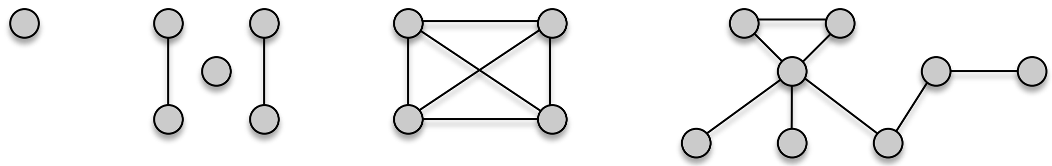
Grafos

- Grafos são estruturas de dados muito importantes na Ciência da Computação.
- Existe uma infinidade de problemas de grande interesse, tanto teórico quanto prático, que são definidos em termos de grafos.
- Grafos podem ser vistos como uma generalização de árvores. Árvores têm limitações importante:
  - Podem representar apenas relacionamentos hierárquicos, como as relações entre pai e filho.
  - Outras relações (como irmão, por exemplo) podem ser representadas apenas indiretamente.
- Num grafo estas limitações não existem.

Grafos

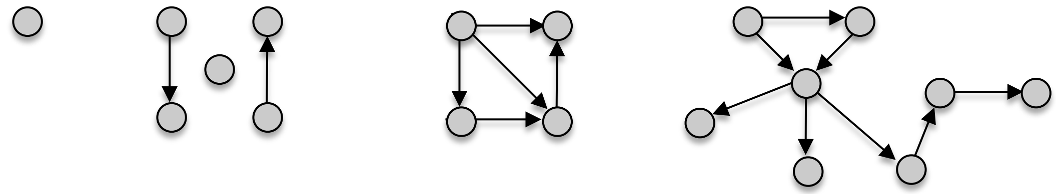
- Intuitivamente, um grafo é constituído por um conjunto de **vértices** (ou **nós**) e um conjunto de **arestas** que correspondem a conexões entre os vértices do grafo.



- Formalmente, um **grafo simples** G = (V, A) consiste de um conjunto não vazio V de **vértices** e de um conjunto A de **arestas** da forma $\{v_i, v_j\}$, com $v_i \in V$ e $v_j \in V$. Note que, neste caso, não há distinção entre uma aresta que conecta os nós v_i e v_j e uma aresta que conecta os nós v_j e v_i.
- Um grafo simples é também conhecido como grafo não-orientado.

Grafos

- Um **grafo orientado** é um grafo G = (V, A) no qual o conjunto A de arestas (neste caso, chamadas de **arcos**) é formado por pares (v_i, v_j), com $v_i \in V$ e $v_j \in V$.
- Também conhecido como **grafo dirigido** ou **digrafo**.
- No caso de digrafos, a aresta (v_i, v_j) é diferente da aresta (v_j, v_i) e será representada por uma seta, que tem **origem** em v_i e **destino** em v_j.



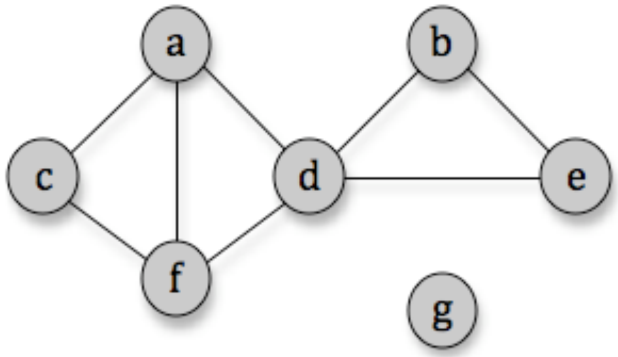
- Não vamos fazer distinção entre arestas (não-orientadas) e arcos (orientados), a menos que seja necessário. Uma aresta entre os vértices v_i e v_j será representada como (v_iv_j).

Grafos

- Um **caminho** de v_1 a v_n em um grafo G = (V, A) é uma sequência de arestas (v_1v_2), (v_2v_3), ..., $(v_{n-1}v_n)$.
- Se $v_1 = v_n$ e nenhuma aresta é repetida, o caminho é denominado **circuito**.
- Se todos os vértices em um circuito são diferentes, o circuito é denominado **ciclo**.
- Um grafo G = (V, A) é chamado de **grafo ponderado** se cada aresta de A possui um valor (interpretado como peso, custo, distância, comprimento, etc.).
- Um grafo G = (V, A), com |V| = n, é chamado de **grafo completo** (ou **clique**), e denotado por K_n, se existe uma aresta de A conectando qualquer par de vértices distintos de V. Observe que em K_n existem (n (n-1))/2 arestas.

