordo com o teorema anterior, quando o algoritmo termina, as arestas em  $S$  formam uma árvore geradora mínima.

## Algoritmo de Prim - Exemplo



## Algoritmo de Prim - Implementação

{— Entram aqui os operadores de uma das implementações para grafos —}

{— bem como o operador Constroi do TAD HEAP —}

```
procedure AgmPrim (var Grafo: TipoGrafo; var Raiz: TipoValorVertice);
var Antecessor: array[TipoValorVertice] of integer;
    p          : array[TipoValorVertice] of TipoPeso;
    Itensheap  : array[TipoValorVertice] of boolean;
    Pos       : array[TipoValorVertice] of TipoValorVertice;
    A         : Vetor;
    u, v      : TipoValorVertice;
```

```
procedure Refaz (Esq, Dir : Indice; var A : Vetor);
label 999;
var i: Indice; j: integer; x: Item;
begin
    i := Esq; j := 2 * i; x := A[i];
    while j <= Dir do
        begin
            if j < Dir
            then if p[A[j].Chave] > p[A[j + 1].Chave] then j := j + 1;
            if p[x.Chave] <= p[A[j].Chave] then goto 999;
            A[i] := A[j]; Pos[A[j].Chave] := i;
            i := j; j := 2 * i;
        end;
    999 : A[i] := x; Pos[x.Chave] := i;
end; { Refaz }
```

## Algoritmo de Prim - Implementação

```
function RetiraMin (var A: Vetor): Item;
begin
    if n < 1
    then writeln('Erro: heap vazio')
    else begin
        RetiraMin := A[1];
        A[1] := A[n]; Pos[A[n].chave] := 1;
        n := n - 1;
        Refaz (1, n, A);
    end;
end; { RetiraMin }

procedure DiminuiChave (i: Indice; ChaveNova: TipoPeso; var A: Vetor);
var x: Item;
begin
    if ChaveNova > p[A[i].Chave]
    then writeln('Erro: ChaveNova maior que a chave atual')
    else begin
        p[A[i].Chave] := ChaveNova;
        while (i > 1) and (p[A[i div 2].Chave] > p[A[i].Chave])
        do begin
            x := A[i div 2];
            A[i div 2] := A[i]; Pos[A[i].Chave] := i div 2;
            A[i] := x; Pos[x.Chave] := i;
            i := i div 2;
        end;
    end;
end; { DiminuiChave }
```

## Algoritmo de Prim - Implementação

```
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 }
```

## Algoritmo de Prim - Implementação

- Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na AGM residem no *heap A*.
- O *heap* contém os vértices, mas a condição do *heap* é mantida pelo peso da aresta através do arranjo  $p[v]$  (*heap* indireto).
- $Pos[v]$  fornece a posição do vértice  $v$  dentro do *heap A*, para que o vértice  $v$  possa ser acessado a um custo  $O(1)$ , necessário para a operação DiminuiChave.
- $Antecessor[v]$  armazena o antecessor de  $v$  na árvore.
- Quando o algoritmo termina,  $A$  está vazia e a AGM está de forma implícita como  $S = \{(v, Antecessor[v]) : v \in V - \{Raiz\}\}$

