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

PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

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.

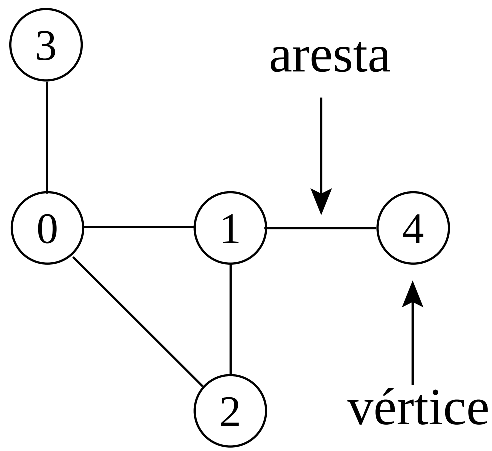
Aplicações

- 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.

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

Conceitos Básicos

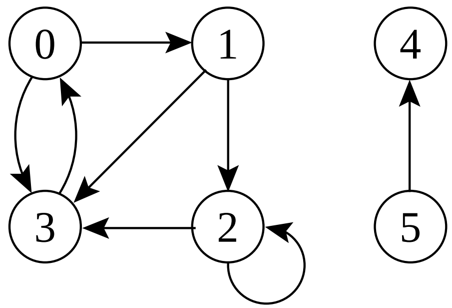
- **Grafo**: conjunto de vértices e arestas.
- **Vértice**: objeto simples que pode ter nome e outros atributos.
- **Aresta**: conexão entre dois vértices.



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

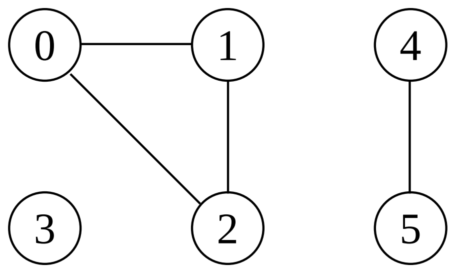
Grafos Direcionados

- 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*.



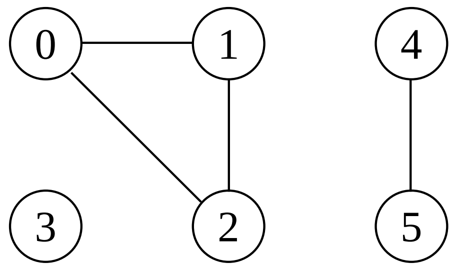
Grafos Não Direcionados

- 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.

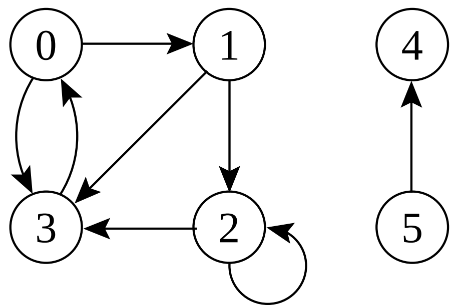


Grau de um Vértice

- 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.



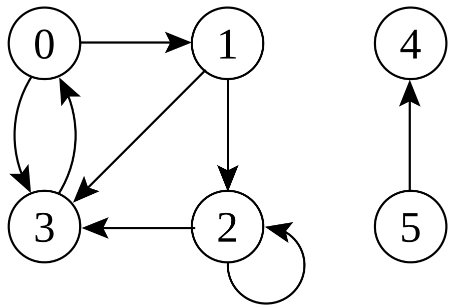
- Em grafos direcionados
  - O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).
  - Ex.: O vértice 2 tem *in-degree* 2, *out-degree* 2 e grau 4.



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 $(v_0, v_1, v_2, \ldots, v_k)$ 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.

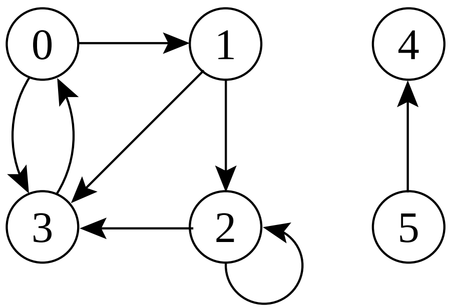
Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



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

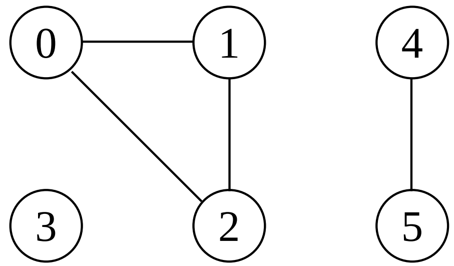
Ex.: O caminho $(0, 1, 2, 3, 0)$ forma um ciclo. O caminho $(0, 1, 3, 0)$ forma o mesmo ciclo que os caminhos $(1, 3, 0, 1)$ e $(3, 0, 1, 3)$.



