ESTRUTURAS DE DADOS II

prof. Alexandre César Muniz de Oliveira

1. Grafos
2. Ordenação
3. Busca

Sugestão bibliográfica:

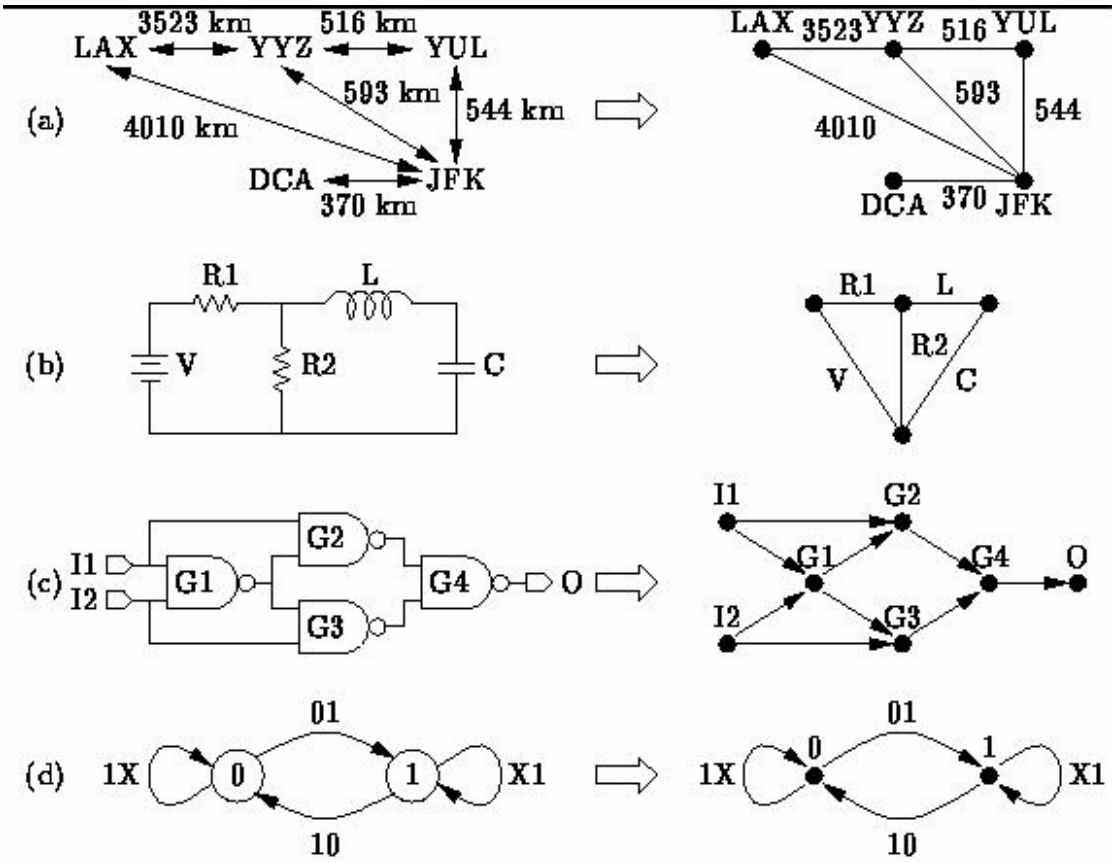
- **ESTRUTURAS DE DADOS USANDO C Aaron M. Tenenbaum, et alli**
- **DATA STRUCTURES AND ALGORITHMS WITH OBJECT-ORIENTED DESIGN PATTERNS IN JAVA -BRUNO R. PREISS,**
- **DATA STRUCTURES, AN ADVANCED APPROACH USING C JEFFREY ESAKOV & TOM WEISS**
- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED) MICHAEL GODRICH & ROBERTO TAMASSIA**

GRAFOS

1 Fundamentos

Um grafo G = {V,A}, onde V= conjunto de vértices, não vazio, e A= conjunto de arestas, formadas por pares de vértices a=(v,w) que formam uma relação binária entre eles. **Logo:**

> G={$(v_1,v_2),(v_3,v_4),...,(v_{n-1},v_n)$}, onde cada um destes pares é chamado de aresta e $\{v_1,v_2,...v_n\} \in$ V (vértices de G).



2 Dígrafos

Quando as arestas são pares ordenados de vértices, tem-se um grafo orientado (ou dígrafo). Caso contrário, tem-se um grafo não orientado.

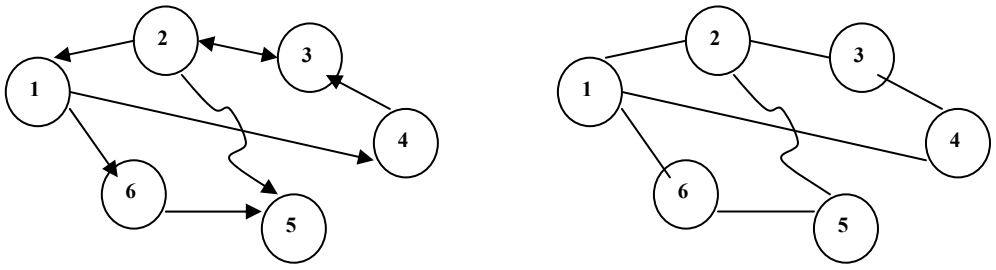
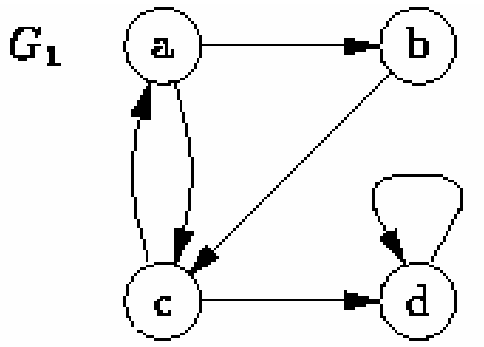
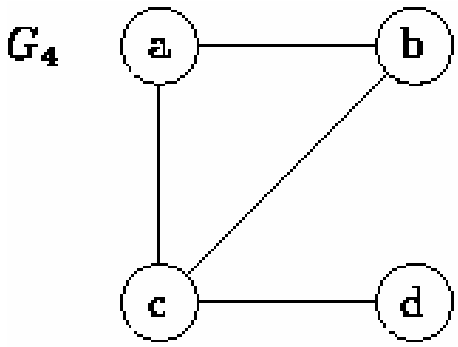


Fig 1 – Grafo orientados e não orientados



$$\mathcal{V}_1 = \{a, b, c, d\}$$
$$\mathcal{E}_1 = \{(a, b), (a, c), (b, c), (c, a), (c, d), (d, d)\}.$$



$$\mathcal{V}_4 = \{a, b, c, d\}$$
$$\mathcal{E}_4 = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$$

3 Completos

Quando o número máximo de arestas é atingido. Nesse caso, com base no número de vértices, pode-se calcular o número de arestas. Num grafo completo orientado é n (n-1). Os grafos completos não orientados são também denominados cliques.

Um clique pode ser representado por Kn, onde n = número de vértices do grafo completo.

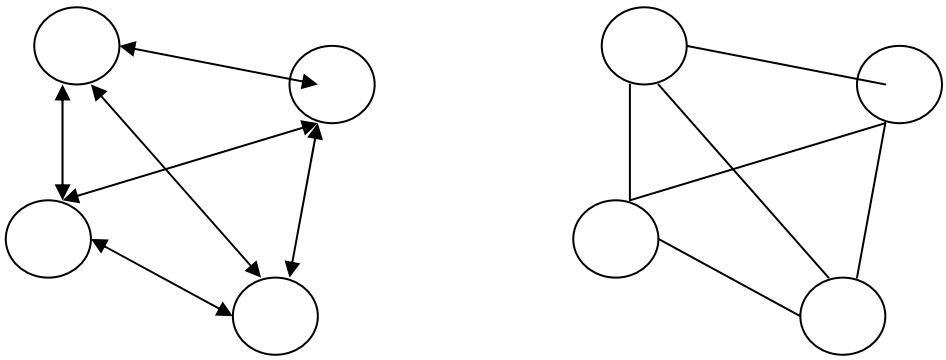


Fig 2 – Grafos completos

4 Grafo Bipartido

Um grafo G é bipartido quando for possível obter uma partição de conjuntos de vértices V com dois subconjuntos V1 e V2, de forma que, não existam arestas unindo elementos de V1 ou unindo elementos de V2.

Grafos completos bipartidos podem ser denotados por Kpq, onde p e q são as cardinalidades das duas partições do grafo. Se o grafo da Fig 3 fosse não orientado poderia ser denotado por K3,2.

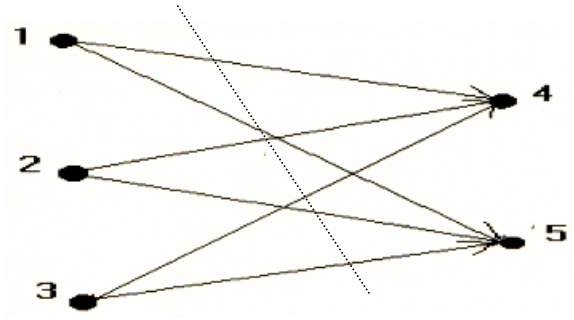


Fig 3 – Grafo Bipartido

5 Grafo Ponderado ou Rede

Um Grafo G, orientado ou não, é dito ponderado (ou valorado) se tem valores (pesos) associados aos vértices e/ou arestas. Estes valores, em geral, podem corresponder a distâncias, capacidade, localização ou custos em função das características ou propriedades de cada elemento de G. Em um dígrafo, quando se destacam um nó fonte e outro nó sumidouro, tem-se uma rede.