## Algoritmo de Prim - Análise

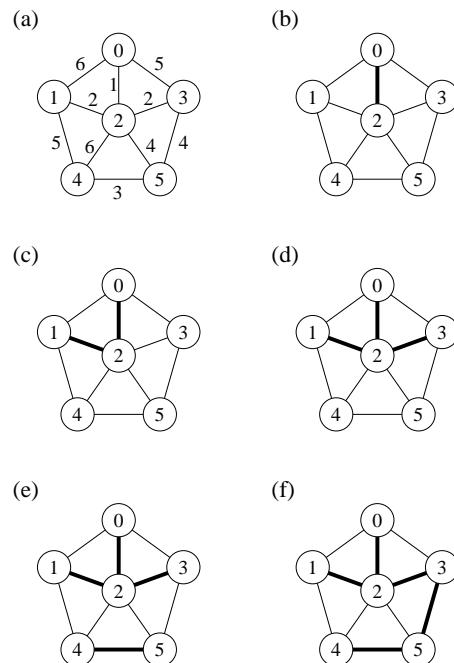
- O corpo do anel **while** é executado  $|V|$  vezes.
- O procedimento *Refaz* tem custo  $O(\log |V|)$ .
- Logo, o tempo total para executar a operação retira o item com menor peso é  $O(|V| \log |V|)$ .
- O **while** mais interno para percorrer a lista de adjacentes é  $O(|A|)$  (soma dos comprimentos de todas as listas de adjacência é  $2|A|$ ).
- O teste para verificar se o vértice  $v$  pertence ao *heap*  $A$  tem custo  $O(1)$ .
- Após testar se  $v$  pertence ao *heap*  $A$  e o peso da aresta  $(u, v)$  é menor do que  $p[v]$ , o antecessor de  $v$  é armazenado em *Antecessor* e uma operação *DiminuiChave* é realizada sobre o *heap*  $A$  na posição  $Pos[v]$ , a qual tem custo  $O(\log |V|)$ .
- Logo, o tempo total para executar o algoritmo de Prim é  
 $O(|V| \log |V| + |A| \log |V|) = O(|A| \log |V|)$ .

## AGM - Algoritmo de Kruskal

- Sejam  $C_1$  e  $C_2$  duas árvores conectadas por  $(u, v)$ :
  - Como  $(u, v)$  tem de ser uma aresta leve conectando  $C_1$  com alguma outra árvore,  $(u, v)$  é uma aresta segura para  $C_1$ .
- 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.

## AGM - Algoritmo de Kruskal

- Pode ser derivado do algoritmo genérico.
- $S$  é uma floresta e a aresta segura adicionada a  $S$  é sempre uma aresta de menor peso que conecta dois componentes distintos.
- Considera as arestas ordenadas pelo peso.



## Algoritmo de Kruskal - Implementação

- 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.
- Os elementos de um conjunto são representados por um objeto. Operações:
  - CriaConjunto(x): cria novo conjunto cujo único membro,  $x$ , é seu representante. Para que os conjuntos sejam disjuntos,  $x$  não pode pertencer a outro conjunto.
  - União(x, y): une conjuntos dinâmicos contendo  $x$  ( $C_x$ ) e  $y$  ( $C_y$ ) em novo conjunto, cujo representante pode ser  $x$  ou  $y$ . Como os conjuntos na coleção devem ser disjuntos,  $C_x$  e  $C_y$  são destruídos.
  - EncontreConjunto(x): retorna apontador para o representante do conjunto (único) contendo  $x$ .

## Algoritmo de Kruskal - Implementação

- Primeiro refinamento:

```

procedure Kruskal;
1.   $S := \emptyset$ ;
2.  for  $v := 0$  to Grafo.NumVertices-1 do CriaConjunto ( $v$ );
3.  Ordena as arestas de  $A$  pelo peso;
4.  for cada  $(u, v)$  de  $A$  tomadas em ordem ascendente de peso do
5.      if EncontreConjunto ( $u$ )  $\neq$  EncontreConjunto ( $v$ )
          then begin
6.           $S := S + \{(u, v)\}$ ;
7.          Uniao ( $u, v$ );
          end;
end;

```

- A implementação das operações União e EncontraConjunto deve ser realizada de forma eficiente.
- Esse problema é conhecido na literatura como **União-EncontraConjunto**.

## Caminhos Mais Curtos - Aplicação

- Um motorista procura o caminho mais curto entre Diamantina e Ouro Preto. Possui mapa com as distâncias entre cada par de interseções adjacentes.
- Modelagem:
  - $G = (V, A)$ : grafo direcionado ponderado, mapa rodoviário.
  - $V$ : interseções.
  - $A$ : segmentos de estrada entre interseções
  - $p(u, v)$ : peso de cada aresta, distância entre interseções.
- Peso de um caminho:  $p(c) = \sum_{i=1}^k p(v_{i-1}, v_i)$
- Caminho mais curto:

$$\delta(u, v) = \begin{cases} \min \{p(c) : u \rightsquigarrow v\} & \text{se existir caminho de } u \text{ a } v \\ \infty & \text{caso contrário} \end{cases}$$

- **Caminho mais curto** do vértice  $u$  ao vértice  $v$ : qualquer caminho  $c$  com peso  $p(c) = \delta(u, v)$ .