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.



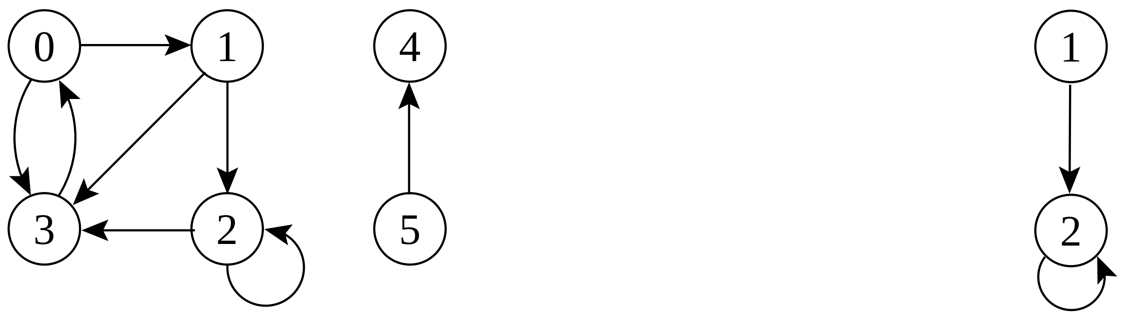
PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

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 $\{1, 2, 4, 5\}$.



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.

PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

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.

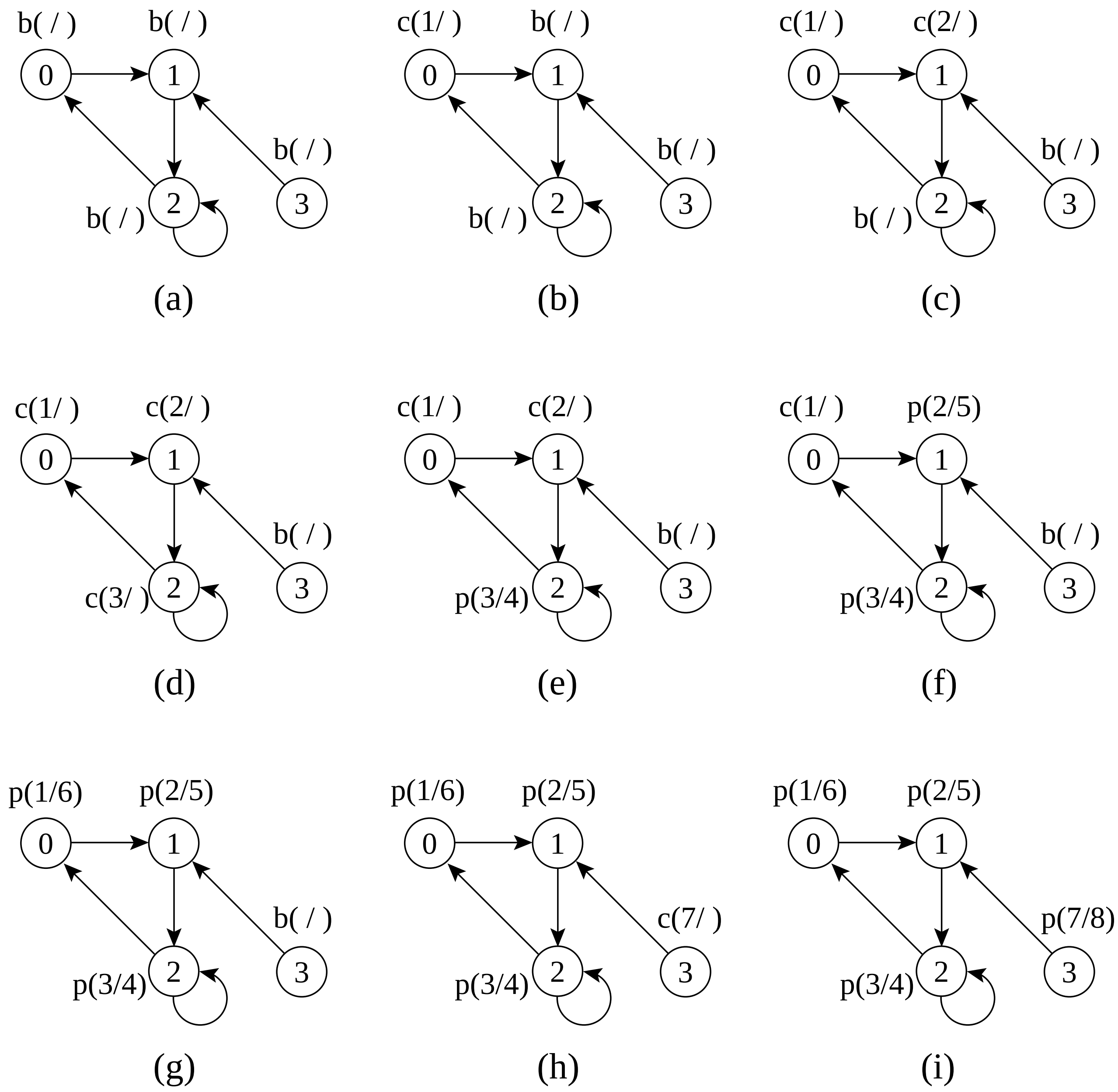
- Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto.