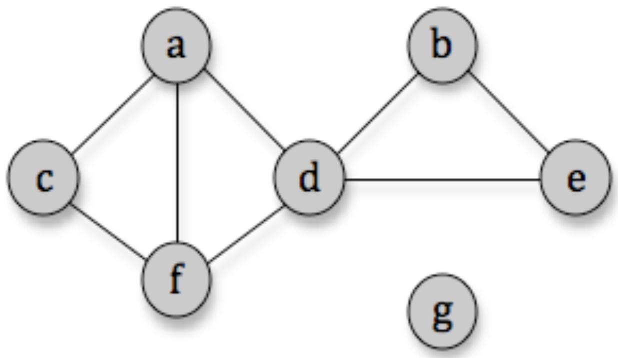
Grafos

- Um **subgrafo** G' de um grafo G = (V, A) é um grafo (V', A') tal que $V' \subseteq V$ e $A' \subseteq A$.
- Seja G = (V, A) um grafo. Dizemos que dois vértices v_i e v_j de V são **adjacentes** se a aresta $(v_i v_j) \in A$.
- Uma aresta $(v_i v_j)$ é **incidente** aos vértices v_i e v_j (no caso de digrafos, pode-se considerar como incidente a um vértice v apenas as arestas que têm destino em v).
- O **grau** de um vértice v, denominado por grau(v), é o número de arestas incidentes a v. Se grau(v) = 0, v é denominado **vértice isolado**.

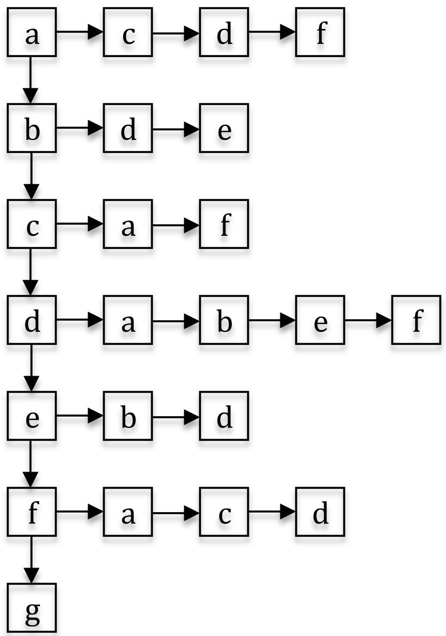


Representações de grafos

- Uma representação simples: **lista de adjacências** que especifica, para cada vértice do grafo, os vértices que são adjacentes a ele.
- A lista de adjacências pode ser implementada como uma tabela ou como uma lista encadeada.

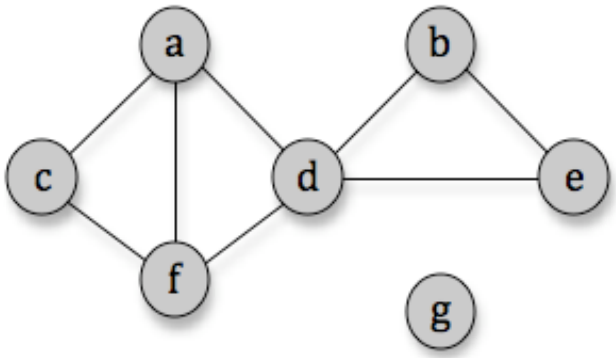


a	c, d, f
b	d, e
c	a, f
d	a, b, e, f
e	b, d
f	a, c, d
g	



Representações de grafos

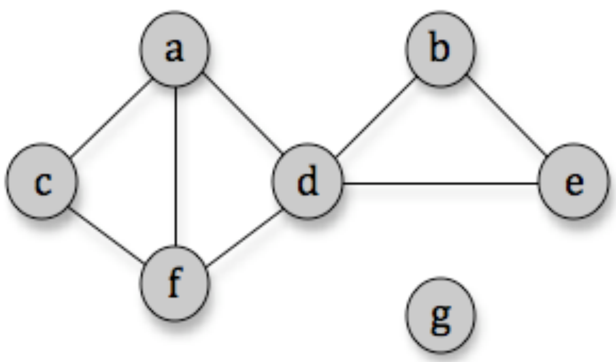
- Outra representação: uma matriz, que pode ser uma matriz de adjacência ou uma matriz de incidência.
- A **matriz de adjacência** de G = (V, A), com |V| = n, é uma matriz binária n × n em que cada elemento $\mathbf{adj_{ij}}$ é tal que:
  - adj_{ij} = 1, se existe uma aresta $(v_i v_j) \in A$;
  - adj_{ij} = 0, caso contrário.
- **Exemplo**:



	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

Representações de grafos

- A **matriz de incidência** de um grafo G = (V, A), com |V| = n e |A| = m, é uma matriz binária $n \times m$ em que cada elemento $\mathbf{inc_{ij}}$ é tal que:
  - $inc_{ij} = 1$, se a aresta a_j é incidente ao vértice v_i
  - $inc_{ij} = 0$, caso contrário.
- **Exemplo**:



	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

Representações de grafos

- Qual é a melhor representação? Depende do problema:
  - Se for preciso processar vértices adjacentes a um dado vértice v, a lista de adjacências exige grau(v) passos, enquanto a matriz de adjacência exige |V| passos.
  - Incluir ou excluir um vértice adjacente a v exige manutenção da lista encadeada na representação por lista de adjacências. No caso da representação por matriz, isto exige apenas a troca de 0 para 1 (inclusão) ou de 1 para 0 (exclusão) em um elemento da matriz.
- Seja um grafo não ponderado descrito por sua matriz de adjacência **adj**. Considere a expressão:

```
(adj[i][k] == 1 && adj[k][j] == 1)
```

Esta expressão será verdadeira **se e somente se** existir uma aresta entre os nós i e k e uma aresta entre os nós k e j, ou seja, se e somente se existir um **caminho de comprimento 2** entre os nós i e j, passando por k.

