Algoritmos em Grafos*

Última alteração: 24 de Setembro de 2010

*Transparências elaboradas por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Conteúdo do Capítulo

7.1 Definições Básicas

7.2 O Tipo Abstrato de Dados Grafo

- 7.2.1 Implementação por meio de Matrizes de Adjacência
- 7.2.2 Implementação por meio de Listas de Adjacência Usando Apontadores
- 7.2.3 Implementação por meio de Listas de Adjacência Usando Arranjos
- 7.2.4 Programa Teste para as Três Implementações

7.3 Busca em Profundidade

7.4 Verificar se Grafo é Acíclico

- 7.4.1 Usando Busca em Profundidade
- 7.4.1 Usando o Tipo Abstrato de Dados Hipergrafo

7.5 Busca em Largura

7.6 Ordenação Topológica

7.7 Componentes Fortemente Conectados

7.8 Árvore Geradora Mínima

- 7.8.1 Algoritmo Genérico para Obter a Árvore Geradora Mínima
- 7.8.2 Algoritmo de Prim
- 7.8.2 Algoritmo de Kruskal

7.9 Caminhos mais Curtos

7.10 O Tipo Abstrato de Dados Hipergrafo

- 7.10.1 Implementação por meio de Matrizes de Incidência
- 7.10.1 Implementação por meio de Listas de Incidência Usando Arranjos

Motivação

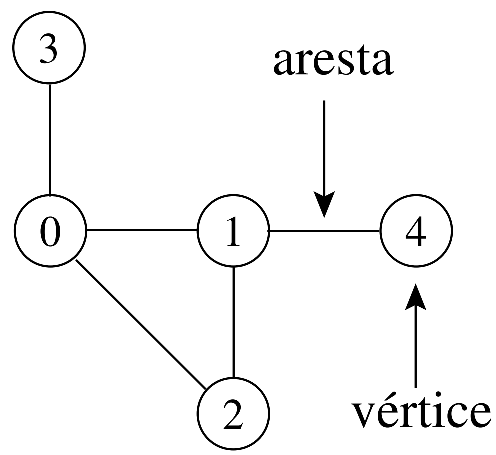
- Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
  - Existe um caminho para ir de um objeto a outro seguindo as conexões?
  - Qual é a menor distância entre um objeto e outro objeto?
  - Quantos outros objetos podem ser alcançados a partir de um determinado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

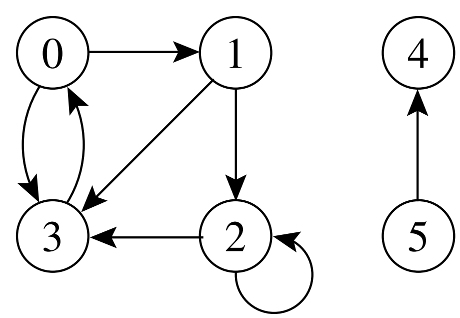
- **Grafo**: conjunto de vértices e arestas.
- **Vértice**: objeto simples que pode ter nome e outros atributos.
- **Aresta**: conexão entre dois vértices.



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

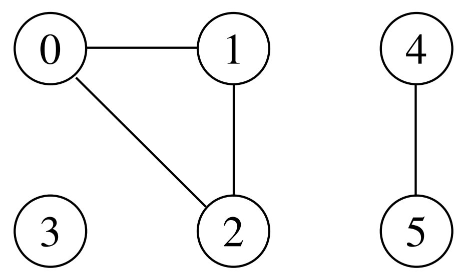
Grafos Direcionados

- 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*.



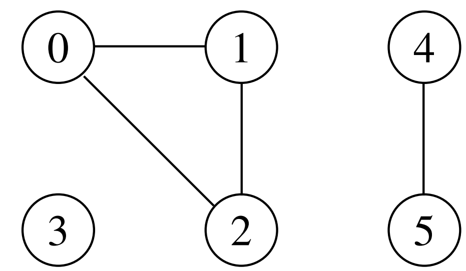
Grafos Não Direcionados

- 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.

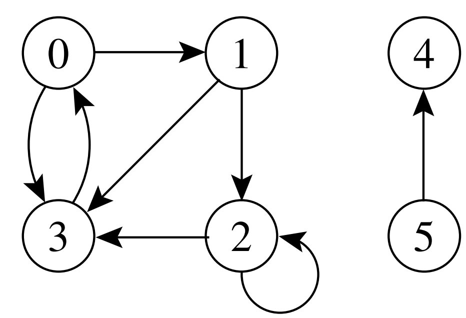


Grau de um Vértice

- 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.



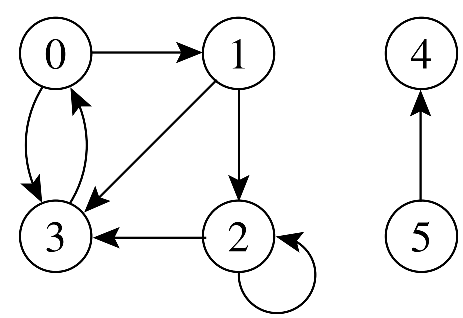
- Em grafos direcionados
  - O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).
  - Ex.: O vértice 2 tem *in-degree* 2, *out-degree* 2 e grau 4.



Caminho entre Vértices

- Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma 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 **simples** se todos os vértices do caminho são distintos.

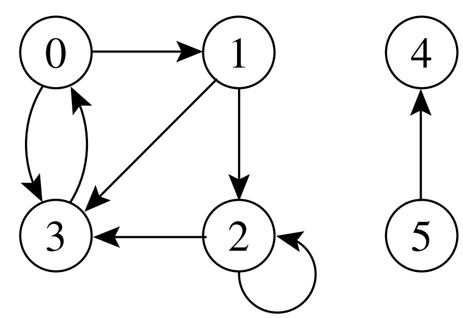
Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



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 $i = 0, 1, \ldots, k-1$.

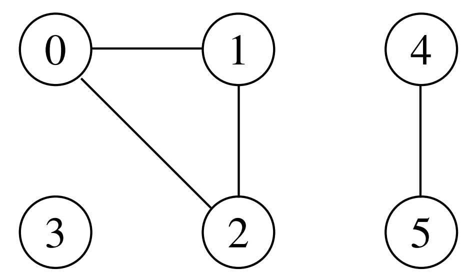
Ex.: O caminho $(0, 1, 2, 3, 0)$ forma um ciclo. O caminho$(0, 1, 3, 0)$ forma o mesmo ciclo que os caminhos $(1, 3, 0, 1)$ e $(3, 0, 1, 3)$.



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.

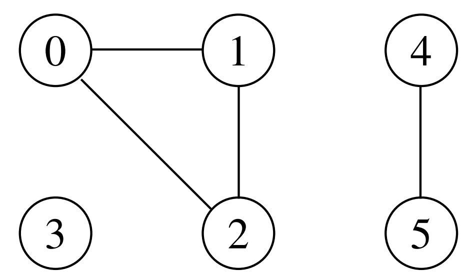
Ex.: O caminho $(0, 1, 2, 0)$ é um ciclo.



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.

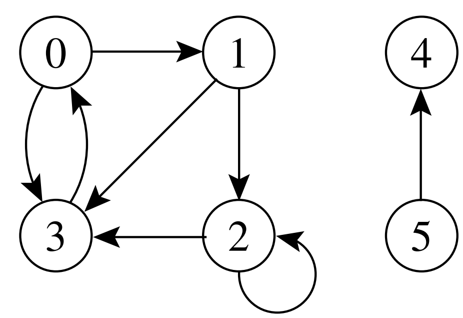
Ex.: Os componentes são: $\{0, 1, 2\}$, $\{4, 5\}$ e $\{3\}$.



Componentes Fortemente Conectados

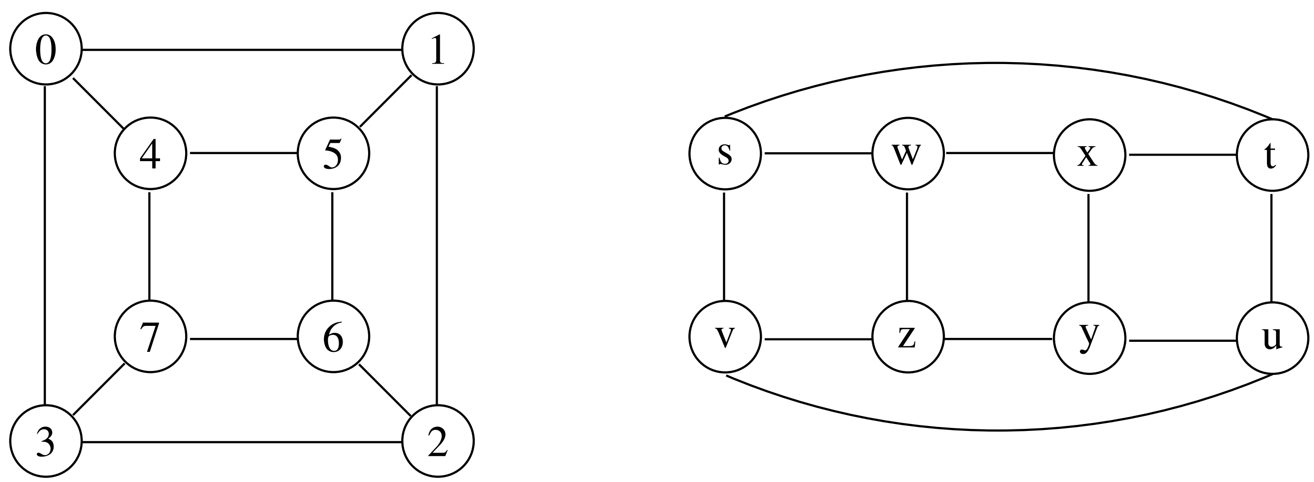
- Um grafo direcionado $G = (V, A)$ é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado são conjuntos de vértices sob a relação “são mutuamente alcançáveis”.
- Um **grafo direcionado fortemente 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 $f : V \to V'$ tal que $(u, v) \in A$ se e somente se $(f(u), f(v)) \in A'$.
- Em outras palavras, é possível re-rotular os vértices de G para serem rótulos de G' mantendo as arestas correspondentes em G e G'.

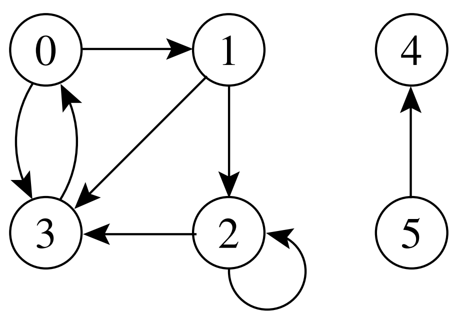


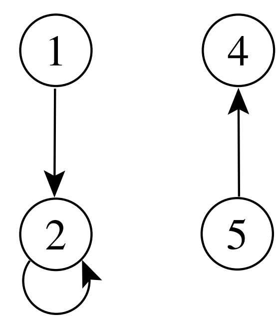
Subgrafos

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

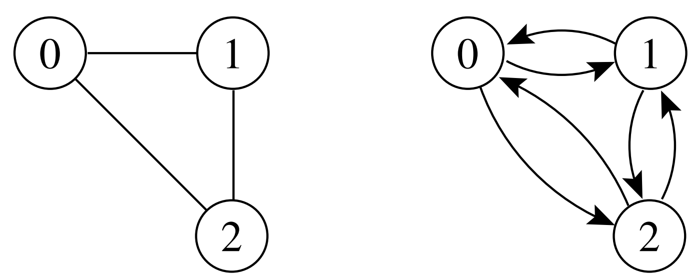
Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.





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.

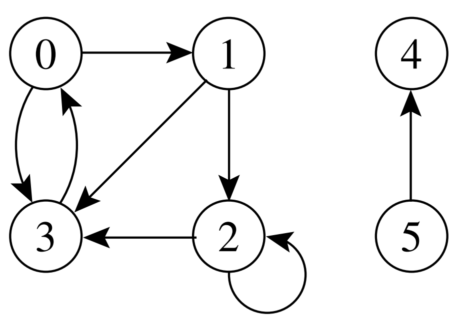


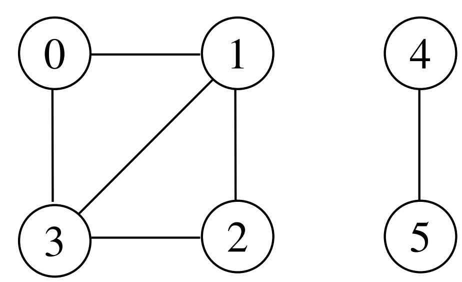
Versão Não Direcionada de um Grafo Direcionado

- 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

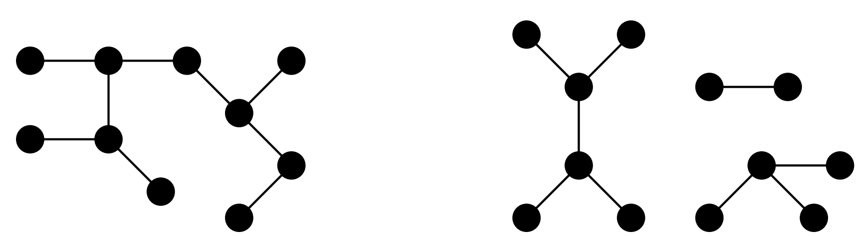
- **Grafo ponderado**: possui pesos associados às arestas.
- **Grafo bipartido**: grafo não direcionado $G = (V, A)$ no qual V pode ser particionado em dois conjuntos V_1 e V_2 tal que $(u, v) \in A$ implica que $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$ (todas as arestas ligam os dois conjuntos V_1 e V_2).
- **Hipergrafo**: grafo não direcionado em que cada aresta conecta um número arbitrário de vértices.
  - Hipergrafos são utilizados na Seção 5.5.4 sobre ***hashing*** **perfeito**.
  - Na Seção 7.10 é apresentada uma estrutura de dados mais adequada para representar um hipergrafo.

