Grafos

Conceito

Abstração que permite codificar relacionamentos entre pares de objetos

Que objetos? Qualquer um! Ex. pessoas, cidades, empresas, países, páginas web, filmes, etc...

Que relacionamentos? Qualquer um! Ex. amizade, conectividade, produção, língua falada, etc.

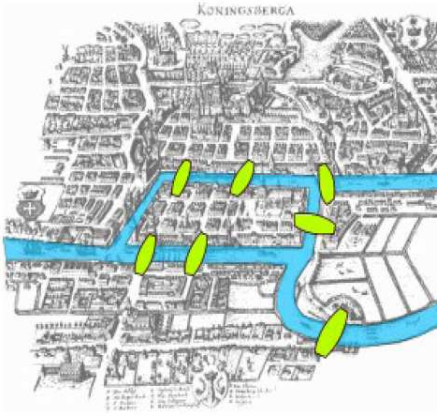
Abstração que permite codificar relacionamentos entre pares de objetos

Objetos → vértices do grafo

Relacionamentos → arestas do grafo

Início da Teoria dos Grafos

Problema: as 7 pontes de Königsberg



Desafio popular na cidade:

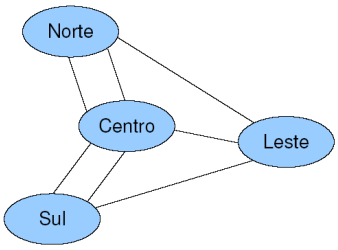
Partir de um ponto, atravessar as 7 pontes uma única vez, retornando ao ponto de partida!

Euler resolveu este problema em 1736!

Abstração via grafos

Objetos: áreas contíguas de terra;

Arestas: ponte entre áreas podendo ter mais de uma ligando regiões;



Tal trajeto existe? Não!

Conceitos Básicos da Teoria de Grafos

GRAFO

Um grafo *G(V,E)* é definido pelo par de conjuntos *V* e *E*, onde:

V - conjunto não vazio: os **vértices** ou **nodos** do grafo;

E - conjunto de pares ordenados *e* = (v,w), v e w ∈ V: as **arestas** do grafo.

Seja, por exemplo, o grafo G(V,E) dado por:

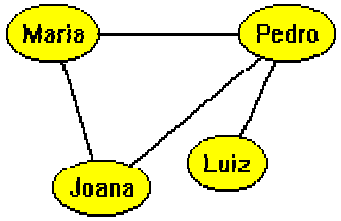
V = { p | p é uma pessoa } e E = { (v,w) | < v é amigo de w > }

Esta definição representa toda uma família de grafos. Um exemplo de elemento desta família (G_1) é dado por:

V = { Maria, Pedro, Joana, Luiz }

E = { (Maria, Pedro) , (Joana, Maria) , (Pedro, Luiz) , (Joana, Pedro) }

G_1:



Neste exemplo estamos considerando que a relação *<v é amigo de w>* é uma **relação simétrica**, ou seja, se *<v é amigo de w>* então *<w é amigo de v>*.

Como conseqüência, as arestas que ligam os vértices não possuem qualquer orientação

DIGRAFO (Grafo Orientado)

Considere, agora, o grafo definido por:

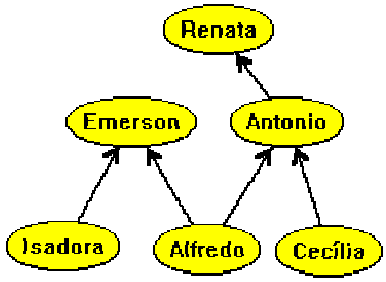
V = { p | p é uma pessoa da família Castro } e E = { (v,w) | < v é pai/mãe de w >}

Um exemplo de deste grafo (ver G_2) é:

V = { Emerson, Isadora, Renata, Antonio, Rosane, Cecília, Alfredo }

E = {(Isadora, Emerson), (Antonio, Renata), (Alfredo, Emerson), (Cecília, Antonio), (Alfredo, Antonio)}

G_2:



A relação definida por *E* **não é simétrica**, pois se

<v é pai/mãe de w>, não é o caso de *<w é pai/mãe de v>.*

Há, portanto, uma orientação na relação, com um correspondente efeito na representação gráfica de G.

O grafo acima é dito ser um **grafo orientado** (ou **dígrafo**), sendo que as conexões entre os vértices são chamadas de **arcos**.

ORDEM

A ordem de um grafo *G* é dada pela cardinalidade do conjunto de vértices, ou seja, pelo número de vértices de *G*. Nos exemplos:

ordem(G_1) = 4

ordem(G_2) = 6

ADJACÊNCIA

Em um grafo simples (a exemplo de G_1) dois vértices v e w são adjacentes (ou vizinhos) se há uma aresta $a=(v,w)$ em G. Está aresta é dita ser incidente a ambos, v e w. É o caso dos vértices *Maria* e *Pedro* em G_1. No caso do grafo ser dirigido (a exemplo de G_2), a adjacência (vizinhança) é especializada em:

Sucessor: um vértice w é sucessor de v se há um arco que parte de v e chega em w. Em G_2, por exemplo, diz-se que *Emerson* e *Antonio* são sucessores de *Alfredo*.

Antecessor: um vértice v é antecessor de w se há um arco que parte de v e chega em w. Em G_2, por exemplo, diz-se que *Alfredo* e *Cecília* são antecessores de *Antonio*.

GRAU

O grau de um vértice é dado pelo número de arestas que lhe são incidentes. Em G_1, por exemplo:

- grau(*Pedro*) = 3
- grau(*Maria*) = 2

No caso do grafo ser dirigido (a exemplo de G_2), a noção de grau é especializada em:

Grau de emissão: o grau de emissão de um vértice v corresponde ao número de arcos que partem de v. Em G_2, por exemplo:

- grauDeEmissão(*Antonio*) = 1
- grauDeEmissao(*Alfredo*) = 2
- grauDeEmissao(*Renata*) = 0

Grau de recepção: o grau de recepção de um vértice v corresponde ao número de arcos que chegam a v. Em G_2, por exemplo:

- grauDeRecepção(*Antonio*) = 2
- grauDeRecepção(*Alfredo*) = 0
- grauDeRecepção(*Renata*) = 1

FONTE

Um vértice v é uma fonte se grauDeRecepção(v) = 0.

É o caso dos vértices *Isadora*, *Alfredo* e *Cecília* em G_2.

SUMIDOURO

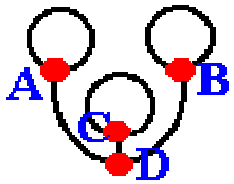
Um vértice v é um sumidouro se grauDeEmissão(v) = 0.

É o caso dos vértices *Renata* e *Emerson* em G_2.

LAÇO

Um laço é uma aresta ou arco do tipo $a=(v,v)$, ou seja, que relaciona um vértice a ele próprio. Em G_3 há três ocorrências de laços para um grafo não orientado.

G_3:

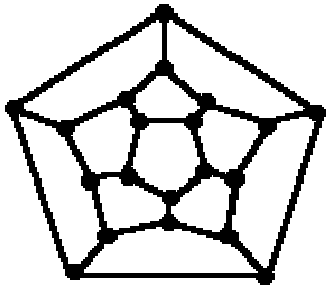


GRAFO REGULAR

Um grafo é dito ser regular quando todos os seus vértices têm o mesmo grau.

O grafo G_4, por exemplo, é dito ser um grafo regular-3, pois todos os seus vértices têm grau 3.

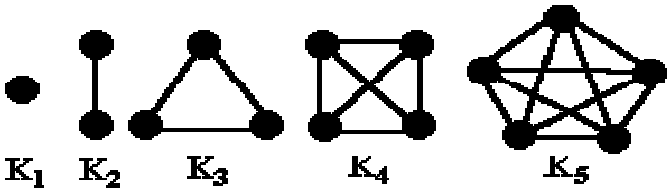
G_4:



GRAFO COMPLETO

Um grafo é dito ser completo quando há uma aresta entre cada par de seus vértices.

Estes grafos são designados por K_n, onde n é a ordem do grafo.



Um grafo K_n possui o número máximo possível de arestas para um dado ***n***. Ele é, também *regular-(n-1)*, pois todos os seus vértices tem grau *n-1*.

GRAFO BIPARTIDO

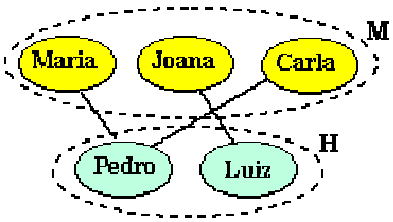
Um grafo é dito ser bipartido quando seu conjunto de vértices V puder ser particionado em dois subconjuntos V_1 e V_2, tais que toda aresta de G une um vértice de V_1 a outro de V_2.

Para exemplificar, sejam os conjuntos H={h | h é um homem} e M={m | h é um mulher} e o grafo $G(V,E)$ (ver o exemplo G_5) onde:

$V = H \cup M$

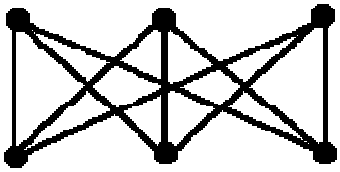
E = {(v,w) | (v ∈ H e w ∈ M) ou (v ∈ M e w ∈ H) e <v foi namorado de w>}

G_5:



O grafo G_6 é uma $K_{3,3}$, ou seja, um grafo **bipartido completo** que contém duas partições de 3 vértices cada.

G_6:



***K*3,3**

Ele é completo, pois todos os vértices de uma partição estão ligados a todos os vértices da outra partição.

GRAFO ROTULADO

Um grafo $G(V,A)$ é dito ser rotulado em vértices (ou arestas) quando a cada vértice (ou aresta) estiver associado um rótulo. G_5 é um exemplo de grafo rotulado.

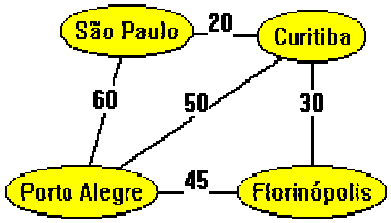
GRAFO VALORADO

Um grafo $G(V,E)$ é dito ser valorado quando existe uma ou mais funções relacionando V e/ou E com um conjunto de números.

Para exemplificar (ver o grafo G_7), seja $G(V,E)$ onde:

- V = {v | v é uma cidade com aeroporto}
- E = {(v,w,t) | <há linha aérea ligando v a w, sendo t o tempo esperado de vôo>}

G_7:

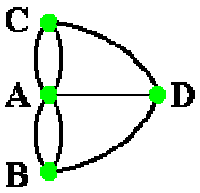


MULTIGRAFO

Um grafo $G(V,E)$ é dito ser um multigrafo quando existem múltiplas arestas entre pares de vértices de G.

No grafo G_8, por exemplo, há duas arestas entre os vértices A e C e entre os vértices A e B, caracterizando-o como um multigrafo.

G_8:

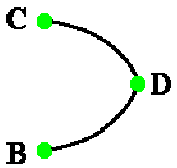


SUBGRAFO

Um grafo $G_s(V_s, E_s)$ é dito ser subgrafo de um grafo $G(V,E)$ quando $V_s \subseteq V$ e $E_s \subseteq E$.

O grafo G_9, por exemplo, é subgrafo de G_8.

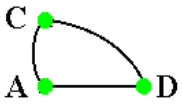
G_9:



Subgrafos interessantes:

Clique: Uma clique é um subgrafo que é completo, por exemplo G_{9b}

G_{9b}:



Conjunto independente de vértices: Um subgrafo induzido de G que não contém nenhuma aresta, por exemplo G_{9c}.

G_{9c}:

CAMINHO

Um caminho é uma cadeia na qual todos os arcos possuem a mesma orientação. Aplica-se, portanto, somente a grafos orientados. A seqüência de vértices (x_1, x_2, x_5, x_6, x_3) é um exemplo de caminho em G_{11}.

CICLO

Um ciclo é uma cadeia simples e fechada (o vértice inicial é o mesmo que o vértice final).

A seqüência de vértices (x_1, x_2, x_3, x_6, x_5, x_4, x_1) é um exemplo de ciclo elementar em G_{11}.

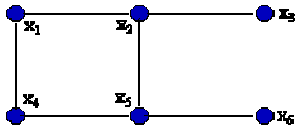
CIRCUITO

Um circuito é um caminho simples e fechado. A seqüência de vértices (x_1, x_2, x_5, x_4, x_1) é um exemplo de circuito elementar em G_{11}.