Representações de grafos

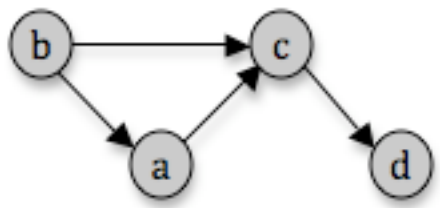
- E a expressão:

```
(adj[i][0] == 1 && adj[0][j] == 1) ||
(adj[i][1] == 1 && adj[1][j] == 1) ||
                   ...
(adj[i][n-1] == 1 && adj[n-1][j] == 1)
```

- Esta expressão será verdadeira somente se existir um caminho de comprimento 2 entre os nós i e j, passando por qualquer um dos nós 0, 1, ..., n-1. Em outras palavras, a expressão será verdadeira somente se existir um caminho de comprimento 2 entre os nós i e j.
- Seja **adj2** uma matriz tal que adj2[i][j] = 1 se a expressão acima for verdadeira e adj2[i][j] = 0, caso contrário. Esta matriz pode ser imaginada como a matriz de caminhos de comprimento 2 do grafo. **Como calcular adj2**?

Representações de grafos

Exemplo:



	a	b	c	d
a	0	0	1	0
b	1	0	1	0
c	0	0	0	1
d	0	0	0	0

- Pelo grafo, percebe-se claramente que existem caminhos de comprimento 2 entre os nós:
  - a e d e portanto, adj2[a][d] = 1
  - b e c e portanto, adj2[b][c] = 1
  - b e d e portanto, adj2[b][d] = 1

	a	b	c	d
a	0	0	0	1
b	0	0	1	1
c	0	0	0	0
d	0	0	0	0

- Para calcular **adj2** basta multiplicar adj por si mesma:

$$\begin{vmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} \times \begin{vmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Representações de grafos

- De modo semelhante, podemos definir **adj3** (matriz de caminhos de comprimento 3), **adj4** (matriz de caminhos de comprimento 4) e assim por diante.
- Em termos gerais, para calcular uma matriz de caminhos de comprimento **c**, basta efetuar o produto da matriz de caminhos de comprimento **c-1** pela matriz de adjacência.
- Imagine que desejamos saber se existe um caminho de comprimento menor ou igual a 3 entre dois nós i e j de um grafo. Se existir, tal caminho terá comprimento 1, 2 ou 3.

  adj[i][j] == 1 || adj2[i][j] == 1 || adj3[i][j] == 1
- Considere que desejamos formar uma matriz C tal que C[i][j] = 1, se e somente se existir um caminho entre i e j de qualquer tamanho (C[i][j] = 0, caso contrário). **Como calcular C**?

Representações de grafos

- Imagine um grafo com n nós.

  Se existir um caminho de comprimento m > n entre i e j neste grafo, então algum nó neste caminho, digamos k, aparece mais de uma vez, o que configura um ciclo. Neste caso, removendo o ciclo de k até k, obtém-se outro caminho entre i e j de comprimento ≤ n. Logo:

  C[i][j] = (adj[i][j] == 1 || ... || adjn[i][j] == 1)

- A matriz C é conhecida como **fecho transitivo** da matriz adj.

- Portanto, para calcular C basta calcular as matrizes **adj2**, **adj3**, ..., **adjn** e efetuar a disjunção dos valores.

Fecho transitivo de um grafo

Este algoritmo é **O(n^4)**, pois calcular o produto é O(n^3) e em **FechoTransitivo** a função **produto** é chamada n vezes.

```
matriz FechoTransitivo(matriz adj)
{
  int i,j,k;
  matriz p,C;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
      p[i][j] = adj[i][j];
      C[i][j] = adj[i][j];
    }

  for (k = 1; k < n; k++)
  {
    p = produto(p,adj);
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        C[i][j] = (C[i][j] || p[i][j]);
  }
}
```

Algoritmo de Warshall

- Uma forma mais eficiente para calcular o fecho transitivo de um grafo é conhecido como **algoritmo de Warshall**.
- Seja $C_k[i][j] = 1$ se e somente se existir um caminho entre os nós i e j que não passe por nenhum nó com numeração maior do que k (exceto, possivelmente, i e j).
- Como obter C_{k+1} a partir de C_k?
- Se $C_k[i][j] = 1$, então $C_{k+1}[i][j] = 1$.
- Se $C_k[i][j] = 0$, então $C_{k+1}[i][j] = 1$ somente se existir um caminho entre i e j passando pelo nó k+1, pois não existe caminho entre i e j que passe pelos nós de 1 a k. Logo, existe um caminho entre i e k+1 e um caminho entre k+1 e j, passando somente pelos nós de 1 a k.
- Logo: $C_{k+1}[i][j] = 1 \quad \Leftrightarrow \quad \begin{cases} C_k[i][j] = 1, \quad ou \\ C_k[i][k+1] = 1 \quad e \quad C_k[k+1][j] = 1 \end{cases}$