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. *InsereAresta(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.

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.

Implementação da Operação “Obter Lista de Adjacentes”

- É comum encontrar um pseudo comando do tipo:

  for u $\in$ ListaAdjacentes (v) **do** { faz algo com u }

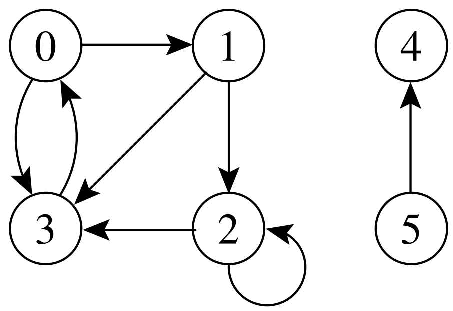
- O trecho de programa abaixo apresenta um possível refinamento do pseudo comando acima.

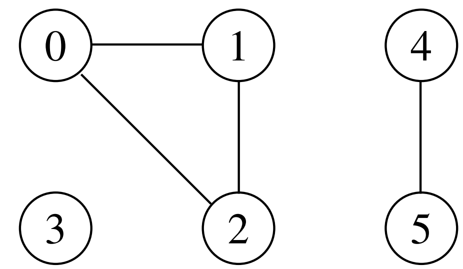
```
if (!ListaAdjVazia(v,Grafo))
{ Aux = PrimeiroListaAdj(v,Grafo);
  FimListaAdj = FALSE;
  while(!FimListaAdj)
    ProxAdj(&v, Grafo, &u, &Peso, &Aux, &FimListaAdj);
}
```

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: Estrutura de Dados

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

```
#define MAXNUMVERTICES  100
#define MAXNUMARESTAS   4500

typedef int TipoValorVertice;
typedef int  TipoPeso;
typedef struct TipoGrafo {
  TipoPeso Mat[MAXNUMVERTICES + 1][MAXNUMVERTICES + 1];
  int NumVertices;
  int NumArestas;
} TipoGrafo;
typedef int  TipoApontador;
```

Matriz de Adjacência: Operadores

```
void FGVazio(TipoGrafo *Grafo)
{ short i, j;
  for (i = 0; i <= Grafo->NumVertices; i++)
    { for (j = 0; j <=Grafo->NumVertices; j++) Grafo->Mat[i][j] = 0; }
}

void InsereAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ Grafo->Mat[*V1][*V2] = *Peso; }

short ExisteAresta(TipoValorVertice Vertice1,
                   TipoValorVertice Vertice2, TipoGrafo *Grafo)
{ return (Grafo->Mat[Vertice1][Vertice2] > 0); }
```

Matriz de Adjacência: Operadores

```
/* Operadores para obter a lista de adjacentes */
short ListaAdjVazia(TipoValorVertice *Vertice, TipoGrafo *Grafo)
{ TipoApontador Aux = 0;
  short ListaVazia = TRUE;
  while (Aux < Grafo->NumVertices && ListaVazia)
    if (Grafo->Mat[*Vertice][Aux] > 0) ListaVazia = FALSE;
    else Aux++;
  return (ListaVazia == TRUE);
}
```

Matriz de Adjacência: Operadores

```
/* Operadores para obter a lista de adjacentes */
TipoApontador PrimeiroListaAdj (TipoValorVertice *Vertice, TipoGrafo *Grafo)
{ TipoValorVertice Result;
  TipoApontador Aux = 0;
  short ListaVazia = TRUE;
  while (Aux < Grafo->NumVertices && ListaVazia)
    { if (Grafo->Mat[*Vertice][Aux] > 0) { Result = Aux; ListaVazia = FALSE; }
      else Aux++;
    }
  if (Aux == Grafo->NumVertices)
    printf("Erro: Lista adjacencia vazia (PrimeiroListaAdj)\n");
  return Result;
}
```

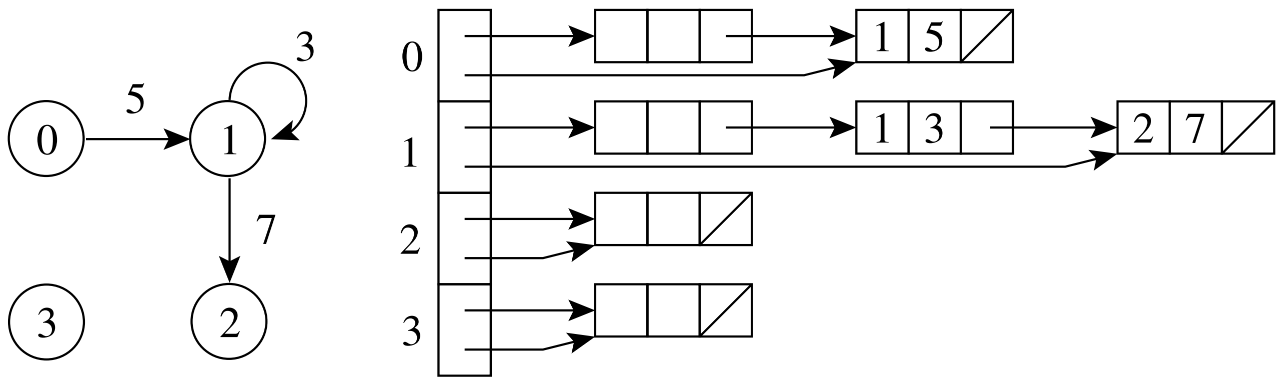
Matriz de Adjacência: Operadores

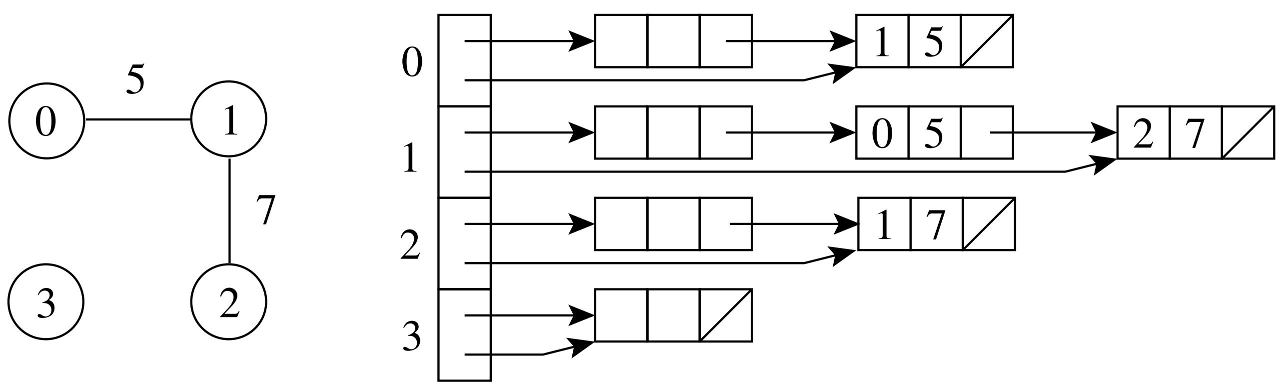
```
/* Operadores para obter a lista de adjacentes */
void ProxAdj(TipoValorVertice *Vertice, TipoGrafo *Grafo,
             TipoValorVertice *Adj, TipoPeso *Peso,
             TipoApontador *Prox, short *FimListaAdj)
{ /* Retorna Adj apontado por Prox */
  *Adj = *Prox;
  *Peso = Grafo->Mat[*Vertice][*Prox];
  (*Prox)++;
  while (*Prox < Grafo->NumVertices &&
         Grafo->Mat[*Vertice][*Prox] == 0) (*Prox)++;
  if (*Prox == Grafo->NumVertices) *FimListaAdj = TRUE;
}
```

Matriz de Adjacência: Operadores

```
void RetiraAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ if (Grafo->Mat[*V1][*V2] == 0) printf("Aresta nao existe\n");
  else { *Peso = Grafo->Mat[*V1][*V2]; Grafo->Mat[*V1][*V2] = 0; }
}
void LiberaGrafo(TipoGrafo *Grafo)
{  /* Nao faz nada no caso de matrizes de adjacencia */ }
void ImprimeGrafo(TipoGrafo *Grafo)
{ short i, j; printf("   ");
  for (i = 0; i <= Grafo->NumVertices - 1; i++) printf("%3d", i);
  printf("\n");
  for (i = 0; i <=  Grafo->NumVertices - 1; i++)
    { printf("%3d", i);
      for (j = 0; j <=Grafo->NumVertices - 1; j++)
        printf("%3d", Grafo->Mat[i][j]);
      printf("\n");
    }
}
```

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 os vértices adjacentes a u em G.

Listas de Adjacência Usando Apontadores: 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 Apontadores (1)

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500
typedef int TipoValorVertice;
typedef int TipoPeso;
typedef struct TipoItem {
  TipoValorVertice Vertice;
  TipoPeso Peso;
} TipoItem;
typedef struct TipoCelula *TipoApontador;
struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;
```

Listas de Adjacência Usando Apontadores (2)

```
typedef struct TipoLista {
  TipoApontador Primeiro, Ultimo;
} TipoLista;
typedef struct TipoGrafo {
  TipoLista Adj[MAXNUMVERTICES + 1];
  TipoValorVertice NumVertices;
  short NumArestas;
} TipoGrafo;
```

- 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.

Listas de Adjacência Usando Apontadores: Operadores

```
/*-- Entram aqui os operadores FLVazia, Vazia, Insere, Retira e Imprime--*/
/*-- do TAD Lista de Apontadores do Programa 3.4                       --*/
void FGVazio(TipoGrafo *Grafo)
{ long i;
  for (i = 0; i < Grafo->NumVertices; i++) FLVazia(&Grafo->Adj[i]);
}

void InsereAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ TipoItem x;
  x.Vertice = *V2;
  x.Peso = *Peso;
  Insere(&x, &Grafo->Adj[*V1]);
}
```

Listas de Adjacência usando Apontadores

```
short ExisteAresta(TipoValorVertice Vertice1,
                   TipoValorVertice Vertice2,
                   TipoGrafo *Grafo)
{ TipoApontador Aux;
  short EncontrouAresta = FALSE;
  Aux = Grafo->Adj[Vertice1].Primeiro->Prox;
  while (Aux != NULL && EncontrouAresta == FALSE)
    { if (Vertice2 == Aux->Item.Vertice) EncontrouAresta = TRUE;
      Aux = Aux->Prox;
    }
  return EncontrouAresta;
}
```

Listas de Adjacência Usando Apontadores: Operadores

```
/* Operadores para obter a lista de adjacentes */
short ListaAdjVazia(TipoValorVertice *Vertice, TipoGrafo *Grafo)
{ return (Grafo->Adj[*Vertice].Primeiro == Grafo->Adj[*Vertice].Ultimo);
}

TipoApontador PrimeiroListaAdj(TipoValorVertice *Vertice, TipoGrafo *Grafo)
{ return (Grafo->Adj[*Vertice].Primeiro->Prox); }

void ProxAdj(TipoValorVertice *Vertice, TipoGrafo *Grafo,
             TipoValorVertice *Adj, TipoPeso *Peso,
             TipoApontador *Prox, short *FimListaAdj)
{ /* Retorna Adj e Peso do Item apontado por Prox */
  *Adj = (*Prox)->Item.Vertice;
  *Peso = (*Prox)->Item.Peso;
  *Prox = (*Prox)->Prox;
  if (*Prox == NULL) *FimListaAdj = TRUE;
}
```

Listas de Adjacência Usando Apontadores: Operadores

```
void RetiraAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ TipoApontador AuxAnterior, Aux;
  short EncontrouAresta = FALSE;
  TipoItem x;
  AuxAnterior = Grafo->Adj[*V1].Primeiro;
  Aux = Grafo->Adj[*V1].Primeiro->Prox;
  while (Aux != NULL && EncontrouAresta == FALSE)
    { if (*V2 == Aux->Item.Vertice)
      { Retira(AuxAnterior, &Grafo->Adj[*V1], &x);
        Grafo->NumArestas--;
        EncontrouAresta = TRUE;
      }
      AuxAnterior = Aux;
      Aux = Aux->Prox;
    }
}
```

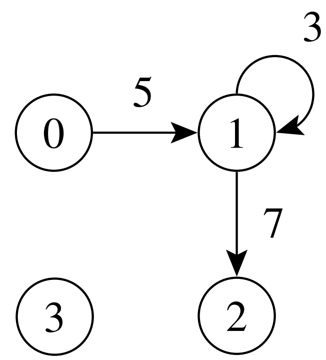
Listas de Adjacência Usando Apontadores: Operadores

```
void LiberaGrafo(TipoGrafo *Grafo)
{ TipoApontador AuxAnterior, Aux;
  for (i = 0; i < GRAfo->NumVertices; i++)
    { Aux = Grafo->Adj[i].Primeiro->Prox;
      free(Grafo->Adj[i].Primeiro);    /*Libera celula cabeca*/
      Grafo->Adj[i].Primeiro=NULL;
      while (Aux != NULL)
        { AuxAnterior = Aux;
          Aux = Aux->Prox;
          free(AuxAnterior);
        }
    }
  Grafo->NumVertices = 0;
}
```

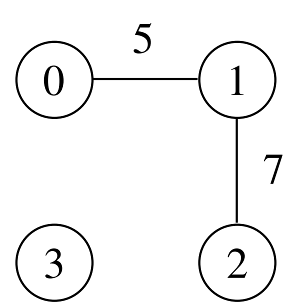
Listas de Adjacência Usando Apontadores: Operadores

