Algoritmos em Grafos*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Motivação

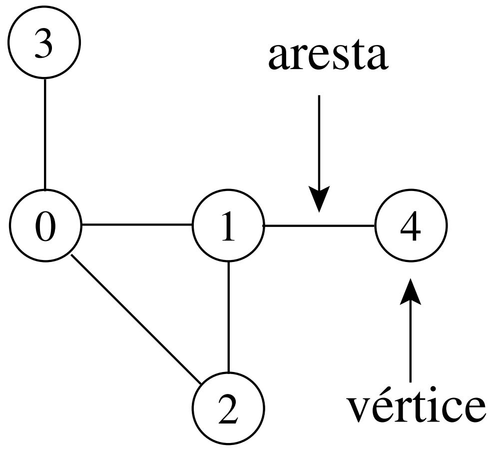
- Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
  - Existe um caminho para ir de um objeto a outro seguindo as conexões?
  - Qual é a menor distância entre um objeto e outro objeto?
  - Quantos outros objetos podem ser alcançados a partir de um determinado 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

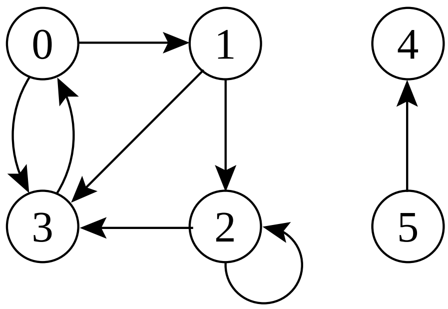
- **Grafo**: conjunto de vértices e arestas.
- **Vértice**: objeto simples que pode ter nome e outros atributos.
- **Aresta**: conexão entre dois vértices.



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

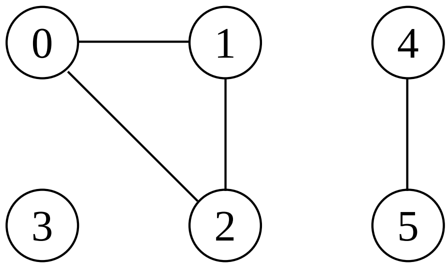
Grafos Direcionados

- 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

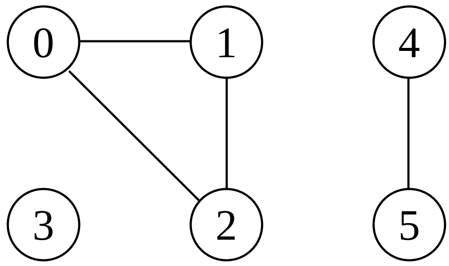
- Um grafo não direcionado G é um par (V, A), onde o conjunto de arestas A é constituído de pares de vértices não ordenados.
  - As arestas (u, v) e (v, u) são consideradas como uma única aresta. A relação de adjacência é simétrica.
  - *Self-loops* não são permitidos.



Grau de um Vértice

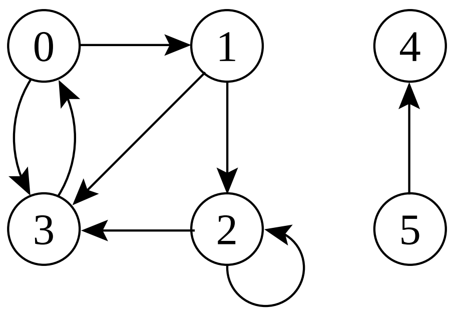
- Em grafos não direcionados:
  - O grau de um vértice é o número de arestas que incidem nele.
  - Um vérice de grau zero é dito **isolado** ou **não conectado**.

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*).

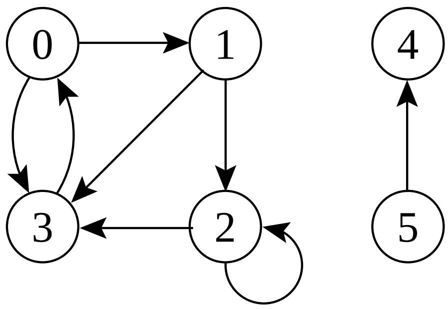
Ex.: O vértice 2 tem *in-degree* 2, *out-degree* 2 e grau 4.



Caminho entre Vértices

- Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma seqüência de vértices $(v_0, v_1, v_2, \ldots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \ldots, k$.
- O comprimento de um caminho é o número de arestas nele, isto é, o caminho contém os vértices $v_0, v_1, v_2, \ldots, v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.
- Se existir um caminho c de x a y então y é **alcançável** a partir de x via c.
- Um caminho é **simples** se todos os vértices do caminho são distintos.

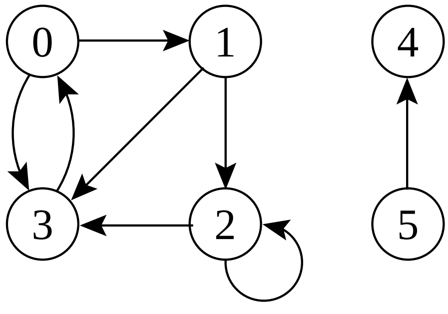
Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



Ciclos

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

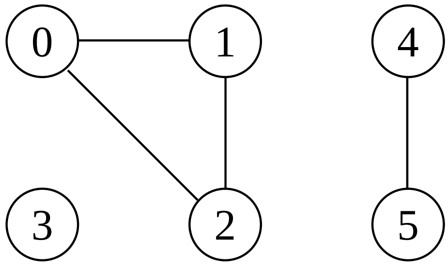
Ex.: O caminho $(0, 1, 2, 3, 0)$ forma um ciclo. O caminho$(0, 1, 3, 0)$ forma o mesmo ciclo que os caminhos $(1, 3, 0, 1)$ e $(3, 0, 1, 3)$.



Ciclos

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

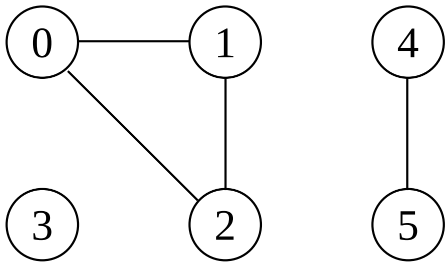
Ex.: O caminho $(0, 1, 2, 0)$ é um ciclo.



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 $\{3\}$.



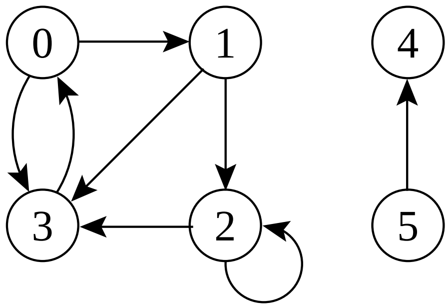
Componentes Fortemente Conectados

- 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".

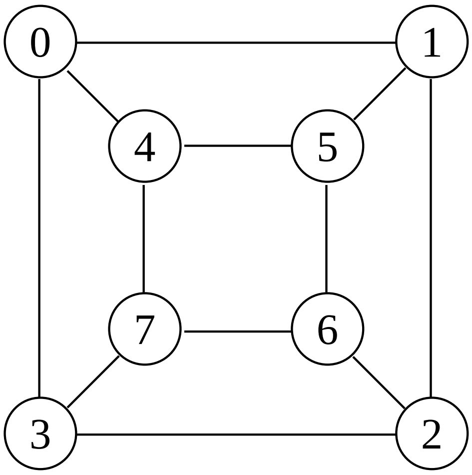
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.

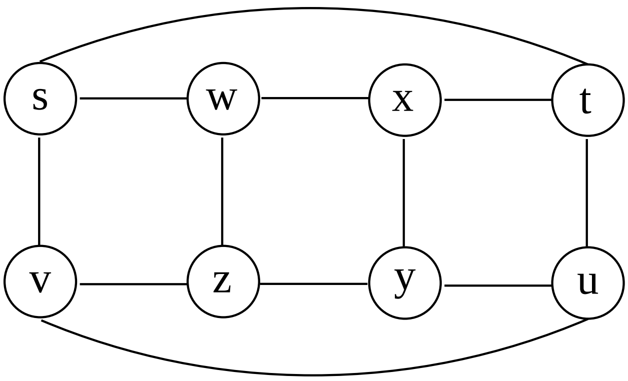
Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.



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 $(f(u), f(v)) \in A'$.

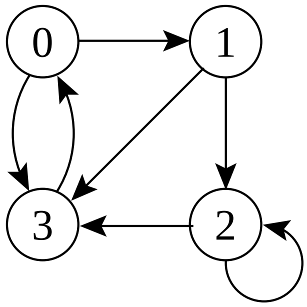


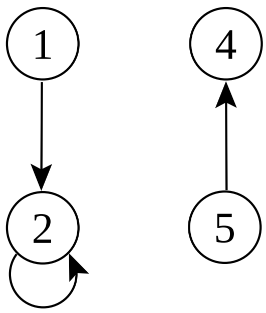


Subgrafos

- 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 $A' = \{(u, v) \in A | u, v \in V'\}$.

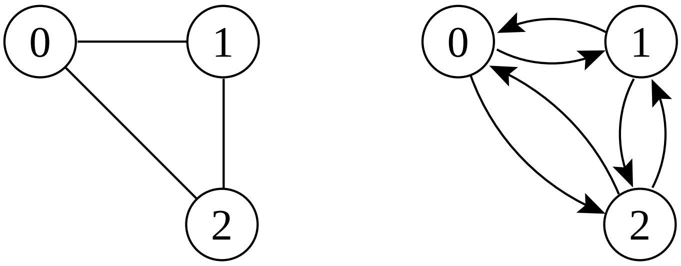
Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.





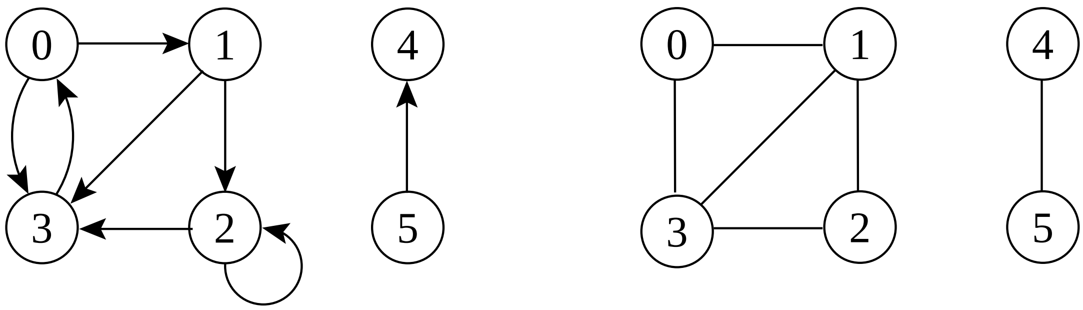
Versão Direcionada de um Grafo Não Direcionado

- 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

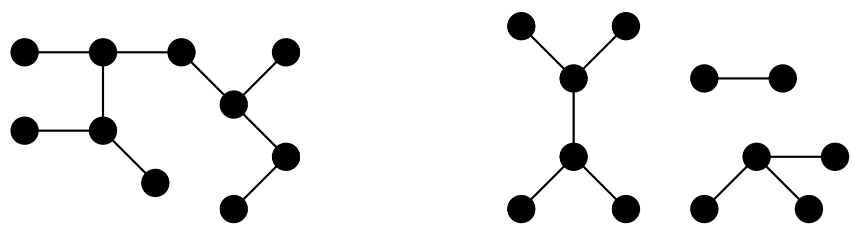
- **Grafo ponderado**: possui pesos associados às arestas.
- **Grafo bipartido**: grafo não direcionado $G = (V, A)$ no qual V pode ser particionado em dois conjuntos V_1 e V_2 tal que $(u, v) \in A$ implica que $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$ (todas as arestas ligam os dois conjuntos 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. Criar um grafo vazio.
2. Inserir uma aresta no grafo.
3. Verificar se existe determinada aresta no grafo.
4. Obter a lista de vértices adjacentes a determinado vértice.
5. Retirar uma aresta do grafo.
6. Imprimir um grafo.
7. Obter o número de vértices do grafo.
8. Obter o transposto de um grafo direcionado.
9. Obter a aresta de menor peso de um grafo.