Algoritmo de Warshall

- Evidentemente, $C_0[i][j] = adj[i][j]$, pois a única maneira de ir do nó i para o nó j sem passar por qualquer outro nó é seguir diretamente de i para j.
- Além disso, $C_n[i][j] = C[i][j]$, pois qualquer caminho entre os nós i e j não irá passar por um nó com numeração maior do que n (os nós estão numerados de 1 a n).

```
matriz FechoTransitivo(matriz adj)
{
  int i,j,k;
  matriz C;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      C[i][j] = adj[i][j];

  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        C[i][j] = (C[i][j] || (C[i][k] && C[k][j]));

  return C;
}
```

Note que este algoritmo é **$O(n^3)$**.

Percursos em grafos

- Percorrer um grafo consiste em visitar cada um de seus vértices apenas uma vez.
- Como no caso de árvores, existem dois tipos de percursos em grafos: a busca em profundidade e a busca em largura.
- O algoritmo de busca em profundidade usa de uma pilha (explícita ou implícitamente, devido à recursão) e o algoritmo de busca em largura usa uma fila.
- **Busca em profundidade**: cada vértice v é visitado e então, cada vértice ainda não visitado adjacente a v é visitado. Se v não tem vértices adjacentes ou se todos eles já foram visitados, volta-se para o predecessor de v. O percurso termina quando este processo levar ao vértice no qual o percurso começou. Caso exista um vértice v ainda não visitado, o percurso é reiniciado para v.

Busca em profundidade

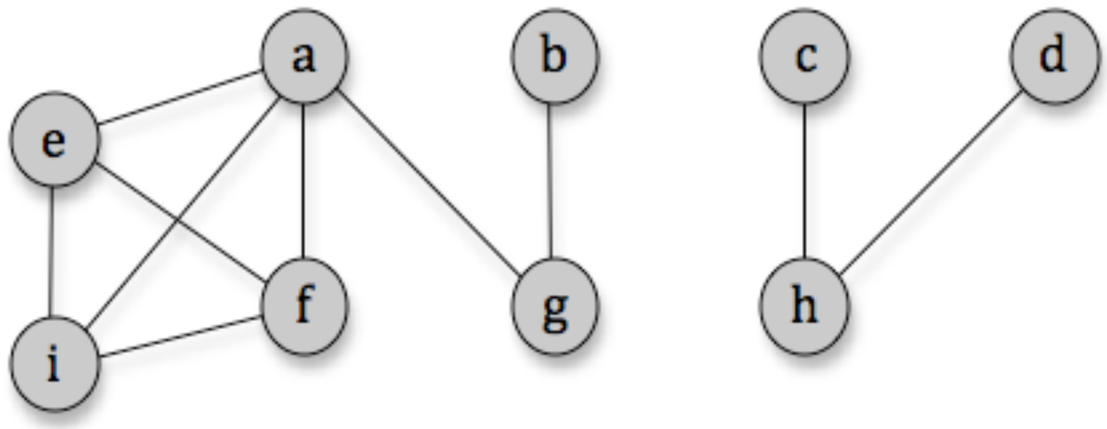
Notar que o **while** em **BuscaEmProfundidade** garante o percurso mesmo para grafos desconexos. Para grafos conexos, o laço do while será executado apenas 1 vez.

```
void BuscaEmProfundidade()
{
  for (cada v ∈ V)
  {
    num[v] = 0;
  }
  arestas = ∅;
  i = 1;
  while (existe v ∈ V tal que num[v] == 0)
  {
    Profundidade(v,i);
  }
  mostrar(arestas);
}
```

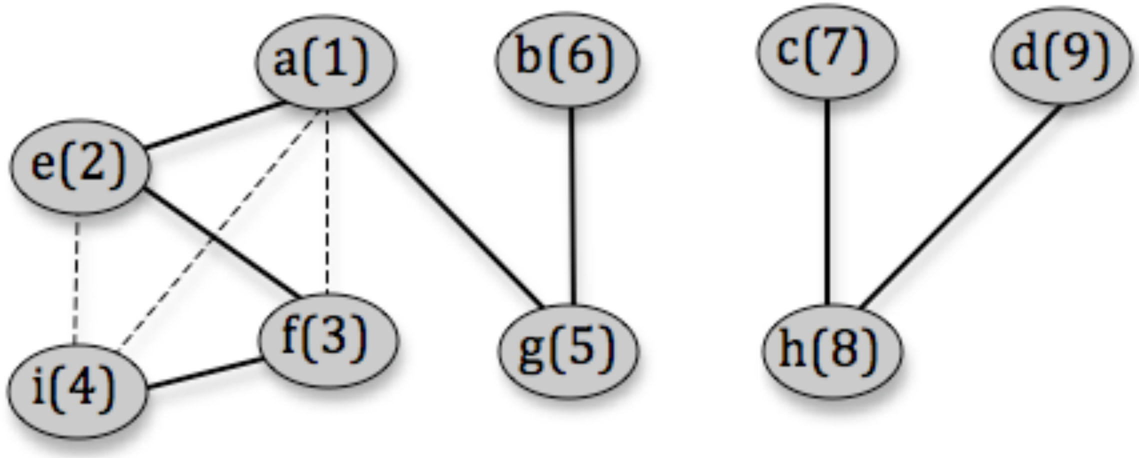
```
void Profundidade(v,i)
{
  num[v] = i++;
  for (todos u adjacentes a v)
  {
    if (num[u] == 0)
    {
      arestas = arestas ∪ (vu);
      Profundidade(u,i);
    }
  }
}
```

Busca em profundidade

Exemplo:



- Qual será a numeração dos vértices e o conjunto arestas?