## AGM - Análise do Algoritmo de Kruskal

- A inicialização do conjunto  $S$  tem custo  $O(1)$ .
- Ordenar arestas (linha 3) custa  $O(|A| \log |A|)$ .
- A linha 2 realiza  $|V|$  operações CriaConjunto.
- O anel (linhas 4-7) realiza  $O(|A|)$  operações EncontreConjunto e Uniao, a um custo  $O((|V| + |A|)\alpha(|V|))$  onde  $\alpha(|V|)$  é uma função que cresce lentamente ( $\alpha(|V|) < 4$ ).
- O limite inferior para construir uma estrutura dinâmica envolvendo  $m$  operações EncontreConjunto e Uniao e  $n$  operações CriaConjunto é  $m\alpha(n)$ .
- Como  $G$  é conectado temos que  $|A| \geq |V| - 1$ , e assim as operações sobre conjuntos disjuntos custam  $O(|A|\alpha(|V|))$ .
- Como  $\alpha(|V|) = O(\log |A|) = O(\log |V|)$ , o tempo total do algoritmo de Kruskal é  $O(|A| \log |A|)$ .
- Como  $|A| < |V|^2$ , então  $\log |A| = O(\log |V|)$ , e o custo do algoritmo de Kruskal é também  $O(|A| \log |V|)$ .

## Caminhos Mais Curtos

- **Caminhos mais curtos a partir de uma origem**: dado um grafo ponderado  $G = (V, A)$ , desejamos obter o caminho mais curto a partir de um dado vértice origem  $s \in V$  até cada  $v \in V$ .
- Muitos problemas podem ser resolvidos pelo algoritmo para o problema origem única:
  - **Caminhos mais curtos com destino único**: reduzido ao problema origem única invertendo a direção de cada aresta do grafo.
  - **Caminhos mais curtos entre um par de vértices**: o algoritmo para origem única é a melhor opção conhecida.
  - **Caminhos mais curtos entre todos os pares de vértices**: resolvido aplicando o algoritmo origem única  $|V|$  vezes, uma vez para cada vértice origem.

## Caminhos Mais Curtos

- A representação de caminhos mais curtos pode ser realizada pela variável *Antecessor*.
- Para cada vértice  $v \in V$  o  $Antecessor[v]$  é um outro vértice  $u \in V$  ou *nil* (-1).
- O algoritmo atribui a *Antecessor* os rótulos de vértices de uma cadeia de antecessores com origem em  $v$  e que anda para trás ao longo de um caminho mais curto até o vértice origem  $s$ .
- Dado um vértice  $v$  no qual  $Antecessor[v] \neq nil$ , o procedimento *ImprimeCaminho* pode imprimir o caminho mais curto de  $s$  até  $v$ .
- Os valores em  $Antecessor[v]$ , em um passo intermediário, não indicam necessariamente caminhos mais curtos.
- Entretanto, ao final do processamento, *Antecessor* contém uma árvore de caminhos mais curtos definidos em termos dos pesos de cada aresta de  $G$ , ao invés do número de arestas.
- Caminhos mais curtos não são necessariamente únicos.

## Algoritmo de Dijkstra

- Mantém um conjunto  $S$  de vértices cujos caminhos mais curtos até um vértice origem já são conhecidos.
- Produz uma árvore de caminhos mais curtos de um vértice origem  $s$  para todos os vértices que são alcançáveis a partir de  $s$ .
- Utiliza a técnica de **relaxamento**:
  - Para cada vértice  $v \in V$  o atributo  $p[v]$  é um limite superior do peso de um caminho mais curto do vértice origem  $s$  até  $v$ .
  - O vetor  $p[v]$  contém uma estimativa de um caminho mais curto.
- O primeiro passo do algoritmo é inicializar os antecessores e as estimativas de caminhos mais curtos:
  - $Antecessor[v] = nil$  para todo vértice  $v \in V$ ,
  - $p[u] = 0$ , para o vértice origem  $s$ , e
  - $p[v] = \infty$  para  $v \in V - \{s\}$ .