Operação "Obter Lista de Adjacentes"

1. Verificar se a lista de adjacentes de um vértice v está vazia. Retorna *true* se a lista de adjacentes de v está vazia.

2. Obter o primeiro vértice adjacente a um vértice v, caso exista. Retorna o endereço do primeiro vértice na lista de adjacentes de v.

3. Obter o próximo vértice adjacente a um vértice v, caso exista. Retorna a próxima aresta que o vértice v participa.

Implementação da Operação “Obter Lista de Adjacentes”

- É comum encontrar um pseudo comando do tipo:

  for u $\in$ iista de adjacentes (v) **do** { faz algo com u }

- O trecho de programa abaixo apresenta um possível refinamento do pseudo comando acima.

```
if (!grafo.listaAdjVazia (v)) {
  Aresta aux = grafo.primeiroListaAdj (v);
  while (aux != null) {
    int u = aux.vertice2 (); int peso = aux.peso ();
    aux = grafo.proxAdj (v);
  }
}
```

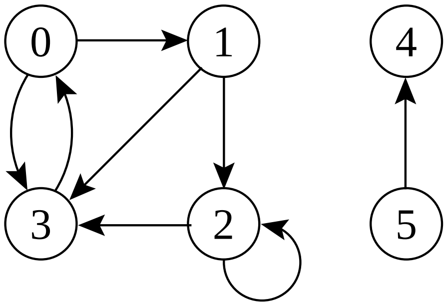
Matriz de Adjacência

- 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*.

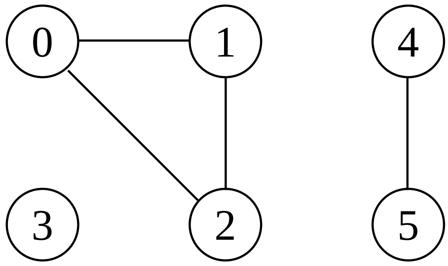
- Se não existir uma aresta de i para j então é necessário utilizar um valor que não possa ser usado como rótulo ou peso.

Matriz de Adjacência - Exemplo



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

(a)



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

(b)

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.

```java
package cap7.matrizadj;
public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso; }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private int mat[][]; // pesos do tipo inteiro
  private int numVertices;
  private int pos[]; // posição atual ao se percorrer os adjs de um vértice v
  public Grafo (int numVertices) {
    this.mat = new int[numVertices][numVertices];
    this.pos = new int[numVertices]; this.numVertices = numVertices;
    for (int i = 0; i < this.numVertices; i++) {
      for (int j = 0; j < this.numVertices; j++) this.mat[i][j] = 0;
      this.pos[i] = -1;
    }
  }
  public void insereAresta (int v1, int v2, int peso) {
    this.mat[v1][v2] = peso; }
  public boolean existeAresta (int v1, int v2) {
    return (this.mat[v1][v2] > 0);
  }
```

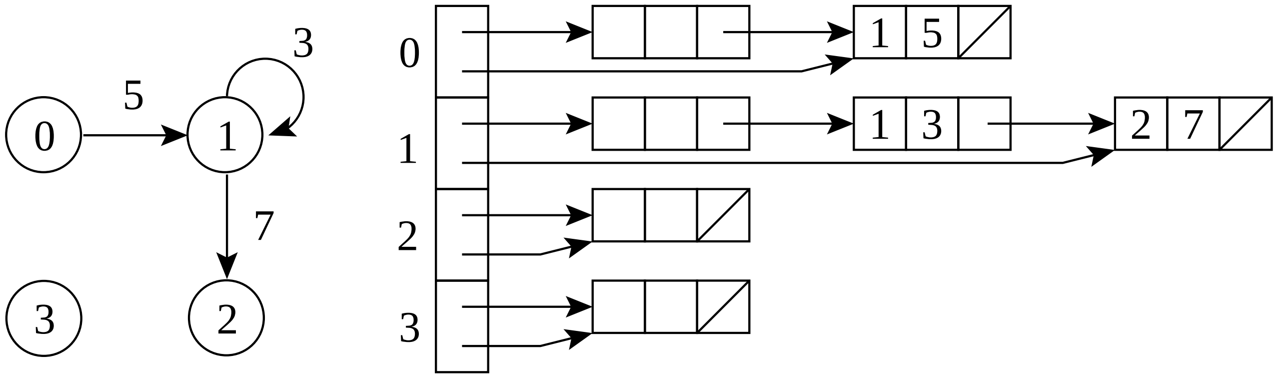
Matriz de Adjacência - Implementação

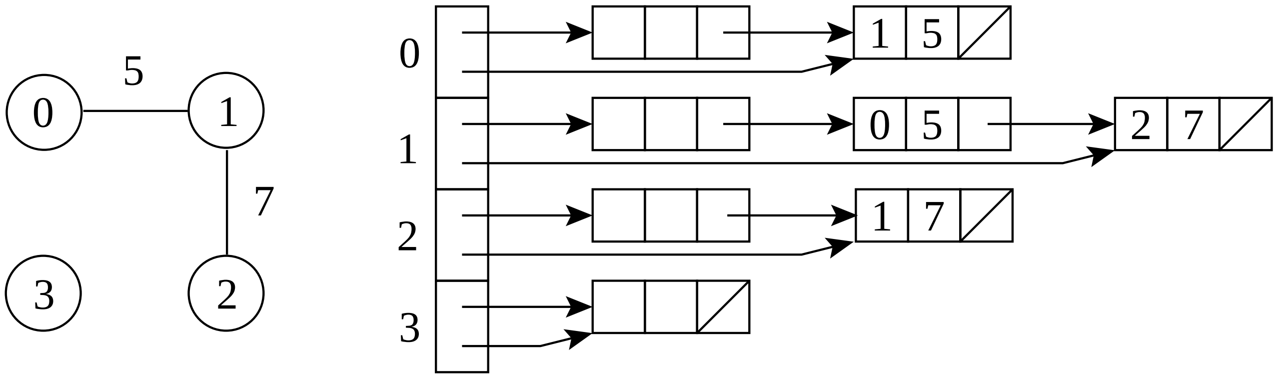
```
public boolean listaAdjVazia (int v) {
  for (int i =0; i < this.numVertices; i++)
    if (this.mat[v][i] > 0) return false;
  return true;
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  this.pos[v] = −1; return this.proxAdj (v);
}
public Aresta proxAdj (int v) {
 // Retorna a próxima aresta que o vértice v participa ou
 // null se a lista de adjacência de v estiver no fim
  this.pos[v] ++;
  while ((this.pos[v] < this.numVertices) &&
         (this.mat[v][this.pos[v]] == 0)) this.pos[v]++;
  if (this.pos[v] == this.numVertices) return null;
  else return new Aresta (v, this.pos[v], this.mat[v][this.pos[v]]);
}
public Aresta retiraAresta (int v1, int v2) {
  if (this.mat[v1][v2] == 0) return null; // Aresta não existe
  else {
    Aresta aresta = new Aresta (v1, v2, this.mat[v1][v2]);
    this.mat[v1][v2] = 0; return aresta;
  }
}
```

Matriz de Adjacência - Implementação

```
  public void imprime () {
    System.out.print ("    ");
    for (int i = 0; i < this.numVertices; i++)
      System.out.print (i + "    ");
    System.out.println ();
    for (int i = 0; i < this.numVertices; i++) {
      System.out.print (i + "  ");
      for (int j = 0; j < this.numVertices; j++)
        System.out.print (this.mat[i][j] + "    ");
      System.out.println ();
    }
  }
  public int numVertices () {
    return this.numVertices;
  }
}
```

Listas de Adjacência usando Estruturas Auto-Referenciadas





- 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 **esparsos**, 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 Estruturas Auto-Referenciadas - Implementação

- A seguir apresentamos a implementação do **tipo abstrato de dados grafo** utilizando listas encadeadas implementadas por meio de estruturas auto-referenciadas para as sete primeiras operações definidas anteriormente.
- A classe $Aresta$ representa as informações de uma aresta para que os usuários da classe $Grafo$ possam acessá-las.
- A classe $Celula$ é utilizada para representar uma entrada na lista de adjacência de um vértice do grafo.
- O método $equals$ é usado para verificar se um vértice qualquer v é adjacente a um outro vértice u ao se percorrer a lista de adjacentes de u.