```
void ImprimeGrafo(TipoGrafo *Grafo)
{ int i;
  TipoApontador Aux;
  for (i = 0; i < Grafo->NumVertices; i++)
    { printf("Vertice%2d:", i);
      if (!Vazia(Grafo->Adj[i]))
      { Aux = Grafo->Adj[i].Primeiro->Prox;
        while (Aux != NULL)
          { printf("%3d (%d)", Aux->Item.Vertice, Aux->Item.Peso);
            Aux = Aux->Prox;
          }
      }
      putchar('\n');
    }
}
```

Listas de Adjacência Usando Arranjos



		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



		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

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500
#define TRUE 1
#define FALSE 0
#define MAXTAM (MAXNUMVERTICES + MAXNUMARESTAS * 2)

typedef int TipoValorVertice;
typedef int TipoPeso;
typedef int  TipoTam;
typedef struct TipoGrafo {
  TipoTam Cab[MAXTAM + 1];
  TipoTam Prox[MAXTAM + 1];
  TipoTam Peso[MAXTAM + 1];
  TipoTam ProxDisponivel;
  char NumVertices;
  short NumArestas;
} TipoGrafo;
typedef short TipoApontador;
```

Listas de Adjacência Usando Arranjos: Operadores

```
void FGVazio(TipoGrafo *Grafo)
{ short i;
  for (i = 0; i <= Grafo->NumVertices; i++)
    { Grafo->Prox[i] = 0;  Grafo->Cab[i] = i;
      Grafo->ProxDisponivel = Grafo->NumVertices;
    }
}
void InsereAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ short Pos;
  Pos = Grafo->ProxDisponivel;
  if (Grafo->ProxDisponivel == MAXTAM)
  { printf("nao ha espaco disponivel para a aresta\n"); return; }
  Grafo->ProxDisponivel++;
  Grafo->Prox[Grafo->Cab[*V1]] = Pos;
  Grafo->Cab[Pos] = *V2;  Grafo->Cab[*V1] = Pos;
  Grafo->Prox[Pos] = 0;  Grafo->Peso[Pos] = *Peso;
}
```

Listas de Adjacência Usando Arranjos: Operadores

```
short ExisteAresta(TipoValorVertice Vertice1,
                   TipoValorVertice Vertice2, TipoGrafo *Grafo)
{ TipoApontador Aux;
  short EncontrouAresta = FALSE;
  Aux = Grafo->Prox[Vertice1];
  while (Aux != 0 && EncontrouAresta == FALSE)
    { if (Vertice2 == Grafo->Cab[Aux])
      EncontrouAresta = TRUE;
      Aux = Grafo->Prox[Aux];
    }
  return EncontrouAresta;
}
```

Listas de Adjacência Usando Arranjos: Operadores

```
/* Operadores para obter a lista de adjacentes */
short ListaAdjVazia(TipoValorVertice *Vertice, TipoGrafo *Grafo)
{ return (Grafo->Prox[*Vertice] == 0); }

TipoApontador PrimeiroListaAdj(TipoValorVertice *Vertice,
                               TipoGrafo *Grafo)
{ return (Grafo->Prox[*Vertice]); }

void ProxAdj(TipoValorVertice *Vertice, TipoGrafo *Grafo,
             TipoValorVertice *Adj, TipoPeso *Peso,
             TipoApontador *Prox, short *FimListaAdj)
{ /* Retorna Adj apontado por Prox */
  *Adj = Grafo->Cab[*Prox];  *Peso = Grafo->Peso[*Prox];
  *Prox = Grafo->Prox[*Prox];
  if (*Prox == 0) *FimListaAdj = TRUE;
}
```

Listas de Adjacência Usando Arranjos: Operadores

```
void RetiraAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ TipoApontador Aux, AuxAnterior;  short EncontrouAresta = FALSE;
  AuxAnterior = *V1;  Aux = Grafo->Prox[*V1];
  while (Aux != 0 && EncontrouAresta == FALSE)
    { if (*V2 == Grafo->Cab[Aux]) EncontrouAresta = TRUE;
      else { AuxAnterior = Aux; Aux = Grafo->Prox[Aux]; }
    }
  if (EncontrouAresta) /* Apenas marca como retirado */
  { Grafo->Cab[Aux] = MAXNUMVERTICES + MAXNUMARESTAS * 2;
  }
  else printf("Aresta nao existe\n");
}
```

Listas de Adjacência Usando Arranjos: Operadores

```
void LiberaGrafo(TipoGrafo *Grafo)
{  /* Nao faz nada no caso de posicoes contiguas */ }

void ImprimeGrafo(TipoGrafo *Grafo)
{ short i, forlim;
  printf("     Cab Prox Peso\n");
  forlim = Grafo->NumVertices + Grafo->NumArestas * 2;
  for (i = 0; i <= forlim - 1; i++)
    printf("%2d%4d%4d%4d\n", i, Grafo->Cab[i],
           Grafo->Prox[i], Grafo->Peso[i]);
}
```

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.

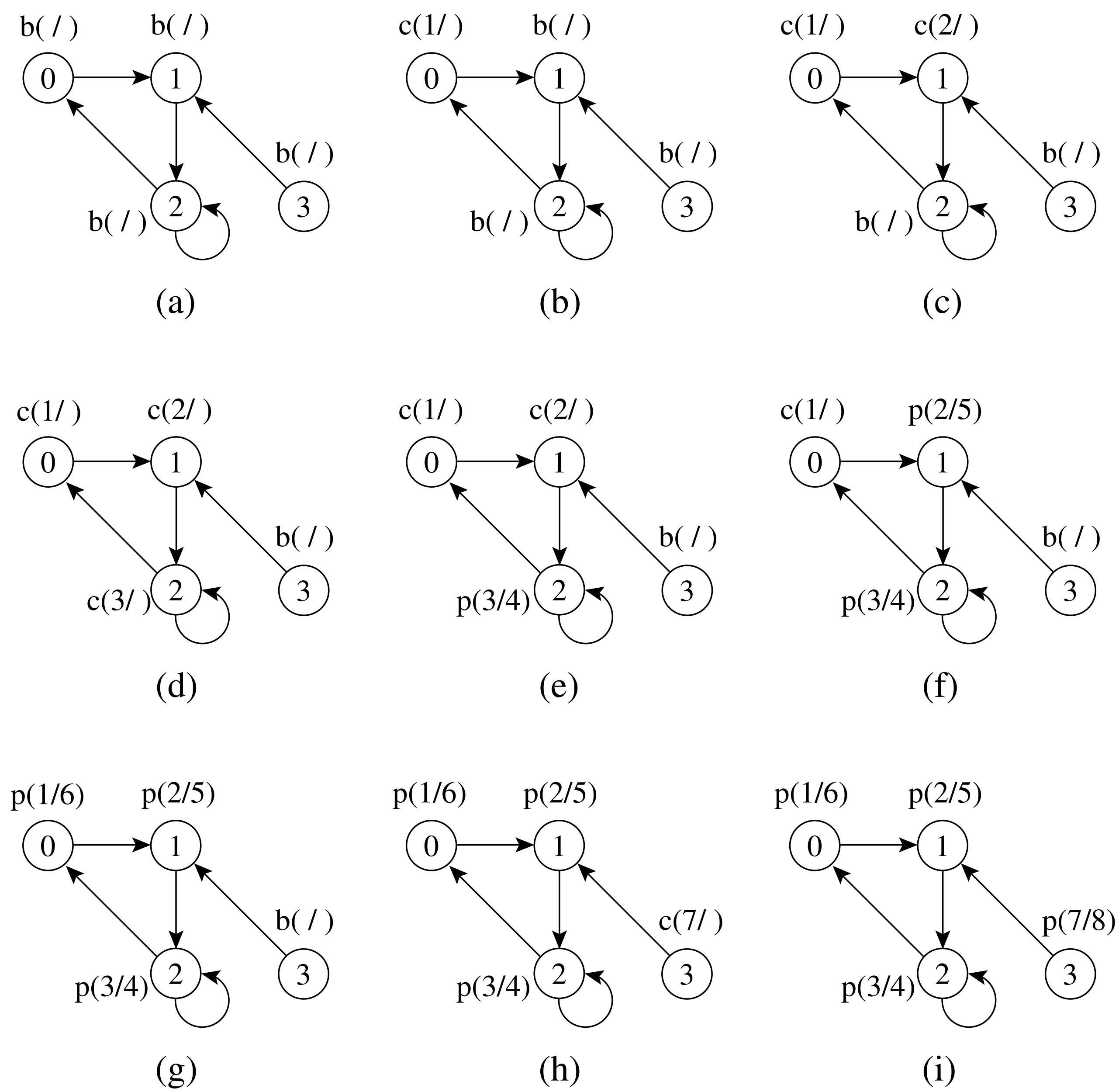
Busca em Profundidade: Implementação

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
               TipoValorTempo* Tempo, TipoValorTempo* d,
               TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo); FimListaAdj = FALSE;
    while (!FimListaAdj)
      { ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
        if (Cor[v] == branco)
        { Antecessor[v] = u; VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
        }
      }
  }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Busca em Profundidade: Implementação

```
void BuscaEmProfundidade(TipoGrafo *Grafo)
{ TipoValorVertice x;
  TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES+1];
  short Antecessor[MAXNUMVERTICES+1];
  Tempo = 0;
  for (x = 0; x <= Grafo->NumVertices - 1; x++)
    { Cor[x] = branco;
      Antecessor[x] = -1;
    }
  for (x = 0; x <= Grafo->NumVertices - 1; x++)
    { if (Cor[x] == branco)
      VisitaDfs(x, Grafo, &Tempo, d, t, Cor, Antecessor);
    }
}
```

Busca em Profundidade: Exemplo



Busca em Profundidade: Análise

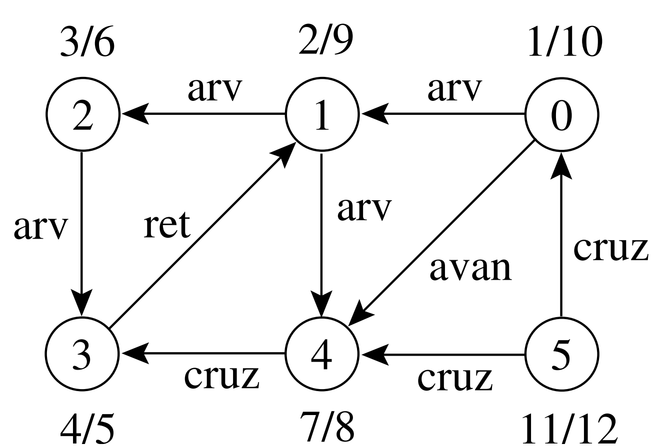
- Os dois anéis da *BuscaEmProfundidade* têm custo $O(|V|)$ cada um, a menos da chamada do procedimento *VisitaDfs*(u) no segundo anel.
- O procedimento *VisitaDfs* é chamado exatamente uma vez para cada vértice $u \in V$, desde que *VisitaDfs* é chamado apenas para vértices brancos e a primeira ação 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

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 Usando Busca em Profundidade

- 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.
- O algoritmo BuscaEmProfundidade pode ser alterado para descobrir arestas de retorno. Para isso, basta verificar se um vértice v adjacente a um vértice u apresenta a cor cinza na primeira vez que a aresta (u, v) é percorrida.
- O algoritmo tem custo linear no número de vértices e de arestas de um grafo $G = (V, A)$ que pode ser utilizado para verificar se G é **a**cíclico.

Teste para Verificar se Grafo é Acíclico Usando o Tipo Abstrato de Dados Hipergrafo

- **Hipergrafos ou** $r-$**grafos** $G_r(V, A)$ são apresentados na Seção 7.10 (Slide 119).
- Representação: por meio de estruturas de dados orientadas a arestas em que para cada vértice v do grafo é mantida uma lista das arestas que incidem sobre o vértice v.
- Existem duas representações usuais para hipergrafos: **m**atrizes de incidência e **l**istas de incidência. Aqui utilizaremos a implementação de listas de incidência usando arranjos apresentada na Seção 7.10.2.
- O programa a seguir utiliza a seguinte propriedade de r-grafos:

  Um r-grafo é **a**cíclico se e somente se a remoção repetida de arestas contendo apenas vértices de grau 1 (vértices sobre os quais incide apenas uma aresta) elimina todas as arestas do grafo.

Teste para Verificar se Grafo é Acíclico Usando o Tipo Abstrato de Dados Hipergrafo

- O procedimento a seguir recebe o grafo e retorna no vetor L as arestas retiradas do grafo na ordem em foram retiradas.
- O procedimento primeiro procura os vértices de grau 1 e os coloca em uma fila. A seguir, enquanto a fila não estiver vazia, desenfileira um vértice e retira a aresta incidente ao vértice.
- Se a aresta retirada tinha algum outro vértice de grau 2, então esse vértice muda para grau 1 e é enfileirado.
- Se ao final não restar nenhuma aresta, então o grafo é acíclico. O custo do procedimento GrafoAciclico é $O(|V| + |A|)$.