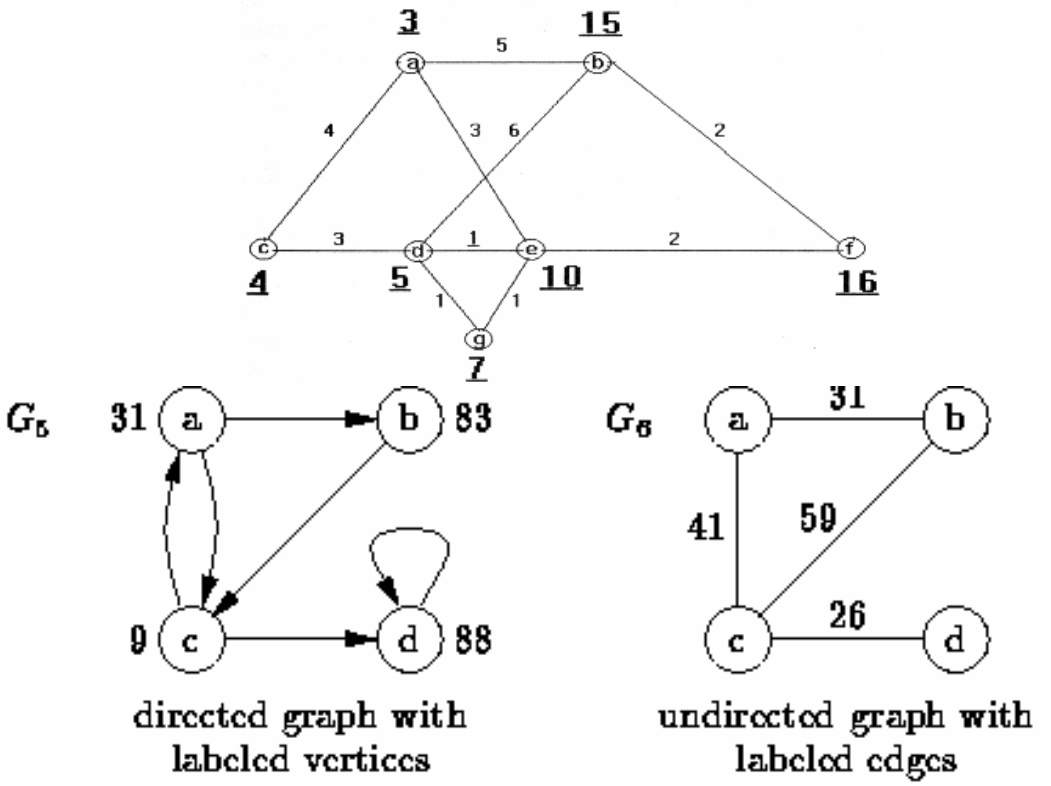


Fig 4 – Grafo ponderado

6 Grau de um grafo

Dado um grafo G=(V,E), se <vi,vj> ∈ E, vi ≠ vj, então os vértices vi e vj são adjacentes. O grau de um vértice vi ∈ V é igual ao número de vértices adjacentes a vi e grau g(G) = max { g(v1), g (v2), g(v3), ..., g(vn) } para vi ∈ V.

vertex *v*	$\mathcal{A}(v)$	out-degree	$\mathcal{I}(v)$	in-degree
a	$\{(a,b),(a,c)\}$	2	$\{(c,a)\}$	1
b	$\{(b,c)\}$	1	$\{(a,b)\}$	1
c	$\{(c,a),(c,d)\}$	2	$\{(a,c),(b,c)\}$	2
d	$\{(d,d)\}$	1	$\{(c,d),(d,d)\}$	2

7 Caminho, percurso, ciclo, circuito e comprimento

Um caminho de um vértice vi0 para vik é uma seqüência de arestas $<v_{i0}, v_{i1}>$, $<v_{i1}, v_{i2}>$, ... $<v_{ik-1}, v_{ik}>$. Um caminho é dito elementar se passa exatamente uma vez por cada vértice e é simples se passa uma vez por cada aresta. Quando o grafo não é orientado o conceito de caminho é substituído por cadeia. Percurso pode ser utilizado genericamente nos dois casos.

Se $v_{i0} = v_{ik}$, diz-se que o percurso é fechado e forma um ciclo (não orientado) ou circuito (orientado).

O comprimento de um percurso é a soma dos pesos associados a cada uma das arestas (ou vértices) envolvidos (para um grafo valorado) ou o número de arestas que compõe o percurso (para um grafo não valorado).

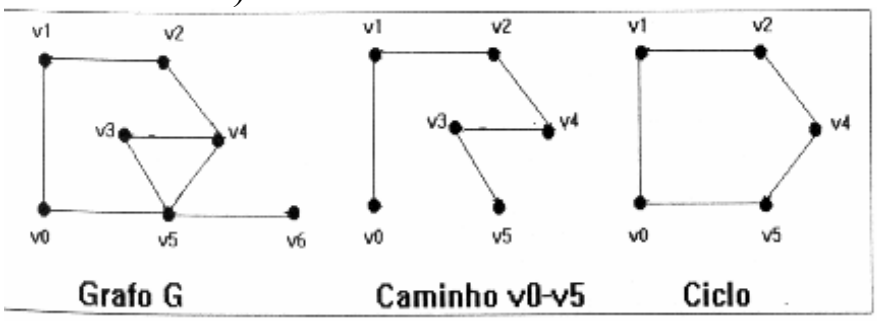


Fig 5 - Percursos

Ciclo Hamiltoniano e Euleriano

Um ciclo e(G) **simples** que passa por **todas** as arestas de um grafo G é dito **Euleriano**.

Um ciclo h(G) **elementar** que passa por **todos** os vértices de um grafo G é dito **Hamiltoniano.**

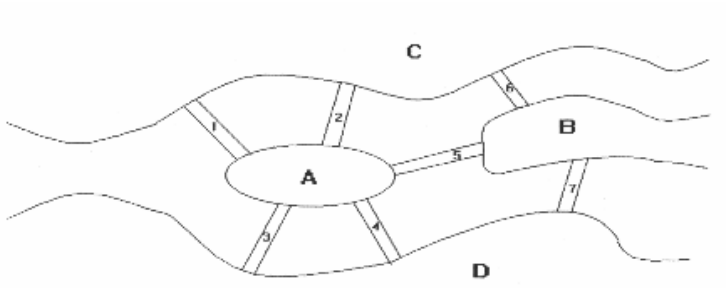


Fig 6 – Pontes de Koenigsberg, junto com as terras A,B,C e D não formam um grafo de Euler, mas possue um ciclo Hamiltoniano h(G) = {<A,C>,<C,B>,<B,D>,<D,A>}

Grafo Desconexo

Um grafo G não orientado é **conexo** se para todo par de vértices (v,w) existe pelo menos uma cadeia entre eles. Caso haja pelo menos um par de vértice de G que não possua uma cadeia entre eles, diz-se que G é desconexo.

Em grafos orientados não existe a obrigatoriedade de haver um caminho entre todo par de vértice. Se isso ocorrer, diz-se que o grafo é **fortemente conexo.**

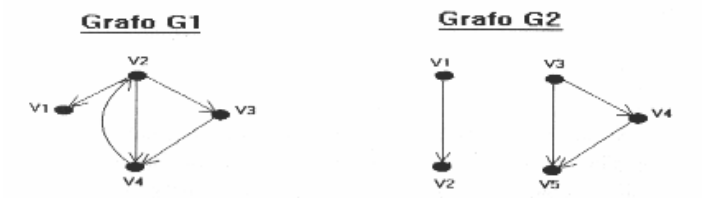


Fig 7 – Grafo conexo (G1) e desconexo (G2)

8 Grafo Planar

Arestas podem se interceptar em um ponto onde não há um vértice. Um grafo G onde toda intercessão de arestas corresponde a um vértice é dito **planar**.

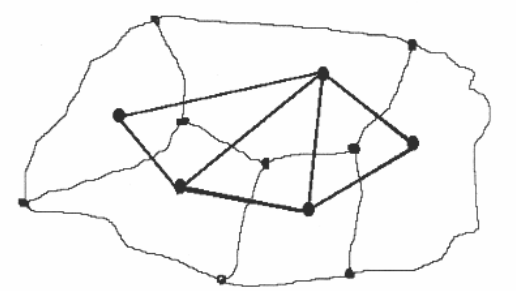


Fig 8 – Grafo planar

Grafo Árvore

A definição clássica de árvore:

- estrutura de dados que possui uma estrutura hierárquica entre seus elementos
- conjunto finito de nós onde um deles é denominado raiz e os demais, recursivamente, são também árvore (sub-árvores)

Existem variações de conceitos para árvores que não guardam relação de conectividade. Árvores não conexas, isto é, *orchards*. Uma árvore conexa que possui um nó raiz, é chamada árvore enraizada (rooted tree). E cada sub-árvore de uma árvore é por si uma árvore que formam um conjunto de árvores que são conhecidas como floresta de uma árvore.

Com relação a grafos, pode-se dizer que uma árvore de n nós é um grafo conexo, sem ciclos e com exatamente (n-1) arestas. Por não haver ciclos, existe uma só cadeia que liga dois vértices (v e w) diferentes.