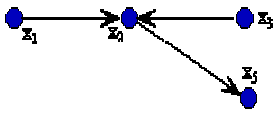
GRAFO CONEXO

Um grafo $G(V,A)$ é dito ser conexo se há pelo menos uma cadeia ligando cada par de vértices deste grafo G.

G_{12}:



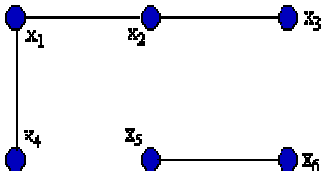
G_{13}:



GRAFO DESCONEXO

Um grafo $G(V,E)$ é dito ser desconexo se há pelo menos um par de vértices que não está ligado por nenhuma cadeia.

G_{14}:

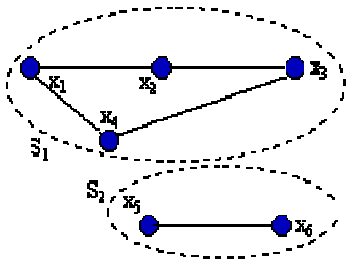


COMPONENTE CONEXA

Um grafo $G(V,E)$ desconexo é formado por pelo menos dois subgrafos conexos, disjuntos em relação aos vértices e maximais em relação à inclusão.

Cada um destes subgrafos conexos é disto ser uma componente conexa de G.

G_{15}:

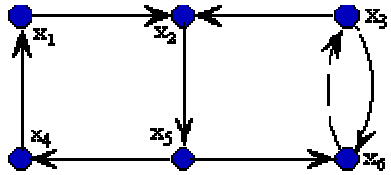


GRAFO FORTEMENTE CONEXO

No caso de grafos orientados, um grafo é dito ser fortemente conexo (f-conexo) se todo par de vértices está ligado por pelo menos um caminho em cada sentido, ou seja, se cada par de vértices participa de um circuito.

Isto significa que cada vértice pode ser alcançável partindo-se de qualquer outro vértice do grafo.

G_{16}:

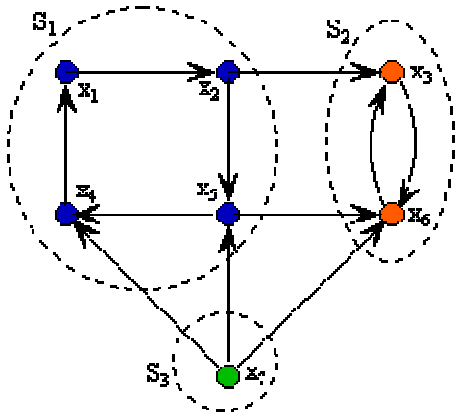


COMPONENTE FORTEMENTE CONEXA

Um grafo *G*(*V,E*) que não é fortemente conexo é formado por pelo menos dois subgrafos fortemente conexos, disjuntos em relação aos vértices e maximais em relação à inclusão.

Cada um destes subgrafos é dito ser uma componente fortemente conexa de *G*, a exemplo dos subgrafos identificados por S_1, S_2 e S_3 em G_{17}.

G_{17}:



VÉRTICE DE CORTE

Um vértice é dito ser um vértice de corte se sua remoção (juntamente com as arestas a ele conectadas) provoca uma redução na conexidade do grafo. Os vértices x_2 em G_{13} e G_{14} são exemplos de vértices de corte.

PONTE

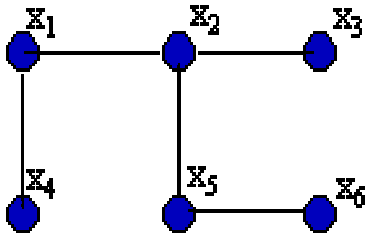
Uma aresta é dita ser um a ponte se sua remoção provoca uma redução na conexidade do grafo. As arestas (x_1, x_2) em G_{13} e G_{14} são exemplos de pontes.

ÁRVORE

Uma árvore é um grafo conexo sem ciclos. Seja *G(V,E)* um grafo com ordem *n* > 2; as propriedades seguintes são equivalentes para caracterizar G como uma árvore:

1. *G* é conexo e sem ciclos;
2. *G* é sem ciclos e tem n-1 arestas;
3. *G* é conexo e tem n-1 arestas;
4. *G* é sem ciclos e por adição de uma aresta se cria um ciclo e somente um;
5. *G* é conexo, mas deixa de sê-lo se uma aresta é suprimida (todas as arestas são pontes);
6. todo par de vértices de *G* é unido por uma e somente uma cadeia simples.

G_{20}:

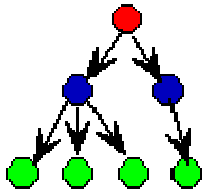


ARBORESCÊNCIA

Uma arborescência é uma árvore que possui uma raiz.

Aplica-se, portanto, somente a grafos orientados.

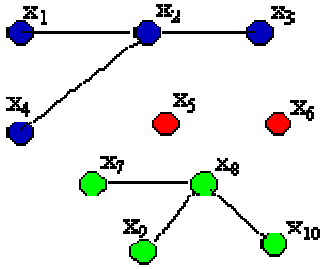
G_{21}:



FLORESTA

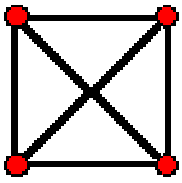
Uma floresta é um grafo cujas componentes conexas são árvores.

G_{22}:

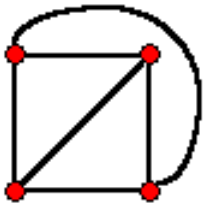


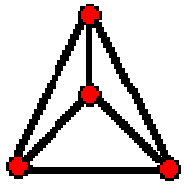
GRAFO PLANAR

Um grafo $G(V,E)$ é dito ser planar quando existe alguma forma de se dispor seus vértices em um plano de tal modo que nenhum par de arestas se cruze.



Ao lado aparecem três representações gráficas distintas para uma K_4 (grafo completo de ordem 4).

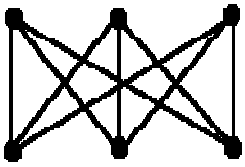




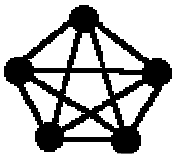
Apesar de haver um cruzamento de arestas na primeira das representações gráficas, a K_4 é um grafo planar, pois admite pelo menos uma representação num plano sem que haja cruzamento de arestas (duas possíveis representações aparecem nas figuras ao lado).

Já uma K_5 e uma $K_{3,3}$ são exemplos de grafos não planares.

Estes dois grafos não admitem representações planares.



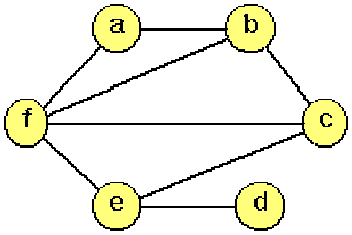
$K_{3,3}$

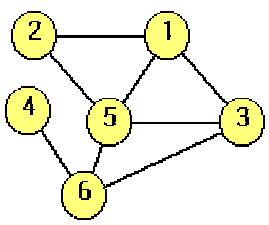


K_5

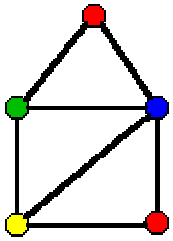
ISOMORFISMO

Sejam os grafos $G_1(V_1,E_1)$ e $G_2(V_2,E_2)$, um isomorfismo de G_1 sobre G_2 é um mapeamento bijetivo $f: V_1 \rightarrow V_2$ tal que $\{x,y\} \in E_1$ se e somente se $\{f(x),f(y)\} \in E_2$, para todo $x,y \in V_1$. Os grafos abaixo são isomorfos, pois há a função { (a →2), (b →1), (c →3), (d →4), (e →6), (f →5) } que satisfaz a condição descrita.





COLORAÇÃO

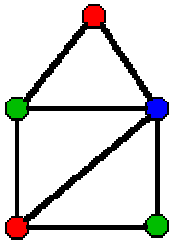


Seja *G(V,E)* um grafo e *C* um conjunto de cores. Uma coloração de *G* é uma atribuição de alguma cor de *C* para cada vértice de *V*, de tal modo que a dois vértices adjacentes sejam atribuídas cores diferentes.

Assim sendo, uma coloração de *G* é uma função $f: V \rightarrow C$ tal que para cada par de vértices $v,w \in V$ tem se $(v,w) \in E \rightarrow f(v) \neq f(w)$.

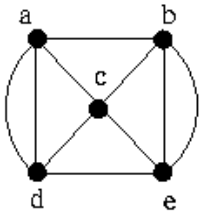
Uma *k*-coloração de *G* é uma coloração que utiliza um total de *k* cores. O exemplo ao lado mostra um 4-coloração para o grafo.

NÚMERO CROMÁTICO



Denomina-se número cromático *X(G)* de um grafo *G* ao menor número de cores *k*, para o qual existe uma *k*-coloração de *G*. O exemplo ao lado mostra uma 3-coloração para o grafo, que é o número cromático deste grafo.

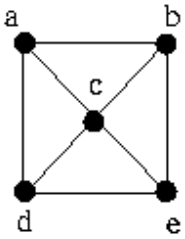
GRAFO EULERIANO



É aquele que possui um ciclo que contem todas as suas arestas. Este ciclo é dito ser um ciclo euleriano.

Circuito euleriano: é aquele que parte de um vértice v, passa por todas as arestas uma só vez.

GRAFO HAMILTONIANO

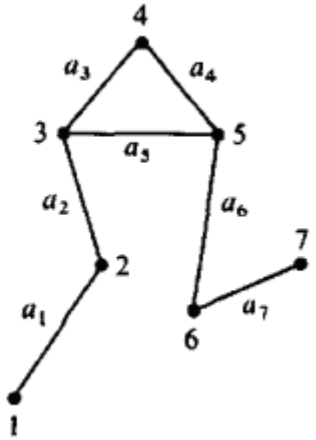


É aquele que possui um ciclo que contem todos os vértices.

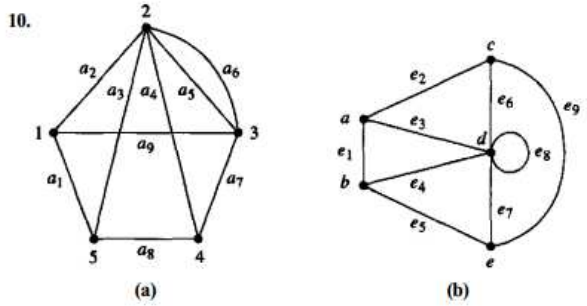
Circuito Hamiltoniano: inclui todas os vértices uma única vez (exceto o inicial = final). Pode não incluir todas as arestas, ou seja, pode não gerar um circuito euleriano.

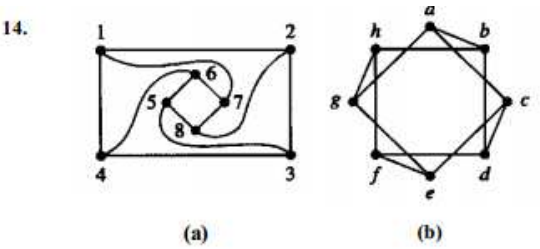
Exercícios

1. Responda as seguintes perguntas sobre o grafo mostrado
   a. Este grafo é simples?
   b. Este grafo é completo?
   c . Este grafo é conexo?
   d. Existem dois caminhos entre os vértices 3 e 6?
   e. Este grafo possui algum ciclo?
   f. O grafo possui algum vértice cuja remoção o tornaria uma árvore?
   g. O grafo possui algum vértice cuja remoção o tornaria desconexo?



2. Diga se os dois grafos apresentados são ou não isomorfos. Se forem, apresente a função que estabelece o isomorfismo entre eles; caso contrário, explique o porquê.



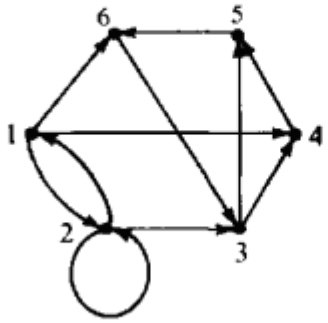


3. Cinco lobistas políticos estão visitando sete membros do Congresso (chamados de A a G) no mesmo dia. Os membros do Congresso que os cinco lobistas precisam visitar são:

1 :A,B,D 2:B,C ,F 3:A,B,D,G 4:E,G 5:D,E,F

Cada membro do Congresso estará disponível por uma hora. Qual o menor número de intervalos de visita que deve ser usado afim de que não haja conflitos entre os lobistas? (Dica: Trate isto como um problema de coloração de grafos.) O que ocorreria se o lobista 3 descobrisse que não precisa visitar o membro B, e o lobista 5 descobrisse que não precisa visitar o membro D?