Teste para Verificar se Grafo é Acíclico Usando o Tipo Abstrato de Dados Hipergrafo

```
program GrafoAciclico;
/** Entram aqui os tipos do Programa 3.17 (ou do Programa 3.19 **/
/** Entram aqui tipos do Programa 7.25 (Slide 137) **/
  int i, j;
  TipoAresta Aresta;
  TipoGrafo Grafo;
  TipoArranjoArestas L;
  short GAciclico;
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e          **/
/** Desenfileira do Programa 3.18 (ou do Programa 3.20            **/
/** Entram aqui os operadores ArestasIguais, FGVazio, InsereAresta,  **/
/** RetiraAresta e ImprimeGrafo do Programa 7.26 (Slide 138) **/
short VerticeGrauUm(TipoValorVertice *V,
                    TipoGrafo *Grafo)
{ return (Grafo->Prim[*V] >= 0) && (Grafo->Prox[Grafo->Prim[*V]] == INDEFINIDO);
}
```

Teste para Verificar se Grafo é Acíclico Usando o Tipo Abstrato de Dados Hipergrafo (1)

```
void GrafoAciclico (TipoGrafo *Grafo,
                    TipoArranjoArestas L, short *GAciclico)
{ TipoValorVertice j = 0; TipoValorAresta A1;
  TipoItem x; TipoFila Fila; TipoValorAresta NArestas;
  TipoAresta Aresta; NArestas = Grafo->NumArestas;
  FFVazia (&Fila);
  while (j < Grafo->NumVertices)
  { if (VerticeGrauUm (&j, Grafo))
    { x.Chave = j; Enfileira (x, &Fila); }
    j++;
  }
  while (!Vazia(&Fila) && (NArestas > 0))
  { Desenfileira (&Fila, &x);
```

Teste para Verificar se Grafo é Acíclico Usando o Tipo Abstrato de Dados Hipergrafo (2)

```
    if (Grafo->Prim[x.Chave] >= 0)
    { A1 = Grafo->Prim[x.Chave] % Grafo->NumArestas;
      Aresta = RetiraAresta(&Grafo->Arestas[A1], Grafo);
      L[Grafo->NumArestas - NArestas] = Aresta;
      NArestas = NArestas - 1;
      if (NArestas > 0)
      { for (j = 0; j < Grafo->r; j++)
        { if (VerticeGrauUm(&Aresta.Vertices[j], Grafo))
          { x.Chave = Aresta.Vertices[j]; Enfileira (x, &Fila);
          }
        }
      }
    }
  }
  if (NArestas == 0) *GAciclico = TRUE;
  else *GAciclico = FALSE;
}
```

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

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MaxNumvertices + 1];
  TipoCor Cor[MaxNumvertices + 1];
  int Antecessor[MaxNumvertices + 1];
  for (x = 0; x <= Grafo -> NumVertices - 1; x++)
    { Cor[x] = branco; Dist[x] = Infinito; Antecessor[x] = -1; }
  for (x = 0; x <= Grafo -> NumVertices - 1; x++)
    { if (Cor[x] == branco)
        VisitaBfs (x, Grafo, Dist, Cor, Antecessor);
    }
}
```

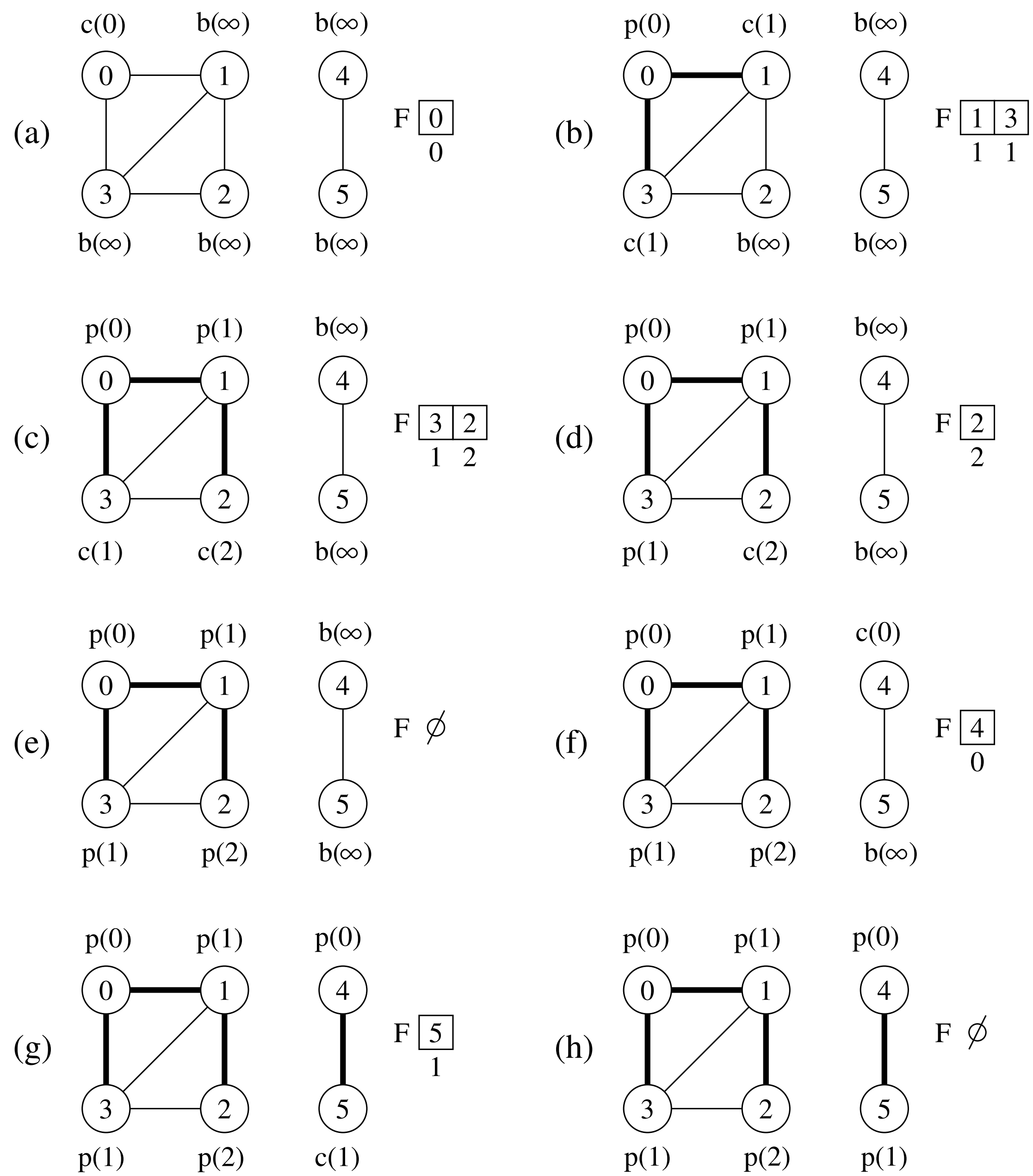
Busca em Largura: Implementação

```
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do **/
/** do Programa 3.18 ou do Programa 3.20, dependendo da implementação **/
/** da busca em largura usar arranjos ou apontadores, respectivamente **/
void VisitaBfs(TipoValorVertice u, TipoGrafo *Grafo,
               int *Dist, TipoCor *Cor, int *Antecessor)
{ TipoValorVertice v;  Apontador Aux;  short FimListaAdj;
  TipoPeso Peso;  TipoItem Item;  TipoFila Fila;
  Cor[u] = cinza;  Dist[u] = 0;
  FFVazia(&Fila);
  Item.Vertice = u;  Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

Busca em Largura: Implementação

```
while (!FilaVazia(Fila))
  { Desenfileira(&Fila, &Item);
    u = Item.Vertice;
    if (!ListaAdjVazia(&u, Grafo))
    { Aux = PrimeiroListaAdj(&u, Grafo);
      FimListaAdj = FALSE;
      while (FimListaAdj == FALSE)
        { ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
          if (Cor[v] != branco) continue;
          Cor[v] = cinza;  Dist[v] = Dist[u] + 1;
          Antecessor[v] = u; Item.Vertice = v;
          Item.Peso = Peso; Enfileira(Item, &Fila);
        }
    }
    Cor[u] = preto;
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);
    ImprimeFila(Fila); getchar();
  }
} /* 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 $O(|V| + |A|)$.

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 a seguir imprime os vértices do caminho mais curto entre o vértice origem e outro vértice qualquer do grafo.

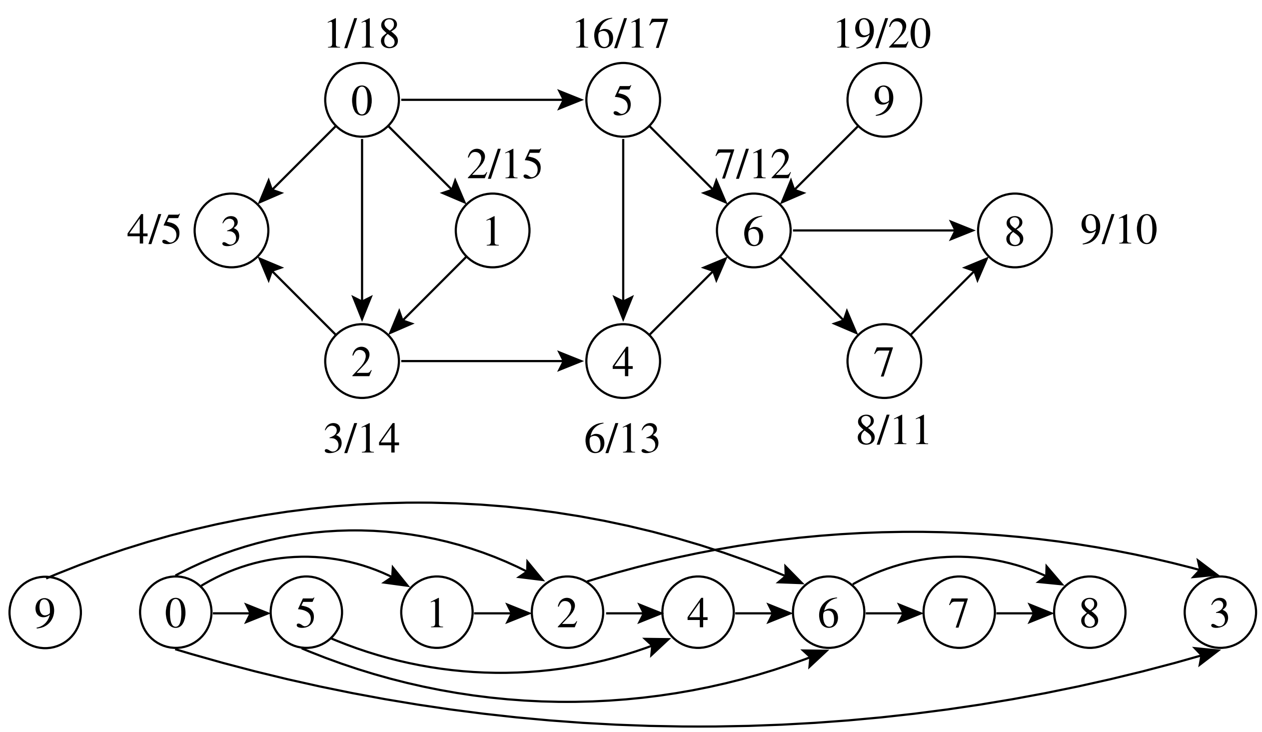
```
void ImprimeCaminho(TipoValorVertice Origem, TipoValorVertice v,
                    TipoGrafo *Grafo, int * Dist, TipoCor *Cor,
                    int *Antecessor)
{ if (Origem == v) { printf("%d ", Origem); return; }
  if (Antecessor[v] == -1)
  printf("Nao existe caminho de %d ate %d", Origem, v);
  else { ImprimeCaminho(Origem,Antecessor[v], Grafo, Dist, Cor, Antecessor);
         printf("%d ", 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. 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

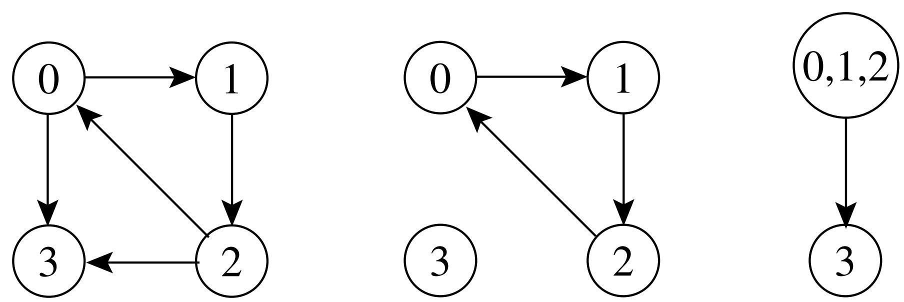
- Basta inserir uma chamada para o procedimento *InsLista* no procedimento *BuscaDfs*, logo após o momento em que o tempo de término $t[u]$ é obtido e o vértice é pintado de preto.
- Ao final, basta retornar a lista obtida (ou imprimí-la usando o procedimento Imprime do Programa 3.4).

```
void InsLista (TipoItem *Item, TipoLista *Lista)
{ /*-- Insere antes do primeiro item da lista --*/
  TipoApontador Aux;
  Aux = Lista->Primeiro->Prox;
  Lista->Primeiro->Prox = (TipoApontador)malloc(sizeof(tipoCelula));
  Lista->Primeiro->Prox->Item = Item;
  Lista->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. Chama *BuscaEmProfundidade*(G) para obter os tempos de término $t[u]$ para cada vértice u.

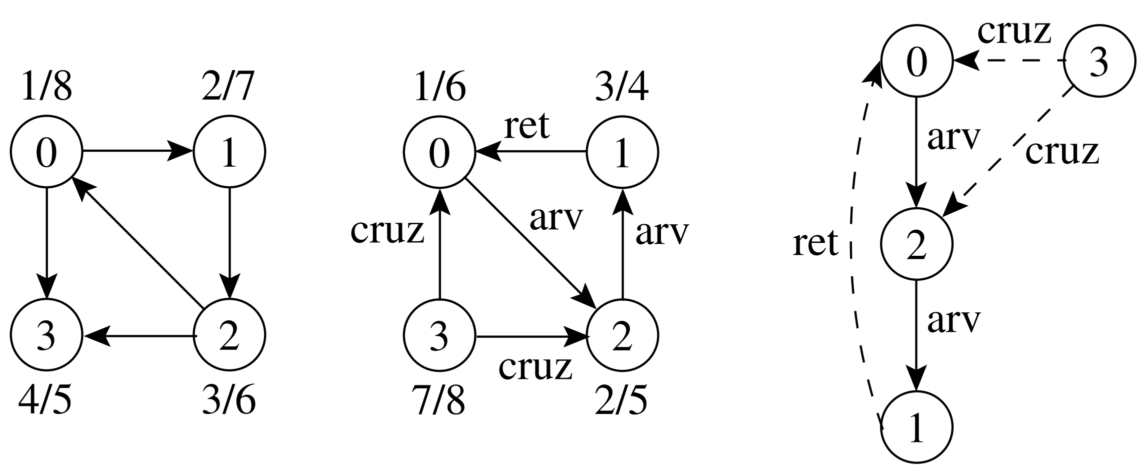
2. Obtem G^T.

3. Chama *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.

4. 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).



(a) (b) (c)

Componentes Fortemente Conectados: Implementação