9 Representação de grafos

- **Matricial**

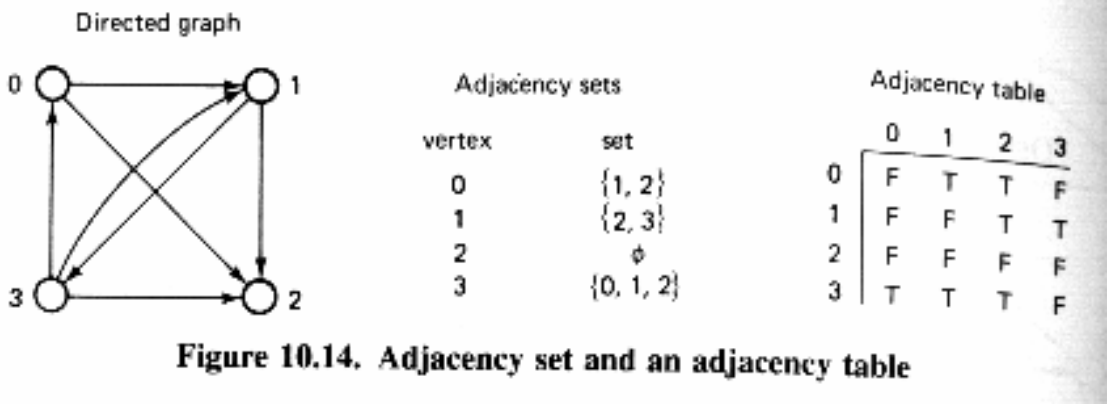


Figure 10.14. Adjacency set and an adjacency table

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

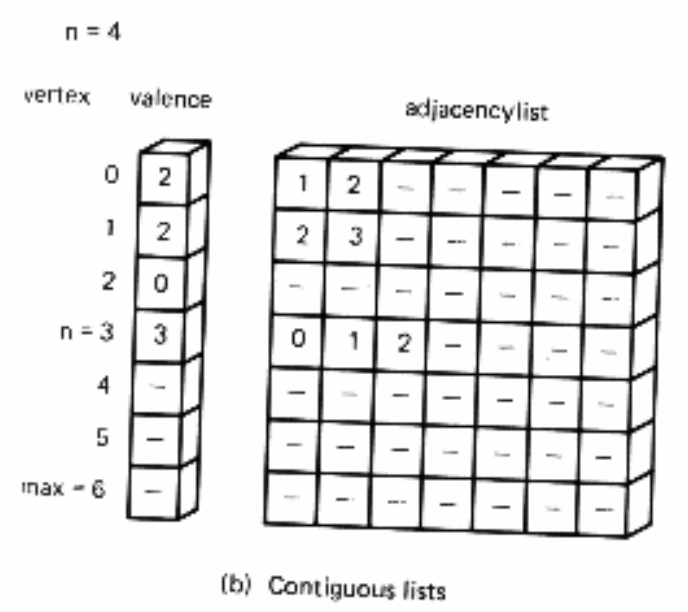
$$A_6 = \begin{bmatrix} \infty & 31 & 41 & \infty \\ 31 & \infty & 59 & \infty \\ 41 & 59 & \infty & 26 \\ \infty & \infty & 26 & \infty \end{bmatrix}$$

```
typedef int AdjacencyList_type [MAX];

typedef struct graph_tag {
        int n;
        int valence [MAX];
        AdjacencyList_type A [MAX];
} Graph_type;
```

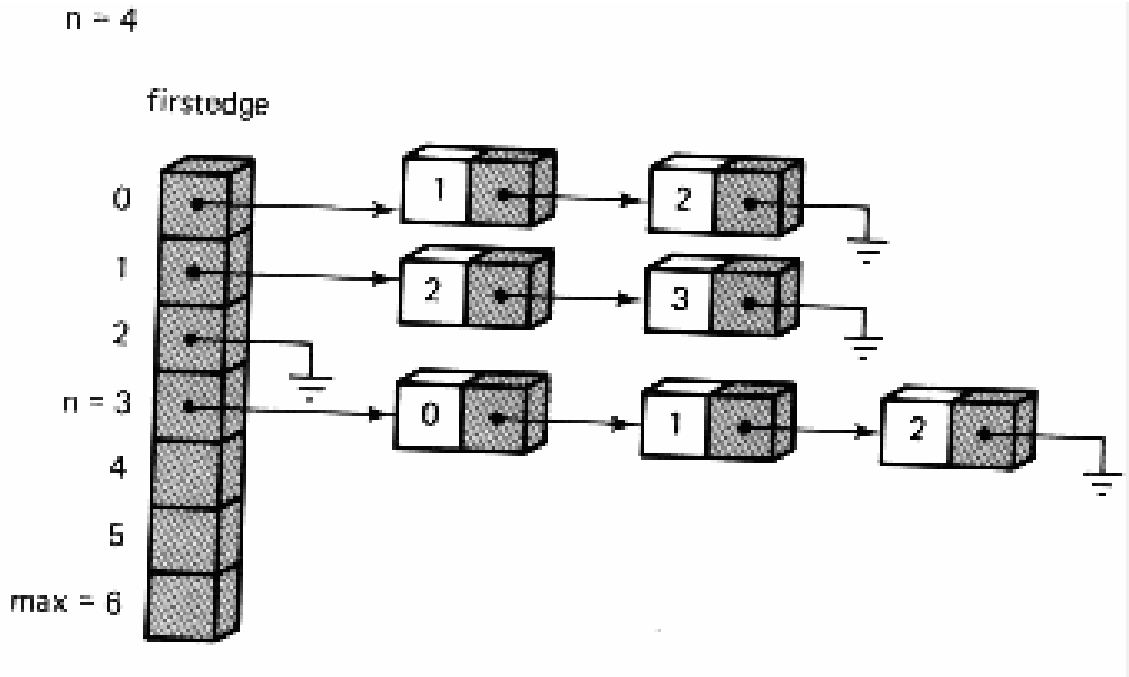
- **Grafo esparso e grafo denso**

$G = (V, E)$ in which $|E| = O(|V|)$.

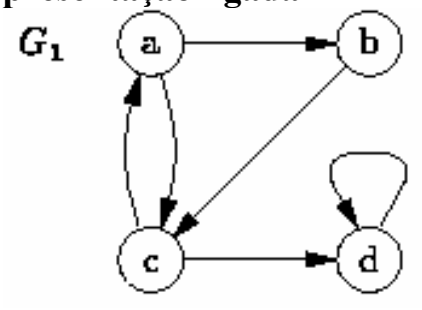
$G = (V, E)$ in which $|E| = \Theta(|V|^2)$.