## Árvore de caminhos mais curtos

- Uma árvore de caminhos mais curtos com raiz em  $u \in V$  é um subgrafo direcionado  $G' = (V', A')$ , onde  $V' \subseteq V$  e  $A' \subseteq A$ , tal que:
  1.  $V'$  é o conjunto de vértices alcançáveis a partir de  $s \in G$ ,
  2.  $G'$  forma uma árvore de raiz  $s$ ,
  3. para todos os vértices  $v \in V'$ , o caminho simples de  $s$  até  $v$  é um caminho mais curto de  $s$  até  $v$  em  $G$ .

## Relaxamento

- O **relaxamento** de uma aresta  $(u, v)$  consiste em verificar se é possível melhorar o melhor caminho até  $v$  obtido até o momento se passarmos por  $u$ .
- Se isto acontecer,  $p[v]$  e  $Antecessor[v]$  devem ser atualizados.

```

if  $p[v] > p[u] + \text{peso da aresta } (u,v)$ 
then  $p[v] = p[u] + \text{peso da aresta } (u,v)$ 
     $Antecessor[v] := u$ 

```

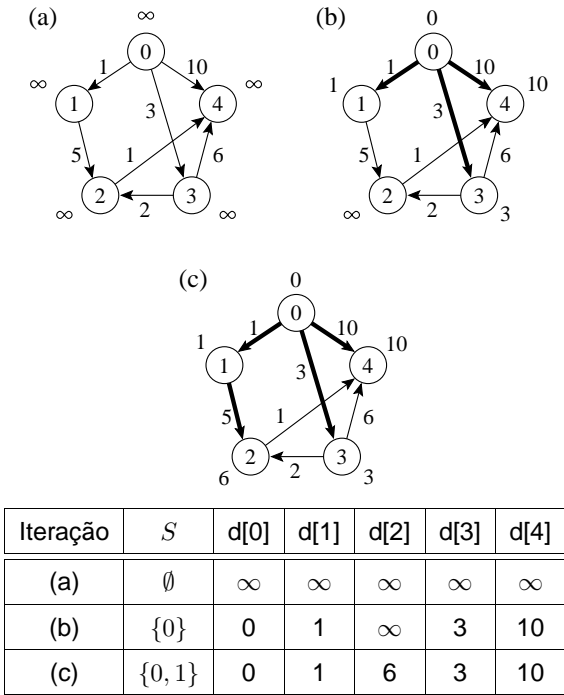


Algoritmo de Dijkstra - 1º Refinamento

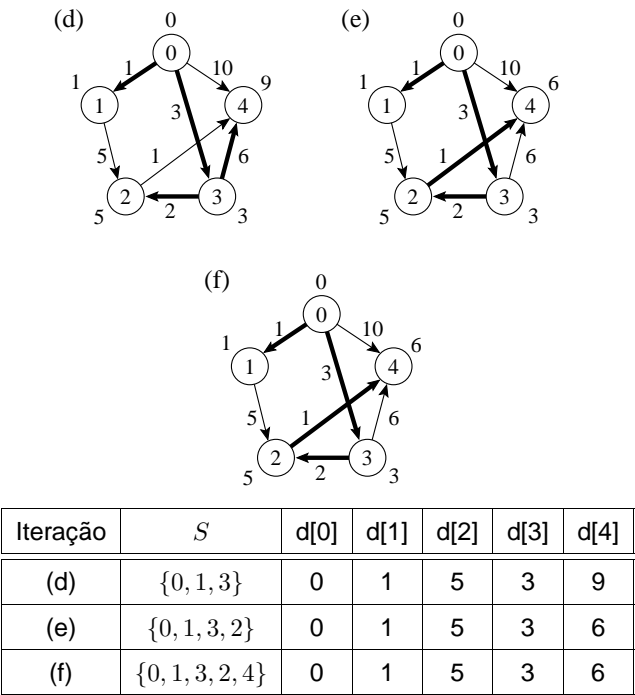
```
procedure Dijkstra (Grafo, Raiz);
1. for v := 0 to Grafo.NumVertices-1 do
2.   p[v] := Infinito;
3.   Antecessor[v] := -1;
4. p[Raiz] := 0;
5. Constrói heap no vetor A;
6. S := ∅;
7. While heap > 1 do
8.   u := RetiraMin(A);
9.   S := S + u
10.  for v ∈ ListaAdjacentes[u] do
11.    if p[v] > p[u] + peso da aresta (u,v)
12.      then p[v] = p[u] + peso da aresta (u,v)
13.        Antecessor[v] := u
```