Notar que a busca em profundidade produz uma **árvore** (ou um conjunto de árvores). Esta árvore é conhecida como **árvore de espalhamento** (*spanning tree*).

Percursos em grafos

- Para um grafo G = (V, A), a complexidade do algoritmo de busca em profundidade é O(|V|+|A|), pois:
  - Inicializar num[v] para cada vértice v exige |V| passos;
  - **while (existe v ∈ V tal que num[v] == 0)** pode exigir até |V| passos;
  - **Profundidade(v,i)** é chamada grau(v) vezes para cada v, portanto, o número total de chamadas é 2 |A|.

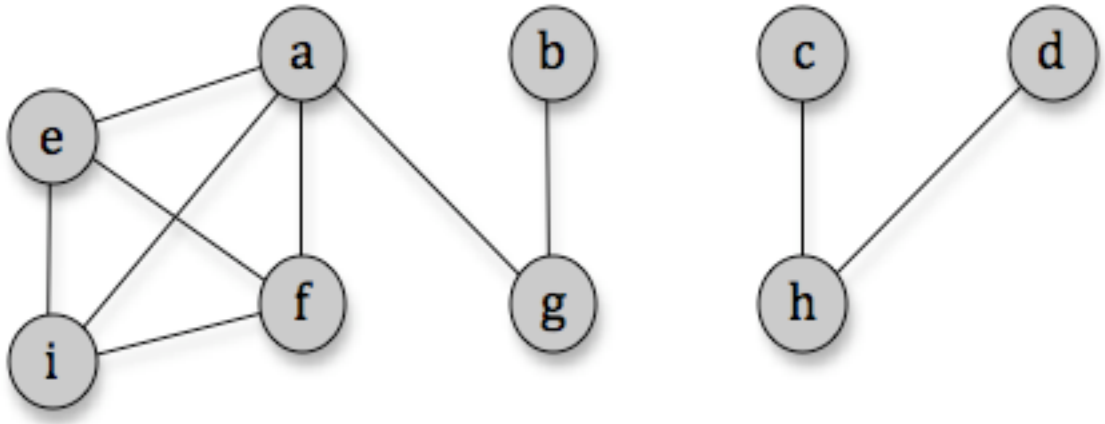
- Outro tipo de percurso em grafos é a **busca em largura**. O algoritmo de busca em largura procura visitar todos os vizinhos de um vértice v antes de prosseguir para outros vértices.

Busca em largura

```
void BuscaEmLargura()
{
  for (cada v ∈ V) num[v] = 0;
  arestas = ∅; fila = NULL; i = 0;
  while (existe v ∈ V tal que num[v] == 0)
  {
    num[v] = i++;
    incluir(fila,v);
    while (fila != NULL)
    {
      v = excluir(fila);
      for (todos os vértices u adjacentes a v)
        if (num[u] == 0)
        {
          num[u] = i++;
          arestas = arestas ∪ (vu);
          incluir(fila,u);
        }
    }
  }
}
```

Busca em largura

Exemplo:



- Qual será a numeração dos vértices e o conjunto arestas?

v	u	num	arestas	fila
a		num[a] = 1		a
a				NULL
	e	num[e] = 2	(ae)	e
	f	num[f] = 3	(ae)(af)	e f
	g	num[g] = 4	(ae)(af)(ag)	e f g
	i	num[i] = 5	(ae)(af)(ag)(ai)	e f g i
e				f g i
f				g i
g				i
	b	num[b] = 6	(ae)(af)(ag)(ai)(gb)	i b
i				b
b				NULL
c		num[c] = 7		c
c				NULL
	h	num[h] = 8	(ae)(af)(ag)(ai)(gb)(ch)	h
h				NULL
	d	num[d] = 9	(ae)(af)(ag)(ai)(gb)(ch)(hd)	d
d				NULL

Percursos em grafos

- Para um grafo G = (V, A), a complexidade do algoritmo de busca em largura também é O(|V|+|A|), pois:
  - Inicializar num[v] para cada vértice v exige |V| passos;
  - **while (existe v $\in$ V tal que num[v] == 0)** pode exigir até |V| passos. Deve-se notar que cada vértice é colocado na (e retirado da) fila apenas uma vez e que as operações de incluir e excluir da fila podem ser executadas em tempo O(1);
  - **for (todos os vértices u adjacentes a v)** exige grau(v) iterações para cada v e, portanto, o número total de iterações é 2|A|.