4. Use o grafo direcionado da figura a seguir para responder as perguntas abaixo
   a. Quais vértices são alcançáveis a partir do vértice 3?
   b. Qual é o comprimento do menor caminho entre 3 e 6?
   c. Qual é o caminho do vértice 1 ao vértice 6 com comprimento 8?



5. **a.** Desenhe K_6. b. Desenhe $K_{3,4}$.

Algoritmos em Grafos

Representação computacional

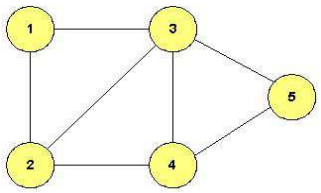
Para especificar um grafo (dígrafo) é preciso dizer quais são seus vértices e quais são suas arestas (arcos). Quanto aos vértices podemos especificar como 1,2,3,...,etc. E quanto as arestas (ou arcos) temos a matriz de adjacências ou, as listas de adjacências.

Matriz de adjacências:

- Trata-se de uma matriz quadrada, digamos Adj, cujas linhas e colunas são indexadas pelos vértices.
- Para cada par *u,v* de vértices, Adj[u,v] = 1 se existe aresta (arcos) de *u* para *v* e Adj[u,v] = 0 em caso contrário.

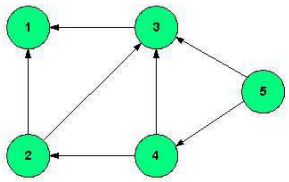
Exemplos de matriz de adjacência:

Grafo não orientado:



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

Grafo orientado:



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

Representação computacional:

A estrutura de dados deve informar:

- O número de vértices;
- A quantidade de arestas (ou arcos);
- A matriz de adjacências
- Valores e/ou atributos associados a cada aresta (arco)

Ilustrando a estrutura teríamos o Tipo Abstrato de Dado (TAD) grafos (ou dígrafos):

V = nº vértices A = nº arestas

Adj

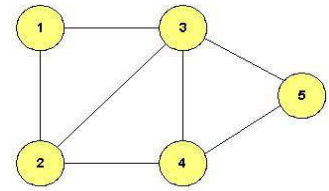
(1,1)	(1,2)	(1,3)	...
(2,1)	(2,2)	(2,3)	...
(3,1)	(3,2)	(3,3)	...
...	...	...	...

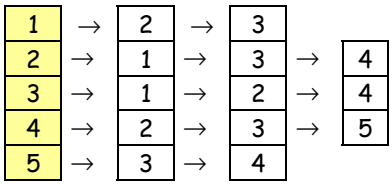
Listas de adjacências:

- Para cada vértice *u* temos uma lista linear Adj(u) implementada com ponteiros, que contém todos os vértices *v* tais que *(u,v)* é uma aresta (arco).
- Podemos supor que cada nó da lista pode ter um campo vert e um campo prox.
- O endereço do primeiro nó da lista é Adj(u).
- Se p é o endereço de um nó então vert(p) é o vértice armazenado no nó e prox(p) é o endereço do próximo nó da lista.
- Se p é o endereço do último nó da lista então prox(p) = NIL.

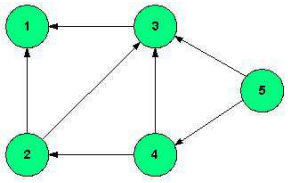
Exemplos de listas de adjacência:

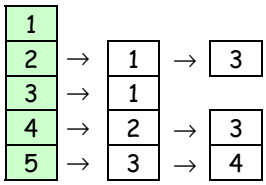
Grafo não orientado:





Grafo orientado:



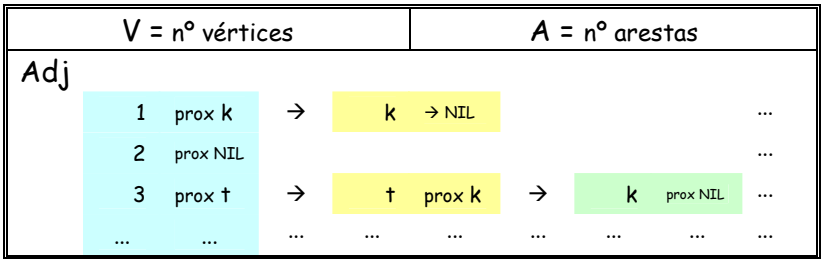


Representação computacional:

A estrutura de dados deve informar:

- O número de vértices;
- A quantidade de arestas (ou arcos);
- As listas de adjacências
- Valores e/ou atributos associados a cada aresta (arco)

Ilustrando a estrutura teríamos o Tipo Abstrato de Dado (TAD) grafos (ou dígrafos):



Caminhos

Um caminho em um grafo é uma seqüência de vértices em que cada dois vértices consecutivos são ligados por uma aresta: é uma seqüência $(v_0, v_1, ..., v_{k-1}, v_k)$ de vértices tal que, para todo *i*, o par (v_{i-1}, v_i) é uma aresta. Um caminho é *simples* se não tem vértices repetidos.

Território (ou domínio) de um vértice

Um vértice *x* é acessível a partir de um vértice *r* se existe um caminho de *r* a *x*. O território de um vértice *r* é o conjunto de todos os vértices acessíveis a partir de *r,* denotado por *T(r).*

problema: dado um vértice r de um grafo, encontrar o território de r.

A solução do problema será dada por um vetor "*cor*" indexado pelos vértices: para cada vértice *u*, teremos cor[u] = PRETO se e só se *u* está no território de *r*. Um algoritmo simples resolve o problema.

No início de cada iteração temos duas classes de vértices, a classe dos vértices PRETOs e a classe dos BRANCOs.

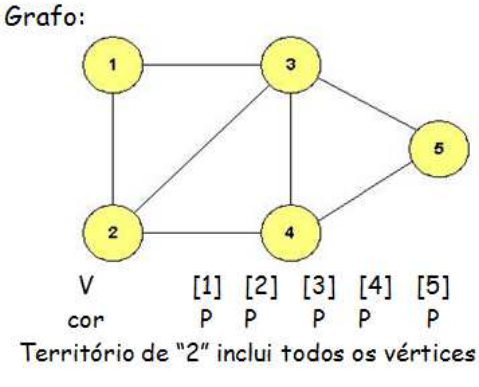
Todo vértice PRETO está no território de *r*. Cada iteração escolhe uma aresta que vai de um vértice PRETO a um vértice BRANCO e pinta esse último vértice de PRETO.

Para administrar o processo, convém introduzir uma terceira cor: CINZA.

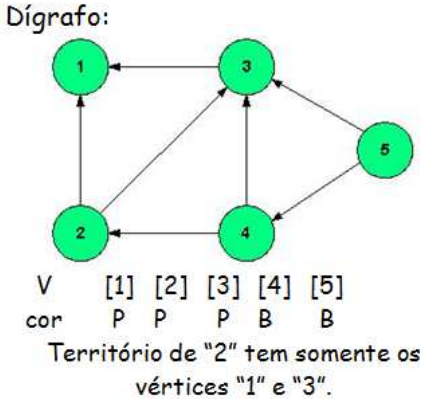
O algoritmo abaixo tem como parâmetro de entrada o grafo G (lista de adjacências Adj e o número de vértices n) e o vértice r e, supõe que os vértices são 1, 2, ..., n.

```
TERRITÓRIO(G,r)
1. para u ← 1 até n faça cor[u] ← BRANCO
2. cor[r] ← CINZA
3. enquanto existe u tal que cor[u] = CINZA faça
4.         para cada v em Adj[u] faça
5.                 se cor[v] = BRANCO
6.                         então cor[v] ← CINZA
7.         cor[u] ← PRETO
8. devolve cor
```

Exemplos de aplicação do algoritmo a partir do vértice "2" apresentam:



Território de "2" inclui todos os vértices



Território de "2" tem somente os vértices "1" e "3".

Busca em Largura

A busca em largura ou *pesquisa primeiro na extensão* – do inglês breadth-first search: BFS – é um dos algoritmos mais simples para se pesquisar um (dí)grafo possibilitando descobrir distâncias entre vértices.

O comprimento de um caminho é o número de arestas (arcos) do caminho:

- A distância de um vértice *x* a um vértice *y* é o comprimento de um caminho de comprimento mínimo que começa em *x* e termina em *y*.
- Naturalmente a distância de *x* a y pode não ser igual à distância de *y* a *x*.

A distância de *x* a *y* é *infinita* se não existe caminho algum. É importante ressaltar que a afirmação "a distância de *x* a *y* é k" equivale a duas afirmações:

(i) existe um caminho de *x* a *y* cujo comprimento é *k*,

(ii) não existe caminho de *x* a *y* cujo comprimento seja menor que *k.*

Problema: *Dado um vértice s de um grafo, encontrar a distância de s a cada um dos demais vértices.*

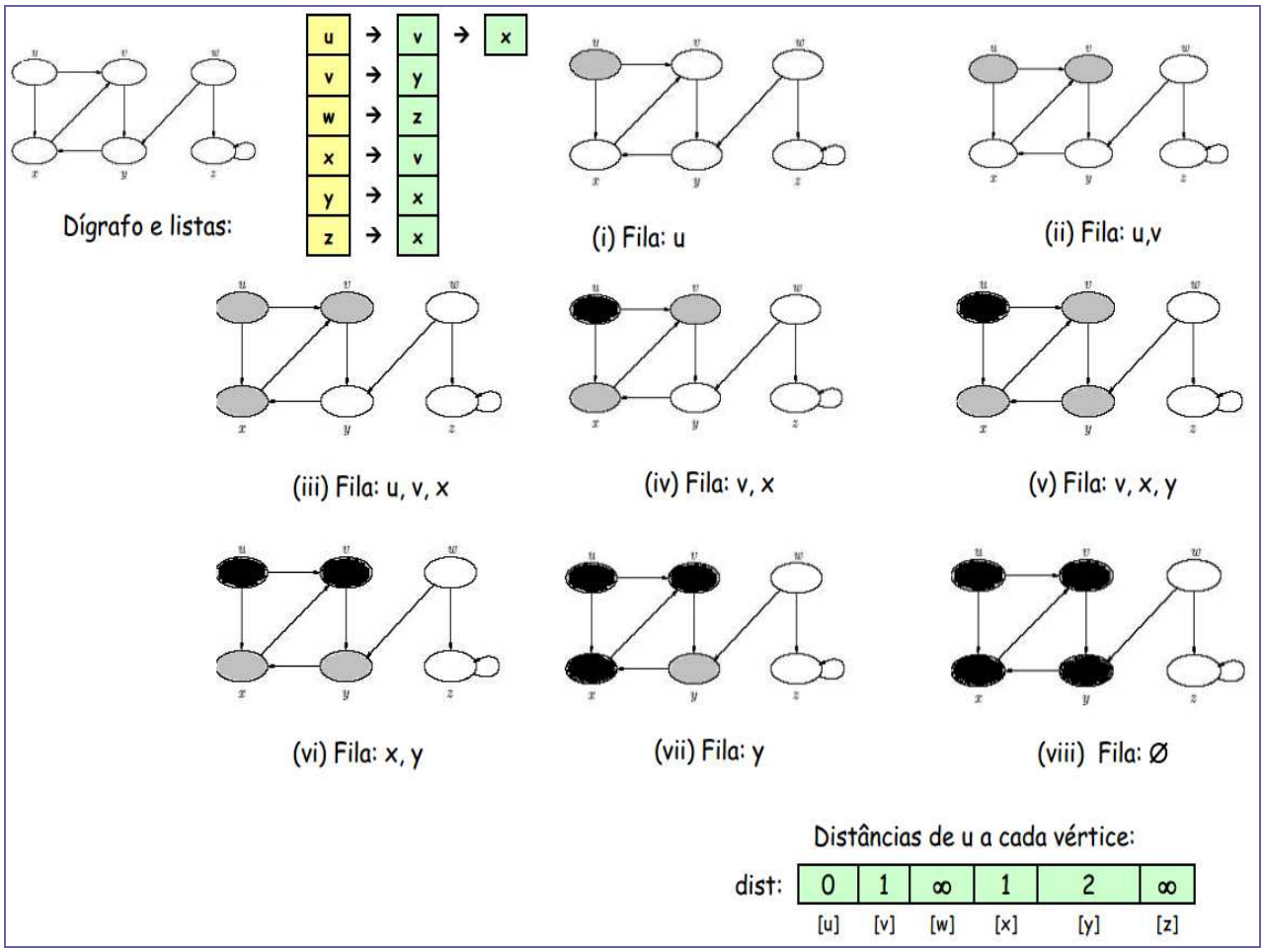
O algoritmo da busca em largura explora o grafo a partir de x:

- Todos os vértices são inicialmente identificados com a cor BRANCO.
- À medida que o algoritmo progride, um vértice pode se tornar CINZA e depois PRETO.
- Um vértice se torna CINZA quando é atingido pela primeira vez e permanece CINZA enquanto seus vizinhos não tiverem sido todos explorados. Porém é preciso que os vértices de cor CINZA sejam examinados *na mesma ordem em que foram encontrados*.
- Para administrar essa ordem, vamos guardar os vértices de cor CINZA em uma fila.