- Invariante: o número de elementos do *heap* é igual a  $V - S$  no início do anel **while**.
- A cada iteração do **while**, um vértice  $u$  é extraído do *heap* e adicionado ao conjunto  $S$ , mantendo assim o invariante.
- *RetiraMin* obtém o vértice  $u$  com o caminho mais curto estimado até o momento e adiciona ao conjunto  $S$ .
- No anel da linha 10, a operação de relaxamento é realizada sobre cada aresta  $(u, v)$  adjacente ao vértice  $u$ .

Algoritmo de Dijkstra - Exemplo



Algoritmo de Dijkstra - Exemplo



Algoritmo de Dijkstra

- Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na árvore de caminhos mais curtos residem no *heap*  $A$  baseada no campo  $p$ .
- Para cada vértice  $v$ ,  $p[v]$  é o caminho mais curto obtido até o momento, de  $v$  até o vértice raiz.
- O *heap* mantém os vértices, mas a condição do *heap* é mantida pelo caminho mais curto estimado até o momento através do arranjo  $p[v]$ , o *heap* é indireto.
- O arranjo  $Pos[v]$  fornece a posição do vértice  $v$  dentro do *heap*  $A$ , permitindo assim que o vértice  $v$  possa ser acessado a um custo  $O(1)$  para a operação *DiminuiChaveInd*.

## Algoritmo de Dijkstra - Implementação

```

procedure Dijkstra (var Grafo: TipoGrafo; var Raiz: TipoValorVertice);
var Antecessor: array[TipoValorVertice] of integer;
    P          : array[TipoValorVertice] of TipoPeso;
    ItensHeap   : array[TipoValorVertice] of boolean;
    Pos         : array[TipoValorVertice] of TipoValorVertice;
    A           : Vetor;
    u, v        : TipoValorVertice;

```

{— Entram aqui os operadores de uma das implementações de grafos, bem como o operador Constroi da implementação de filas de prioridades, assim como os operadores RefazInd, RetiraMinInd e DiminuiChaveInd do Programa Constroi—}

```

begin { Dijkstra }
  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; {Tamanho do heap}
  p[Raiz] := 0;
  Constroi(A);

```

```

:
:

```

## Porque o Algoritmo de Dijkstra Funciona

- O algoritmo usa uma estratégia gulosa: sempre escolher o vértice mais leve (ou o mais perto) em  $V - S$  para adicionar ao conjunto solução  $S$ ,
- O algoritmo de Dijkstra sempre obtém os caminhos mais curtos, pois cada vez que um vértice é adicionado ao conjunto  $S$  temos que  $p[u] = \delta(Raiz, u)$ .

## Algoritmo de Dijkstra - Implementação

```

:
:
while n >= 1 do {enquanto heap nao vazio}
  begin
    u := RetiraMinInd(A).Chave;
    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 p[v] > p[u] + Peso
              then begin
                p[v] := p[u] + Peso; Antecessor[v] := u;
                DiminuiChaveInd(Pos[v], p[v], A);
                write( 'Caminho: v[ ', v, ' ] v[ ', Antecessor[v], ' ] ',
                  ' d[ ', p[v], ' ] '); readln;
              end;
            end;
          end;
        end;
      end;
    end; { Dijkstra }

```