```
void GrafoTransposto(TipoGrafo *Grafo, TipoGrafo *GrafoT)
{ TipoValorVertice v, Adj;
  TipoPeso Peso;
  TipoApontador Aux;
  FGVazio(GrafoT);
  GrafoT->NumVertices = Grafo->NumVertices;
  GrafoT->NumArestas = Grafo->NumArestas;
  for (v = 0; v <= Grafo->NumVertices - 1; v++)
    { if (!ListaAdjVazia(&v, Grafo))
      { Aux = PrimeiroListaAdj(&v, Grafo);
        FimListaAdj = FALSE;
        while (!FimListaAdj)
          { ProxAdj(&v, Grafo, &Adj, &Peso, &Aux, &FimListaAdj);
            InsereAresta(&Adj, &v, &Peso, GrafoT);
          }
      }
    }
}
```

Componentes Fortemente Conectados: Implementação

```
typedef struct TipoTempoTermino {
  TipoValorTempo t[MAXNUMVERTICES + 1];
  short Restantes[MAXNUMVERTICES + 1];
  TipoValorVertice NumRestantes;
} TipoTempoTermino;

TipoValorVertice MaxTT(TipoTempoTermino *TT, TipoGrafo *Grafo)
{ TipoValorVertice Result; short i = 0, Temp;
  while (!TT->Restantes[i]) i++;
  Temp = TT->t[i]; Result = i;
  for (i = 0; i <= Grafo->NumVertices - 1; i++)
    { if (TT->Restantes[i])
      { if (Temp < TT->t[i]) { Temp = TT->t[i]; Result = i; }
      }
    }
  return Result;
}
```

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

Componentes Fortemente Conectados: Implementação

```
void BuscaEmProfundidadeCfc(TipoGrafo *Grafo, TipoTempoTermino *TT)
{ TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES + 1];
  short Antecessor[MAXNUMVERTICES + 1];
  TipoValorVertice x, VRaiz; Tempo = 0;
  for (x = 0; x <= Grafo->NumVertices - 1; x++)
    { Cor[x] = branco; Antecessor[x] = -1; }
  TT->NumRestantes = Grafo->NumVertices;
  for (x = 0; x <= Grafo->NumVertices - 1; x++)
    TT->Restantes[x] = TRUE;
  while (TT->NumRestantes > 0)
  { VRaiz = MaxTT(TT, Grafo);
    printf("Raiz da proxima arvore:%2d\n", VRaiz);
    VisitaDfs2 (VRaiz, Grafo, TT, &Tempo, d, t, Cor, Antecessor);
  }
}
```

Componentes Fortemente Conectados: Implementação

```
void VisitaDfs2(TipoValorVertice u, TipoGrafo *Grafo,
                TipoTempoTermino *TT, TipoValorTempo *Tempo,
                TipoValorTempo *d, TipoValorTempo *t,
                TipoCor *Cor, short *Antecessor)
{ short FimListaAdj; TipoPeso Peso; TipoApontador Aux;  TipoValorVertice v;
  Cor[u] = cinza;
  (*Tempo)++; d[u] = (*Tempo);
  TT->Restantes[u] = FALSE;
  TT->NumRestantes--;
  printf("Visita%2d Tempo descoberta:%2d cinza\n",u,d[u]);
  getchar();
```

Componentes Fortemente Conectados: Implementação

```
    if (!ListaAdjVazia(&u, Grafo))
    { Aux = PrimeiroListaAdj(&u, Grafo);
      FimListaAdj = FALSE;
      while (!FimListaAdj)
      { ProxAdj(&u, Grafo, &v, &Peso, &Aux, &FimListaAdj);
        if (Cor[v] == branco)
        { Antecessor[v] = u;
          VisitaDfs2 (v, Grafo, TT, Tempo, d, t, Cor, Antecessor);
        }
      }
    }
    Cor[u] = preto; (*Tempo)++;
    t[u] = (*Tempo);
    printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]);
    getchar();
}
```

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

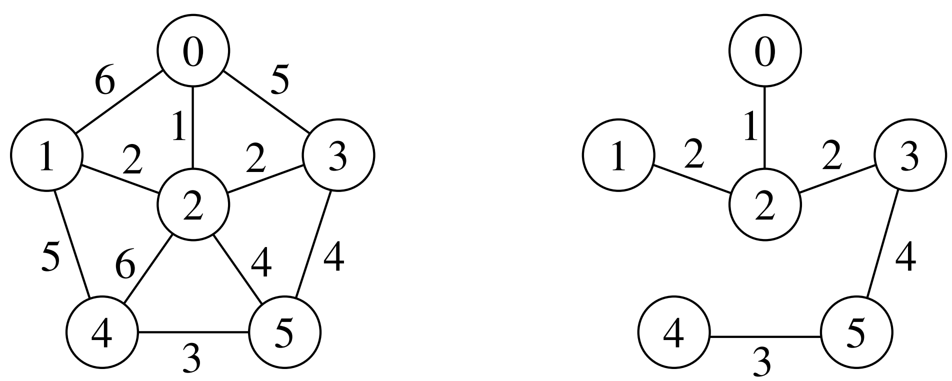
Árvore Geradora Mínima

- 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

- 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)

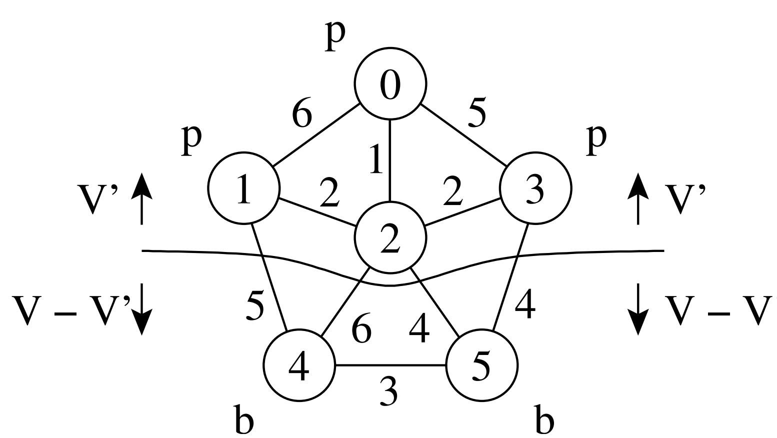
AGM - Algoritmo Genérico

```
void GenericoAGM()
{  S = ∅;
   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;
}
```

- 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**.
- 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*.



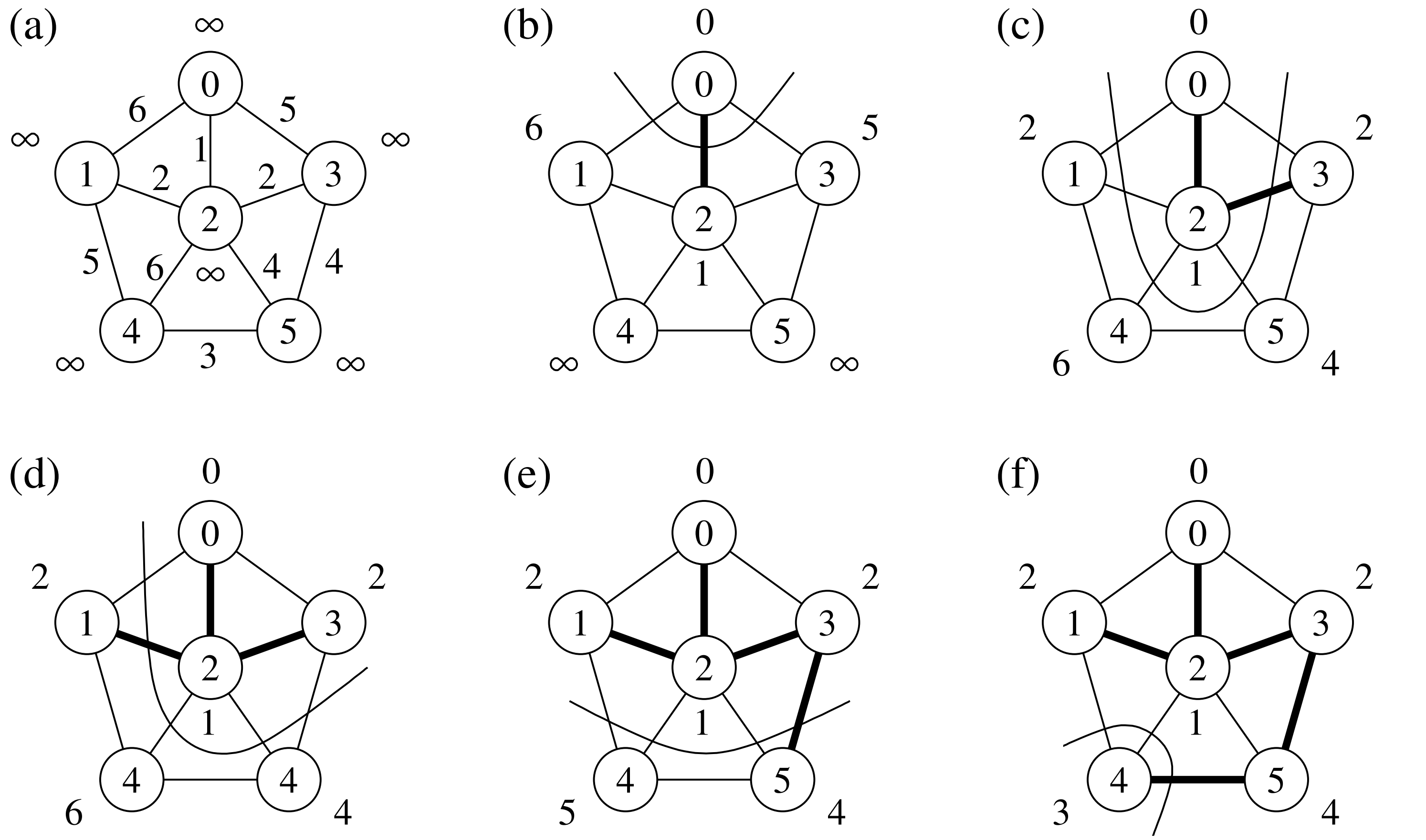
AGM - Teorema para Reconhecer Arestas Seguras

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

Algoritmo de Prim para Obter Uma AGM

- 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



Prim: Operadores para Manter o *Heap* Indireto (1)

```
/** Entra aqui o operador Constroi da Seção 4.1.5 (Programa 4.10) **/
/** Trocando a chamada Refaz (Esq, n , A) por RefazInd (Esq, n, A) **/
void RefazInd(TipoIndice Esq, TipoIndice Dir, TipoItem *A,
              TipoPeso *P, TipoValorVertice *Pos)
{ TipoIndice i = Esq; int j = i * 2; TipoItem x; x = A[i];
  while (j <= Dir)
  { if (j < Dir)
    { if (P[A[j].Chave] > P[A[j+1].Chave]) j++; }
    if (P[x.Chave] <= P[A[j].Chave]) goto L999;
    A[i] = A[j]; Pos[A[j].Chave] = i; i = j;
    j = i * 2;
  }
  L999: A[i] = x;
  Pos[x.Chave] = i;
}
```

Prim: Operadores para Manter o *Heap* Indireto (2)

```
TipoItem RetiraMinInd(TipoItem *A, TipoPeso *P, TipoValorVertice *Pos)
{ TipoItem Result;
  if (n < 1) { printf("Erro: heap vazio\n"); return Result; }
  Result = A[1]; A[1] = A[n];
  Pos[A[n].Chave] = 1; n--;
  RefazInd(1, n, A, P, Pos);
  return Result;
}
```

Prim: Operadores para Manter o *Heap* Indireto (3)

```
void DiminuiChaveInd(TipoIndice i, TipoPeso ChaveNova, TipoItem *A,
                     TipoPeso *P, TipoValorVertice *Pos)
{ TipoItem x;
  if (ChaveNova > P[A[i].Chave])
  { printf("Erro: ChaveNova maior que a chave atual\n");
    return;
  }
  P[A[i].Chave] = ChaveNova;
  while (i > 1 && P[A[i / 2].Chave] > P[A[i].Chave])
    { x = A[i / 2]; A[i / 2] = A[i];
      Pos[A[i].Chave] = i / 2; A[i] = x;
      Pos[x.Chave] = i; i /= 2;
    }
}
```

Algoritmo de Prim: Implementação