O algoritmo recebe um grafo G, armazenado em listas de adjacências e, um vértice *x*, devolvendo o vetor dist[1..n] onde para cada vértice *v*, o número *dist(v)* é a distância de *x* a *v*.

```
BFS(G,x)
 1.  para u ← 1 até n faça
 2.        cor[u] ← BRANCO
 3.        dist[u] ← ∞
 4.  cor[x] ← CINZA
 5.  dist[x] = 0
 6.  Q ← Inicializa-Fila(Q,x)
 7.   enquanto Q ≠ Ø faça
 8.        u ← Primeiro-da-Fila(Q)
 9.        para cada v em Adj[u] faça
10.             se cor[v] = BRANCO
11.                  então cor[v] ← CINZA
12.                      dist[v] ← dist[u]+1
13.                      Insira-na-Fila(Q,v)
14.        Remova-da-Fila(Q)
15.        cor[u] ← PRETO
16.  devolve dist[1..n]
```

Exemplo: abaixo temos o andamento do algoritmo BFS em um dígrafo começando a busca por "u"



Ao longo do algoritmo, *Q* é uma *fila de vértices* que é o segredo do funcionamento do algoritmo. O comando:

- Inicializa-Fila(Q,x) cria uma fila com um só elemento igual a *x,*
- Primeiro-da-Fila(Q) devolve o primeiro elemento da fila *Q* mas não retira esse elemento da fila,
- Insira-na-Fila(Q,v) insere *v* no fim da fila *Q* e,
- Remova-da-Fila(Q) remove o primeiro elemento.

O algoritmo funciona corretamente?

Para responder esta pergunta é preciso entender a situação no início de cada iteração do processo iterativo que começa na linha 7 de BFS. No início de cada iteração:

- todo vértice CINZA está em *Q* e todo vértice em *Q* é CINZA;
- para todo vértice *v* que seja CINZA ou PRETO, o valor de *d[v]* é finito e igual à distância de *x* a *v*;
- para todo vértice *v* que seja BRANCO, temos *d[v]* = ∞;
- não existe arco *(u,v)* tal que *u* é PRETO e *v* é BRANCO.

Portanto, quando a execução do algoritmo termina, o vetor *dist[1..n]* informa as distâncias corretas de *x* a cada um dos demais vértices.

Considere que o primeiro vértice de *Q* é q_1, o segundo é q_2, etc., o último é q_k e, ainda, que $d[q_1] = k$. Então,

1. $d[q_1] \leq d[q_2] \leq \ldots \leq d[q_r] \leq k+1$
2. se *x* é PRETO então $d[x] \leq k$

Eficiência do algoritmo

Quanto tempo o algoritmo consome no pior caso? Vamos medir esse tempo em relação a dois parâmetros: o número *n* de vértices e o número *m* de arestas.

O tempo gasto com as operações de inicialização nas linhas 1 a 3 não passa de *O(n).*

Convém não tentar estimar, em separado, o tempo gasto com *cada* execução do bloco de linhas 7-15. É melhor estimar o tempo *total* gasto com cada *tipo* de operação.

Cada vértice entra na fila no máximo uma vez e sai da fila no máximo uma vez. Portanto, o tempo total dedicado às operações de manipulação da fila (linhas 6, 8, 13 e 14) é *O(n).*

A lista de adjacências de cada vértice é percorrida (linha 9) no máximo uma vez. A soma dos comprimentos de todas as listas de adjacências é igual ao número de arestas do grafo.

Portanto, o tempo total gasto em todas as varreduras de listas de adjacências (linhas 9 a 13) é *O(m),* supondo *m* arestas (arcos).

Conclusão: o consumo de tempo de BFS é *O(n+m)* proporcional ao número de vértices e arestas.

Arborescência da busca em largura

Na busca em largura a partir de um dado vértice x obtêm-se uma árvore que reporta a seqüência de visitas no (dí)grafo. Essa árvore será definida como "arborescência" para reservar o termo "árvore" para outras aplicações.

Para montar a árvore, sempre que um vértice v é descoberto durante a busca na lista (matriz) de adjacências de outro vértice já visitado u, a busca em largura (BFS) memoriza este evento ao definir um predecessor de v como u, armazenado em um vetor de predecessores (pred[v] ← u).

O vetor predecessor (pred) da busca em largura apresentará a arborescência enraizada no vértice x.

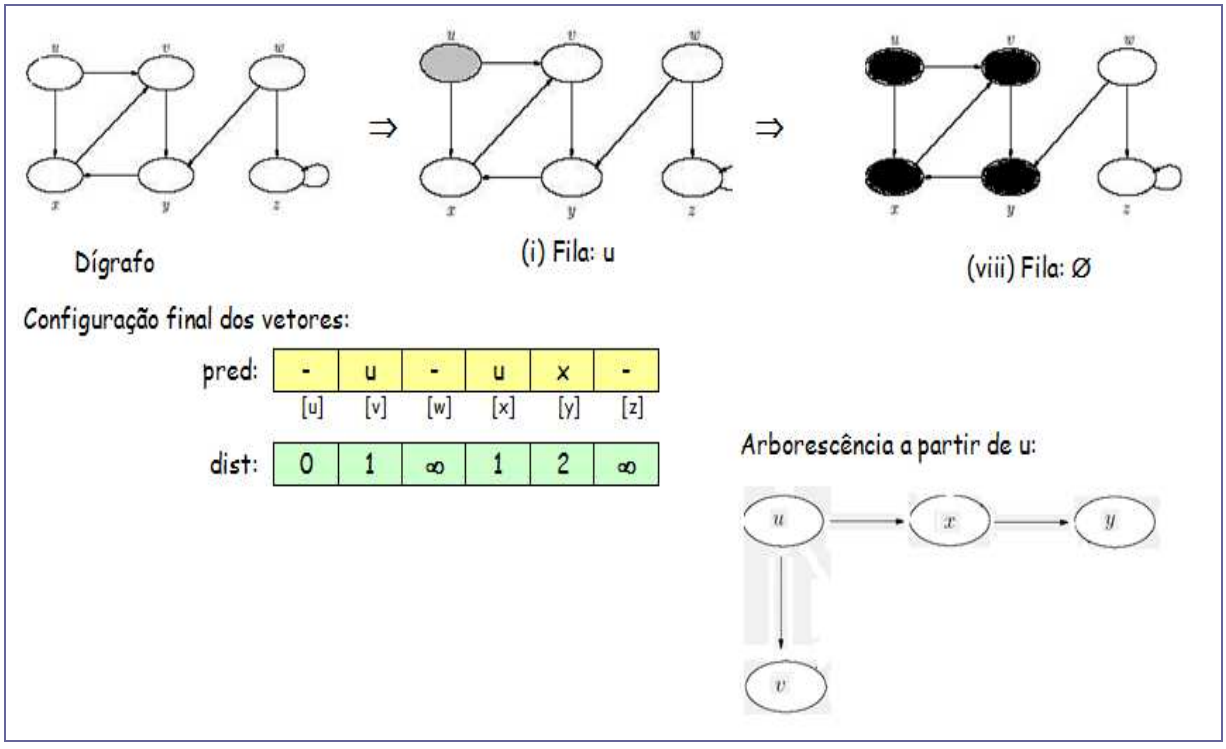
No início de cada iteração cada vértice v que está na fila é uma folha da arborescência representada por pred(v).

Vamos rever o algoritmo BFS introduzindo o vetor pred[1..n]:

```
BFS(G,x)
 1.  para u ← 1 até n faça
 2.        cor[u] ← BRANCO
 3.        dist[u] ← ∞
 4.        pred[u] ← NIL
 5.  cor[x] ← CINZA
 6.  dist[x] = 0
 7.  Q ← Inicializa-Fila(Q,x)
 8.   enquanto Q ≠ Ø faça
 9.        u ← Primeiro-da-Fila(Q)
10.        para cada v em Adj[u] faça
11.             se cor[v] = BRANCO
12.                  então cor[v] ← CINZA
13.                       dist[v] ← dist[u]+1
14.                       pred[v] ← u
15.                       Insira-na-Fila(Q,v)
16.        Remova-da-Fila(Q)
17.        cor[u] ← PRETO
18.  devolve dist[1..n], pred[1..n]
```

Na linha 14 do algoritmo BFS: a atribuição armazena os predecessores e, assim é construída uma arborescência da busca em largura.

Exemplo: abaixo temos o andamento do algoritmo BFS em um dígrafo começando a busca por "u"



Caminhos mínimos

Agora que pesquisamos as distâncias em um grafo, podemos pedir mais informações:

Problema: Dados um vértice x de um grafo, encontrar um caminho de comprimento mínimo de " x " a cada um dos demais vértices.

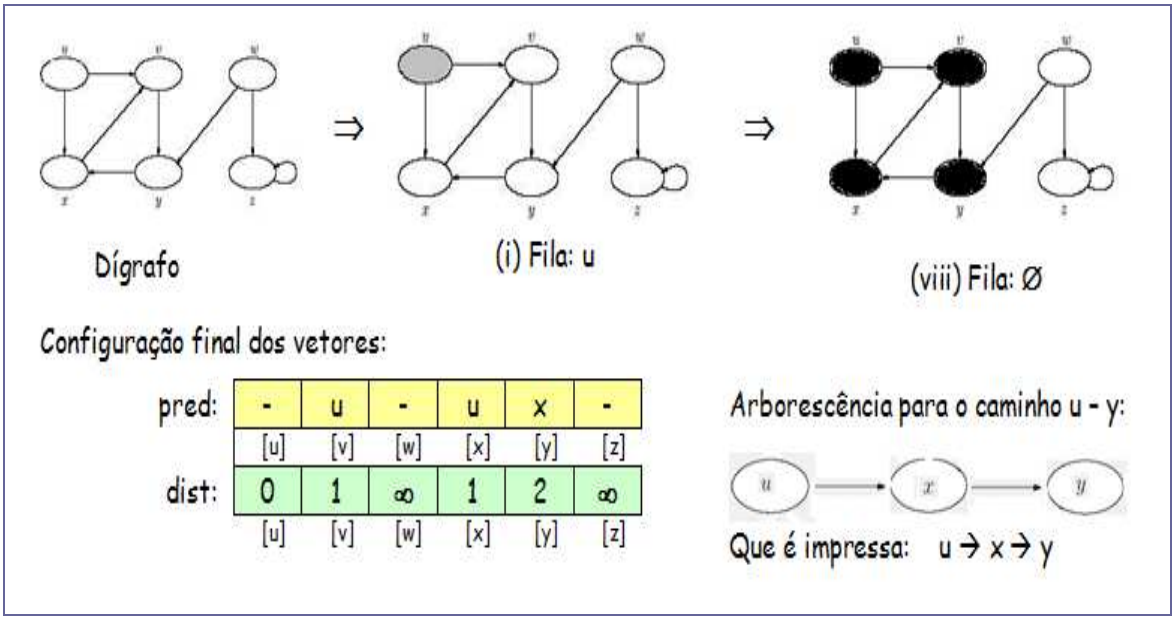
Como todos esses caminhos podem ser representados?