Caminhos de menor peso em grafos

- Determinar o caminho de menor peso em grafos ponderados é um problema clássico e muitos algoritmos têm sido propostos.
- Nestes algoritmos, para determinar o caminho de menor peso entre os vértices u e v, a informação sobre os pesos de u a vértices intermediários w precisa ser registrada. Isto pode ser feito associando um rótulo aos vértices.
- Dependendo de como os rótulos são atualizados, os algoritmos de caminho de menor peso são divididos em duas classes:
  - Os algoritmos de estabelecimento de rótulos
  - Os algoritmos de correção de rótulos

Caminhos de menor peso em grafos

- Nos **algoritmos de estabelecimento de rótulos**, a cada iteração, um vértice v recebe um valor definitivo, ou seja, um valor que permanece imutável até o fim da execução (este valor corresponde ao menor peso entre o vértice inicial e v). Isso limita a aplicação destes algoritmos a grafos que possuem apenas **pesos não-negativos**.
- Nos **algoritmos de correção de rótulos**, o valor de qualquer vértice pode ser alterado durante a execução do algoritmo. Neste caso, o algoritmo pode ser aplicado a grafos com pesos negativos (mas sem ciclos negativos, ou seja, ciclos compostos de arestas com pesos que somam um valor negativo).
- Estas duas classes de algoritmos, no entanto, podem ser subordinadas à um mesmo esquema geral.

Caminhos de menor peso em grafos

```
CaminhoMaisCurto(grafo G, vertice s)
{
  for (cada v ∈ V)
    valor[v] = INFINITO;
  valor[s] = 0;
  inicializar FaltaVerificar;
  while (FaltaVerificar != vazio)
  {
    v = um vértice de FaltaVerificar;
    excluir v de FaltaVerificar;
    for (todos os vértices u adjacentes a v)
    {
      if (valor[u] > valor[v] + peso(vu))
      {
        valor[u] = valor[v] + peso(vu);
        predecessor(u) = v;
        FaltaVerificar = FaltaVerificar ∪ {u};
      }
    }
  }
}
```

Neste algoritmo, **G** é um grafo ponderado e **s** é o vértice inicial do caminho.

O **rótulo** de um vértice consiste de dois elementos:

(valor, predecessor)

O esquema não especifica:

- A organização do conjunto **FaltaVerificar**;
- A escolha de v em "**v = um vértice de FaltaVerificar**";

Caminhos de menor peso em grafos

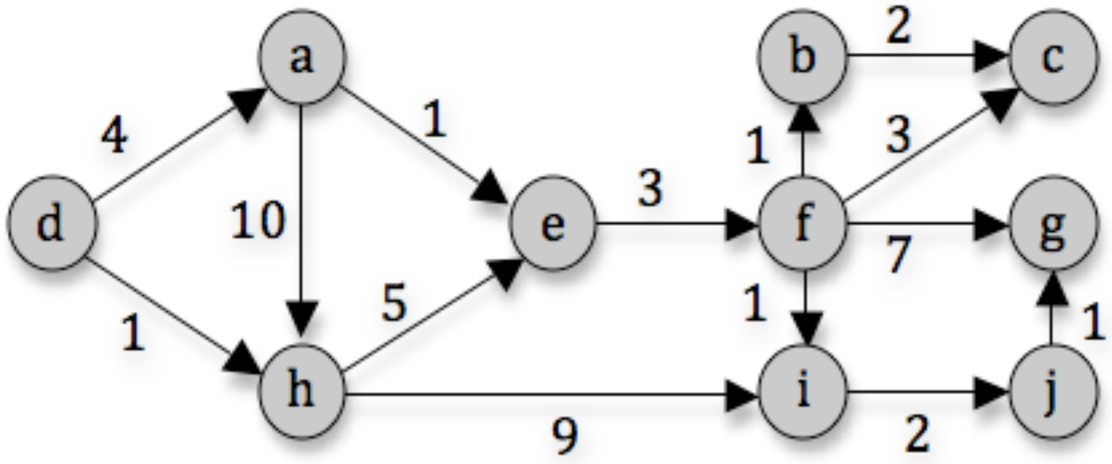
- Um dos primeiros algoritmos de estabelecimento de rótulos foi desenvolvido por Dijkstra.

```
AlgoritmoDijkstra(grafo G, vertice s)
{
  for (cada v ∈ V)
  {
    valor[v] = INFINITO;
  }
  valor[s] = 0;
  FaltaVerificar = V;
  while (FaltaVerificar != vazio)
  {
    v = vértice de FaltaVerificar com menor valor[v];
    excluir v de FaltaVerificar;
    for (todos os vértices u adjacentes a v)
    {
      if (valor[u] > valor[v] + peso(vu))
      {
        valor[u] = valor[v] + peso(vu);
        predecessor(u) = v;
      }
    }
  }
}
```

Notar que a estrutura de **FaltaVerificar** não está especificada neste algoritmo. A eficiência do algoritmo depende de quão rapidamente será possível recuperar desta estrutura o vértice com o **menor** valor.