```
{−− Entram aqui operadores do tipo grafo do Slide 28 ou Slide 37 ou Slide 45, −−}
{−− e os operadores RefazInd, RetiraMinInd e DiminuiChaveInd do Slide 93 −−}
void AgmPrim(TipoGrafo *Grafo, TipoValorVertice *Raiz)
{ int  Antecessor[MAXNUMVERTICES + 1];
  short  Itensheap[MAXNUMVERTICES + 1];
  Vetor A;
  TipoPeso P[MAXNUMVERTICES + 1];
  TipoValorVertice Pos[MAXNUMVERTICES + 1], u, v;
  TipoItem TEMP;
  for (u = 0; u <= Grafo->NumVertices; u++)
  {  /*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;
  }
  n = Grafo->NumVertices; P[*Raiz] = 0;
  Constroi(A, P, Pos);
```

Algoritmo de Prim: Implementação

```
while (n >= 1)  /*enquanto heap nao vazio*/
{ TEMP = RetiraMinInd(A, P, Pos);
  u = TEMP.Chave; Itensheap[u] = FALSE;
  if (u != *Raiz)
  printf("Aresta de arvore: v[%d] v[%d]",u,Antecessor[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj)
    { ProxAdj(&u, Grafo, &v, &Peso, &Aux, &FimListaAdj);
      if (Itensheap[v] && Peso < P[v])
      { Antecessor[v] = u;
        DiminuiChaveInd(Pos[v], Peso, A, P, Pos);
      }
    }
  }
}
}
```

Algoritmo de Prim: Implementação

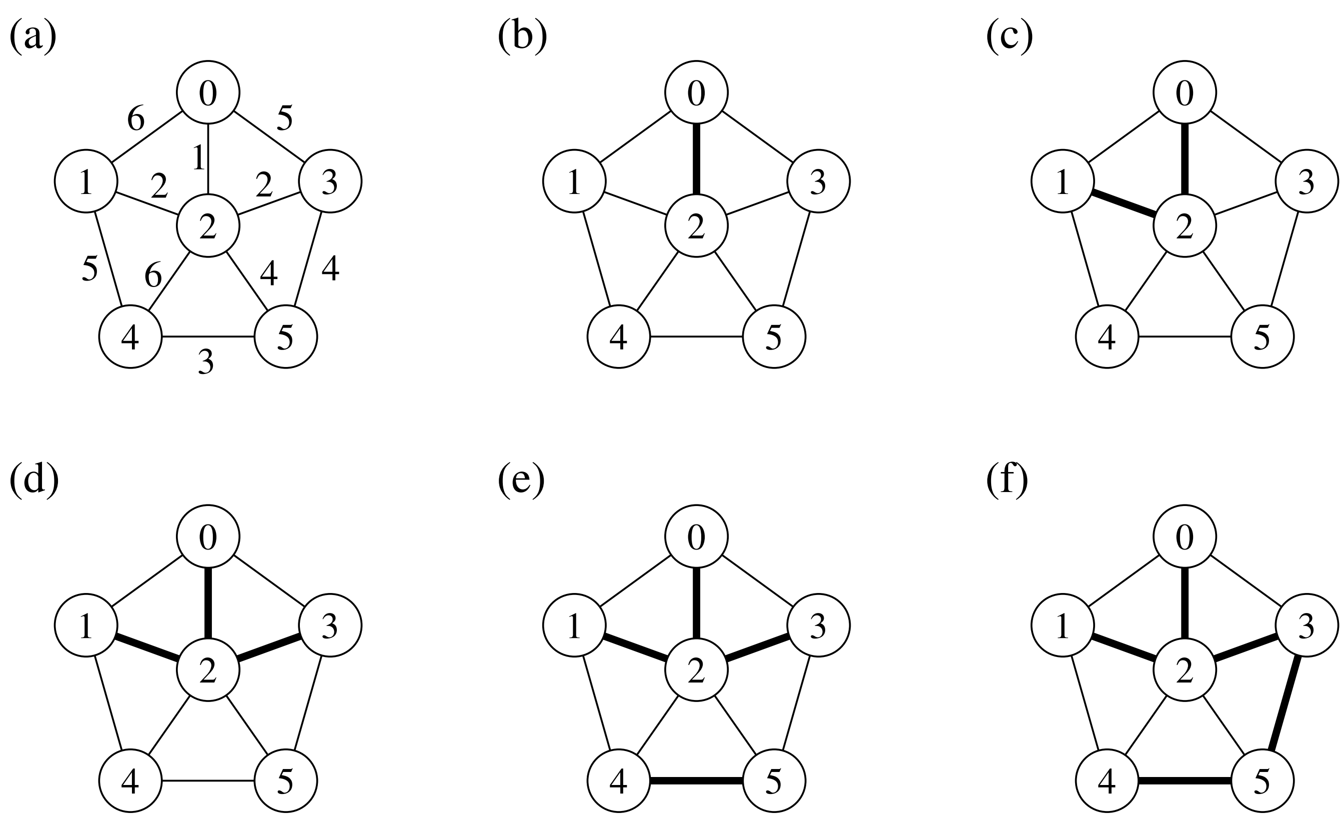
- Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na AGM residem no *heap* A.
- O *heap* contém os vértices, mas a condição do *heap* é mantida pelo peso da aresta através do arranjo $p[v]$ (*heap* indireto).
- $Pos[v]$ fornece a posição do vértice v dentro 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 custa $O(1)$.
- 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

- 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 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:

```
void Kruskal();
{
1. S = ∅;
2. for(v=0;v < Grafo.NumVertices) CriaConjunto (v);
3. Ordena as arestas de A pelo peso;
4. for (cada (u,v) de A tomadas em ordem ascendente de peso)
5.      if (EncontreConjunto (u) != EncontreConjunto (v) )
6.      { S = S+{(u,v)};
7.        Uniao (u,v);
        }
}
```

- A implementação das operações União 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 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: 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:

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

- **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 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.

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:
  - $Antecessor[v] = nil$ 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; }
```

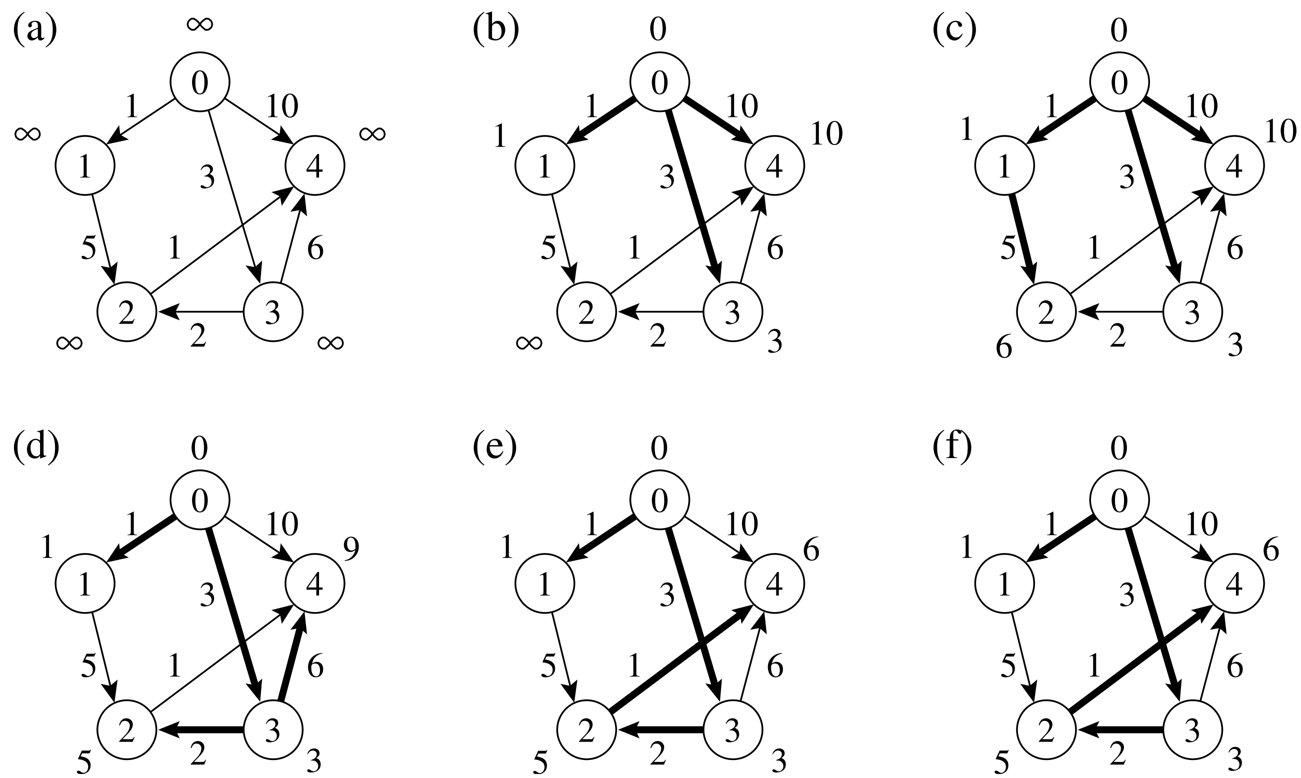
Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra(Grafo, Raiz)
{
for(v=0;v < Grafo.NumVertices;v++)
   p[v] = Infinito;
   Antecessor[v] = −1;
p[Raiz] = 0;
Constroi heap no vetor A;
S = ∅;
while (heap > 1)
   u = RetiraMin(A);
   S = S + u;
   for (v ∈ ListaAdjacentes[u])
      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

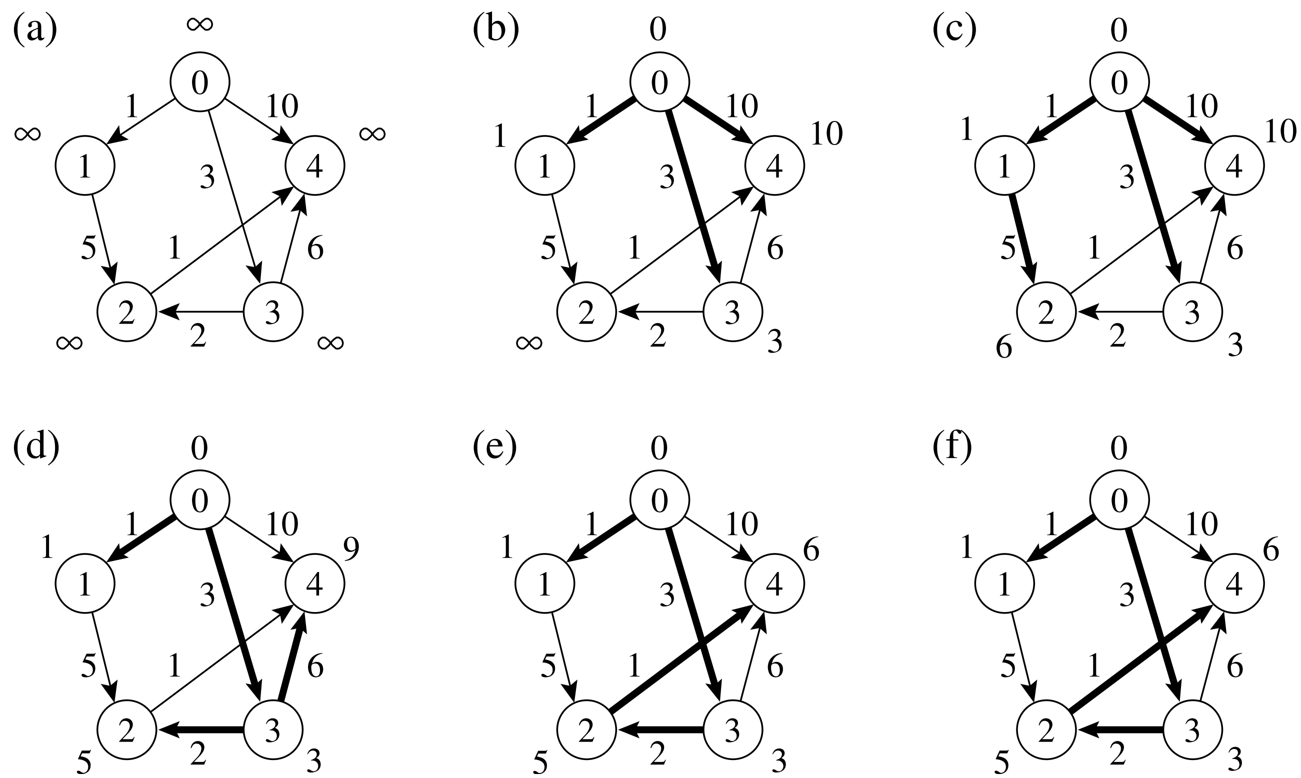
- Invariante: o número de elementos do *heap* é igual a $V - S$ no início do anel **while**.
- A cada iteração do **while**, um vértice u é extraído do *heap* e adicionado ao conjunto S, mantendo assim o invariante.
- *RetiraMin* obtém o vértice u com o caminho mais curto estimado até o momento e adiciona 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



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



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

Algoritmo de Dijkstra: Implementação

```
/** Entram aqui os operadores de uma das implementações de grafos, bem como o ope-
rador Constroi da implementação de filas de prioridades, assim como os operadores Re-
fazInd, RetiraMinInd e DiminuiChaveInd do Programa Constroi **/
void Dijkstra(TipoGrafo *Grafo, TipoValorVertice *Raiz)
{ TipoPeso P[MAXNUMVERTICES + 1];
  TipoValorVertice Pos[MAXNUMVERTICES + 1];
  long Antecessor[MAXNUMVERTICES + 1];
  short Itensheap[MAXNUMVERTICES + 1];
  TipoVetor A; TipoValorVertice u, v; TipoItem temp;
  for (u = 0; u <= Grafo->NumVertices; u++)
  { /*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;
  }
  n = Grafo->NumVertices;
  P[*(Raiz)] = 0;
  Constroi(A, P, Pos);
```