Resposta: por meio de uma *arborescência de caminhos mínimos* (ou seja, a árvore que reporta a seqüência de visitas no (dí)grafo na busca em largura representada por um vetor de predecessores, pred[1..n].

O procedimento abaixo imprime os vértices em um caminho mínimo, desde x até v, supondo que BFS já foi executado para calcular a arborescência de caminhos mínimos:

```
Caminho-Mínimo-BFS(G,x, v)
1.  se v = x
2.       então imprime x
3.       senão se pred(v) = NIL
4.                 então escreve " não há caminho "
5.                 senão Caminho-Mínimo-BFS(G, x, pred(v))
6.                       escreve "v → "
```

Exemplo: revendo o andamento do algoritmo BFS em um dígrafo começando a busca por "u"



Pesquisa primeiro na profundidade: a busca em profundidade

A busca em profundidade – do inglês depth-first search: DFS - é um algoritmo para caminhar no (dí)grafo. Na busca em profundidade, as arestas (arcos) são exploradas a partir do vértice visitado mais recentemente.

O algoritmo DFS recebe um (dí)grafo cujos vértices são coloridos durante a busca:

- BRANCO: antes da busca;
- CINZA: quando o vértice for visitado e,
- PRETO: quando os vértices adjacentes já foram visitados.

DFS emprega uma variável global "tempo", que marca as visitas em cada vértice em dois instantes:

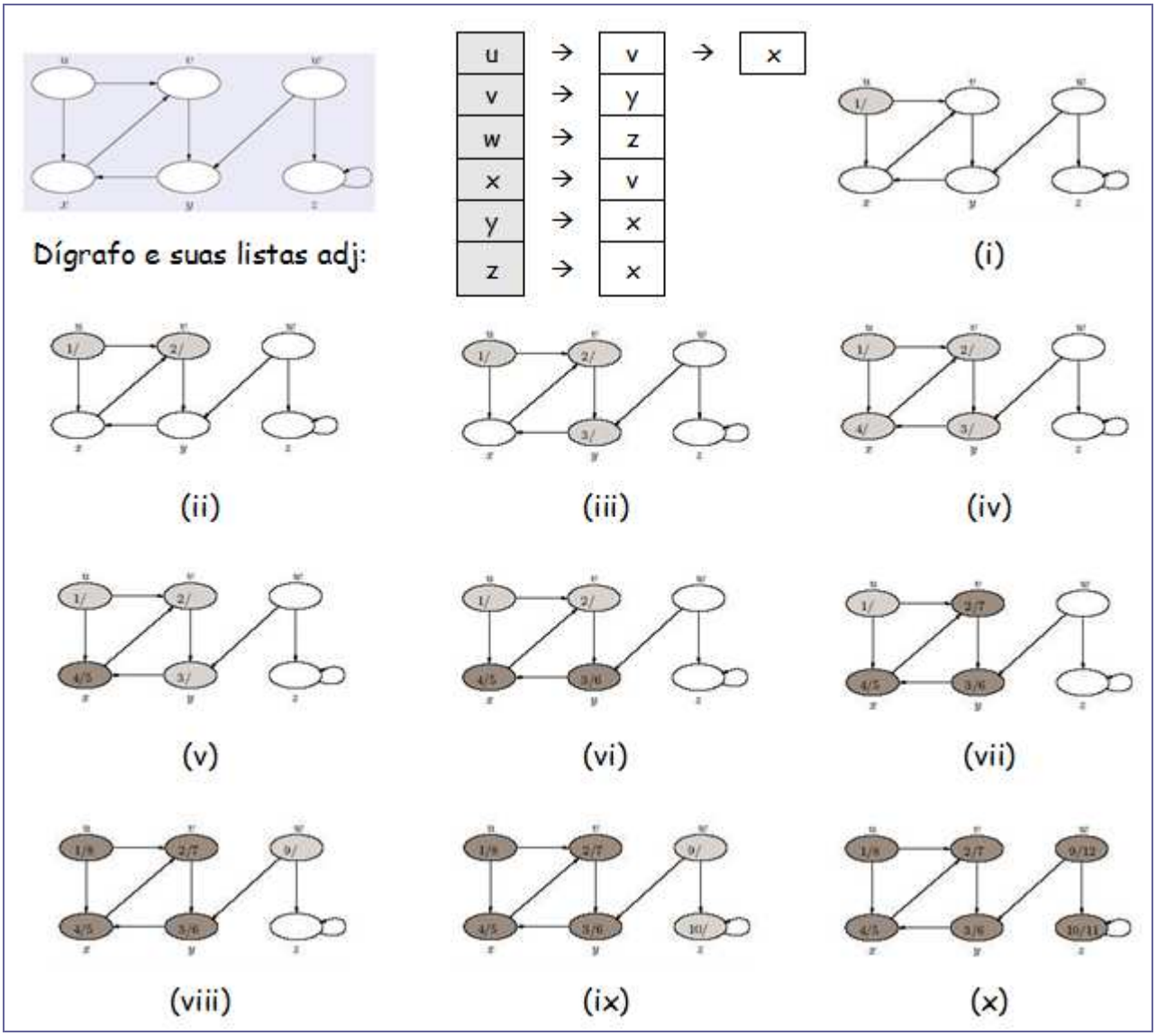
- d(v) ← tempo, no instante em que v foi visitado (pintado de CINZA),
- f(v) ← tempo, no instante em que a busca pelos vértices na lista de adjacências de v foi completada sendo então pintado de PRETO.

Algoritmo para uma busca completa em um (dí)grafo representado em listas de adjacências:

```
DFS(G)
1.  para cada vértice u em G faça
2.      cor[u] = BRANCO
3.  tempo ← 0
4.  para cada vértice u em G faça
5.          se cor(u) = branco
6.              então Visita_DFS(u)

Visita-DFS(G, u)
1.   cor[u] ← CINZA
2.  tempo ← tempo + 1
3.  d(u)  ← tempo
4.  para cada v em Adj(u) faça
5.          se cor[v] = BRANCO
6.                  então Visita_DFS(v)
7.  cor[u] ← PRETO
8.  tempo ← tempo + 1
9.  f(u)  ← tempo
```

No exemplo abaixo temos o andamento do algoritmo DFS em um dígrafo:



DFS inicia a busca realizada pelo procedimento recursivo Visita_DFS, pelo primeiro vértice da lista (em geral o menor valor, no exemplo o vértice "u"). Note que a execução do laço, linhas 4 a 6, de DFS garante a visita em todos os vértices do (dí)grafo.

Qual é o tempo de execução de DFS ?

Os laços das linhas 1-2 e 4-6 são executados em tempo proporcional ao número de vértices **n:** $\Theta(n)$, enquanto o procedimento Visita_DFS(v) é chamado exatamente uma vez para cada vértice de G, pois é chamado apenas para vértices de cor BRANCO e na primeira vez que isto acontece ele é pintado de CINZA.

Durante uma execução das linhas 4-7 do procedimento Visita_DFS(v) é executado |Adj(v)| vezes, podendo chegar a $\Theta(m)$ considerando o que temos m arestas no (dí)grafo.

Portanto, o tempo total de execução de DFS é $\Theta(n+m)$.

Propriedades da busca em profundidade

Na busca em profundidade de um grafo (direcionado ou não) G = (V, E), para quaisquer dois vértices **u** e **v** vale uma das três condições:

- os intervalos [d(u), f(u)] e [d(v), f(v)] são disjuntos e **u** não é descendente de v na floresta da busca em profundidade, por exemplo "v" e "w" no dígrafo acima;
- o intervalo [d(u), f(u)] está contido em [d(v), f(v)] e **u** é um descendente de **v** na floresta da busca em profundidade, no exemplo os vértices "x" e "y";
- o intervalo [d(v), f(v)] está contido em [d(u), f(u)] e **v** é um descendente de **u** na floresta da busca em profundidade, no exemplo os vértices "u" e "v".

Podemos classificar quatro tipos de arestas a partir do efeito da busca em profundidade:

- ***aresta de árvore***: 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), no exemplo anterior temos as arestas (u,v), (v,y), etc.
- ***aresta reversa***: são arestas (u,v) conectando um vértice u com um predecessor v em uma árvore de busca em profundidade, no exemplo a aresta (x,v) .
- ***aresta direta***: são arestas (u,v) que não pertencem à árvore de busca em profundidade, mas conectam um vértice u a um descendente v que pertence à árvore de busca em profundidade, no exemplo a aresta (u,x).
- ***aresta de cruzamento*:** são todas as outras arestas, que podem conectar vértices na mesma árvore de busca em profundidade ou em duas árvores de busca em profundidade, no exemplo a aresta (w,y).

Arborescência da busca em profundidade

Na busca em profundidade obtêm-se uma árvore que reporta a seqüência de visitas no (dí)grafo. Para montar a árvore, sempre que um vértice v é descoberto durante a busca na lista (matriz) de adjacências de outro vértice já visitado u, a busca em profundidade (DFS) memoriza este evento ao definir um predecessor de v como u, armazenado em um vetor de predecessores (pred[v] ← u). O vetor predecessor (pred) da busca em profundidade (DFS) pode ser composto de várias arborescências.

Definindo: Arborescência é um dígrafo em que

- não existem vértices com grau de entrada maior que 1;
- existe exatamente um vértice com grau de entrada 0;
- cada um dos vértices é término de um caminho com origem no único vértice que tem grau de entrada nulo.

O único vértice com grau de entrada nulo é a raiz da arborescência. Todos os vértices diferentes da raiz têm grau de entrada igual a 1.

Para obter a arborescência modificamos o algoritmo DFS incluindo o vetor pred[1..n]:

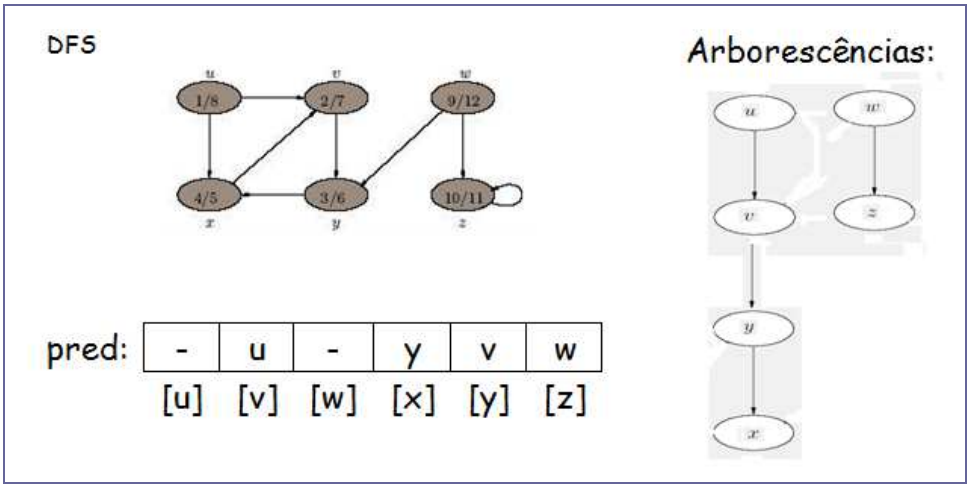
DFS(G)

```
1.  para cada vértice u em G faça
2.        cor[u] = BRANCO
3.        pred[u] ← NIL
4.  tempo ← 0
5.  para cada vértice u em G faça
6.           se cor(u) = BRANCO
7.                 então Visita_DFS(u)
8.  devolve pred[1..n]
```

Visita-DFS(G, u)

```
1.   cor[u] ← CINZA
2.  tempo ← tempo + 1
3.  d(u)  ← tempo
4.  para cada v em Adj(u) faça
5.           se cor[v] = BRANCO
6.                    então pred[v] ← u
7.                               Visita_DFS(v)
7.  cor[u] ← PRETO
8.  tempo ← tempo + 1
9.  f(u)  ← tempo
```

Para o dígrafo anterior obtemos as arborescências:

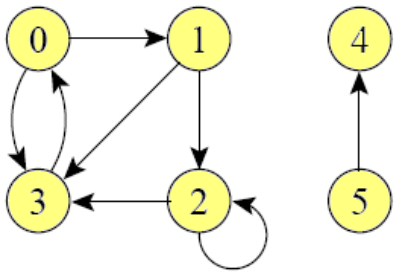


Aplicações empregando a Busca em Profundidade

Caminhos

Um caminho em um (dí)grafo é uma seqüência de vértices em que cada dois vértices consecutivos são ligados por uma aresta (arco): é uma seqüência $(v_0, v_1, \ldots, v_{k-1}, v_k)$ de vértices tal que, para todo *i,* o par (v_{i-1}, v_i) é uma aresta(arco). Um caminho é *simples* se não tem vértices repetidos. O comprimento de um *caminho* é o número de arestas (arcos) na seqüência de vértices.