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

```
package cap7.listaadj.autoreferencia;
import cap3.autoreferencia.Lista;

public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private static class Celula {
    int vertice, peso;
    public Celula (int v, int p) {this.vertice = v; this.peso = p;}
    public boolean equals (Object obj) {
      Celula item = (Celula) obj;
      return (this.vertice == item.vertice);
    }
  }
  private Lista adj[];
  private int numVertices;
  public Grafo (int numVertices) {
    this.adj = new Lista[numVertices]; this.numVertices = numVertices;
    for (int i = 0; i < this.numVertices; i++) this.adj[i] = new Lista();
  }
  public void insereAresta (int v1, int v2, int peso) {
    Celula item = new Celula (v2, peso);
    this.adj[v1].insere (item);
  }
```

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

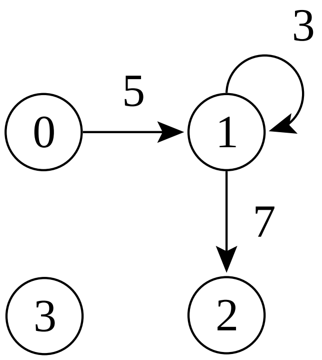
```
public boolean existeAresta (int v1, int v2) {
  Celula item = new Celula (v2, 0);
  return (this.adj[v1].pesquisa (item) != null);
}
public boolean listaAdjVazia (int v) {
  return this.adj[v].vazia ();
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  Celula item = (Celula) this.adj[v].primeiro ();
  return item != null ? new Aresta (v, item.vertice, item.peso): null;
}
public Aresta proxAdj (int v) {
  // Retorna a próxima aresta que o vértice v participa ou
  // null se a lista de adjacência de v estiver no fim
  Celula item = (Celula) this.adj[v].proximo ();
  return item != null ? new Aresta (v, item.vertice, item.peso): null;
}
public Aresta retiraAresta (int v1, int v2) throws Exception {
  Celula chave = new Celula (v2, 0);
  Celula item = (Celula) this.adj[v1].retira (chave);
  return item != null ? new Aresta (v1, v2, item.peso): null;
}
```

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

```
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      Celula item = (Celula) this.adj[i].primeiro ();
      while (item != null) {
        System.out.println ("  " + item.vertice + " (" +item.peso+ ")");
        item = (Celula) this.adj[i].proximo ();
      }
    }
  }
  public int numVertices () {
    return this.numVertices;
  }
}
```

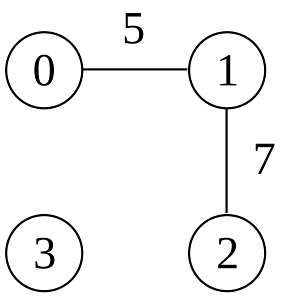
Listas de Adjacência usando Arranjos

(a)



		cab	prox	peso
V	0	4	4	
	1	6	5	
	2	2	0	
	3	3	0	
A	4	1	0	5
	5	1	6	3
	6	2	0	7

(b)



		cab	prox	peso
V	0	4	4	
	1	6	5	
	2	7	7	
	3	3	0	
A	4	1	0	5
	5	0	6	5
	6	2	0	7
	7	1	0	7

- *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

```
package cap7.listaadj.arranjo;
public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private int cab[], prox[], peso[];
  private int pos[]; // posição atual ao se percorrer os adjs de um vértice v
  private int numVertices, proxDisponivel;
  public Grafo (int numVertices, int numArestas) {
    int tam = numVertices + 2*numArestas;
    this.cab  = new int[tam]; this.prox = new int[tam];
    this.peso = new int[tam]; this.numVertices = numVertices;
    this.pos = new int[this.numVertices];
    for (int i = 0; i < this.numVertices; i++) {
      this.prox[i] = 0;
      this.cab[i] = i;
      this.peso[i] = 0;
      this.pos[i] = i;
    }
    this.proxDisponivel = this.numVertices;
  }
```

Listas de Adjacência usando Arranjos - Implementação

```
public void insereAresta (int v1, int v2, int peso) {
  if (this.proxDisponivel == this.cab.length)
    System.out.println ("Nao ha espaco disponivel para a aresta");
  else {
    int ind = this.proxDisponivel++;
    this.prox[this.cab[v1]] = ind;
    this.cab[ind] = v2; this.cab[v1] = ind;
    this.prox[ind] = 0; this.peso[ind] = peso;
  }
}
public boolean existeAresta (int v1, int v2) {
  for (int i = this.prox[v1]; i != 0; i = this.prox[i])
    if (this.cab[i] == v2) return true;
  return false;
}
public boolean listaAdjVazia (int v) {
  return (this.prox[v] == 0);
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  this.pos[v] = v;
  return this.proxAdj (v);
}
public Aresta proxAdj (int v) {
  // Retorna a próxima aresta que o vértice v participa ou
  // null se a lista de adjacência de v estiver no fim
  this.pos[v] = this.prox[this.pos[v]];
  if (this.pos[v] == 0) return null;
  else return new Aresta (v,this.cab[pos[v]],this.peso[pos[v]]);
}
```

Listas de Adjacência usando Arranjos - Implementação

```
  public Aresta retiraAresta (int v1, int v2) {
    int i;
    for (i = v1; this.prox[i] != 0; i = this.prox[i])
      if (this.cab[this.prox[i]] == v2) break;
    int ind = this.prox[i];
    if (this.cab[ind] == v2) { // encontrou aresta
      Aresta aresta = new Aresta(v1, v2, this.peso[ind]);
      this.cab[ind] = this.cab.length; // marca como removido
      if (this.prox[ind] == 0) this.cab[v1] = i; // último vértice
      this.prox[i] = this.prox[ind];
      return aresta;
    } else return null;
  }
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      for (int j = this.prox[i]; j != 0; j = this.prox[j])
        System.out.println ("  " + this.cab[j]+" (" +this.peso[j]+ ")");
    }
  }
  public int numVertices () { return this.numVertices; }
}
```

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.

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

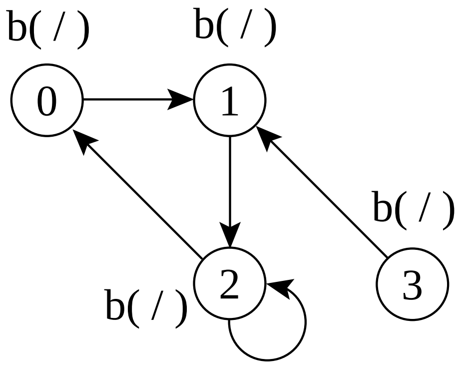
```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class BuscaEmProfundidade {
  public static final byte branco = 0;
  public static byte cinza        = 1;
  public static byte preto        = 2;
  private int d[], t[], antecessor[];
  private Grafo grafo;

  public BuscaEmProfundidade (Grafo grafo) {
    this.grafo = grafo; int n = this.grafo.numVertices();
    d = new int[n]; t = new int[n]; antecessor = new int[n];
  }
  private int visitaDfs (int u, int tempo, int cor[]) {
    cor[u] = cinza; this.d[u] = ++tempo;
    if (!this.grafo.listaAdjVazia (u)) {
      Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
      while (a != null) {
        int v = a.v2 ();
        if (cor[v] == branco) {
          this.antecessor[v] = u;
          tempo = this.visitaDfs (v, tempo, cor);
        }
        a = this.grafo.proxAdj (u);
      }
    }
    cor[u] = preto; this.t[u] = ++tempo;
    return tempo;
  }
```

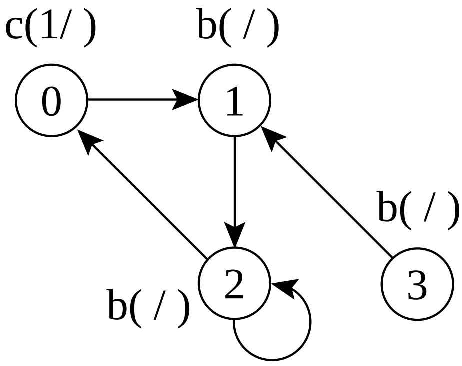
Busca em Profundidade - Implementação

```
  public void buscaEmProfundidade () {
    int tempo = 0; int cor[] = new int[this.grafo.numVertices ()];
    for (int u = 0; u < grafo.numVertices (); u++) {
      cor[u] = branco; this.antecessor[u] = −1;
    }
    for (int u = 0; u < grafo.numVertices (); u++)
      if (cor[u] == branco) tempo = this.visitaDfs (u, tempo, cor);
  }
  public int d (int v) { return this.d[v]; }
  public int t (int v) { return this.t[v]; }
  public int antecessor (int v) { return this.antecessor[v]; }
}
```

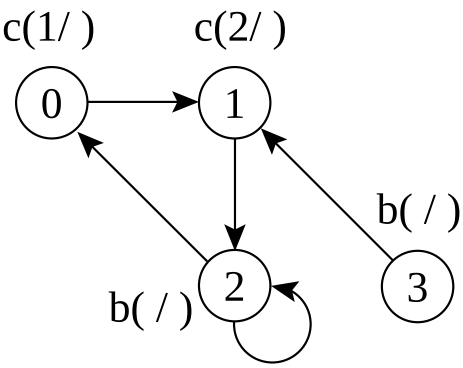
Busca em Profundidade - Exemplo



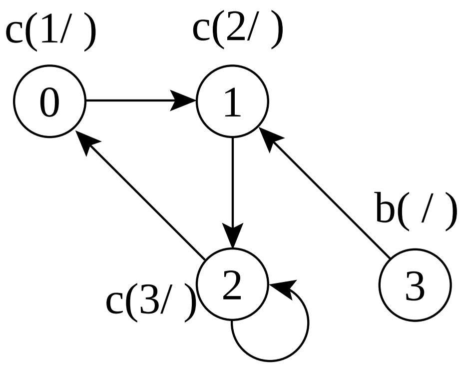
(a)



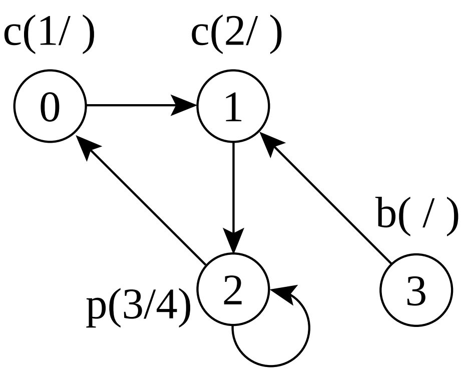
(b)



(c)

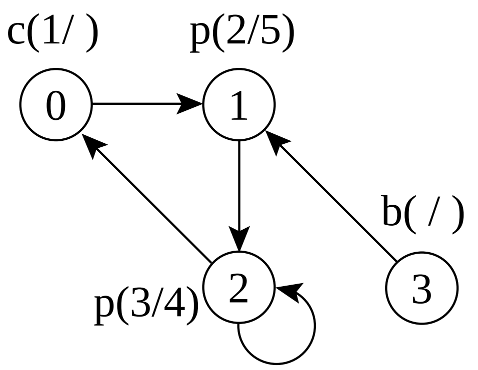


(d)

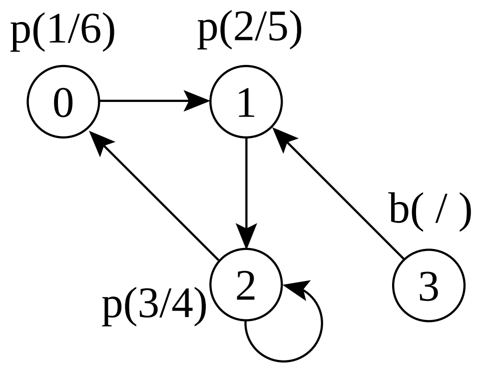


(e)

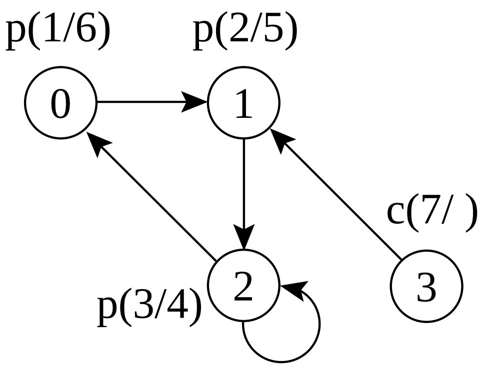
Busca em Profundidade - Exemplo



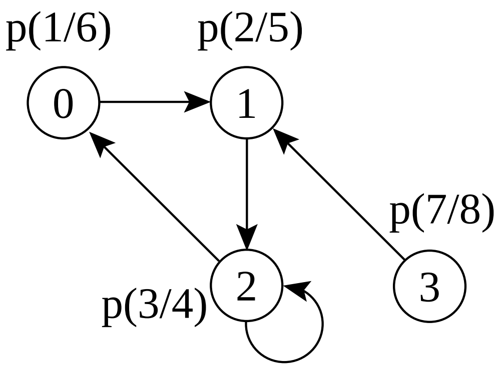
(f)



(g)



(h)



(i)

Busca em Profundidade - Análise

- Os dois anéis do método $buscaEmProfundidade$ têm custo $O(|V|)$ cada um, a menos da chamada do método $visitaDfs$(u, $tempo$, cor) no segundo anel.