- 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: Exemplo



Busca em Largura

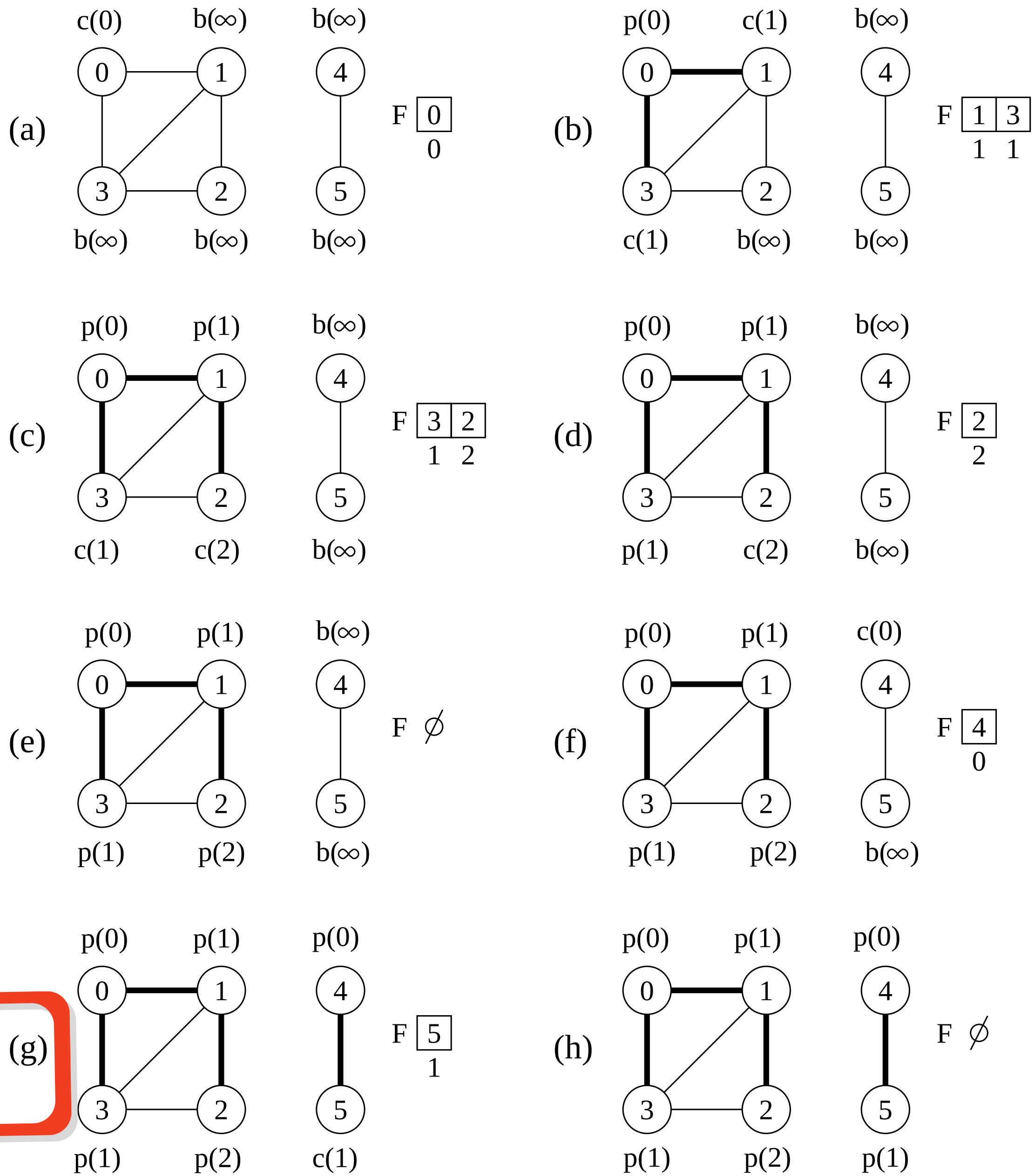
- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.
- O grafo $G(V, A)$ pode ser direcionado ou não direcionado.

PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

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: Exemplo



PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

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

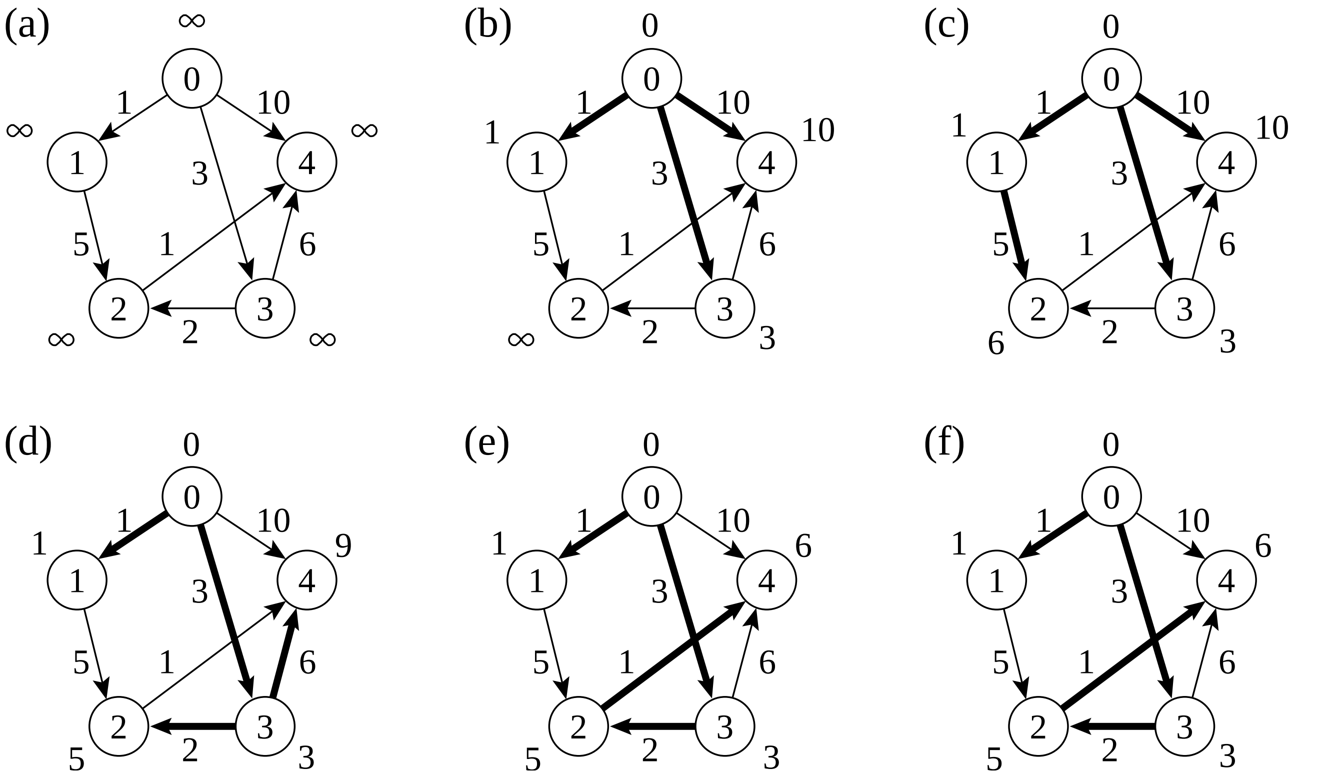
```
void Dijkstra(Grafo, Raiz)
{
for(v=0;v < Grafo.NumVertices;v++)
   p[v] = Infinito;
   Antecessor[v] = −1;
p[Raiz] = 0;
Constroi heap no vetor A;
S = ∅;
while (heap > 1)
   u = RetiraMin(A);
   S = S + u;
   for (v ∈ ListaAdjacentes[u])
      if(p[v] > p[u] + peso da aresta(u,v))