Algoritmo de Dijkstra: Implementação

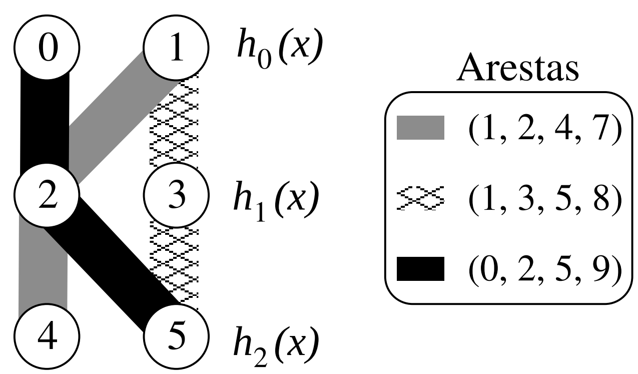
```
  while (n >= 1) /*enquanto heap nao vazio*/
  { temp = RetiraMinInd(A, P, Pos);
    u = temp.Chave; Itensheap[u] = FALSE;
    if (!ListaAdjVazia(&u, Grafo))
    { Aux = PrimeiroListaAdj(&u, Grafo); FimListaAdj = FALSE;
      while (!FimListaAdj)
      { ProxAdj(&u, Grafo, &v, &Peso, &Aux, &FimListaAdj);
        if (P[v] > (P[u] + Peso))
        { P[v] = P[u] + Peso; Antecessor[v] = u;
          DiminuiChaveInd(Pos[v], P[v], A, P, Pos);
          printf("Caminho: v[%d] v[%ld] d[%d]", v, Antecessor[v], P[v]);
          scanf("%*[^\n]"); getchar();
        }
      }
    }
  }
}
```

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 $G_r = (V, A)$ no qual cada aresta $a \in A$ conecta r vértices, sendo r a ordem do hipergrafo.
- Os grafos estudados até agora são 2-grafos (hipergrafos de ordem 2).
- Hipergrafos são utilizados na Seção 5.5.4 sobre ***hashing* perfeito**.
- A figura apresenta um 3-grafo contendo os vértices $\{0, 1, 2, 3, 4, 5\}$, as arestas $\{(1, 2, 4), (1, 3, 5), (0, 2, 5)\}$ e os pesos 7, 8 e 9, respectivamente.



O Tipo Abstrato de Dados Hipergrafo: Operações

1. Criar um hipergrafo vazio. A operação retorna um hipergrafo contendo $|V|$ vértices e nenhuma aresta.
2. Inserir uma aresta no hipergrafo. Recebe a aresta $(V_1, V_2, \ldots, V_r)$ e seu peso para serem inseridos no hipergrafo.
3. Verificar se existe determinada aresta no hipergrafo: retorna *true* se a aresta $(V_1, V_2, \ldots, V_r)$ estiver presente, senão retorna *false*.
4. Obter a lista de arestas incidentes em determinado vértice. Essa operação será tratada separadamente logo a seguir.
5. Retirar uma aresta do hipergrafo. Retira a aresta $(V_1, V_2, \ldots, V_r)$ do hipergrafo e a retorna.
6. Imprimir um hipergrafo.
7. Obter a aresta de menor peso de um hipergrafo. A operação retira a aresta de menor peso dentre as arestas do hipergrafo e a retorna.

O Tipo Abstrato de Dados Hipergrafo: Operações

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

- Para implementar esse operador de forma independente da representação escolhida para a aplicação em pauta, precisamos de três operações sobre hipergrafos, a saber:

  1. Verificar se a lista de arestas incidentes em um vértice v está vazia. A operação retorna *true* se a lista estiver vazia, senão retorna *false*.
  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: Operações

- A forma mais adequada para representar um hipergrafo é por meio de estruturas de dados em que para cada vértice v do grafo é mantida uma lista das arestas que incidem sobre o vértice v, o que implica a representação explícita de cada aresta do hipergrafo.
- Essa é uma estrutura orientada a arestas e não a vértices como as representações apresentadas até agora.
- Existem duas representações usuais para hipergrafos: as matrizes de incidência e as listas de incidência.

Implementação por Matrizes de Incidência

- A **m**atriz de incidência de $G_r = (V, A)$ contendo n vértices e m arestas é uma matriz $n \times m$ de *bits*, em que $A[i, j] = 1$ se o vértice i participar da aresta j.
- Para hipergrafos ponderados, $A[i, j]$ contém o rótulo ou peso associado à aresta e a matriz não é de *bits*.
- Se o vértice i não participar da aresta j, então é necessário utilizar um valor que não possa ser usado como rótulo ou peso, tal como 0 ou branco.
- A figura ilustra a representação por matrizes de incidência para o hipergrafo do slide 119.

	0	1	2
0			9
1	7	8	
2	7		9
3		8	
4	7		
5		8	9

Implementação por Matrizes de Incidência

- A representação por matrizes de incidência demanda muita memória para hipergrafos **d**ensos, em que $|A|$ é próximo de $|V|^2$.
- Nessa representação, o tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$.
- Logo, essa representação é muito útil para algoritmos em que necessitamos saber com rapidez se um vértice participa de determinada aresta.
- A maior desvantagem é que a matriz necessita $\Omega(|V|^3)$ de espaço. Isso significa que simplesmente ler ou examinar a matriz tem complexidade de tempo $O(|V|^3)$.

Implementação por Matrizes de Incidência

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500
#define MAXR 5
typedef int TipoValorVertice;
typedef int TipoValorAresta;
typedef int Tipor;
typedef int TipoPesoAresta;
typedef TipoValorVertice TipoArranjoVertices[MAXR];
typedef struct TipoAresta {
  TipoArranjoVertices Vertices;
  TipoPesoAresta Peso;
} TipoAresta;
typedef struct TipoGrafo {
  TipoPesoAresta Mat[MAXNUMVERTICES][MAXNUMARESTAS];
  TipoValorVertice NumVertices;
  TipoValorAresta NumArestas;
  TipoValorAresta ProxDisponivel;
  Tipor r;
} TipoGrafo;
```

Implementação por Matrizes de Incidência

- No campo Mat os itens são armazenados em um **a**rray de duas dimensões de tamanho suficiente para armazenar o grafo.
- As constantes MaxNumVertices e MaxNumArestas definem o maior número de vértices e de arestas que o grafo pode ter e r define o número de vértices de cada aresta.
- Uma possível implementação para as primeiras seis operações definidas anteriormente é mostrada no slide a seguir.
- O procedimento ArestasIguais permite a comparação de duas arestas, a um custo $O(r)$.
- O procedimento InsereAresta tem custo $O(r)$ e os procedimentos ExisteAresta e RetiraAresta têm custo $r \times |A|$, o que pode ser considerado $O(|A|)$ porque r é geralmente uma constante pequena.

Implementação por Matrizes de Incidência

```
short ArestasIguais (TipoArranjoVertices * Vertices,
                     TipoValorAresta NumAresta,
                     TipoGrafo * Grafo)
{ short Aux = TRUE;   Tipor v = 0;
  while (v < Grafo->r && Aux == TRUE)
  { if (Grafo->Mat[(*Vertices)[v]][NumAresta]<=0) Aux = FALSE;
    v = v + 1;
  }
  return Aux;
}

void FGVazio (TipoGrafo * Grafo)
{ int i,j;
  Grafo->ProxDisponivel = 0;
  for (i = 0; i < Grafo->NumVertices; i++)
    for (j = 0; j < Grafo->NumArestas; j++) Grafo->Mat[i][j] = 0;
}
```

Implementação por Matrizes de Incidência

```
void InsereAresta (TipoAresta * Aresta, TipoGrafo * Grafo)
{ int i;
  if (Grafo->ProxDisponivel == MAXNUMARESTAS)
  printf("Nao ha espaco disponivel para a aresta\n");
  else
  { for (i = 0; i < Grafo->r; i++)
    Grafo->Mat[Aresta->Vertices[i]][Grafo->ProxDisponivel]=Aresta->Peso;
    Grafo->ProxDisponivel = Grafo->ProxDisponivel + 1;
  }
}
short ExisteAresta (TipoAresta * Aresta, TipoGrafo * Grafo)
{ TipoValorAresta ArestaAtual = 0;
  short EncontrouAresta = FALSE;
  while (ArestaAtual < Grafo->NumArestas &&
         EncontrouAresta == FALSE)
  { EncontrouAresta =
      ArestasIguais(&(Aresta->Vertices), ArestaAtual, Grafo);
    ArestaAtual = ArestaAtual + 1;
  }
  return EncontrouAresta;
}
```

Implementação por Matrizes de Incidência

```
TipoAresta RetiraAresta (TipoAresta * Aresta, TipoGrafo * Grafo)
{ TipoValorAresta ArestaAtual = 0;
  int i; short EncontrouAresta = FALSE;
  while (ArestaAtual<Grafo->NumArestas& EncontrouAresta == FALSE)
  { if (ArestasIguais(&(Aresta->Vertices), ArestaAtual, Grafo))
    { EncontrouAresta = TRUE;
      Aresta->Peso = Grafo->Mat[Aresta->Vertices[0]][ArestaAtual];
      for (i = 0; i < Grafo->r; i++)
        Grafo->Mat[Aresta->Vertices[i]][ArestaAtual] = -1;
    }
    ArestaAtual = ArestaAtual + 1;
  }
  return *Aresta;
}
```

Implementação por Matrizes de Incidência

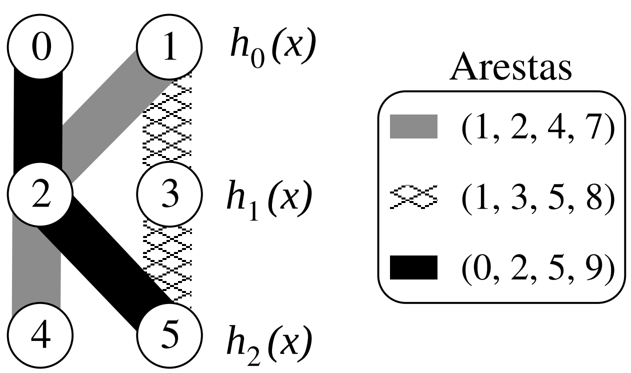
```
void ImprimeGrafo (TipoGrafo * Grafo)
{ int i,j; printf("   ");
  for (i = 0; i < Grafo->NumArestas; i++) printf("%3d", i);
  printf("\n");
  for (i = 0; i < Grafo->NumVertices; i++)
    { printf("%3d", i);
      for (j = 0; j < Grafo->NumArestas; j++)
        printf("%3d", Grafo->Mat[i][j]);
      printf("\n");
    }
}
short ListaIncVazia (TipoValorVertice * Vertice, TipoGrafo * Grafo)
{ short ListaVazia = TRUE; TipoApontador ArestaAtual = 0;
  while (ArestaAtual < Grafo->NumArestas && ListaVazia == TRUE)
    { if (Grafo->Mat[*Vertice][ArestaAtual] > 0) ListaVazia = FALSE;
      else ArestaAtual = ArestaAtual + 1;
    }
  return ListaVazia;
}
```

Implementação por Matrizes de Incidência