- O método $visitaDfs$ é chamado exatamente uma vez para cada vértice $u \in V$, desde que $visitaDfs$ seja chamado apenas para vértices brancos, e a primeira ação é pintar o vértice de cinza.

- Durante a execução de $visitaDfs$(u, $tempo$, cor), 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 do método $buscaEmProfundidade$ é $O(|V| + |A|)$.

Classificação de Arestas

- Existem:

1. **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).

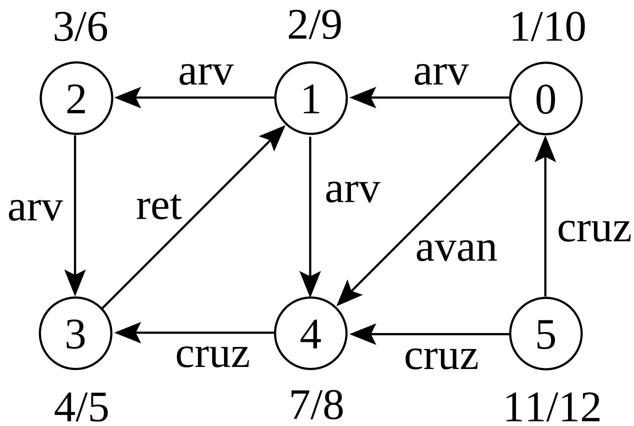
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).

3. **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.

4. **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

- 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.

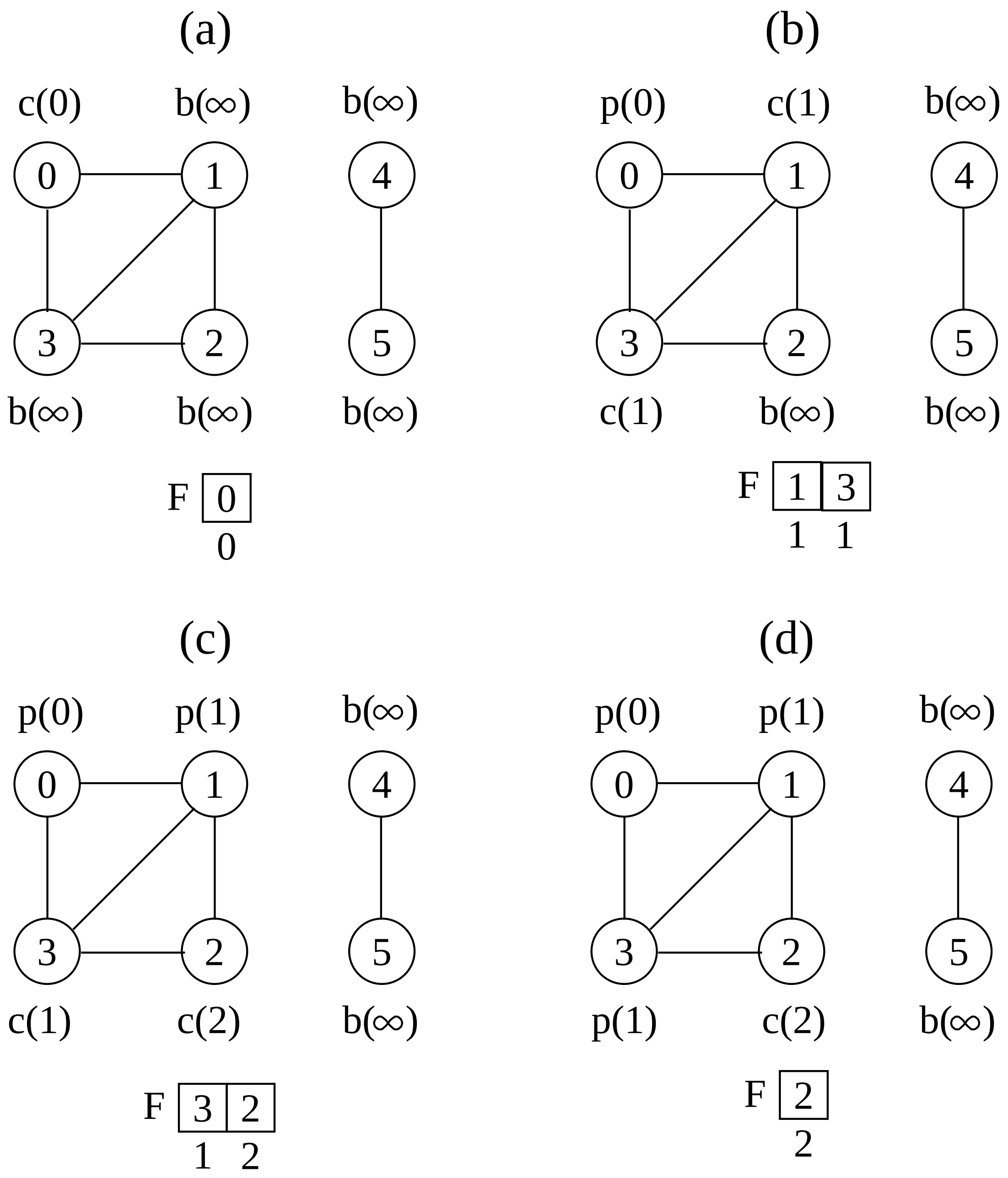
Busca em Largura - Implementação

```
package cap7;
import cap3.autoreferencia.Fila;
import cap7.listaadj.autoreferencia.Grafo;
public class BuscaEmLargura {
  public static final byte branco = 0;
  public static byte cinza       = 1;
  public static byte preto       = 2;
  private int d[], antecessor[];
  private Grafo grafo;
  public BuscaEmLargura (Grafo grafo) {
    this.grafo = grafo; int n = this.grafo.numVertices();
    this.d = new int[n]; this.antecessor = new int[n];
  }
  private void visitaBfs (int u, int cor[]) throws Exception {
    cor[u] = cinza; this.d[u] = 0;
    Fila fila = new Fila (); fila.enfileira (new Integer (u));
    while (!fila.vazia ()) {
      Integer aux = (Integer)fila.desenfileira (); u = aux.intValue();
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
        while (a != null) {
          int v = a.v2 ();
          if (cor[v] == branco) {
            cor[v] = cinza; this.d[v] = this.d[u] + 1;
            this.antecessor[v] = u; fila.enfileira (new Integer (v));
          }
          a = this.grafo.proxAdj (u);
        }
      }
      cor[u] = preto;
    }
  }
```

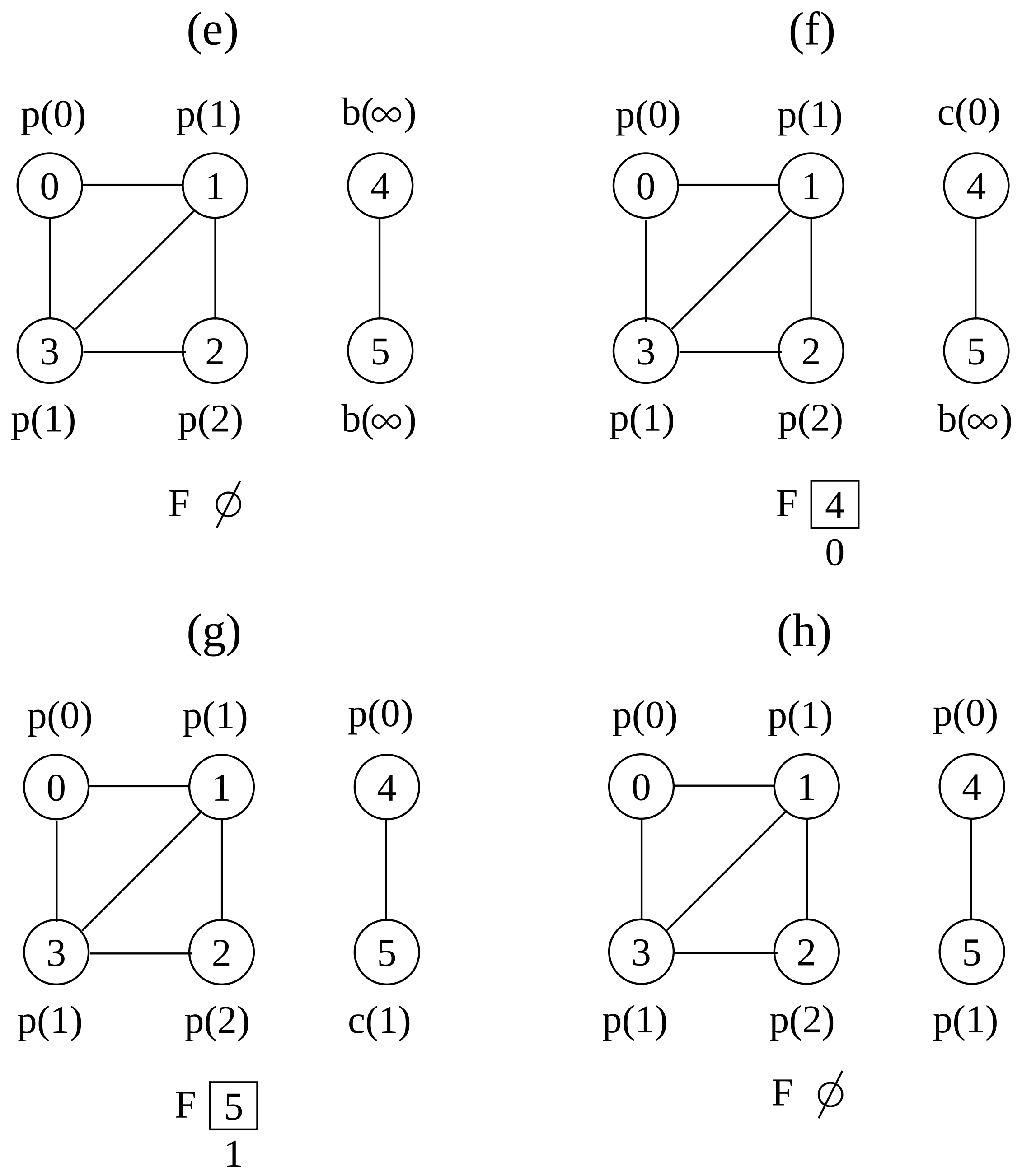
Busca em Largura - Implementação

```
  public void buscaEmLargura () throws Exception {
    int cor[] = new int[this.grafo.numVertices ()];
    for (int u = 0; u < grafo.numVertices (); u++) {
      cor[u] = branco; this.d[u] = Integer.MAX_VALUE;
      this.antecessor[u] = −1;
    }
    for (int u = 0; u < grafo.numVertices (); u++)
      if (cor[u] == branco) this.visitaBfs (u, cor);
  }
  public int d (int v) { return this.d[v]; }
  public int antecessor (int v) { return this.antecessor[v]; }
}
```

Busca em Largura - Exemplo



Busca em Largura - Exemplo



Busca em Largura - Análise (para listas de adjacência)

- O custo de inicialização do primeiro anel no método $buscaEmLargura$ é $O(|V|)$.
- O custo do segundo anel é também $O(|V|)$.
- Método $visitaBfs$: 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 $O(|V| + |A|)$.