Exemplo: no dígrafo abaixo, o caminho (0; 1; 2; 3) é simples e tem comprimento 3, já o caminho (1; 3; 0; 3) não é simples.



Um algoritmo de pesquisa de caminhos entre dois vértices num (dí)grafo, representado em listas de adjacência, empregando o formato da busca em profundidade:

```
Caminho(G, s,t)
1  para cada vétice u em G faça
2        cor[u] ← BRANCO
3  devolva Visita-Caminho(G, s, t).

Visita-Caminho(G, u, t)
1  se u = t
2     então devolve 1
3  cor[u] ← CINZA
4  para cada v em Adj(u) faça
5        se cor[v] = BRANCO
6             então se Visita-Caminho(G,v,t) = 1
7                        então devolva 1
8  devolva 0
```

Note que a aplicação do algoritmo no dígrafo acima sinalizaria que há caminhos entre 0 e 2, 0 e 3, 1 e 3 más também indicaria a não existência de caminhos entre 0,1,2 ou 3 e 4,5.

Componentes de grafos

Um conjunto *X* de vértices é *isolado* se não existe aresta alguma com uma ponta em *X* e outra fora. Há dois casos degenerados que vale a pena destacar: o conjunto de todos os vértices e o conjunto vazio são sempre isolados.

Um *componente* de um grafo é um conjunto isolado não vazio *mínimo:* digamos que A é o conjunto de *todos* os conjuntos isolados distintos de ∅.

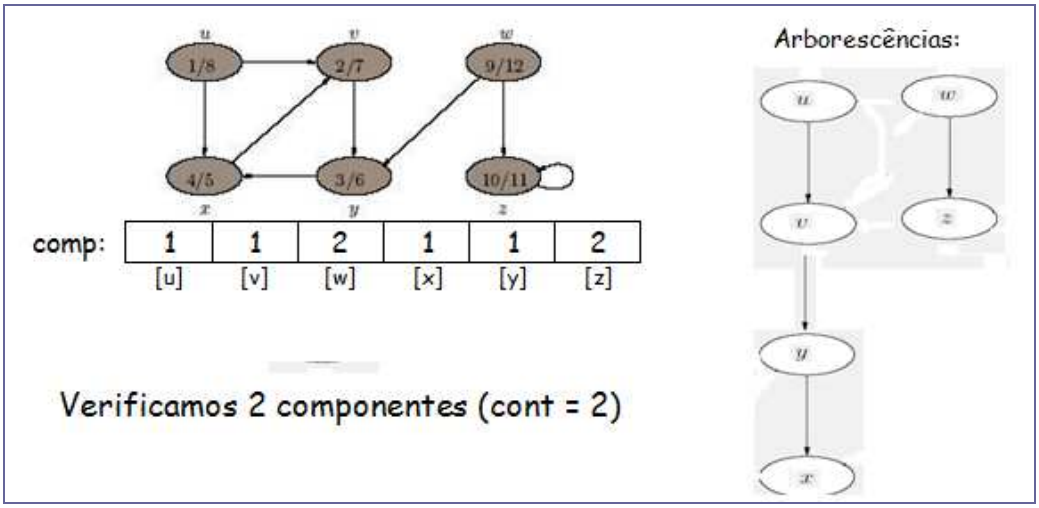
Um elemento *X* de A é *mínimo* se não contém algum outro elemento de A, portanto, um componente é um conjunto isolado que não inclui propriamente outro conjunto isolado, exceto o vazio.

Podemos empregar a busca em profundidade para determinar as componentes de um (dí)grafo, basta uma modificação no algoritmo DFS para contar as componentes:

Componentes-DFS(G)
1. para cada vértice u em G faça
2. comp[u] = 0
3. cont ← 0
4. para cada vértice u em G faça
5. se comp[u] = 0
6. então cont ← cont + 1
7. Visita-Componentes-DFS(G, u, cont)
8. devolve cont

Visita-Componentes-DFS(G, u, cont)
1. comp[u] ← cont
2. para cada v em Adj(u) faça
3. se comp[v] = 0
4. então Visita-Componentes-DFS(G, u, cont)

Exemplos: verificando as componentes nos exemplos anteriores



Ciclos

Um ciclo é um caminho $(x_0, x_1, ..., x_{k-1}, x_k)$ tal que $x_0, x_1, ..., x_{k-1}$ são distintos dois a dois más $x_k = x_0$, ou seja, é um caminho fechado de comprimento pelo menos 2, sem vértices repetidos, exceto o último (que coincide com o primeiro).

No dígrafo o ciclo é *orientado*: todos os arcos *v - w* "apontam pra frente" sendo *w* sucessor de *v*.

No grafo o ciclo é trivial se tem comprimento 2, pois usam os dois arcos de uma mesma aresta (grafo = dígrafo simétrico). Por essa razão os ciclos triviais são ignorados.

Um grafo é acíclico se não tem ciclo. Existem várias situações em que o teste para verificar se um grafo é acíclico é importante.

O algoritmo DFS pode ser usado para a tarefa: *verificar se o (dí)grafo é acíclico ou contém um ou mais ciclos*.

Se um ***arco reverso*** é encontrado durante a busca de profundidade, então ***o dígrafo tem ciclo.***

Um *sorvedouro* é um vértice do qual não saem arestas, ou seja, um vértice que não é ponta inicial de nenhuma aresta. Se um grafo não tem sorvedouro então certamente tem algum ciclo (embora a recíproca não seja verdadeira).

Algoritmos para pesquisar ciclos:

1 – pesquisa em dígrafos representados por listas de adjacências

```
Ciclos-em-dígrafos(G)
1.  para cada vértice u em G faça
2.          para cada v em Adj(u) faça
3.                  ciclo ← Caminho(G, v, u)
4.                  se ciclo = 1
5.                      então devolve 1
6.  devolve 0
```

Empregamos a função Caminho(G, x, y), apresentada na seção anterior, para verificar a existência de caminhos (0 = não, 1 = sim). A indicação da existência de ciclos também será sinalizada por 0 ou 1.

2 – pesquisa em grafos representados por listas de adjacência:

```
Ciclos-em-grafos(G)
1.  para cada vértice u em G faça
2.          para cada v em Adj(u) faça
3.                  se u < v
4.                      então  Remove-aresta(G, v, u)
5.                             ciclo ← Caminho(G, v, u)
6.                              Inclui-aresta(G, v, u)
7.                              se ciclo = 1
8.                                  então devolve 1
9.  devolve 0
```

Essa versão para grafos acrescenta a remoção e a re-inclusão de uma aresta *v-u* para verificar se existe um ciclo não trivial entre dois, empregando a mesma sinalização de existência ou não.

Se empregarmos diretamente a estratégia da busca em profundidade pode-se encontrar ciclos de uma maneira mais eficiente. Os algoritmos a seguir têm por base a seguinte observação: *em relação a qualquer floresta de busca em profundidade, todo arco de retorno pertence a um ciclo e todo ciclo tem um arco de retorno.*

1 – pesquisa mais eficiente em dígrafos representados por listas de adjacências

```
Ciclos-em-dígrafos-DFS(G)
 1. para cada vértice u em G faça
 2.       ord(u) ← 0
 3. tempo ← 0
 4. para cada vértice u em G faça
 5.          se ord(u) = 0
 6.             então  se Visita-Ciclos-em-dígrafos-DFS-d(G,u) = 1
 7.                          então devolve 1
 8. devolve 0.

Visita-ciclos-em-dígrafos-DFS(G,u)
 1. tempo ← tempo + 1
 2. ord(u)  ← tempo
 3. para cada v em Adj(u) faça
 4.          se ord[v] = 0
 5.             então se Visita-ciclos-em-dígrafos-DFS(G,v) = 1
 6.                          então devolve 1
 7.             senão se ord(v) < ord(u)
 8.                          então devolve 1
 9. devolve 0.
```

A indicação de ciclos será sinalizada por: 0 = dígrafo acíclico ou 1 = existe um ou mais ciclos.

2 – pesquisa eficiente de ciclos não – triviais em grafos representados por listas de adjacência:

```
Ciclos-em-grafos-DFS(G)
 1. para cada vértice u em G faça
 2.       ord(u) ← 0
 3. tempo ← 0
 4. para cada vértice u em G faça
 5.          se ord(u) = 0
 6.             então  pred(u) ← u
 7.                    se Visita-ciclos-em-grafos-DFS(G,u) = 1
 8.                          então devolve 1
 9. devolve 0.

Visita-Ciclos-em-grafos-DFS(G,u)
 1. tempo ← tempo + 1
 2. ord(u)  ← tempo
 3. para cada v em Adj(u) faça
 4.          se ord[v] = 0
 5.             então pred(v) ← u
 6.                   se Visita-Ciclos-em-Grafos-DFS(G,v) = 1
 7.                          então devolve 1
 8.             senão se ord(v) < ord(u)  E  v ≠  pred(u)
 9.                          então devolve 1
10. devolve 0.
```

Note que para evitar os ciclos triviais é utilizado o vetor de predecessores (pred[ ]) que dá acesso á floresta da busca em profundidade na linha 8.

Grafos bipartidos e ciclos ímpares

Um grafo é bipartido se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte.

Outra maneira de formular a definição: *um grafo é bipartido se for possível atribuir uma de duas cores a cada vértice de tal forma que as pontas de cada aresta tenham cores diferentes.*

Verifica-se facilmente que grafos que têm ciclos de comprimento ímpar não são bipartidos. É mais difícil provar que todo grafo "sem ciclos ímpares" admite bipartição.

Vejamos um algoritmo de pesquisa que prova desse fato, bem como a prova de sua correção e, é estruturado como uma busca em profundidade.

O algoritmo é aplicado a um grafo representado por listas de adjacências, e em sua pesquisa devolve 1 se o grafo é bipartido e devolve 0 em caso contrário. Além disso, se o grafo é bipartido, a função atribui uma "cor" a cada vértice, de tal forma que toda aresta tenha pontas de cores diferentes.

As cores dos vértices, representadas por 0 e 1, são registradas no vetor cor[ ] indexado pelos vértices:

```
Bipartido-DFS(G)
 1.  para cada vértice u em G faça
 2.    cor(u) ← -1
 3.  ref ← 0
 4.  para cada vértice u em G faça
 5.        se cor(u) = -1
 6.           então se Visita-Bipartido-DFS(G, u, 0) = 0
 7.                       então devolve 0
 8.  devolve 1.


Visita-Bipartido-DFS(G, u, ref)
 1.  cor(u)  ← 1 - ref
 2.  para cada v em Adj(u) faça
 3.        se cor[v] = -1
 4.           então se Visita-Bipartido-DFS(G, v, 1-ref) = 0
 5.                       então devolve 0
 6.           senão se cor(v) = (1 – ref)
 7.                       então devolve 0
 8. devolve 1.
```

Pontes e articulações em grafos

Consideremos agora os grafos que deixam de ser conexos quando perdem uma de suas arestas. Trataremos apenas de grafos, pois os conceitos a serem discutidos não fazem sentido em dígrafos não-simétricos.

Pontes e aresta-biconexão

Uma aresta de um grafo é uma ponte se ela é a única aresta que atravessa algum corte do grafo. Outra denominação: *pontes* são *arestas de corte*, mas não vamos usar essa terminologia.

Em resumo, *uma ponte é uma aresta cuja remoção aumenta o número de componentes do grafo.*

Problema a tratar: "Encontrar as pontes de um grafo dado."

Um grafo é aresta-biconexo se for conexo e não tiver pontes, portanto, é preciso remover pelo menos duas arestas que ele deixe de ser conexo.

Importante: Um grafo é aresta-biconexo se e somente se, para cada par *(x,y)* de seus vértices, existem (pelo menos) dois caminhos de *x* a *y* sem arestas em comum.

Uma componente aresta-biconexa de um grafo é qualquer componente do grafo que se obtém depois que todas as pontes são removidas.

Articulações e biconexão

Uma articulação ou vértice de corte de um grafo é um vértice cuja remoção aumenta o número de componentes.