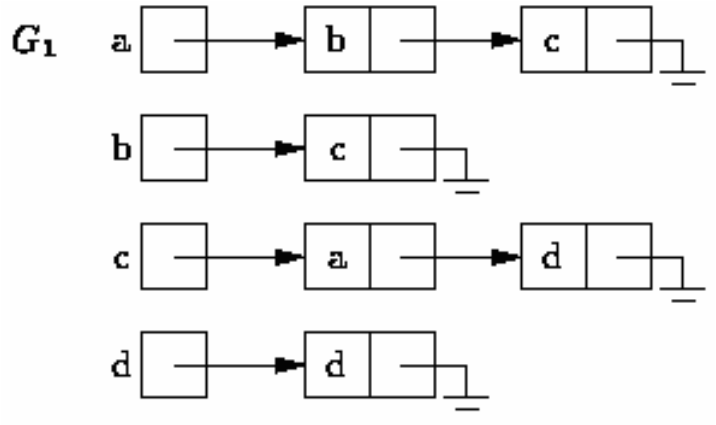


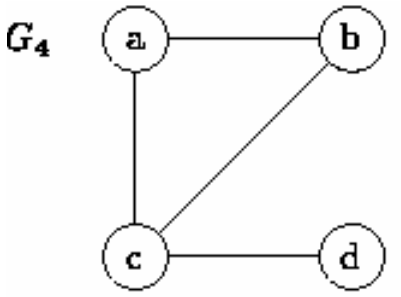
(b) Contiguous lists

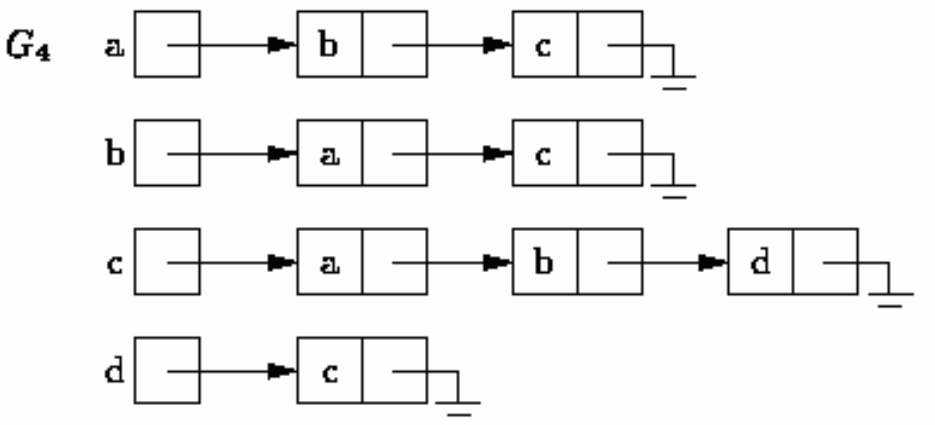


- **Exemplos de representação ligada**





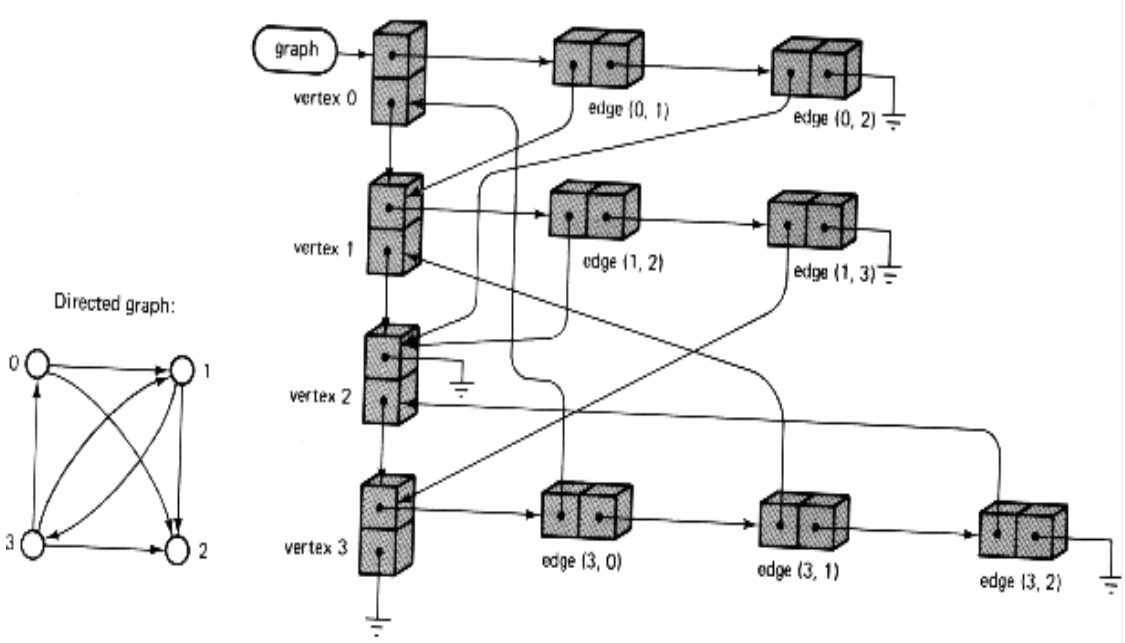




GRAFOS

- **Implementação em C**

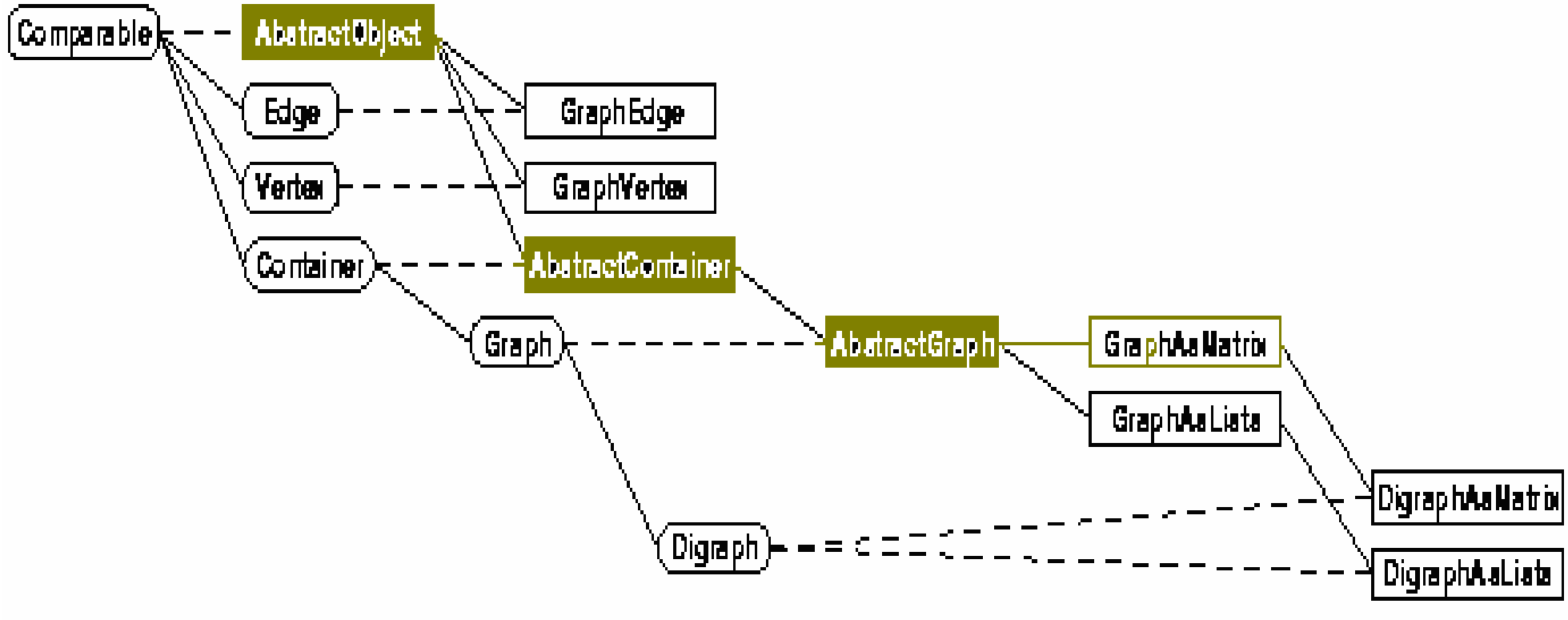
```
typedef struct vertex_tag Vertex_type;
typedef struct edge_tag Edge_type;

struct vertex_tag {
        Edge_type *firstedge;          /* start of the adjacency list
        Vertex_type *nextvertex;       /* next vertex on the linked list
};

struct edge_tag {
        Vertex_type *endpoint;         /* vertex to which the edge points
        Edge_type *nextedge;           /* next edge on the adjacency list
};

typedef Vertex_type *Graph_type;       /* header for the list of vertices
```



(a) Linked lists

GRAFOS

- **Implementação em Java**



```
1  public interface Vertex
2      extends Comparable
3  {
4      int getNumber ();
5      Object getWeight ();
6      Enumeration getIncidentEdges ();
7      Enumeration getEmanatingEdges ();
8      Enumeration getPredecessors ();
9      Enumeration getSuccessors ();
10 }
```

```
1  public interface Edge
2      extends Comparable
3  {
4      Vertex getV0 ();
5      Vertex getV1 ();
6      Object getWeight ();
7      boolean isDirected ();
8      Vertex getMate (Vertex vertex);
9  }
```

GRAFOS

```
1  public interface Graph
2      extends Container
3  {
4      int getNumberOfEdges ();
5      int getNumberOfVertices ();
6      boolean isDirected ();
7      void addVertex (int v);
8      void addVertex (int v, Object weight);
9      Vertex getVertex (int v);
10     void addEdge (int v, int w);
11     void addEdge (int v, int w, Object weight);
12     Edge getEdge (int v, int w);
13     boolean isEdge (int v, int w);
14     boolean isConnected ();
15     boolean isCyclic ();
16     Enumeration getVertices ();
17     Enumeration getEdges ();
18     void depthFirstTraversal (PrePostVisitor visitor, int start);
19     void breadthFirstTraversal (Visitor visitor, int start);
20 }
```

Métodos de acesso e modificadores

getNumberOfEdges
This method returns the number of edges contained by the graph.

getNumberOfVertices
This method returns the number of vertices contained by the graph.

isDirected
This boolean-valued method returns true if the graph is a directed graph.

addVertex
This method inserts a vertex into a graph. All the vertices contained in a given graph must have a unique vertex number. Furthermore, if a graph contains n vertices, those vertices shall be numbered 0, 1, ..., n-1. Therefore, the next vertex inserted into the graph shall have the number n.

getVertex
This method takes an integer, say i where , and returns the vertex contained in the graph.

addEdge
This method inserts an edge into a graph. If the graph contains n vertices, both arguments must fall in the interval [0,n-1].

isEdge
This boolean-valued method takes two integer arguments. It returns true if the graph contains an edge that connects the corresponding vertices.

getEdge
This method takes two integer arguments. It returns the edge instance (if it exists) that connects the corresponding vertices. The behavior of this method is undefined when the edge does not exist. (An implementation will typically throw an exception).

isCyclic
This boolean-valued method returns true if the graph is cyclic.
isConnected. This boolean-valued method returns true if the graph is connected.

getVertices
This method returns an enumeration that can be used to traverse the elements of E.

getEdges
This method returns an enumeration that can be used to traverse the elements of .E

depthFirstTraversal
This methods accepts two arguments--a PrePostVisitor and an integer. The integer specifies the starting vertex for a depth-first traversal of the graph.

breadthFirstTraversal
This methods accepts two arguments--a Visitor and an integer. The integer specifies the starting vertex for a breadth-first traversal of the graph.

```
1  public interface Digraph
2      extends Graph
3  {
4      boolean isStronglyConnected ();
5      void topologicalOrderTraversal (Visitor visitor);
6  }
```

isStronglyConnected
This boolean-valued method returns true if the directed graph is strongly connected. Strong connectedness is discussed in Section .

topologicalOrderTraversal
A topological sort is an ordering of the nodes of a directed graph. This traversal visits the nodes of a directed graph in the order specified by a topological sort

```
1  public abstract class AbstractGraph
2      extends AbstractContainer
3      implements Graph
4  {
5      protected int numberOfVertices;
6      protected int numberOfEdges;
7      protected Vertex[] verter;
8
9      public AbstractGraph (int size)
10         { verter = new Verter [size]; }
11
12     protected final class GraphVertex
13         extends AbstractObject
14         implements Vertex
15     {
16         protected int number;
17         protected Object weight;
18         // ...
19     }
20
21     protected final class GraphEdge
22         extends AbstractObject
23         implements Edge
24     {
25         protected int v0;
26         protected int v1;
27         protected Object weight;
28         // ...
29     }
30
31     protected abstract Enumeration getIncidentEdges (int v);
32     protected abstract Enumeration getEmanatingEdges (int v);
33     // ...
34 }
```

Fechamento Transitivo

Observando a matriz adjacência ADJ, que representa as arestas interligando os vértices de um dado grafo G, pode-se concluir que:

Se (ADJ[i][j] = V)
Existe um caminho de comprimento 1(ou aresta) entre i e j.

Se (ADJ[i][1] = V .E. ADJ[1][j] = V .OU.
ADJ[i][2] = V .E. ADJ[2][j] = V .OU.
ADJ[i][3] = V .E. ADJ[3][j] = V .OU.
...
ADJ[i][N] = V .E. ADJ[N][j] = V .OU.)
Existe pelo menos um caminho de comprimento 2 entre i e j.

O mesmo pode ser feito para mais arestas, o que tornaria o princípio válido para caminhos de qualquer comprimento.