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.

```
public void imprimeCaminho (int origem, int v) {
  if (origem == v) System.out.println (origem);
  else if (this.antecessor[v] == -1)
    System.out.println ("Nao existe caminho de " + origem + " ate " + v);
  else {
    imprimeCaminho (origem, this.antecessor[v]);
    System.out.println (v);
  }
}
```

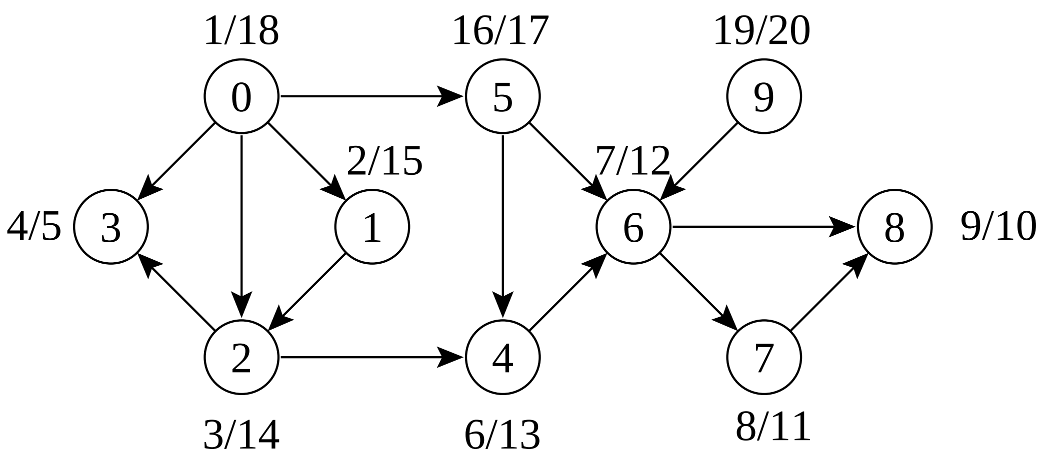
Ordenação Topológica

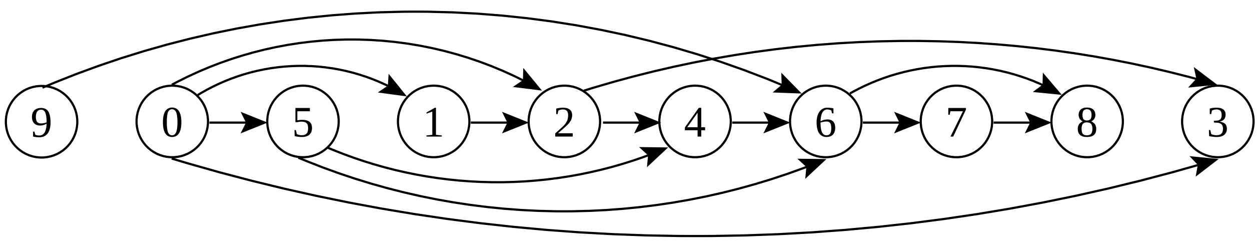
- Ordenação linear de todos os vértices, tal que se G contém uma aresta (u, v) então u aparece antes de v.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.
- Pode ser feita usando a busca 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. Aplicar a busca em profundidade no grafo 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. Retornar 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

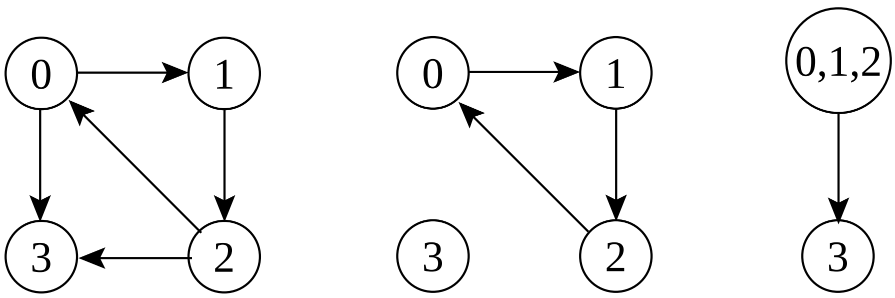
- Basta inserir uma chamada ao método $inserePrimeiro$ no método $buscaDfs$, logo após o momento em que o tempo de término $t[u]$ é obtido e o vértice é pintado de $preto$.
- Ao final, basta retornar a lista obtida.

```
// Insere antes do primeiro item da lista
public void inserePrimeiro (Object item) {
  Celula aux = this.primeiro.prox;
  this.primeiro.prox = new Celula ();
  this.primeiro.prox.item = item;
  this.primeiro.prox.prox = aux;
}
```

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.



(a) (b) (c)

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

1. Aplicar a busca em profundidade no grafo G para obter os tempos de término $t[u]$ para cada vértice u.

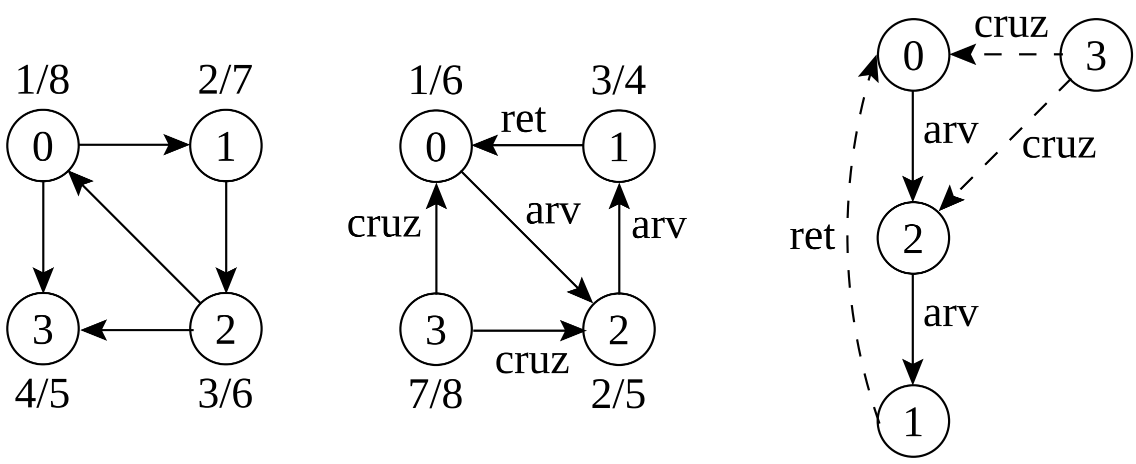
2. Obter G^T.

3. Aplicar a busca em profundidade no grafo G^T, realizando a busca a partir do vértice de maior $t[u]$ obtido na linha 1. Se a busca em profundidade não alcançar todos os vértices, inicie uma nova busca em profundidade a partir do vértice de maior $t[u]$ dentre os vértices restantes.

4. Retornar os vértices de cada árvore da floresta obtida na busca em profundidade na linha 3 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).



(a) (b) (c)

Componentes Fortemente Conectados - Implementação

```
public Grafo grafoTransposto () {
  Grafo grafoT = new Grafo (this.numVertices);
  for (int v = 0; v < this.numVertices; v++)
    if (!this.listaAdjVazia (v)) {
      Aresta adj = this.primeiroListaAdj (v);
      while (adj != null) {
        grafoT.insereAresta (adj.v2 (), adj.v1 (), adj.peso ());
        adj = this.proxAdj (v); }
    }
  return grafoT;
}
```

Componentes Fortemente Conectados - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class Cfc {
  private static class TempoTermino {
    private int numRestantes, t[];
    private boolean restantes[];
    public TempoTermino (int numVertices) {
      t = new int[numVertices];
      restantes = new boolean[numVertices];
      numRestantes = numVertices;
    }
    public int maxTT () {
      int vMax = 0;
      while (!this.restantes[vMax]) vMax++;
      for (int i = 0; i < this.t.length; i++) {
        if (this.restantes[i]) {
          if (this.t[i] > this.t[vMax]) vMax = i;
        }
      }
      return vMax;
    }
  }
  private Grafo grafo;
  public Cfc (Grafo grafo) {
    this.grafo = grafo;
  }
```

Componentes Fortemente Conectados - Implementação