Um grafo é biconexo se é conexo e não tem articulações, portanto, é preciso remover pelo menos 2 vértices de um grafo biconexo para que ele deixe de ser conexo.

Importante: Um grafo é biconexo se e somente se, para cada par *(x,y)* de seus vértices, existem (pelo menos) dois caminhos de *x* e y sem vértices internos em comum (ou seja, *x* e *y* são os únicos vértices comuns aos dois caminhos).

Pesquisa de Pontes

Com uma adaptação do algoritmo de busca em profundidade é possível encontrar as pontes de um grafo de maneira muito eficiente. O ponto de partida é fato que em qualquer grafo, uma aresta é uma ponte se e somente se não faz parte de um ciclo não - trivial. Ou seja, toda aresta é de um (e apenas um) de dois tipos: ou ela é uma ponte ou pertence a um ciclo não - trivial.

Se fizermos uma busca em profundidade no grafo, um dos dois arcos que constituem qualquer ponte será necessariamente um arco de arborescência.

Propriedade: em qualquer busca em profundidade, um arco *v*-*w* da floresta da Busca em Profundidade faz parte (juntamente com *w*-*v*) de uma ponte se e somente se não existe arco de retorno que ligue um descendente de *w* a um ancestral de *v*.

Obs.: os descendentes e ancestrais de que trata a propriedade não são necessariamente próprios. (ancestral próprio de um vértice w é qualquer ancestral de w exceto o próprio w)

Para empregar a propriedade, vamos calcular, para cada vértice *v*, o menor *número de pré - ordem* que pode ser alcançado a partir de *v*.

Suponha, para efeito desta explicação, que um caminho é *interessante* se começa em v, "desce" pela arborescência de busca em profundidade e finalmente percorre no máximo um arco de retorno.

Por exemplo: todo caminho de comprimento 0 que começa em v é interessante ou, um caminho de comprimento 1 que começa em v e percorre um só arco de retorno é interessante. Esse número será denotando esse número por "pré-ord(u)".

Considerando que armazenamos o tempo de descoberta de um vértice em ord(u), vamos supor que o *valor* de um caminho interessante é ord[x], sendo x o último vértice do caminho. Então pré-ord[x] é, por definição, o valor de um caminho interessante de valor mínimo.

É claro que pré-ord[x] ≤ ord[x] para todo vértice x. E para toda aresta *x-y* do grafo:

- se *x-y* é um arco de arborescência (e portanto x é pai de y) então pré-ord[x] ≤ pré-ord[y];
- se *x-y* é uma arco de retorno (e portanto pré-ord [x] > pré-ord [y]) então pré-ord [x] ≤ ord[y].

A propriedade pode ser assim reformulada: *em qualquer floresta de busca em profundidade de um grafo, um arco de arborescência x-y faz parte de uma ponte se e somente se pré-ord[y] = ord[y].*

Algoritmo para pesquisar pontes:

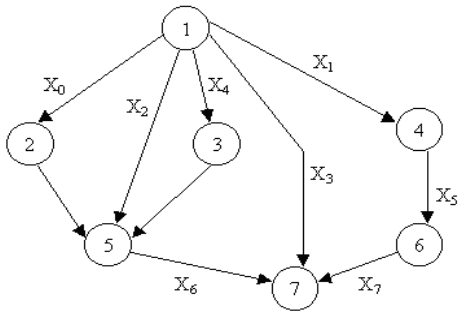
```
Pontes-DFS(G)
 1.  para cada vértice u em G faça
 2.        ord(u) ← 0
 3.  tempo ← conta-ponte ← 0
 4.  para cada vértice u em G faça
 5.         se ord(u) = 0
 6.            então  pred(u) ← u
 7.                   tempo ← tempo + 1
 8.                   Visita-Pontes-DFS(G, u, tempo, conta-ponte).

Visita-Pontes-DFS(G,u,tempo,conta-ponte)
 1.  pré-ord(u) ← ord(u)  ← tempo
 2.  para cada v em Adj(u) faça
 3.         se ord[v] = 0
 4.             então pred(v) ← u
 5.                   tempo ← tempo + 1
 6.                   Visita-Pontes-DFS(G, u, tempo, conta-ponte)
 7.                   se pré-ord(u) > pré-ord(v)
 8.                        então pré-ord(u) ← pré-ord(v)
 9.                   se pré-ord(v) = ord(v)
10.                        então conta-ponte ← conta-ponte + 1
11.                              escreve: "ponte: ", u,"–", v
12.            senão se v ≠ pred(u)  E  pré-ord(u) > ord(v)
13.                        então pré-ord(u) ← ord(v)
```

O teste da linha 12, "v ≠ pred(u)", garante que *u-v* é um arco de retorno (e não um arco - pai). A execução desse algoritmo consome tempo proporcional a |V| + |E|, ou seja, a quantidade de vértices e arestas. A variante dessa função para matrizes de adjacência consome tempo proporcional a $|V|^2$.

Dígrafos Acíclicos (DAG)

Um DAG (direct acyclic graph) é um dígrafo acíclico, ou seja, uma travessia em profundidade no dígrafo G não indica nenhum arco para trás. Exemplo de um DAG:



Dígrafo acíclico G

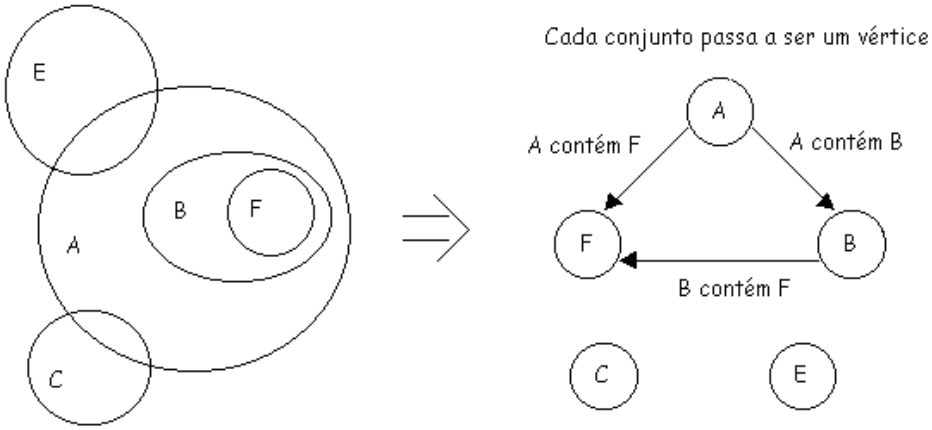
Se considerarmos uma árvore onde a raiz corresponde a um vértice só com saídas e as folhas a vértices só com entradas, podemos verificar que neste caso não estamos na presença de uma árvore porque um filho pode ter vários pais (exemplo: como acontece com o vértice 5 na figura).

Também não é exatamente um grafo pois não contém ciclos. Assim, podemos dizer que um DAG é qualquer coisa entre uma árvore e um grafo, que pode também, conter pesos (no exemplo: X_0, X_1, X_2, ...).

DAGs podem representar: operações de ordem parcial, redes de atividades, compiladores, pré-requisitos, jogos, etc.

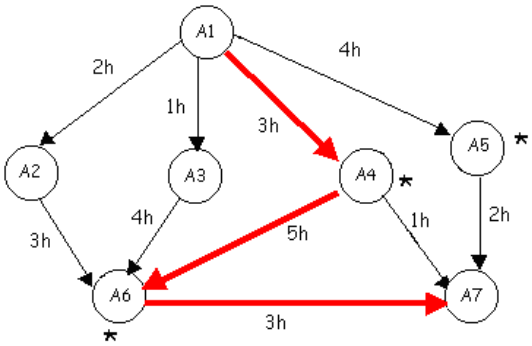
Vejamos alguns exemplos:

- *Operações de Ordem parcial:* exemplo de inclusão:



- *Redes de Atividades*: as redes de atividades não podem ter ciclos, pois o que se pretende é criar uma rede onde cada tarefa deve ser executada até ao fim. Por essa razão utilizam-se os DAG para criar uma rede de atividades. Vejamos um exemplo para compreender do que se trata.

Exemplo: temos uma rede de atividades num processo industrial que pode ser representada na seguinte forma:



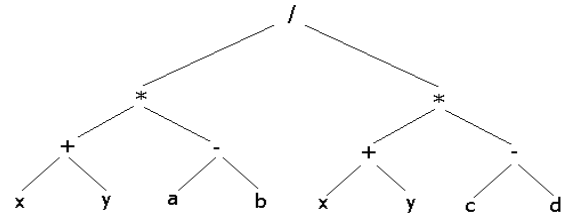
Estamos agora na presença de um DAG, que representa um processo industrial onde cada atividade é representada por um vértice (A1, A2,...,A7), e os pesos correspondem ao tempo necessário para executar uma tarefa (2h, 1h, 4h,...).
Desta forma, não é possível começar uma nova tarefa sem que a tarefa que lhe precede esteja terminada.
Por exemplo, se quisermos chegar à tarefa A7 precisamos executar uma das 3 entradas sendo o tempo necessário para executá-la de 3h. Cabe observar que para uma fase ser ativa, precisaremos sempre do peso máximo e não do peso mínimo.
Como podemos observar no DAG anterior, o peso máximo para chegar à atividade A7 é 11 (caminho a vermelho), ou seja, é a soma dos pesos desde a fonte até à folha, ao qual chamamos de *caminho crítico*.

- *Compiladores*: seja a expressão matemática:

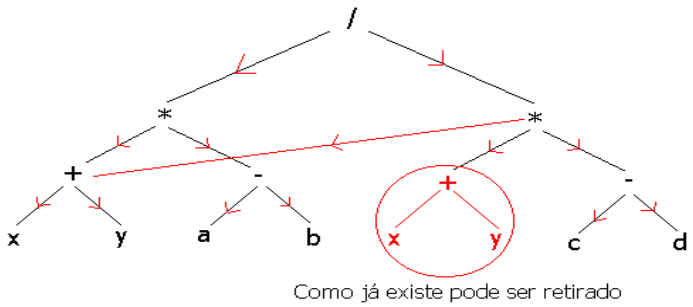
(((x + y)*(a - b))/((x + y)*(c - d))), para processar.

O compilador, dentro de um ciclo, não constrói a expressão inteira a cada iteração. Ao invés disso, constrói uma árvore onde a cada iteração mudam apenas as folhas, ou seja, cria uma árvore de avaliação para todos os valores.

A função prefixa da árvore ficará na forma: **/ * * + x y - a b + x y – c d**
O que dará origem à seguinte árvore:



O DAG pode ser utilizado para otimizar esta árvore, buscando o eixo do fluxo máximo:



Ordenação Topológica

Os DAGs podem ser ordenados utilizando a topologia associada: a ordenação topológica.

A ordenação topológica consiste em ordenar os vértices de um grafo acíclico G=(V,E) de forma que, se existe um caminho do vértice v para o vértice u, então v aparece antes de u na ordenação. De notar que uma ordenação topológica não é única e que ela não é possível se o grafo possuir ciclos.

Um exemplo comum de problemas de ordenação topológica, é a confecção de dicionários, onde desejamos que uma palavra B cuja definição dependa da palavra A, apareça depois de A no dicionário.

Uma forma simples de resolver esse problema é a utilização de uma versão do algoritmo de busca em profundidade que imprime o vértice antes de retornar a chamada. Obteremos assim os vértices em ordem topológica invertida:

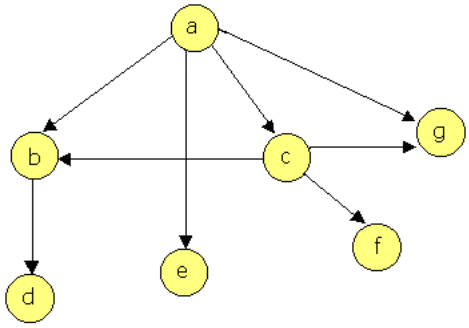
```
OrdTopol-DFS(G)
1.  para cada vértice u em G faça
2.          cor[u] ← BRANCO
3.   topo(u) ← -1
4.  k ← |V|              (nº vértices de G)
5.  para cada vértice u em G faça
6.          se cor(u) = BRANCO
7.              então Visita-OrdTopol-DFS(G, u, topo, k)
8.  devolve topo[1 .. |V|]


Visita-OrdTopol-DFS(G, u, topo, k)
1.   cor[u] ← PRETO
2.  para cada v em Adj(u) faça
3.          se cor[v] = BRANCO
4.                  então  Visita-OrdTopol-DFS(G, v, topo, k)
5.  topo(k) ←  u
6.   k ← k – 1
7.  escreve u
```

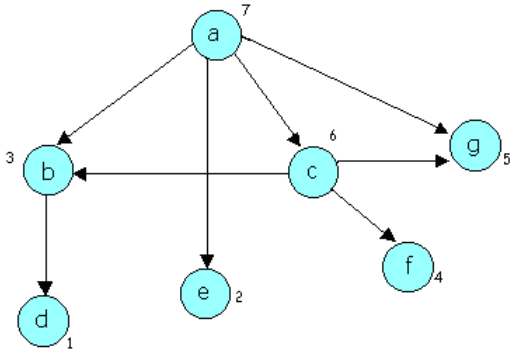
Para verificar que esse algoritmo funciona, é só ver o que acontece quando chega ao final de uma recursão, no momento que ele imprime o vértice. Nesse caso, ele já foi o mais longe possível a partir do vértice inicial (essa é uma característica do algoritmo de busca em profundidade).