Quando ADJ contém 1´s e 0´s, pode-se reescrever

$$ADJ[i][k] = V \ .E. \ ADJ[k][j] = V \ .OU.$$

como

$$ADJ[i][k] \ * \ ADJ[k][j] \ +$$

Assim, a matriz dos caminhos de comprimento 2

$ADJ^2 = ADJ * ADJ$ *(multiplicação de matrizes)*
e de comprimento 3
$ADJ^3 = ADJ^2 * ADJ$ ...
$ADJ^n = ADJ^{n-1} * ADJ$

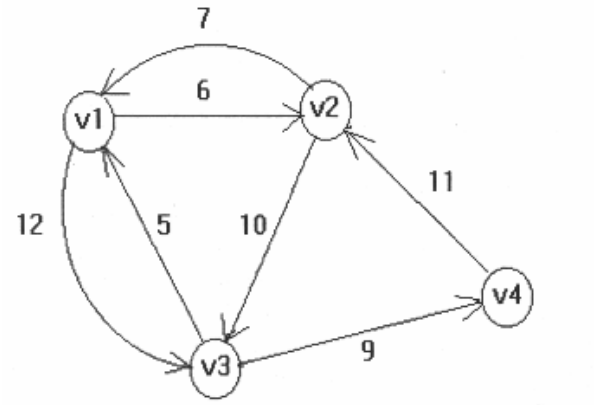
Fechamento transitivo ou matriz caminho é definida como sendo a matriz de todos os caminhos de qualquer comprimento em um grafo G:

$$PATH[i][j] = ADJ[i][j] \ || \ ADJ^2[i][j] \ || \ ADJ^3[i][j] \ || \ ... \ || \ ADJ^n[i][j]$$

10 Matriz Custo

A matriz adjacência pode conter também informações sobre o custo associado a uma aresta. A figura seguir mostra um grafo e a matriz custo inicial com a notação ∞, simbolizando ausência de arestas (custo infinito).

$$C = \begin{bmatrix} 0 & 6 & 12 & \infty \\ 7 & 0 & 10 & \infty \\ 5 & \infty & 0 & 9 \\ \infty & 11 & \infty & 0 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 6 & 12 & 21 \\ 7 & 0 & 10 & 19 \\ 5 & 11 & 0 & 9 \\ 18 & 11 & 21 & 0 \end{bmatrix}$$



A matriz custo mínimo de um grafo *G* considera, de todos os caminhos (de qualquer comprimento) possíveis entre dois vértices, aqueles que possuem menor custo associado. É equivalente ao *fechamento transitivo*, mas não mostra apenas a presença ou não de caminho entre *i e j (vértices de G)*, mas também o menor custo entre *i e j*. Se entre dois vértices não existir caminho de comprimento algum (PATH[i][j]=0), a matriz custo mínimo deverá apresentar ∞ para aquela aresta (i,j).

Portanto a matriz custo mínimo é a matriz dos caminhos de menor custo, obtida a partir da matriz de custo inicial. Assim

```
MatrizCustoMínimo (Cm) := min { Ci , Ci2, Ci3, ... Cin }
  Cm := Ci          // O(n2)
  Para x=1, n
        Para i=1,n
            Para j=1,n
                Se i <> j
                    Para k=1, n
                        Cm(i,j) := min (Cm(i,j) , Cm(i,k) + Cm( k, j )// O(n4)
                    Fim
                Fim
            Fim
        Fim
  Fim
```

11 Percursos em Grafos não orientados

- **Em Profundidade**

```
Profundidade ( vértice v)
Visitado (v) := .T.
Para cada vértice w adjacente a v
            Se (não Visitado (v)) então Profundidade ( w )
Fim
Fim
```

- **Em Largura**

```
Largura ( vértice v)
Visitado (v) := .T.
Inicia (Q)
Faça (.T.)
        Para cada vértice w adjacente a v
                    Se (não Visitado (v)) então
                            Insere (Q, w )
                            Visitado (w) := .T.
                    Fim
        Fim
        Se Vazia(Q) return;
            Senão v := Remove (Q)
        Fim
Fim
```

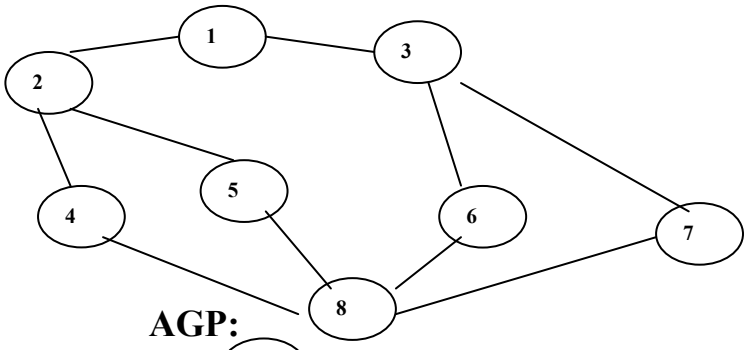
12 Árvores Geradoras

Árvores geradas a partir de percursos em grafos não orientados, onde apenas o nó raiz dessa árvore não possui predecessor. O termo **floresta geradora** é mais genérico, pois trata de um conjunto de árvores geradas nessas mesmas condições, ou seja, mais de um nó não possui predecessor.

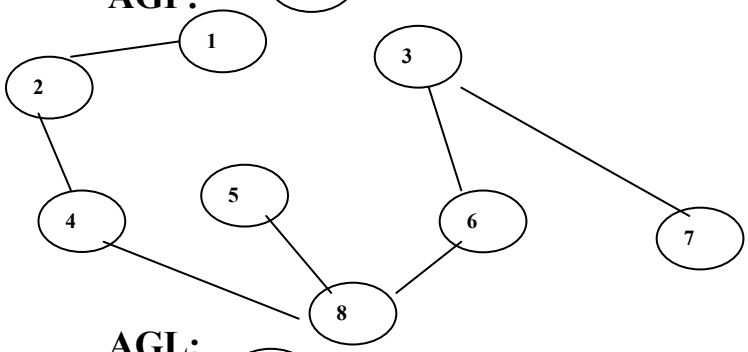
Assim deseja-se encontrar um subgrafo acíclico (árvore) $T \subseteq G$ que conecta todos os vértices de G. Para tanto é necessário que G seja não orientados e conexo. Diz-se que *T* espalha *G* (**spanning tree).**

Assim, pode-se ter árvores geradoras em profundidade (AGP) e árvores geradoras em largura (AGL). Observe os exemplos a seguir:

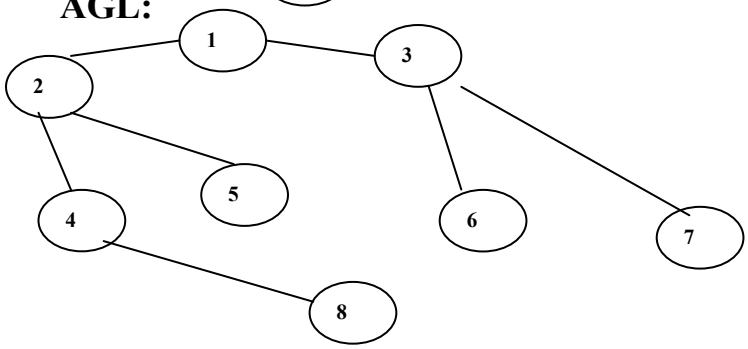
Seja o grafo G:



AGP:



AGL:

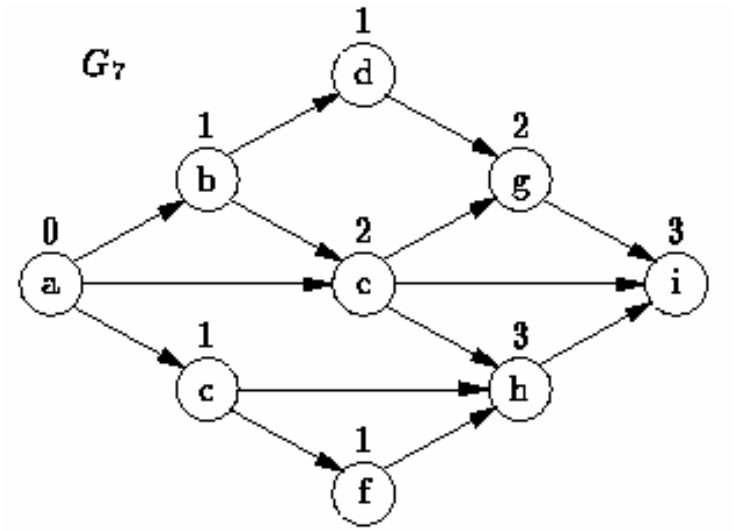


```
1  public abstract class AbstractGraph
2      extends AbstractContainer
3      implements Graph
4  {
5      protected int numberOfVertices;
6      protected int numberOfEdges;
7      protected Vertex[] vertex;
8
9      public void depthFirstTraversal (
10         PrePostVisitor visitor, int start)
11     {
12         boolean[]  visited = new boolean [numberOfVertices];
13         for (int v = 0; v < numberOfVertices; ++v)
14             visited [v] = false;
15         depthFirstTraversal (visitor, vertex [start], visited);
16     }
17
18     private void depthFirstTraversal (
19         PrePostVisitor visitor, Vertex v, boolean[] visited)
20     {
21         if (visitor.isDone ())
22             return;
23         visitor.preVisit (v);
24         visited [v.getNumber ()] = true;
25         Enumeration p = v.getSuccessors ();
26         while (p.hasMoreElements ())
27         {
28             Vertex to = (Vertex) p.nextElement ();
29             if (!visited [to.getNumber ()])
30                 depthFirstTraversal (visitor, to, visited);
31         }
32         visitor.postVisit (v);
33     }
34     // ...
35 }
```

```
1  public abstract class AbstractGraph
2      extends AbstractContainer
3      implements Graph
4  {
5      protected int numberOfVertices;
6      protected int numberOfEdges;
7      protected Vertex[] vertex;
8
9      public void breadthFirstTraversal (
10         Visitor visitor, int start)
11     {
12         boolean[] enqueued = new boolean [numberOfVertices];
13         for (int v = 0; v < numberOfVertices; ++v)
14             enqueued [v] = false;
15
16         Queue queue = new QueueAsLinkedList ();
17
18         enqueued [start] = true;
19         queue.enqueue (vertex [start]);
20         while (!queue.isEmpty () && !visitor.isDone ())
21         {
22             Vertex v = (Vertex) queue.dequeue ();
23             visitor.visit (v);
24             Enumeration p = v.getSuccessors ();
25             while (p.hasMoreElements ())
26             {
27                 Vertex to = (Vertex) p.nextElement ();
28                 if (!enqueued [to.getNumber ()])
29                 {
30                     enqueued [to.getNumber ()] = true;
31                     queue.enqueue (to);
32                 }
33             }
34         }
35     }
36     // ...
37 }
```