```
  private void visitaDfs (Grafo grafo, int u, TempoTermino tt) {
    tt.restantes[u] = false; tt.numRestantes--;
    System.out.println ("  Vertice: "+u);
    if (!grafo.listaAdjVazia (u)) {
      Grafo.Aresta a = grafo.primeiroListaAdj (u);
      while (a != null) {
        int v = a.v2 ();
        if (tt.restantes[v]) { this.visitaDfs (grafo, v, tt); }
        a = grafo.proxAdj (u);
      }
    }
  }
  public void obterCfc () {
    BuscaEmProfundidade dfs = new BuscaEmProfundidade (this.grafo);
    dfs.buscaEmProfundidade ();
    TempoTermino tt = new TempoTermino (this.grafo.numVertices ());
    for (int u = 0; u < this.grafo.numVertices (); u++) {
      tt.t[u] = dfs.t (u); tt.restantes[u] = true;
    }
    Grafo grafoT = this.grafo.grafoTransposto ();
    while (tt.numRestantes > 0) {
      int vRaiz = tt.maxTT ();
      System.out.println ("Raiz da proxima arvore: " + vRaiz);
      this.visitaDfs (grafoT, vRaiz, tt);
    }
  }
}
```

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 $O(|V| + |A|)$.

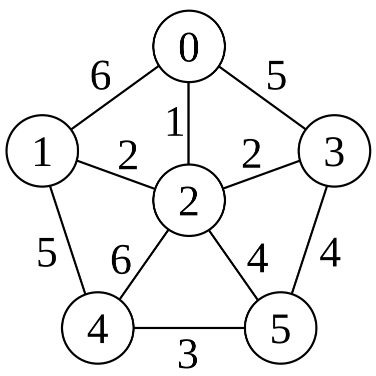
Árvore Geradora Mínima - Aplicação

- 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 $T \subseteq A$ que conecta todos os vértices de G e cujo peso total $p(T) = \sum_{(u,v)\in T} p(u, v)$ é minimizado.

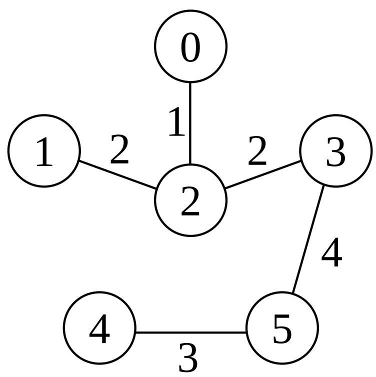
Árvore Geradora Mínima (AGM)

- 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.



(a)



(b)

AGM - Algoritmo Genérico

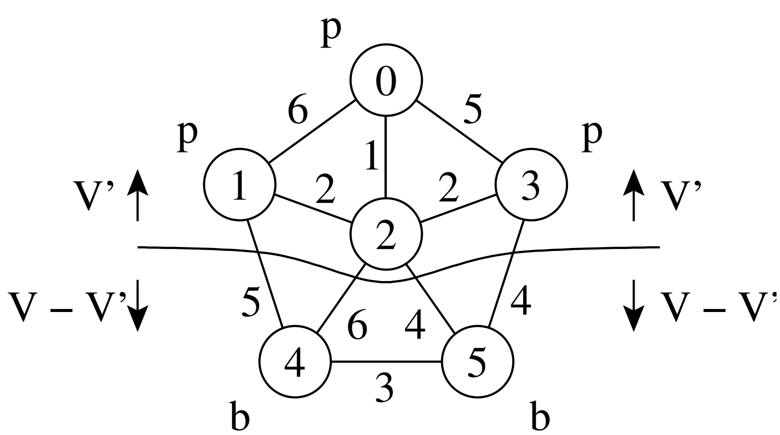
- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração, S é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a S uma aresta (u, v) que não viola o invariante. (u, v) é chamada de uma **aresta segura**.

void GenericoAGM
  $S = \emptyset$;
  while (S não constitui uma árvore geradora mínima)
    (u, v) = seleciona (A);
    if (aresta (u, v) é segura para S) $S = S + \{(u, v)\}$
  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*.



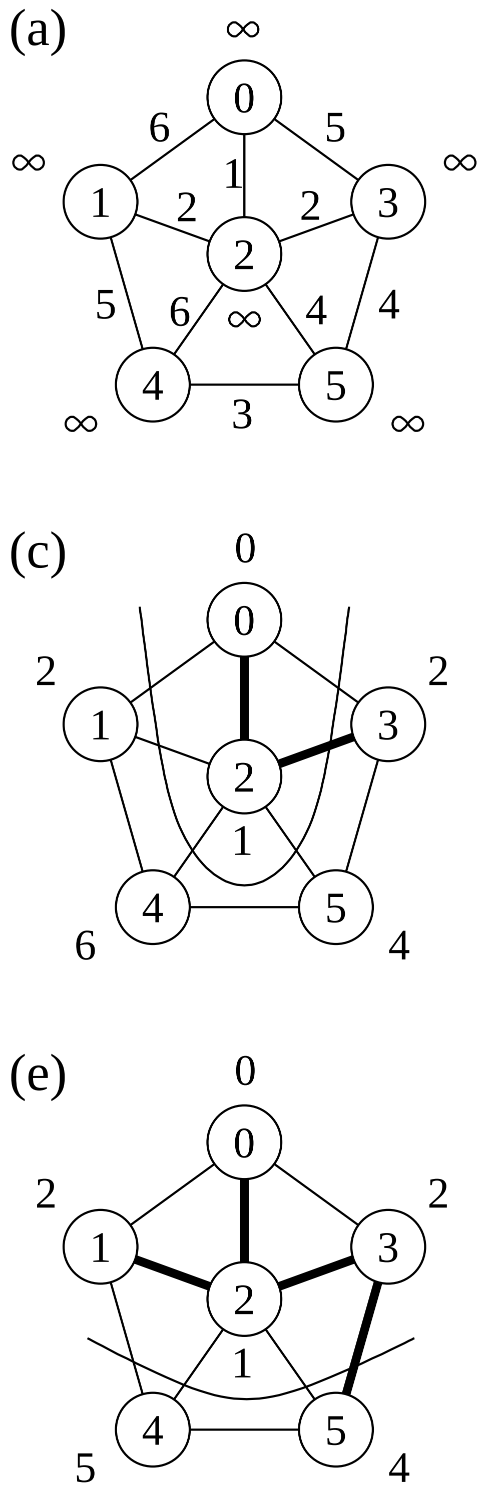
AGM - Teorema para reconhecer arestas seguras

- 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 - Heap Indireto

```
package cap7;
public class FPHeapMinIndireto {
  private double p[];
  private int n, pos[], fp[];

  public FPHeapMinIndireto (double p[], int v[]) {
    this.p = p; this.fp = v; this.n = this.fp.length-1;
    this.pos = new int[this.n];
    for (int u = 0; u < this.n; u++) this.pos[u] = u+1;
  }
  public void refaz (int esq, int dir) {
    int j = esq * 2; int x = this.fp[esq];
    while (j <= dir) {
      if ((j < dir) && (this.p[fp[j]] > this.p[fp[j + 1]])) j++;
      if (this.p[x] <= this.p[fp[j]]) break;
      this.fp[esq] = this.fp[j]; this.pos[fp[j]] = esq;
      esq = j; j = esq * 2;
    }
    this.fp[esq] = x; this.pos[x] = esq;
  }
  public void constroi () {
    int esq = n / 2 + 1;
    while (esq > 1) { esq--; this.refaz (esq, this.n); }
  }
  public int retiraMin () throws Exception {
    int minimo;
    if (this.n < 1) throw new Exception ("Erro: heap vazio");
    else {
      minimo = this.fp[1]; this.fp[1] = this.fp[this.n];
      this.pos[fp[this.n--]] = 1; this.refaz (1, this.n);
    }
    return minimo;
  }
```

Algoritmo de Prim - Heap Indireto

```
public void diminuiChave (int i, double chaveNova) throws Exception {
  i = this.pos[i]; int x = fp[i];
  if (chaveNova < 0)
    throw new Exception ("Erro: chaveNova com valor incorreto");
  this.p[x] = chaveNova;
  while ((i > 1) && (this.p[x] <= this.p[fp[i / 2]])) {
    this.fp[i] = this.fp[i / 2]; this.pos[fp[i / 2]] = i; i /= 2;
  }
  this.fp[i] = x; this.pos[x] = i;
}
boolean vazio () { return this.n == 0; }
```

- O programa acima apresenta a classe $FPHeapMinIndireto$ com as estruturas de dados e as operações necessárias para operar com um *heap* indireto.
- O arranjo $pos[v]$ fornece a posição do vértice v dentro do *heap fp*, permitindo assim que o vértice v possa ser acessado a um custo $O(1)$.
- O acesso ao vértice v é necessário para a operação $diminuiChave$.

Algoritmo de Prim - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class AgmPrim {
  private int antecessor[];
  private double p[];
  private Grafo grafo;
  public AgmPrim (Grafo grafo) { this.grafo = grafo; }
  public void obterAgm (int raiz) throws Exception {
    int n = this.grafo.numVertices();
    this.p = new double[n]; // peso dos vértices
    int vs[] = new int[n+1]; // vértices
    boolean itensHeap[] = new boolean[n]; this.antecessor = new int[n];
    for (int u = 0; u < n; u ++) {
      this.antecessor[u] = –1;
      p[u] = Double.MAX_VALUE; // ∞
      vs[u+1] = u; // Heap indireto a ser construído
      itensHeap[u] = true;
    }
```

Algoritmo de Prim - Implementação

```
    p[raiz] = 0;
    FPHeapMinIndireto heap = new FPHeapMinIndireto (p, vs);
    heap.constroi ();
    while (!heap.vazio ()) {
      int u = heap.retiraMin (); itensHeap[u] = false;
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta adj = grafo.primeiroListaAdj (u);
        while (adj != null) {
          int v = adj.v2 ();
          if (itensHeap[v] && (adj.peso () < this.peso (v))) {
            antecessor[v] = u; heap.diminuiChave (v, adj.peso ());
          }
          adj = grafo.proxAdj (u);
        }
      }
    }
  }

  public int antecessor (int u) { return this.antecessor[u]; }
  public double peso (int u) { return this.p[u]; }

  public void imprime () {
    for (int u = 0; u < this.p.length; u++)
      if (this.antecessor[u] != -1)
        System.out.println ("(" +antecessor[u]+ "," +u+ ") -- p:" +
                            peso (u));
  }
}
```