```
TipoApontador PrimeiroListaInc(TipoValorVertice * Vertice, TipoGrafo * Grafo)
{ TipoApontador ArestaAtual = 0;
  short Continua = TRUE; TipoApontador Resultado = 0;
  while (ArestaAtual < Grafo->NumArestas && Continua == TRUE)
    { if (Grafo->Mat[*Vertice][ArestaAtual] > 0) { Resultado = ArestaAtual; Continua = FALSE; }
      else ArestaAtual = ArestaAtual + 1;
    }
  if (ArestaAtual == Grafo->NumArestas) printf("Erro: Lista incidencia vazia\n");
  return Resultado;
}
void ProxArestaInc (TipoValorVertice * Vertice, TipoGrafo * Grafo,
                    TipoValorAresta * Inc, TipoPesoAresta * Peso,
                    TipoApontador * Prox, short * FimListaAdj)
{ *Inc = *Prox;
  *Peso = Grafo->Mat[*Vertice][*Prox];
  *Prox = *Prox + 1;
  while (*Prox < Grafo->NumArestas && Grafo->Mat[*Vertice][*Prox] == 0) *Prox = *Prox + 1;
  *FimListaAdj = (*Prox == Grafo->NumArestas);
}
```

Implementação por Listas de Incidência Usando Arranjos

- A estrutura de dados usada para representar $G_r = (V, A)$ por meio de listas de incidência foi proposta por Ebert (1987).
- A estrutura usa arranjos para armazenar as arestas e as listas de arestas incidentes a cada vértice. A parte (a) da figura mostra o mesmo 3-grafo de 6 vértices e 3 arestas visto anteriormente e a parte (b) a sua representação por listas de incidência.



(a)

Arestas $|A|=3$

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

Prim $|V|=3$

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

Prox $r|A|=3x3=9$

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

(b)

Implementação por Listas de Incidência Usando Arranjos

- As arestas são armazenadas no arranjo $Arestas$. Em cada posição a do arranjo são armazenados os r vértices da aresta a e o seu $Peso$.
- As listas de arestas incidentes nos vértices do hipergrafo são armazenadas nos arranjos $Prim$ e $Prox$.
- O elemento $Prim[v]$ define o ponto de entrada para a lista de arestas incidentes no vértice v, enquanto $Prox[Prim[v]]$, $Prox[Prox[Prim[v]]]$ e assim por diante definem as arestas subsequentes que contêm v.
- $Prim$ deve possuir $|V|$ entradas, uma para cada vértice.
- $Prox$ deve possuir $r|A|$ entradas, pois cada aresta a é armazenada na lista de arestas incidentes a cada um de seus r vértices.
- A complexidade de espaço é $O(|V| + |A|)$, pois $Arestas$ tem tamanho $O(|A|)$, $Prim$ tem tamanho $O(|V|)$ e $Prox$ tem tamanho $r \times |A| = O(|A|)$, porque r é geralmente uma constante pequena.

Implementação por Listas de Incidência Usando Arranjos

- 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 $Prox[\ldots Prim[v] \ldots] = -1$.
- Assim, 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.
- Por exemplo, ao se percorrer a lista das arestas do vértice 2, os valores $\{5, 3\}$ são obtidos, os quais representam as arestas que contêm o vértice 2 (arestas 2 e 0), ou seja, $\{5 \bmod 3 = 2, 3 \bmod 3 = 0\}$.

Implementação por Listas de Incidência Usando Arranjos

- Os valores armazenados em $Prim$ e $Prox$ são obtidos pela equação $i = a + j|A|$, $0 \leq j \leq r - 1$, e a um índice de uma aresta no arranjo $Arestas$. As entradas de $Prim$ são iniciadas com -1.
- Ao inserir a aresta $a = 0$ contendo os vértices $(1, 2, 4)$, temos que:
  $i = 0 + 0 \times 3 = 0$, $Prox[i = 0] = Prim[1] = -1$ e $Prim[1] = i = 0$,
  $i = 0 + 1 \times 3 = 3$, $Prox[i = 3] = Prim[2] = -1$ e $Prim[2] = i = 3$,
  $i = 0 + 2 \times 3 = 6$, $Prox[i = 6] = Prim[4] = -1$ e $Prim[4] = i = 6$.
- Ao inserir a aresta $a = 1$ contendo os vértices $(1, 3, 5)$ temos que:
  $i = 1 + 0 \times 3 = 1$, $Prox[i = 1] = Prim[1] = 0$ e $Prim[1] = i = 1$,
  $i = 1 + 1 \times 3 = 4$, $Prox[i = 4] = Prim[3] = -1$ e $Prim[3] = i = 4$,
  $i = 1 + 2 \times 3 = 7$, $Prox[i = 7] = Prim[5] = -1$ e $Prim[5] = i = 7$.
- Ao inserir a aresta $a = 2$ contendo os vértices $(0, 2, 5)$ temos que:
  $i = 2 + 0 \times 3 = 2$, $Prox[i = 2] = Prim[0] = -1$ e $Prim[0] = i = 2$,
  $i = 2 + 1 \times 3 = 5$, $Prox[i = 5] = Prim[2] = 3$ e $Prim[2] = i = 5$,
  $i = 2 + 2 \times 3 = 8$, $Prox[i = 8] = Prim[5] = 7$ e $Prim[5] = i = 8$.

Implementação por Listas de Incidência Usando Arranjos

- O programa a seguir apresenta a estrutura de dados utilizando listas de incidência implementadas por meio de arranjos.
- A estrutura de dados contém os três arranjos necessários para representar um hipergrafo, como ilustrado na figura do slide 132:
  - A variável r é utilizada para armazenar a ordem do hipergrafo.
  - A variável NumVertices contém o número de vértices do hipergrafo.
  - A variável NumArestas contém o número de arestas do hipergrafo.
  - A variável ProxDisponivel contém a próxima posição disponível para inserção de uma nova aresta.

Implementação por Listas de Incidência Usando Arranjos

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500
#define MAXR 5
#define MAXTAMPROX MAXR * MAXNUMARESTAS
#define INDEFINIDO -1
typedef int TipoValorVertice;
typedef int TipoValorAresta;
typedef int Tipor;
typedef int TipoMaxTamProx;
typedef int TipoPesoAresta;
typedef TipoValorVertice TipoArranjoVertices[MAXR + 1];
typedef struct TipoAresta {
    TipoArranjoVertices Vertices;
    TipoPesoAresta Peso;
} TipoAresta;
typedef TipoAresta TipoArranjoArestas[MAXNUMARESTAS + 1];
```

Implementação por Listas de Incidência Usando Arranjos

```
typedef struct TipoGrafo {
  TipoArranjoArestas Arestas;
  TipoValorVertice Prim[MAXNUMARESTAS + 1];
  TipoMaxTamProx Prox[MAXTAMPROX + 2];
  TipoMaxTamProx ProxDisponivel;
  TipoValorVertice NumVertices;
  TipoValorAresta NumArestas;
  Tipor r;
} TipoGrafo;
typedef int TipoApontador;
```

Implementação por Listas de Incidência Usando Arranjos

- Uma possível implementação para as primeiras seis operações definidas anteriormente é mostrada no programa a seguir.
- O procedimento ArestasIguais permite a comparação de duas arestas cujos vértices podem estar em qualquer ordem (custo $O(r^2)$).
- O procedimento InsereAresta insere uma aresta no grafo (custo $O(r)$).
- O procedimento ExisteAresta verifica se uma aresta está presente no grafo (custo equivalente ao **g**rau de cada vértice da aresta no grafo).
- O procedimento RetiraAresta primeiro localiza a aresta no grafo, retira a mesma da lista de arestas incidentes a cada vértice em Prim e Prox e marca a aresta como removida no arranjo Arestas. A aresta marcada no arranjo Arestas não é reutilizada, pois não mantemos uma lista de posições vazias.

Implementação por Listas de Incidência Usando Arranjos

```
short ArestasIguais(TipoArranjoVertices V1,
                    TipoValorAresta *NumAresta, TipoGrafo *Grafo)
{ Tipor i = 0, j;
  short Aux = TRUE;
  while (i < Grafo->r && Aux)
  { j = 0;
    while ((V1[i] != Grafo->Arestas[*NumAresta].Vertices[j]) &&
           (j < Grafo->r)) j++;
    if (j == Grafo->r) Aux = FALSE;
    i++;
  }
  return Aux;
}
void FGVazio(TipoGrafo *Grafo)
{ int i;
  Grafo->ProxDisponivel = 0;
  for (i = 0; i < Grafo->NumVertices; i++) Grafo->Prim[i] = -1;
}
```

Implementação por Listas de Incidência Usando Arranjos

```
void InsereAresta(TipoAresta *Aresta, TipoGrafo *Grafo)
{ int i, Ind;
  if (Grafo->ProxDisponivel == MAXNUMARESTAS + 1)
  printf ("Nao ha espaco disponivel para a aresta\n");
  else
  { Grafo->Arestas[Grafo->ProxDisponivel] = *Aresta;
    for (i = 0; i < Grafo->r; i++)
    { Ind = Grafo->ProxDisponivel + i * Grafo->NumArestas;
      Grafo->Prox[Ind] =
        Grafo->Prim[Grafo->Arestas[Grafo->ProxDisponivel].Vertices[i]];
      Grafo->Prim[Grafo->Arestas[Grafo->ProxDisponivel].Vertices[i]]=Ind;
    }
  }
  Grafo->ProxDisponivel++;
}
```

Implementação por Listas de Incidência Usando Arranjos

```
short ExisteAresta(TipoAresta *Aresta,
                   TipoGrafo *Grafo)
{ Tipor v;
  TipoValorAresta A1;
  int Aux;
  short EncontrouAresta;
  EncontrouAresta = FALSE;
  for(v = 0; v < Grafo->r; v++)
  { Aux = Grafo->Prim[Aresta->Vertices[v]];
    while (Aux != -1 && !EncontrouAresta)
      { A1 = Aux % Grafo->NumArestas;
        if (ArestasIguais(Aresta->Vertices, &A1, Grafo))
        EncontrouAresta = TRUE;
        Aux = Grafo->Prox[Aux];
      }
  }
  return EncontrouAresta;
}
```

Implementação por Listas de Incidência Usando Arranjos

```
TipoAresta RetiraAresta(TipoAresta *Aresta, TipoGrafo *Grafo)
{ int Aux, Prev, i; TipoValorAresta A1; Tipor v;
  for (v = 0; v < Grafo->r; v++)
  { Prev = INDEFINIDO;
    Aux = Grafo->Prim[Aresta->Vertices[v]];
    A1 = Aux % Grafo->NumArestas;
    while(Aux >= 0 && !ArestasIguais(Aresta->Vertices, &A1, Grafo))
    { Prev = Aux;
      Aux = Grafo->Prox[Aux];
      A1 = Aux % Grafo->NumArestas;
    }
    if (Aux >= 0)
    { if (Prev == INDEFINIDO) Grafo->Prim[Aresta->Vertices[v]] = Grafo->Prox[Aux];
      else Grafo->Prox[Prev] = Grafo->Prox[Aux];
    }
  }
  TipoAresta Resultado = Grafo->Arestas[A1];
  for (i = 0; i < Grafo->r; i++) Grafo->Arestas[A1].Vertices[i] = INDEFINIDO;
  Grafo->Arestas[A1].Peso = INDEFINIDO;
  return Resultado;
}
```

Implementação por Listas de Incidência Usando Arranjos

```
void ImprimeGrafo(TipoGrafo *Grafo)
{ int i, j;
  printf(" Arestas: Num Aresta, Vertices, Peso \n");
  for (i = 0; i < Grafo->NumArestas; i++)
  { printf ("%2d", i);
    for (j = 0; j < Grafo->r; j++) printf ("%3d", Grafo->Arestas[i].Vertices[j]);
    printf ("%3d\n", Grafo->Arestas[i].Peso);
  }
  printf ("Lista arestas incidentes a cada vertice:\n");
  for (i = 0 ; i < Grafo->NumVertices; i++)
  { printf ("%2d", i);
    j = Grafo->Prim[i];
    while (j != INDEFINIDO)
      { printf ("%3d", j % Grafo->NumArestas);
        j = Grafo->Prox[j];
      }
    printf("\n");
  }
}
```

Implementação por Listas de Incidência Usando Arranjos

```
/* operadores para obter a lista de arestas incidentes a um vertice */
short ListaIncVazia(TipoValorVertice *Vertice,
                    TipoGrafo *Grafo)
{ return Grafo->Prim[*Vertice] == -1; }

TipoApontador PrimeiroListaInc(TipoValorVertice *Vertice,
                               TipoGrafo *Grafo)
{ return Grafo->Prim[*Vertice]; }

void ProxArestaInc(TipoValorVertice *Vertice, TipoGrafo *Grafo,
                   TipoValorAresta *Inc, TipoPesoAresta *Peso,
                          TipoApontador *Prox,  short *FimListaInc)
/* Retorna Inc apontado por Prox */
{ *Inc = *Prox % Grafo->NumArestas;
  *Peso = Grafo->Arestas[*Inc].Peso;
  if (Grafo->Prox[*Prox] == INDEFINIDO)
  *FimListaInc = TRUE;
  else *Prox = Grafo->Prox[*Prox];
}
```

Programa Teste para Operadores do Tipo Abstrato de Dados Hipergrafo

```
/** Entram aqui tipos do Slide 125 ou do Slide 137 **/
/** Entram aqui operadores do Slide 127 ou do Slide 138 **/
int main() {
  TipoApontador Ap;
  int i, j;
  TipoValorAresta Inc;
  TipoValorVertice V1;
  TipoAresta Aresta;
  TipoPesoAresta Peso;
  TipoGrafo Grafo;
  short FimListaInc;
  printf("Hipergrafo r: "); scanf("%d*[^\n]", &Grafo.r);
  printf("No. vertices: "); scanf("%d*[^\n]", &Grafo.NumVertices);
  printf("No. arestas: ");  scanf("%d*[^\n]", &Grafo.NumArestas);
  getchar();
  FGVazio(&Grafo);
```

Programa Teste para Operadores do Tipo Abstrato de Dados Hipergrafo

```
for (i = 0; i < Grafo.NumArestas; i++)
{ printf ("Insere Aresta e Peso: ");
  for (j=0; j < Grafo.r; j++) scanf("%d*[^\n]",&Aresta.Vertices[j]);
  scanf("%d*[^\n]", &Aresta.Peso);
  getchar();
  InsereAresta (&Aresta, &Grafo);
}
// Imprime estrutura de dados
printf ("prim: "); for (i = 0; i < Grafo.NumVertices; i++)
printf ("%3d", Grafo.Prim[i]); printf("\n");
printf ("prox: "); for (i = 0; i < Grafo.NumArestas * Grafo.r; i++)
printf ("%3d", Grafo.Prox[i]); printf("\n");
ImprimeGrafo(&Grafo);
getchar();
printf ("Lista arestas incidentes ao vertice: ");
scanf("%d*[^\n]", &V1);
```

Programa Teste para Operadores do Tipo Abstrato de Dados Hipergrafo

```
if (!ListaIncVazia(&V1, &Grafo))
{ Ap = PrimeiroListaInc(&V1, &Grafo);
  FimListaInc = FALSE;
  while (!FimListaInc)
    { ProxArestaInc (&V1, &Grafo, &Inc, &Peso, &Ap, &FimListaInc);
      printf ("%2d (%d)", Inc % Grafo.NumArestas, Peso);
    }
  printf("\n"); getchar();
}
else printf ("Lista vazia\n");

printf ("Existe aresta: ");
for (j = 0; j < Grafo.r; j++) scanf("%d*[^\n]", &Aresta.Vertices[j]);
getchar();
```

Programa Teste para Operadores do Tipo Abstrato de Dados Hipergrafo

```
  if (ExisteAresta(&Aresta, &Grafo))
    printf ("Sim\n");
  else printf ("Nao\n");
  printf ("Retira aresta: ");
  for (j = 0; j < Grafo.r; j++) scanf("%d*[^\n]", &Aresta.Vertices[j]);
  getchar();
  if (ExisteAresta(&Aresta, &Grafo))
  { Aresta = RetiraAresta(&Aresta, &Grafo);
    printf ("Aresta retirada:");
    for (i = 0; i < Grafo.r; i++) printf ("%3d", Aresta.Vertices[i]);
    printf ("%4d\n", Aresta.Peso);
  }
  else printf ("Aresta nao existe\n");
  ImprimeGrafo(&Grafo);
  return 0;
}
```