13 Ordenação Topológica

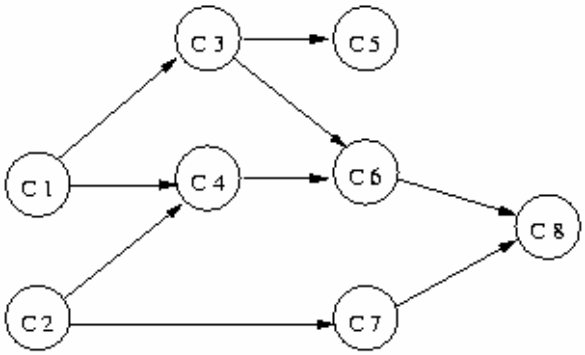
Um grafo acíclico direcionado (DAG) G = (V,E) induz um conjunto S={v1, v2, v3, ..., $v_{|v|}$ } onde v_k aparece uma única vez e S está ordenado de modo a se obter uma seqüência de vértices v_1, v_2, ..., v_n , onde $v_i < v_j$, para i < j. Essa seqüência é uma ordenação topológica.



$$S = \{a, b, c, d, e, f, g, h, i\}$$

$$\mathcal{E} = \{(a,b),(a,c),(a,e),(b,d),(b,e),(c,f),(c,h),(d,g),(e,g),(e,h),(e,i),(f,h)(g,i),(h,i)\}.$$

$$S' = \{a, c, b, f, e, d, h, g, i\}$$
$$S'' = \{a, b, d, e, g, c, f, h, i\}$$
$$S''' = \{a, c, f, h, b, e, d, g, i\}$$
$$\vdots$$



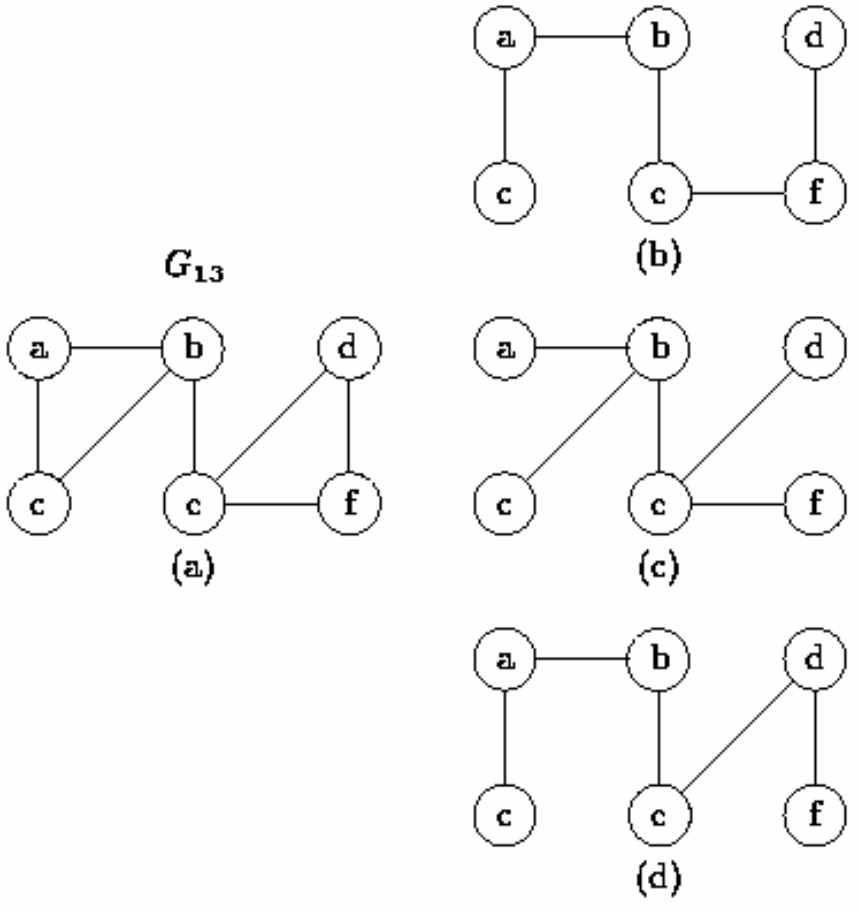
S= {C2,C7,C1,C4,C3,C6,C8,C5}

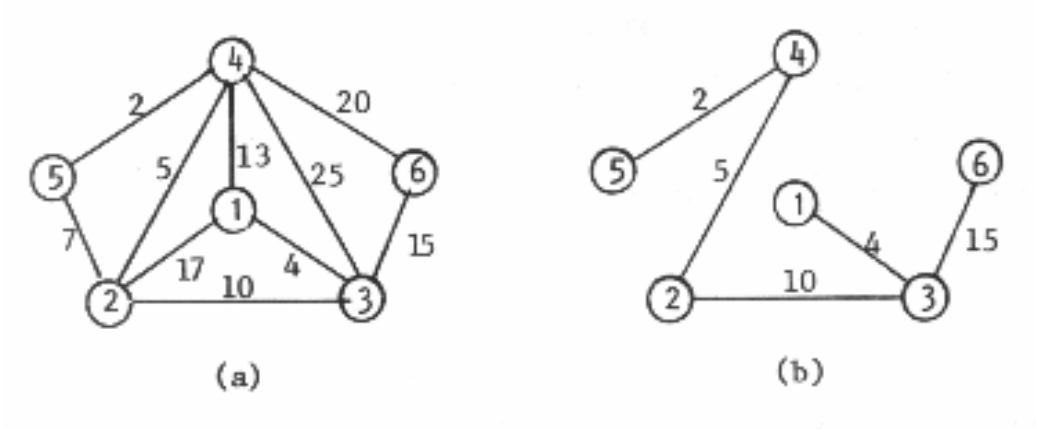
```
1  public abstract class AbstractGraph
2      extends AbstractContainer
3      implements Graph
4  {
5      protected int numberOfVertices;
6      protected int numberOfEdges;
7      protected Vertex[] vertex;
8
9      public void topologicalOrderTraversal (Visitor visitor)
10     {
11         int[] inDegree = new int [numberOfVertices];
12         for (int v = 0; v < numberOfVertices; ++v)
13             inDegree [v] = 0;
14         Enumeration p = getEdges ();
15         while (p.hasMoreElements ())
16         {
17             Edge edge = (Edge) p.nextElement ();
18             Vertex to = edge.getV1 ();
19             ++inDegree [to.getNumber ()];
20         }
21
22         Queue queue = new QueueAsLinkedList ();
23         for (int v = 0; v < numberOfVertices; ++v)
24             if (inDegree [v] == 0)
25                 queue.enqueue (vertex [v]);
26         while (!queue.isEmpty () && !visitor.isDone ())
27         {
28             Vertex v = (Vertex) queue.dequeue ();
29             visitor.visit (v);
30             Enumeration q = v.getSuccessors ();
31             while (q.hasMoreElements ())
32             {
33                 Vertex to = (Vertex) q.nextElement ();
34                 if (--inDegree [to.getNumber ()] == 0)
35                     queue.enqueue (to);
36             }
37         }
38     }
39     // ...
40 }
```

14 Árvore Geradora Mínima (*minimal-spanning-tree)*

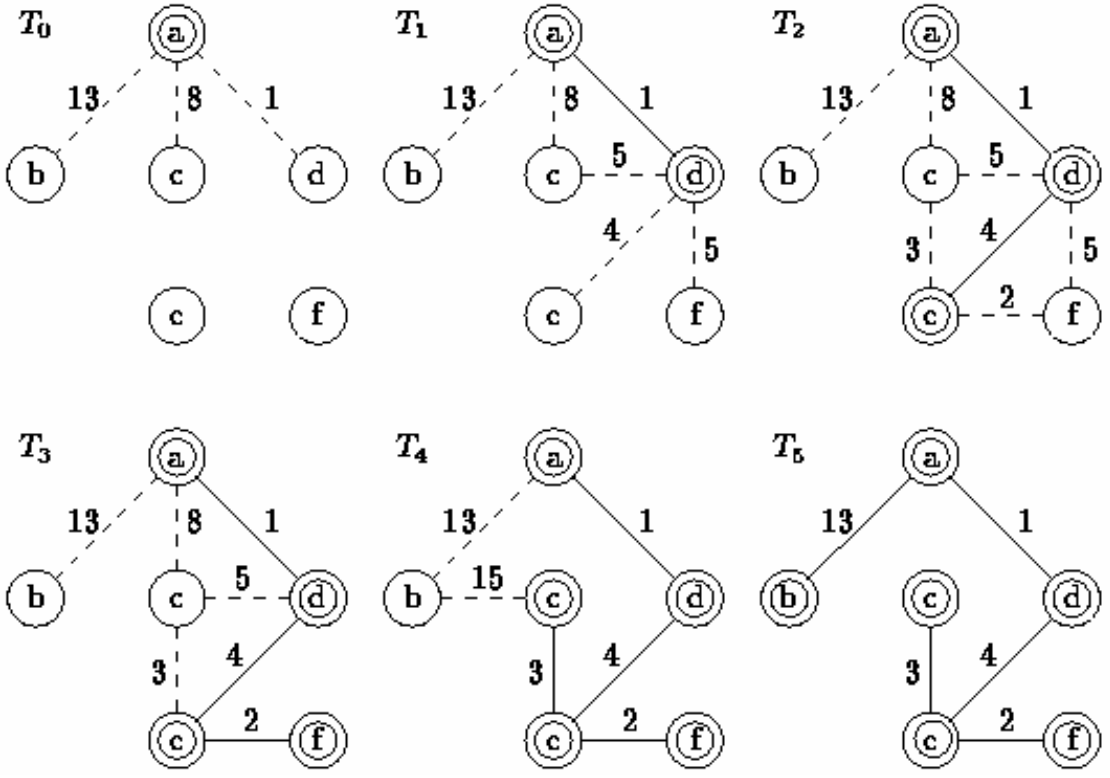
Árvore *T* gerada a partir de um grafo não orientado, conexo e ponderado *G* de forma a minimizar:

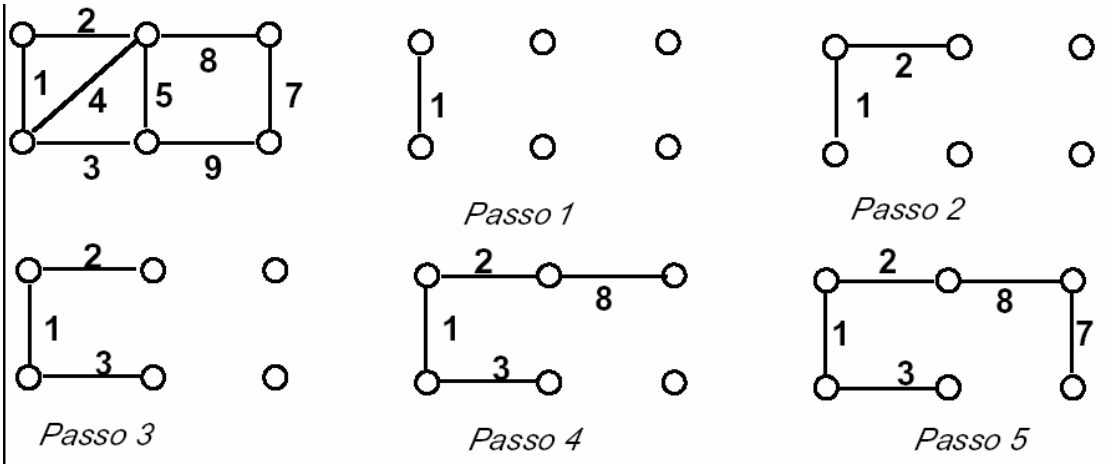
$$w(T) = \sum w(u,v), \text{ onde } (u,v) \in T$$





Algoritmo: Prim-Dijkstra:



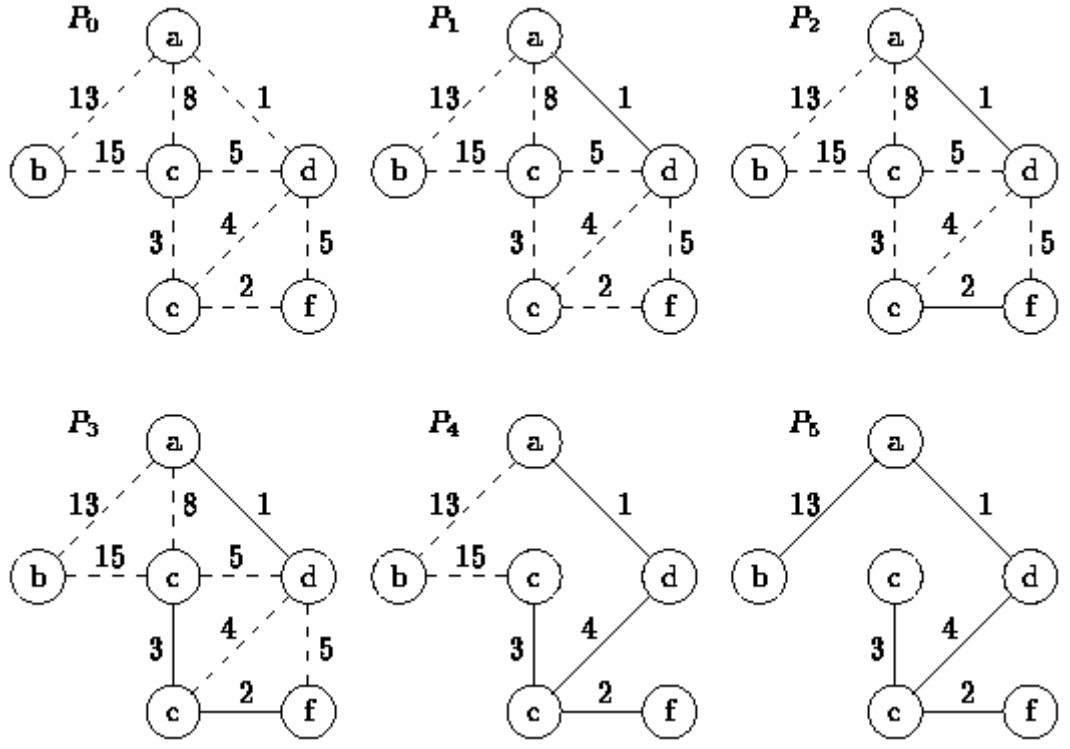


$$O(|\mathcal{V}| + |\mathcal{E}| \log |\mathcal{E}|),$$

$$O(|\mathcal{V}|^2 + |\mathcal{E}| \log |\mathcal{E}|),$$

```
1  public class Algorithms
2  {
3      public static Graph PrimsAlgorithm (Graph g, int s)
4      {
5          int n = g.getNumberOfVertices ();
6          Entry[] table = new Entry [n];
7          for (int v = 0; v < n; ++v)
8              table [v] = new Entry ();
9          table [s].distance = 0;
10         PriorityQueue queue =
11             new BinaryHeap (g.getNumberOfEdges());
12         queue.enqueue (
13             new Association (new Int (0), g.getVertex (s)));
14         while (!queue.isEmpty ())
15         {
16             Association assoc = (Association) queue.dequeueMin();
17             Vertex v0 = (Vertex) assoc.getValue ();
18             int n0 = v0.getNumber ();
19             if (!table [n0].known)
20             {
21                 table [n0].known = true;
22                 Enumeration p = v0.getEmanatingEdges ();
23                 while (p.hasMoreElements ())
24                 {
25                     Edge edge = (Edge) p.nextElement ();
26                     Vertex v1 = edge.getMate (v0);
27                     int n1 = v1.getNumber ();
28                     Int wt = (Int) edge.getWeight ();
29                     int d = wt.intValue ();
30                     if (table [n1].distance > d)
31                     {   table [n1].distance = d;
32                         table [n1].predecessor = n0;
33                         queue.enqueue (
34                             new Association (new Int (d), v1));
35                     }
36                 }
37             }
38         }
39         Graph result = new GraphAsLists (n);
40         for (int v = 0; v < n; ++v)
41             result.addVertex (v);
42         for (int v = 0; v < n; ++v)
43             if (v != s)
44                 result.addEdge (v, table [v].predecessor);
45         return result;
46     }
47 }
```

Kruskal's Algorithm



Algoritmo MST-Krushal(G,w) // O(A log A)
T := ϕ
Enquanto não vazio(G)
Encontre (u,v) , w(u,v) < w(x,y) para ∀(x,y) ∈ G
G := G - {(u,v)}
T := T + {(u,v)}
Se Caminhos (u,v) > 1 // forma ciclo
    T := T - {(u,v)}
Fim
    Fim

$$O(|\mathcal{V}| + |\mathcal{E}|\log|\mathcal{E}| + |\mathcal{E}|\log|\mathcal{V}|),$$

$$O(|\mathcal{V}|^2 + |\mathcal{E}|\log|\mathcal{E}| + |\mathcal{E}|\log|\mathcal{V}|),$$

```
1  public class Algorithms
2  {
3      public static Graph KruskalsAlgorithm (Graph g)
4      {
5          int n = g.getNumberOfVertices ();
6
7          Graph result = new GraphAsLists (n);
8          for (int v = 0; v < n; ++v)
9              result.addVertex (v);
10
11         PriorityQueue queue =
12             new BinaryHeap (g.getNumberOfEdges());
13         Enumeration p = g.getEdges ();
14         while (p.hasMoreElements ())
15         {
16             Edge edge = (Edge) p.nextElement ();
17             Int weight = (Int) edge.getWeight ();
18             queue.enqueue (new Association (weight, edge));
19         }
20
21         Partition partition = new PartitionAsForest (n);
22         while (!queue.isEmpty () && partition.getCount () > 1)
23         {
24             Association assoc = (Association) queue.dequeueMin();
25             Edge edge = (Edge) assoc.getValue ();
26             int n0 = edge.getV0 ().getNumber ();
27             int n1 = edge.getV1 ().getNumber ();
28             Set s = partition.find (n0);
29             Set t = partition.find (n1);
30             if (s != t)
31             {
32                 partition.join (s, t);
33                 result.addEdge (n0, n1);
34             }
35         }
36         return result;
37     }
38 }
```

15 Problema do caminho mínimo

Referência:

- Projeto de Algoritmos. Ornelas Almeida e Nivio Ziviani. 2004

Algoritmo de Djikstra

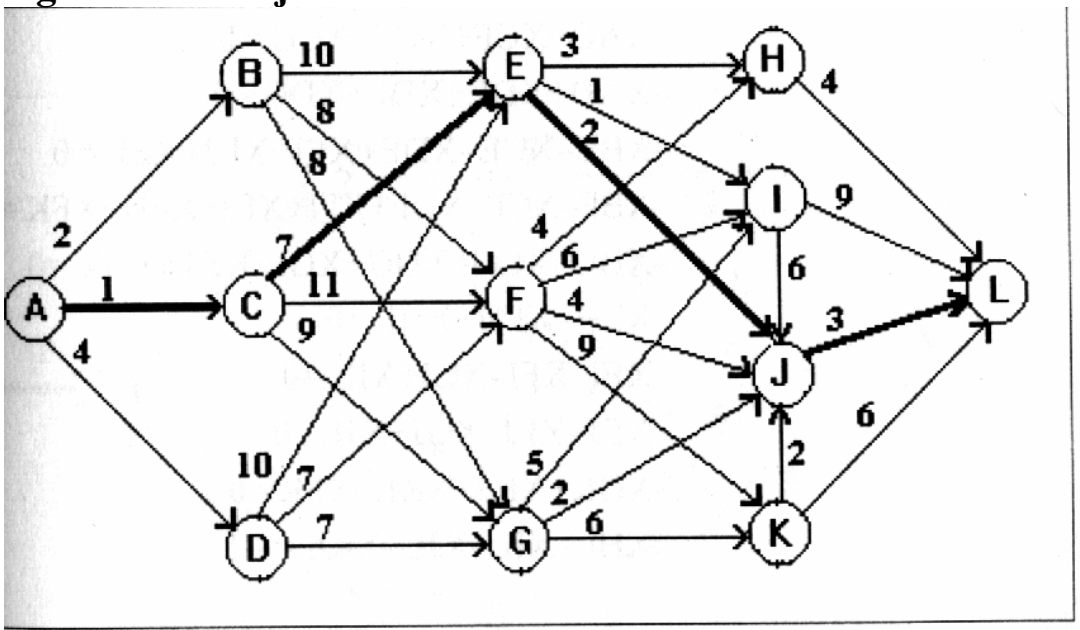


FIGURA 2.24 - Grafo para o Exemplo 2.18 (Caminho Mínimo)[7]

Uma árvore de caminhos mais curtos com raiz em $s \in V$ é um subgrafo direcionado G' *(V', A'),* onde $V' \subseteq V$ e $A' \subseteq A$, tal que:

a. *V'* é o conjunto de vértices alcançáveis a partir de $s \in$ G,

b. G' forma uma árvore de raiz *s,*

c. 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.