         p[v] =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

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

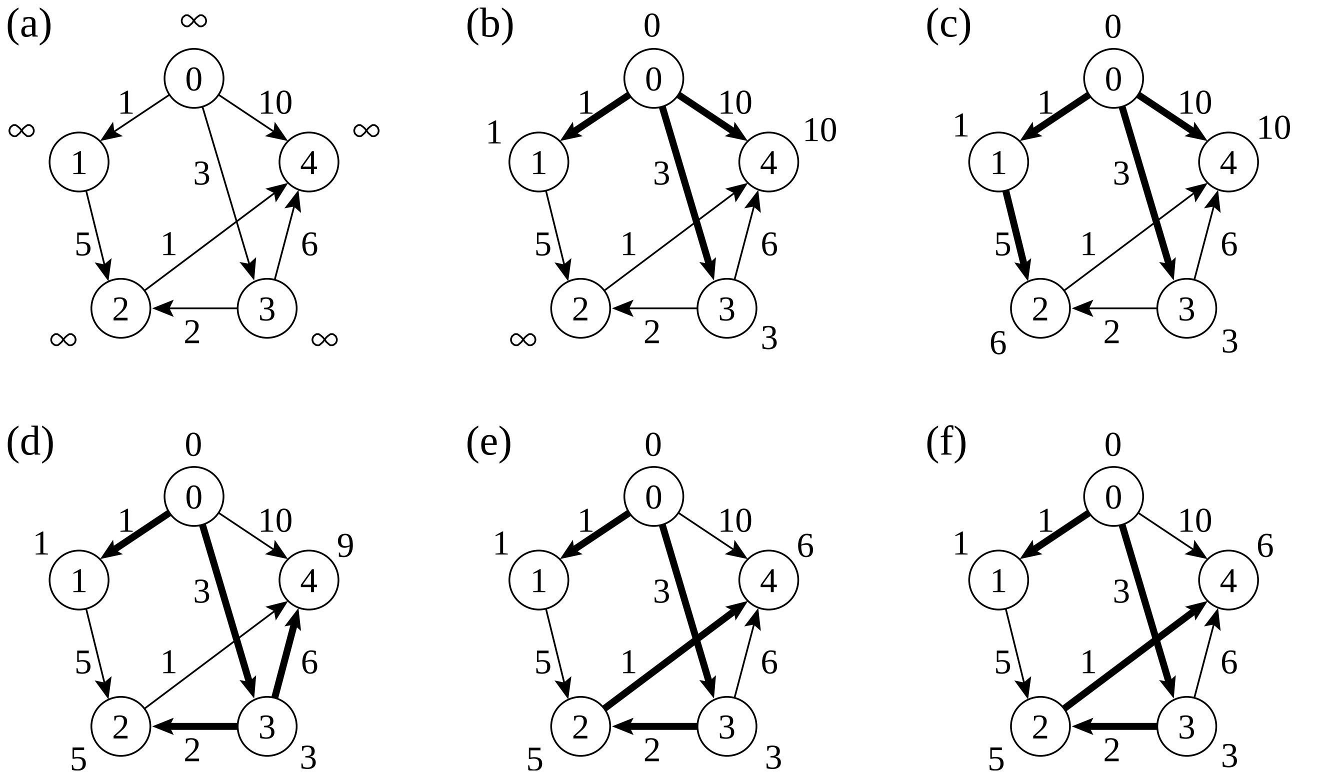
Algoritmo de Dijkstra: Exemplo



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

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

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 operador Constroi da implementação de filas de prioridades, assim como os operadores RefazInd, 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 $p[u] = \delta(Raiz, u)$.