Algoritmo de Dijkstra

- **Exemplo**: considerar **d** como vértice inicial.



iter	0	1	2	3	4	5	6	7	8	9	10
ativo		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

Note que, na iteração 1, o valor de **d** é definitivo. Na iteração 2, o valor de **h** é definitivo, e assim por diante. Considerando os pesos como **distâncias**, os valores finais de cada vértice **v** corresponde à menor distância entre **d** e **v**. Para determinar o caminho mais curto entre esses vértices basta percorrer os predecessores, de **v** até **d**.

Algoritmo de Dijkstra

- A complexidade do algoritmo de Dijkstra é $O(|V|^2)$, pois:
  - Inicializar valor[v] para cada vértice v exige |V| passos;
  - O comando **while** pode exigir até |V| passos;
  - A cada iteração do **while**, encontrar o vértice com menor valor pode exigir até |V| comparações;
  - Como cada vértice é incluído no conjunto **FaltaVerificar** exatamente uma vez, cada aresta da lista de adjacência de v é examinada exatamente uma vez no laço do comando **for**. Como o número total de arestas em todas as listas de adjacências é |A|, existirá um total de |A| iterações do comando **for**.
- Logo: $|V| + |V|\times|V| + |A| = O(|V|^2 + |A|) = O(|V|^2)$

A eficiência deste algoritmo pode ser melhorada organizando **FaltaVerificar** como um *heap*. Com isso, complexidade será $O(|V|\log|V| + |A|)$.

Algoritmo de Dijkstra

- O algoritmo de Dijkstra é eficiente, mas se aplica apenas a grafos com pesos não-negativos. O procedimento a seguir converte um grafo com m nós contendo pesos p_{ij} negativos em um grafo equivalente com pesos não-negativos.

```
ConverteGrafo(grafo G)
{
  for (t = 1; t <= m+1; t++)
  {
    for (i = 1; i <= m; i++)
    {
      c[i] = min{p[i][j], para todo j};
      if (c[i] < 0)
      {
        p[i][j] = p[i][j] - c[i], para todo j;
        p[k][i] = p[k][i] + c[i], para todo k;
      }
    }
    if (p[i][j] >= 0, para todo i e j) parar (convertido);
    if (t == m+1) parar (existe um circuito negativo em G);
  }
}
```

Algoritmo de Ford

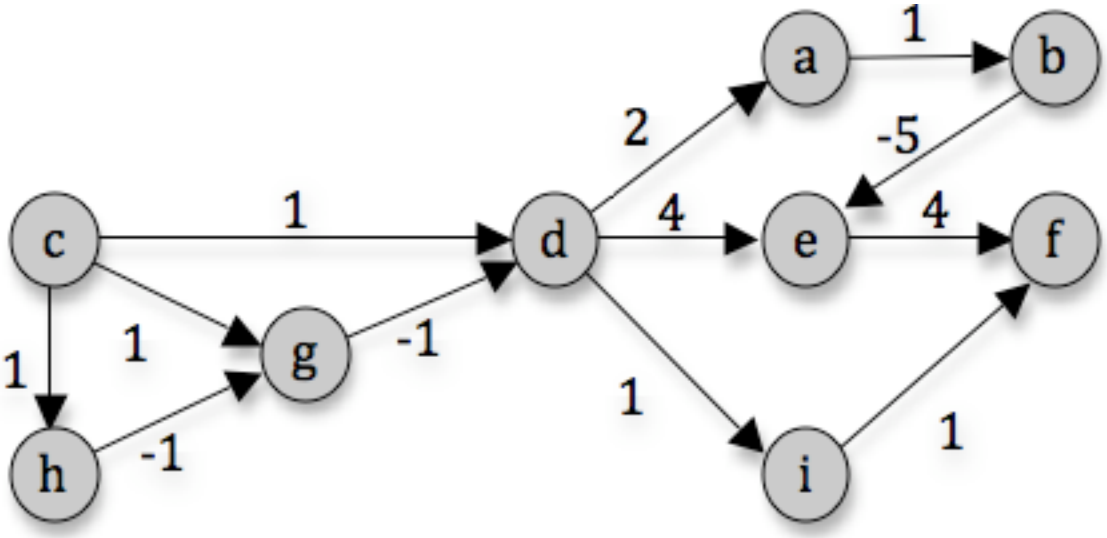
- Um dos primeiros algoritmos de correção de rótulos foi desenvolvido por Ford.

```
AlgoritmoFord(grafo G, vertice s)
{
  for (cada v ∈ V)
    valor[v] = INFINITO;
  valor[s] = 0;
  while (existe aresta (vu): valor[u] > valor[v] + peso(vu))
  {
    valor[u] = valor[v] + peso(vu);
  }
}
```

- Para o comando **while**, deve-se impor uma ordem de monitoramento das arestas (alfabética, por exemplo). O algoritmo procura repetidamente as arestas nesta ordem e ajusta o valor de qualquer vértice sempre que necessário.

Algoritmo de Ford

Exemplo: considerar **c** como vértice inicial.



Considerando a seguinte sequência de arestas:

(ab), (be), (cd), (cg), (ch), (da), (de), (di), (ef), (gd), (hg), (if).