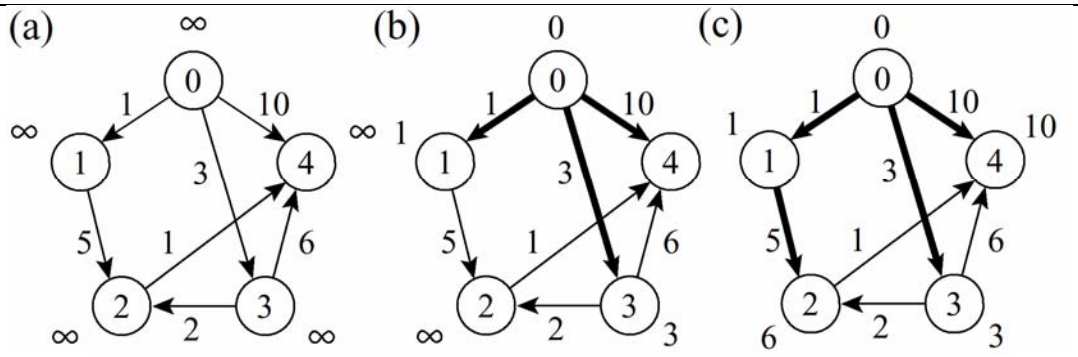
- O primeiro passo do algoritmo é inicializar os antecessores e as estimativas de caminhos mais curtos:

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

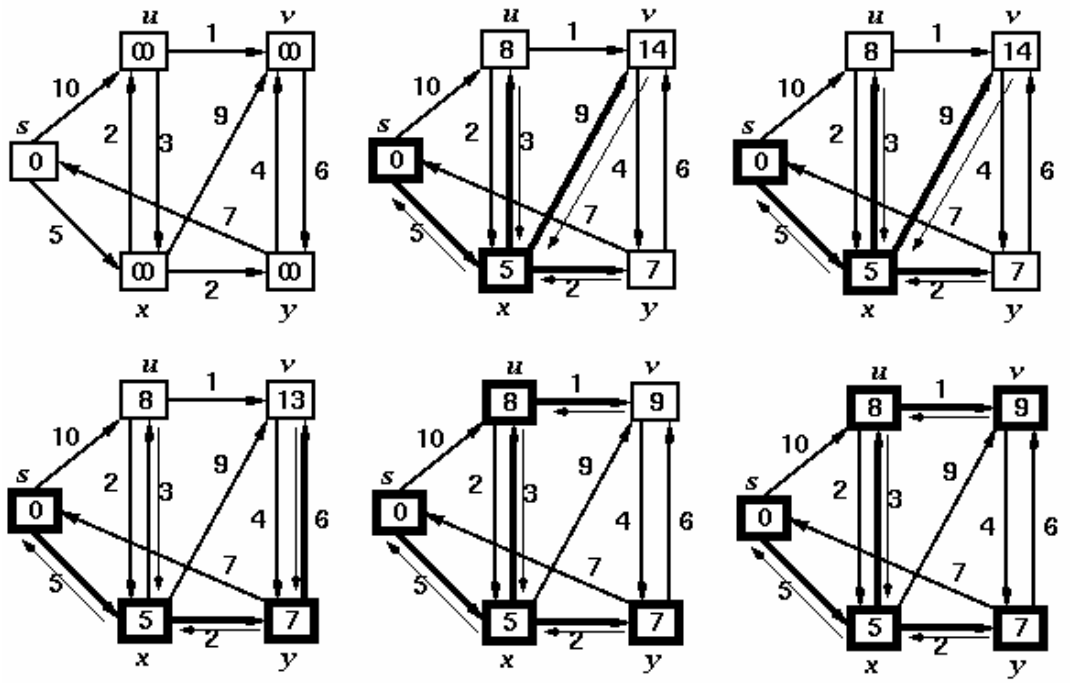
- O relaxamento de uma aresta (u, v) consiste em verificar se é possível melhorar o melhor caminho até v obtido até o momento se passarmos por u.

```
if p[v] > p[u] + peso da aresta (u,v) then
        p[v] = p[u] + peso da aresta (u,v)
        Antecessor[v] := u
```

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



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

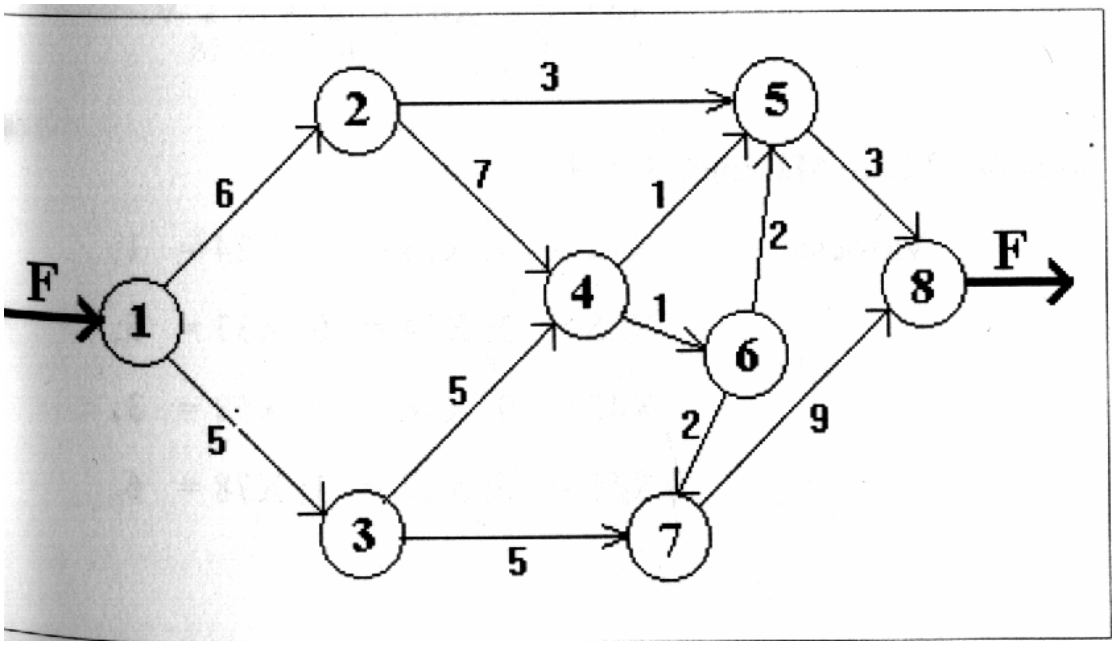


vértices	**s**	**u**	**v**	**x**	**y**	**s**	**u**	**v**	**x**	**y**
estimativas	**0**	10	∞	5	∞	**0**	8	14	5	7
precedentes	**s**	s	-	s	-	**s**	x	x	**s**	x

s	**u**	**v**	**x**	**y**	**s**	**u**	**v**	**x**	**y**	**s**	**u**	**v**	**x**	**y**
0	8	13	5	7	**0**	8	9	5	7	**0**	8	9	5	7
s	x	y	**s**	**x**	**s**	**x**	u	**s**	**x**	**s**	**x**	**u**	**s**	**x**

```
1  public class Algorithms
2  {
3      public static Digraph DijkstrasAlgorithm (Digraph g, int s)
4      {
5          int n = g.getNumberOfVertices ();
6          Entry[] table = new Entry [n];
7          for (int v = 0; v < n; ++v)
8              table [v] = new Entry ();
9          table [s].distance = 0;
10         PriorityQueue queue =
11             new BinaryHeap (g.getNumberOfEdges());
12         queue.enqueue (
13             new Association (new Int (0), g.getVertex (s)));
14         while (!queue.isEmpty ())
15         {
16             Association assoc = (Association) queue.dequeueMin();
17             Vertex v0 = (Vertex) assoc.getValue ();
18             int n0 = v0.getNumber ();
19             if (!table [n0].known)
20             {
21                 table [n0].known = true;
22                 Enumeration p = v0.getEmanatingEdges ();
23                 while (p.hasMoreElements ())
24                 {
25                     Edge edge = (Edge) p.nextElement ();
26                     Vertex v1 = edge.getMate (v0);
27                     int n1 = v1.getNumber ();
28                     Int wt = (Int) edge.getWeight ();
29                     int d = table [n0].distance + wt.intValue ();
30                     if (table [n1].distance > d)
31                     {   table [n1].distance = d;
32                         table [n1].predecessor = n0;
33                         queue.enqueue (
34                             new Association (new Int (d), v1));
35                     }
36                 }
37             }
38         }
39         Digraph result = new DigraphAsLists (n);
40         for (int v = 0; v < n; ++v)
41             result.addVertex (v, new Int (table [v].distance));
42         for (int v = 0; v < n; ++v)
43             if (v != s)
44                 result.addEdge (v, table [v].predecessor);
45         return result;
46     }
47 }
```

Fluxo Máximo



Grafo para o problema do fluxo máximo