Algoritmo de Prim - Implementação

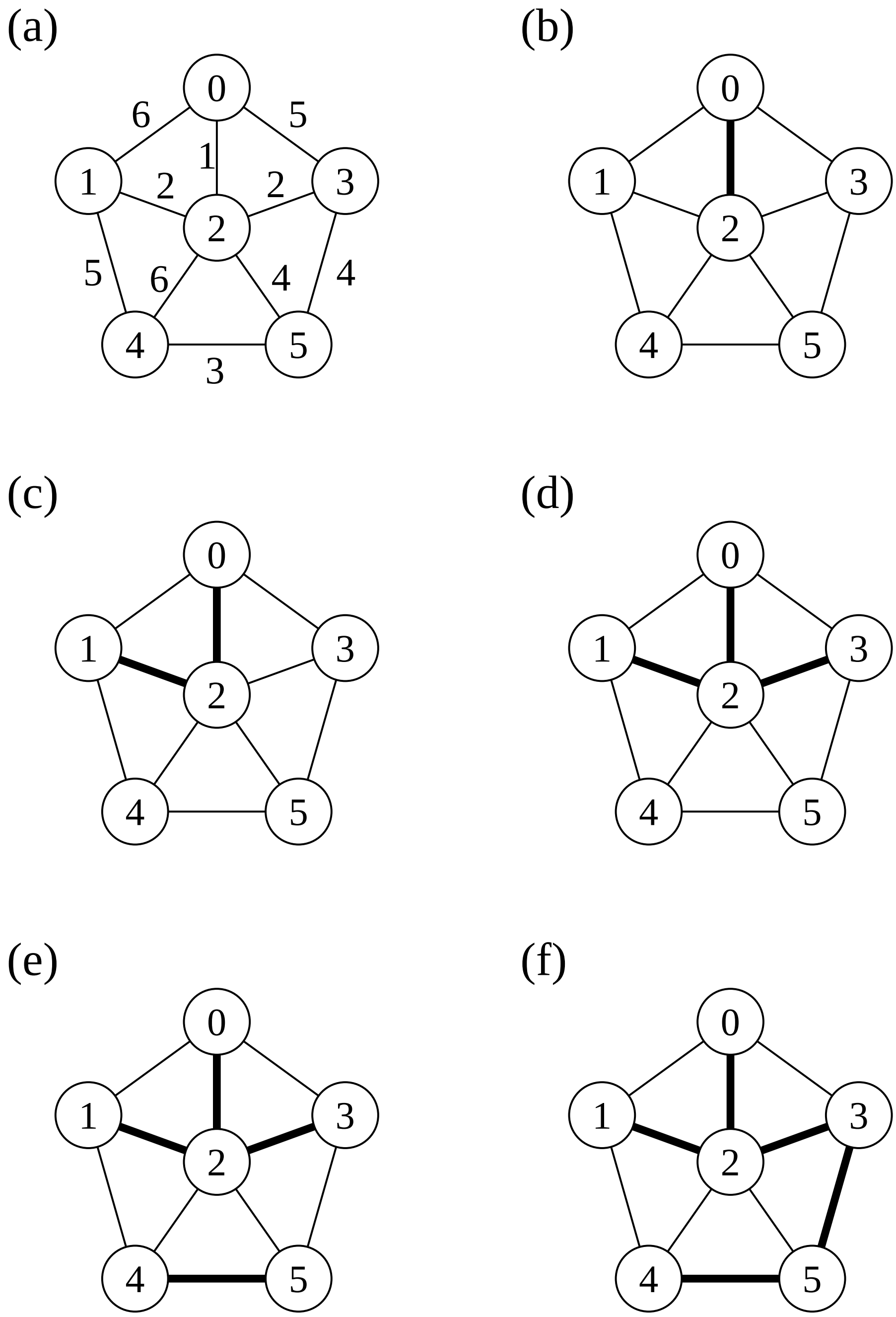
- A classe $AgmPrim$ implementa o algoritmo de Prim, cujo grafo de entrada G é fornecido através do construtor da classe $AgmPrim$.
- O método $obterAgm$ recebe o vértice *raiz* como entrada.
- O campo $antecessor[v]$ armazena o antecessor de v na árvore.
- Quando o algoritmo termina, a fila de prioridades *fp* está vazia, e a árvore geradora mínima S para G é:
- $$S = \{(v, antecessor[v]) : v \in V - \{raiz\}\}\cdot$$
- Os métodos públicos $antecessor$, $peso$ e $imprime$ são utilizados para permitir ao usuário da classe $AgmPrim$ obter o antecessor de um certo vértice, obter o peso associado a um vértice e imprimir as arestas da árvore, respectivamente.

Algoritmo de Prim - Análise

- O corpo do anel **while** é executado $|V|$ vezes.
- O método $refaz$ 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* e o peso da aresta (u, v) é menor do que $p[v]$, o antecessor de v é armazenado em $antecessor[v]$ e uma operação $diminuiChave$ é realizada sobre o *heap* 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

- 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.



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.

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:
  - Criar um novo conjunto cujo único membro é x, o qual passa a ser seu representante.
  - Fazer a união de dois conjuntos dinâmicos cujos representantes são x e y. A operação une os conjuntos dinâmicos que contêm x e y, digamos C_x e C_y, em um novo conjunto que é a união desses dois conjuntos.
  - Encontrar o conjunto de um dado elemento x. Essa operação retorna uma referência ao representante do conjunto (único) contendo x.

Algoritmo de Kruskal - Implementação

- Primeiro refinamento:

```
void Kruskal (Grafo grafo)
  ConjuntoDisjunto conj = new ConjuntoDisjunto ();
  S = ∅;
  for (int v=0; v<grafo.numVertices(); v++) conj.criaConjunto(v);
  Ordena as arestas de A pelo peso;
  for (cada (u, v) de A tomadas em ordem ascendente de peso)
    if (conj.encontraConjunto (u) != conj.encontraConjunto (v))
      S = S + {(u, v)};
      conj.uniao (u, v);
```

- A implementação das operações $uniao$ e $encontraConjunto$ deve ser realizada de forma eficiente.

- Esse problema é conhecido na literatura como **União-EncontraConjunto**.

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 $encontraConjunto$ 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 $encontraConjunto$ 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 - 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:

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

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

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 em um grafo $G = (V, A)$ pode ser realizada por um vetor chamado $antecessor$.
- Para cada vértice $v \in V$ o $antecessor[v]$ é um outro vértice $u \in V$ ou *null* (-1).
- O algoritmo atribui ao $antecessor$ 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 $antecessor[v] \neq null$, o método $imprimeCaminho$ pode imprimir o caminho mais curto de s até v.

Caminhos Mais Curtos

- 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.

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.

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:
  - $antecessor[v] = null$ para todo vértice $v \in V$,
  - $p[u] = 0$, para o vértice origem s, e
  - $p[v] = \infty$ para $v \in V - \{s\}$.

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.

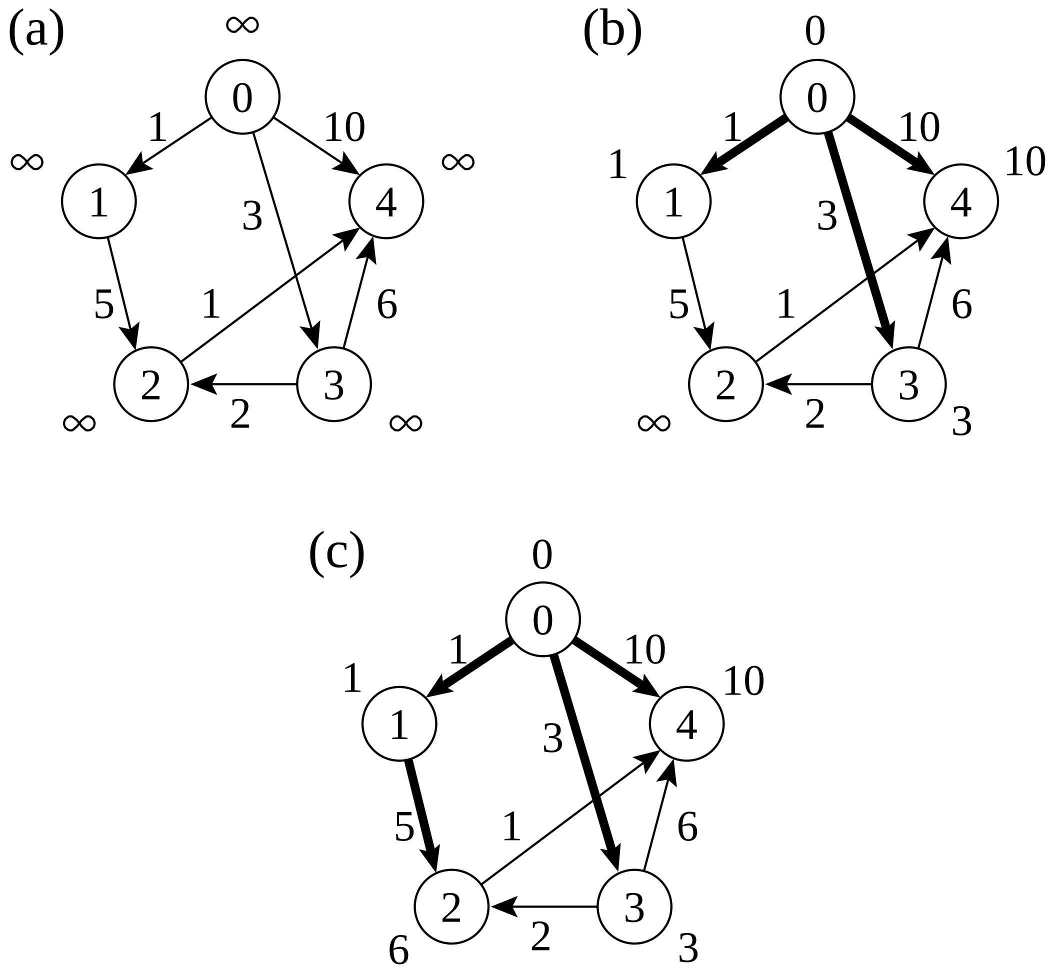
```
if (p[v] > p[u] + peso da aresta (u,v))
  p[v] = p[u] + peso da aresta (u,v);
  antecessor[v] = u;
```

Algoritmo de Dijkstra - 1º Refinamento

```
    dijkstra (Grafo grafo, int raiz)
1.    for (int v = 0; v < grafo.numVertices (); v++)
2.      p[v] = Infinito;
3.      antecessor[v] = −1;
4.    p[raiz] = 0;
5.    Constroi heap sobre vértices do grafo;
6     S = ∅;
7.    while (!heap.vazio ())
8.      u = heap.retiraMin ();
9       S = S + u;
10.     for (v ∈ grafo.listaAdjacentes (u))
11.       if (p[v] > p[u] + peso da aresta (u,v))
12.         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.
- A operação$retiraMin$ 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.

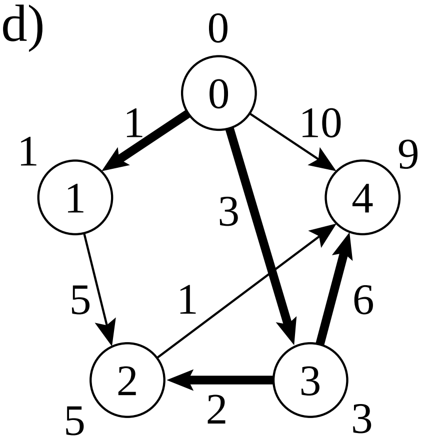
Algoritmo de Dijkstra - Exemplo



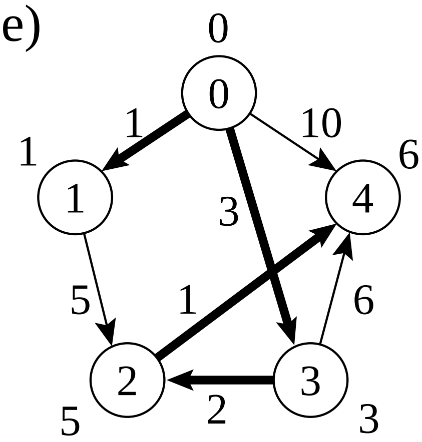
Iteração	S	d[0]	d[1]	d[2]	d[3]	d[4]
(a)	$\emptyset$	∞	∞	∞	∞	∞
(b)	$\{0\}$	0	1	∞	3	10
(c)	$\{0, 1\}$	0	1	6	3	10

Algoritmo de Dijkstra - Exemplo

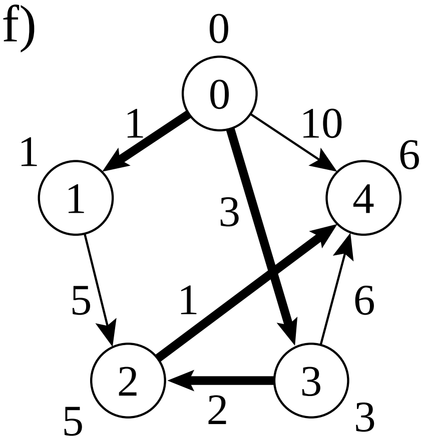
(d)



(e)



(f)



Iteração	S	d[0]	d[1]	d[2]	d[3]	d[4]
(d)	$\{0, 1, 3\}$	0	1	5	3	9
(e)	$\{0, 1, 3, 2\}$	0	1	5	3	6
(f)	$\{0, 1, 3, 2, 4\}$	0	1	5	3	6

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*, permitindo assim que o vértice v possa ser acessado a um custo $O(1)$ para a operação $diminuiChave$.

Algoritmo de Dijkstra - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class Dijkstra {
  private int antecessor[];
  private double p[];
  private Grafo grafo;

  public Dijkstra (Grafo grafo) { this.grafo = grafo; }
  public void obterArvoreCMC (int raiz) throws Exception {
    int n = this.grafo.numVertices();
    this.p = new double[n]; // peso dos vértices
    int vs[] = new int[n+1]; // vértices
    this.antecessor = new int[n];
    for (int u = 0; u < n; u ++) {
      this.antecessor[u] = -1;
      p[u] = Double.MAX_VALUE; // ∞
      vs[u+1] = u; // Heap indireto a ser construído
    }
    p[raiz] = 0;
```