Note que, em uma **mesma iteração**, o valor do vértice pode mudar.

iter	0	1		2	3	4
a	∞	3	3	2	1	
b	∞	∞	∞	4	3	2
c	0					
d	∞	1	0	-1		
e	∞	5	5	-1	-2	-3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2	2	1	0	

Algoritmo de Ford

- A complexidade do algoritmo de Ford é O(|V||A|), pois para cada uma das |A| arestas até |V|-1 vértices podem ser examinados.
- Notar que |V| - 1 é o maior número de arestas em qualquer caminho.
- Neste algoritmo, todas as arestas são verificadas em todas as iterações (na ordem estabelecida pela sequência) e, consequentemente, os valores dos vértices podem ser alterados várias vezes em uma mesma iteração.
- O algoritmo pode se melhorado organizando-se a lista de vértices de modo a limitar o número de visitas por vértice.

Caminho de menor peso entre todos os vértices

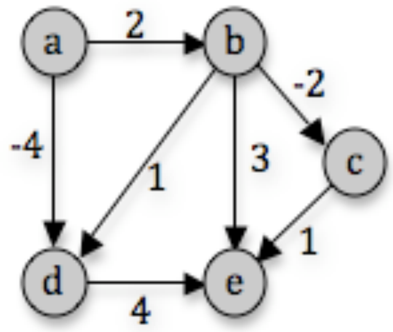
- Os algoritmos de Dijkstra e de Ford determinam os caminhos de menor peso entre um dado vértice inicial e os demais vértices do grafo. E o caminho de menor peso entre qualquer par de vértices do grafo?
- Uma possibilidade para resolver este problema é usar |V| vezes um destes algoritmos, uma vez para cada um dos vértices do grafo como vértice inicial.
- Outra possibilidade é usar o algoritmo de Floyd, que trabalha com uma matriz de adjacências contendo os pesos (positivos ou negativos) das arestas do grafo (ou digrafo).

```
AlgoritmoFloyd(matriz m)
{
  for (i = 0 até |V|-1)
    for (j = 0 até |V|-1)
      for (k = 0 até |V|-1)
        if (m[j][k] > m[j][i] + m[i][k])
          m[j][k] = m[j][i] + m[i][k];
}
```

Note que a complexidade deste algoritmo é $O(|V|^3)$.

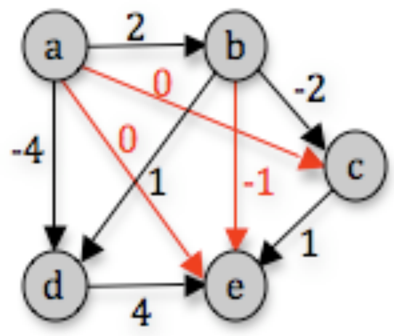
Caminho de menor peso entre todos os vértices

Exemplo:



	a	b	c	d	e
a	0	2	∞	-4	∞
b	∞	0	-2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

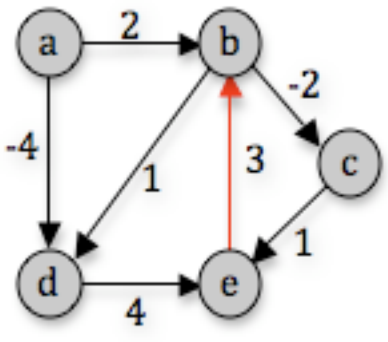
- A cada iteração deste algoritmo novos caminhos de menor peso entre pares de vértices podem ser determinados.



	a	b	c	d	e
a	0	2	0	-4	0
b	∞	0	-2	1	-1
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

Detecção de ciclos

- O algoritmo de Floyd também permite detectar ciclos: basta inicializar a diagonal da matriz de pesos com ∞ (em vez de zero). Se qualquer valor da diagonal for modificado, o grafo contém um ciclo.



	a	b	c	d	e
a	∞	2	0	-4	0
b	∞	2	-2	1	-1
c	∞	4	2	5	1
d	∞	7	5	8	4
e	∞	3	1	4	2

- Note, pela matriz de pesos final obtida pelo algoritmo de Floyd, que como existem elementos da diagonal principal da matriz com valores finitos, o grafo possui ciclos.
- Por exemplo, existe um caminho de valor igual a 8, saindo do vértice **d** e chegando no vértice **d**.

Detecção de ciclos

- Existem algoritmos mais eficientes para a detecção de ciclos, por exemplo, um algoritmo baseado na busca em profundidade para grafos não-orientados.

```
DetectarCiclosEmGrafos(vertice v)
{
  num[v] = i++;
  for (todos os vértices u adjacentes a v)
  {
    if (num[u] == 0)
    {
      arestas = arestas ∪ (vu);
      DetectarCiclosEmGrafos(u);
    }
    else
    if (aresta (uv) ∉ arestas)
    {
      ciclo detectado;
    }
  }
}
```

A complexidade deste algoritmo é O(|A|), pois **DetectarCiclosEmGrafos(u)** será chamada **grau(v)** vezes para cada vértice v e, portanto, o número total de chamadas é 2 |A|.