Algoritmo de Dijkstra - Implementação

```
FPHeapMinIndireto heap = new FPHeapMinIndireto (p, vs);
    heap.constroi ();
    while (!heap.vazio ()) {
      int u = heap.retiraMin ();
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta adj = grafo.primeiroListaAdj (u);
        while (adj != null) {
          int v = adj.v2 ();
          if (this.p[v] > (this.p[u] + adj.peso ())) {
            antecessor[v] = u;
            heap.diminuiChave (v, this.p[u] + adj.peso ());
          }
          adj = grafo.proxAdj (u);
        }
      }
    }
  }
  public int antecessor (int u) { return this.antecessor[u]; }
  public double peso (int u) { return this.p[u]; }
  public void imprimeCaminho (int origem, int v) {
    if (origem == v) System.out.println (origem);
    else if (this.antecessor[v] == -1)
      System.out.println ("Nao existe caminho de " +origem+ " ate " +v);
    else {
      imprimeCaminho (origem, this.antecessor[v]);
      System.out.println (v);
    }
  }
}
```

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 algorimo de Dijkstra sempre obtém os caminhos mais curtos, pois cada vez que um vértice é adicionado ao conjunto S temos que $p[u] = \delta(raiz, u)$.

O Tipo Abstrato de Dados Hipergrafo

- Um **h**ipergrafo ou $r-$grafo é um grafo não direcionado $G = (V, A)$ no qual cada aresta $a \in A$ conecta r vértices, sendo r a ordem do hipergrafo.
- Os grafos estudados até agora são 2-grafos (ou hipergrafos de ordem 2).
- São utilizados para auxiliar na obtenção de funções de transformação perfeitas mínimas.
- A forma mais adequada para representar um hipergrafo é por meio de **l**istas de incidência.
- Em uma representação de um grafo não direcionado usando listas de incidência, para cada vértice v do grafo é mantida uma lista das arestas que incidem sobre o vértice v.
- Essa é uma estrutura orientada a arestas e não a vértices como as representações.
- Isso evita a duplicação das arestas ao se representar um grafo não direcionado pela versão direcionada correspondente.

O Tipo Abstrato de Dados Hipergrafo

- Operações de um tipo abstrato de dados hipergrafo:
  1. Criar um hipergrafo vazio.
  2. Inserir uma aresta no hipergrafo.
  3. Verificar se existe determinada aresta no hipergrafo.
  4. Obter a lista de arestas incidentes em determinado vértice.
  5. Retirar uma aresta do hipergrafo.
  6. Imprimir um hipergrafo.
  7. Obter o número de vértices do hipergrafo.
  8. Obter a aresta de menor peso de um hipergrafo.

O Tipo Abstrato de Dados Hipergrafo

- Uma operação que aparece com freqüência é a de obter a lista de arestas incidentes em determinado vértice.

- Para implementar esse operador precisamos de três operações sobre hipergrafos, a saber:
  1. Verificar se a lista de arestas incidentes em um vértice v está vazia.
  2. Obter a primeira aresta incidente a um vértice v, caso exista.
  3. Obter a próxima aresta incidente a um vértice v, caso exista.

O Tipo Abstrato de Dados Hipergrafo

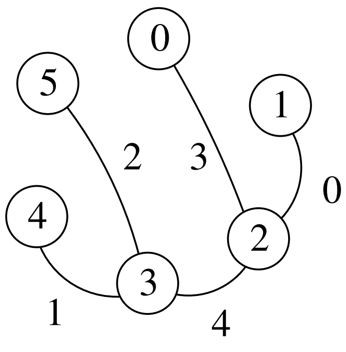
- A estrutura de dados usada para representar o hipergrafo é orientada a arestas
- As arestas são armazenadas em um arranjo chamado $arestas$.
- Em cada índice a do arranjo $arestas$, são armazenados os r vértices da aresta a e o seu $peso$.
- As listas de arestas incidentes nos vértices são armazenadas em dois arranjos: $prim$ (ponto de entrada para a lista de arestas incidentes) e $prox$ (as arestas subseqüentes).
- Valores armazenados nos arranjos $prim$ e $prox$ são obtidos pela equação $a + i|A|$, sendo $0 \leq i \leq r - 1$ e a um índice de uma aresta.
- Para se ter acesso a uma aresta a armazenada em $arestas[a]$ é preciso tomar os valores armazenados nos arranjos $prim$ e $prox$ módulo $|A|$.
- O valor -1 é utilizado para finalizar a lista.
- $prim$ deve possuir $|V|$ entradas.
- $prox$ deve possuir $r|A|$ entradas.

O Tipo Abstrato de Dados Hipergrafo - Exemplo

- Para descobrir quais são as arestas que contêm determinado vértice v é preciso percorrer a lista de arestas que inicia em $prim[v]$ e termina quando $prox[\ldots prim[v] \ldots] = -1$.

- Exemplo, ao se percorrer a lista das arestas do vértice 2, os valores $\{4, 8, 5\}$ são obtidos, os quais representam as arestas que contêm o vértice 2, ou seja, $\{4 \bmod 5 = 4, 8 \bmod 5 = 3, 5 \bmod 5 = 0\}$.

(a)



(b)

	0	1	2	3	4
arestas	(1,2,0)	(3,4,1)	(3,5,2)	(0,2,3)	(2,3,4)

	0	1	2	3	4	5
prim	3	0	4	9	6	7

	0	1	2	3	4	5	6	7	8	9
prox	–1	–1	1	–1	8	–1	–1	–1	5	2

O Tipo Abstrato de Dados Hipergrafo - Implementação

- A variável r é utilizada para armazenar a ordem do hipergrafo.
- $numVertices$ contém o número de vértices do hipergrafo.
- $proxDisponivel$ contém a próxima posição disponível para inserção de uma nova aresta.
- pos é utilizado para reter a posição atual na lista de incidência de um vértice v.

```
package cap7.listincidencia;
public class HiperGrafo {
  public static class Aresta {
    private int vertices[];
    private int peso;
    public Aresta (int vertices[], int peso) {
      this.vertices = vertices;
      this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int vertice (int i) { return this.vertices[i]; }
    public int[] vertices () { return this.vertices; }
    public boolean equals (Object aresta) {
      Aresta a = (Aresta)aresta;
      if (a.vertices.length != this.vertices.length) return false;
      for (int i = 0; i < this.vertices.length; i++)
        if (this.vertices[i] != a.vertices[i]) return false;
      return true;
    }
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
    public String toString () {
      String res = "{"; int i = 0;
      for (i = 0; i < this.vertices.length−1; i++)
        res += this.vertices[i] + ", ";
      res += this.vertices[i] + "} (" + this.peso + ")";
      return res;
    }
  }
  private int numVertices, proxDisponivel, r;
  private Aresta arestas[];
  private int prim[], prox[];
  private int pos[];

  public HiperGrafo (int numVertices, int numArestas, int r) {
    this.arestas = new Aresta[numArestas];
    this.prim = new int[numVertices];
    for (int i = 0; i < numVertices; i++) this.prim[i] = −1;
    this.prox = new int[r∗numArestas];
    this.numVertices = numVertices;
    this.proxDisponivel = 0;
    this.r = r;
    this.pos = new int[numVertices];
  }
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
public void insereAresta (int vertices[], int peso) {
  if (this.proxDisponivel == this.arestas.length)
    System.out.println ("Nao ha espaco disponivel para a aresta");
  else {
    int a = this.proxDisponivel++; int n = this.arestas.length;
    this.arestas[a] = new Aresta (vertices, peso);
    for (int i = 0; i < this.r; i++) {
      int ind = a + i*n;
      this.prox[ind] = this.prim[this.arestas[a].vertices[i]];
      this.prim[this.arestas[a].vertices[i]] = ind;
    }
  }
}
public boolean existeAresta (int vertices[]) {
  for (int v = 0; v < this.r; v++)
    for (int i = this.prim[vertices[v]]; i != -1; i = this.prox[i]) {
      int a = i % this.arestas.length;
      if (this.arestas[a].equals (new Aresta (vertices, 0)))
        return true;
    }
  return false;
}
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
public boolean listaIncVazia (int v) { return (this.prim[v] == -1); }
public Aresta primeiraListaInc (int v) {
  // Retorna a primeira aresta incidente no vértice v ou
  // null se a lista de arestas incidentes em v for vazia
  this.pos[v] = this.prim[v];
  int a = this.pos[v] % this.arestas.length;
  if (a >= 0) return this.arestas[a]; else return null;
}
public Aresta proxInc (int v) {
  // Retorna a próxima aresta incidente no vértice v ou null
  // se a lista de arestas incidentes em v estiver no fim
  this.pos[v] = this.prox[this.pos[v]];
  int a = this.pos[v] % this.arestas.length;
  if (a >= 0) return this.arestas[a]; else return null;
}
public Aresta retiraAresta (int vertices[]) {
  int n = this.arestas.length, a = 0; Aresta aresta = null;
  for (int i = 0; i < this.r; i++) {
    int prev = -1, aux = this.prim[vertices[i]];
    a = aux % n; aresta = new Aresta (vertices, 0);
    while ((aux >= 0) && (!this.arestas[a].equals (aresta))) {
      prev = aux; aux = this.prox[aux]; a = aux % n; }
    if (aux >= 0) { // achou
      if (prev == -1) this.prim[vertices[i]] = this.prox[aux];
      else this.prox[prev] = this.prox[aux];
      aresta = this.arestas[a];
    } else return null; // não achou
  }
  this.arestas[a] = null; // Marca como removido
  return aresta;
}
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      for (int j = this.prim[i]; j != -1; j = this.prox[j]) {
        int a = j % this.arestas.length;
        System.out.println ("  a: " + this.arestas[a]); }
    }
  }
  public int numVertices () { return this.numVertices; }
}
```