Seja *x* o último vértice desse caminho. Não existe vértice *v* tal que $x < v$, porque se for o caso o algoritmo continuaria a busca no mínimo até *v*. Portanto, de todos os próximos vértices que serão impressos, nenhum sucede a *v* na ordem, e obteremos uma ordem topológica invertida.

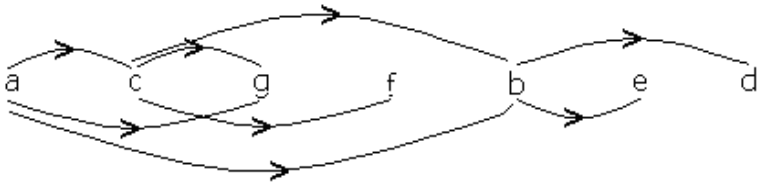
Vejamos um exemplo de ordenação topológica com seguinte DAG:



A execução do algoritmo de busca em profundidade nos dará a ordem de visita de cada vértice, considere que a lista de adjacências deste DAG resulte em:



O objetivo é re - escrever o DAG de forma linear, executando o algoritmo OrdTopo vamos obter os arcos todos na mesma direção:



E o resultado fica registrado no vetor "topo" que apresenta no exemplo o vértice "a" na 1ª posição e o vértice "d" na última.

Há que ter em atenção que a ordenação topológica não é única.

Existe mais de uma forma de escrever um dígrafo por ordem topológica.

No entanto, para realizarmos o algoritmo vamos ter que usar o DFS, sendo o resultado armazenado numa lista.

Outra alternativa para obter uma ordenação topológica é o algoritmo que trabalha com a informação do grau de entrada em cada vértice, denominado algoritmo da eliminação das fontes:

```
OrdTopo-elimfontes(G)
 1.  para cada vértice u em G faça
 2.  grau-entrada[u] ← 0
 3.  para cada vértice u em G faça
 4.        para cada v em Adj(u) faça     (mapeia o grau de entrada de u)
 5.              grau-entrada[v] ← grau-entrada[v] + 1
 6.  CriaFila(Q)
 7.  para cada vértice u em G faça  (somente fontes são inseridas na fila)
 8.       se grau-entrada = 0
 9.          então InsereFila(Q,u)
10.  i ← 1
11.  enquanto Q ≠ ∅ faça
12.        u ← TiraFila(Q)
13.        topo[i] ← u
14.         i ← i + 1
15.         para cada v em Adj(u) faça
16.               grau-entrada[v] ← grau-entrada[v] - 1
17.               se grau-entrada[v] = 0
18.                  então  InsereFila(Q,v)
19.   devolve topo[1 .. n]
```

Busca em profundidade - DFS X busca em largura - BFS

- A diferença marcante entre as buscas BFS e DFS está na estrutura de dados auxiliar empregada por cada uma: BFS usa fila enquanto a DFS usa a pilha (é implícita na versão recursiva);
- A busca DFS visita, tipicamente, todos os vértices do (dí)grafo, enquanto a BFS visita apenas os vértices que podem ser alcançados a partir do vértice inicial.
- A busca DFS é descrita, usualmente, em estilo recursivo enquanto a BFS é descrita em estilo iterativo.
- Espaço de memória e tempo de execução: DFS armazena todos os nós visitados a partir da raiz, ou seja, no máximo o tamanho do (dí)grafo, já a busca BFS armazena todos os possíveis nós processados, desta forma, em termos de espaço e tempo, a DFS pode ser mais eficiente.
- O fato é que, apesar da semelhança entre a siglas, a busca DFS e a busca BFS são muito diferentes e têm aplicações muito diferentes.

Exercícios

1. Considere o seguinte algoritmo que calcula o "grau de saída" *g(u)* de cada vértice *u*. O algoritmo supõe que os vértices do grafo são *1, 2, ..., n*

```
Algorit-exercício-2(n,Adj)
1. para u ← 1 até n faça g[u] ← 0
2. para u ← 1 até n faça g[u] ← 0
3.       para v em Adj[u] faça g[u] ← g[u]+1
4. devolva g[u]
```

a) Mostre que esse algoritmo consome *O(n+nm)* unidades de tempo, onde *m* é o número de arestas.

b) Melhore esta estimativa, mostrando que o algoritmo consome tempo proporcional a *O(n+nm).*

2. Escreva um algoritmo que determine o número de componentes de um grafo.

3. Escreva um algoritmo que decida se um grafo *simétrico* é ou não fortemente conexo. Que coisa o algoritmo pode devolver para comprovar que o grafo é fortemente conexo? Que coisa pode devolver para comprovar que o grafo é desconexo?

4. Escreva um algoritmo que verifica se uma dada ordenação dos vértices é ou não topológica.

5. Prove que todo grafo acíclico admite uma ordem topológica dos vértices.

6. Suponha que os vértices de nosso grafo são 1,2,...,n. Suponha também que a seqüência 1,2,...,n é uma ordem topológica. Que aparência tem a matriz de adjacência do grafo? (A matriz de adjacência, digamos *A*, é definida assim: para cada par *u,v* de vértices, *A[u,v]=1* se *(u,v)* é um arco e *A[u,u]=0* em caso contrário.)

7. Escreva uma versão do algoritmo de ordenação topológica supondo que o grafo tem vértices 1,2,...,n e é representado por sua matriz de adjacência *A* (por definição, para cada par *u,v* de vértices, *A[u,v]=1* se *(u,v)* é um arco e *A[u,u]=0* em caso contrário). Estime o consumo de tempo do algoritmo.

8. utilizando conceitos da teoria de grafos: dadas as seguintes 15 peças de dominó: {1,2}, {1,3}, {1,4}, {1,5}, {1,6}, {2,3}, {2,4}, {2,5}, {2,6}, {3,4}, {3,5}, {3,6}, {4,5}, {4,6}, {5,6}, pergunta-se se é possível enfileirar **todas** estas peças de forma a fechar um ciclo. A notação {a,b} indica que a peça contém os números **a** e **b**.

Árvores geradoras

Suponha que *x* é um vértice de um grafo, uma *árvore geradora* com raiz *x* é um conjunto *A* de arestas que seja mínimo com relação à seguinte propriedade:

Para todo vértice v no território de x, existe um caminho de x a v em A.

A expressão:

- "geradora" diz que *todo* vértice do território de *x* pode ser atingido em *A* a partir de *x.*
- "em *A*" significa que todas as arestas do caminho pertencem a *A*.
- a condição "mínimo" (ou seja, para toda aresta *a* em *A*, o conjunto A - *{a}* não tem a propriedade) na definição de árvore geradora é fundamental.

Uma conseqüência da condição é que, para cada *v* no território de *x*, existe *apenas um* caminho de *x* a *v*.

Se o grafo é *simétrico*, então o território de *x* coincide com o componente que contém *x*.

Note que qualquer dos vértices do componente é raiz de uma árvore geradora que "cobre" o componente.

Vetor de predecessores

Qualquer árvore geradora com raiz *x* pode ser representada por um "vetor de predecessores": pred. Para cada vértice *v* do território de *x*, pred[v] é o *predecessor* de *v* no único caminho que leva de *x* a *v* na árvore.

O conjunto de arestas da árvore será simplesmente o conjunto de todas as arestas (pred(v),v) com *v* diferente de *x.* É claro que pred é um vetor *indexado por vértices com valores que também são vértices,* ou seja, pred[v] também pode ter o valor especial NIL, que indica ausência de predecessor.

Por exemplo, se x = 1 e a árvore tem um caminho (1,2,3,4,5) então pred terá a aparência:

...	1	2	3	4	5	...
	NIL	1	2	3	4	

Para obter um caminho mínimo que termina em *v* basta construir a seqüência:

v, predecessor de v, predecessor do predecessor de v, até chegar a x.

Depois, é só imprimir a seqüência do final para o começo:

```
Imprime_Caminho(pred, x, v)
1. se pred[v] ≠ NIL
2.     então enquanto v ≠ NIL faça
3.                imprime v
4.                v ← pred[v]
```

Obs.: se não existe caminho algum de *x* a *v* o algoritmo não imprime nada.

Construção de uma árvore geradora

Problema: *dado um grafo e um vértice x, encontre uma árvore geradora com raiz x.*

A solução emprega um algoritmo que no início de cada iteração tem duas classes de vértices: vértices PRETOs e vértices BRANCOs, e uma árvore com raiz *x* que "cobre" o conjunto dos vértices PRETOs.

Cada iteração *acrescenta à árvore algum arco que vai de um vértice* PRETO *a um vértice* BRANCO e em seguida pinta esse último vértice de PRETO.

Vamos supor que os vértices do grafo são *1,2,...,n*. Para administrar o processo, convém introduzir uma terceira cor: CINZA.

```
Árvore(G, x)
 1. para u ←1 até n faça
 2.       cor[u] ← BRANCO
 3.       pred[u] ← NIL
 4. cor[x] ← CINZA
 5. enquanto existe u tal que cor[u] = CINZA faça
 6.          para cada v em Adj[u] faça
 7.                   se cor[v] = BRANCO
 8.                       então cor[v] ← CINZA
 9.                            pred[v] ← u
10.          cor[u] ← PRETO
11. devolve pred
```

O algoritmo devolve um vetor pred que representa uma árvore geradora com raiz *x,* considerando que pred[v] = NIL para todo *v* fora do território de *r*.

Qual o tamanho da árvore que o algoritmo produz, supondo que o grafo tem um só componente? → a árvore tem n-1 arestas.

Prova: no fim da linha 4, a árvore tem 0 arestas e "cobre" 1 vértice. A cada passagem pelas linhas 8-9, a árvore ganha uma aresta, *(u,v)*, e passa a "cobrir" um novo vértice (o vértice *v*).

Conclusão: no início de cada nova execução do bloco de linhas 6-10 nossa árvore tem n-1 arestas e "cobre" n vértices.

Mostra-se que *toda e qualquer* árvore geradora com raiz *x* tem n-1 arestas, onde *n* é o número de vértices do território de *x*. De fato, graças à liberdade de escolha do vértice *u* na linha 5, o algoritmo Árvore pode montar *qualquer* árvore geradora.

Árvores geradoras de peso mínimo (Minimum - Spanning Trees)

Imagine que cada aresta *(u,v)* de um *grafo* tem um **peso** numérico *w(u,v)*, que pode ser positivo, negativo ou nulo. O **peso** de um conjunto *E* de arestas é a soma dos pesos das arestas em *E*; o peso de *E* será denotado por *w(E)*.

Onde esses pesos ficam armazenados? → Os valores de *w* ficam dentro das listas de adjacência: na lista *Adj[u]*, junto com a célula que contém um vértice *v*, fica também o número *w(u,v)* e, na matriz de adjacências fica na célula (u,v) que indica a existência da aresta quando não é nula.

> Problema: *dado um grafo, um vértice x, e uma função w que atribui um peso a cada arco, encontrar uma árvore geradora com raiz x que tenha peso mínimo.* (Obs.: o peso de uma árvore é a soma dos pesos de suas arestas.)

No grafo qualquer árvore geradora com raiz *x* "cobre" todos os vértices do componente que contém *x*, ou seja, para cada vértice *v* do componente existe um (e um só) caminho de *x* a *v* na árvore. Se existe uma árvore geradora com raiz *x* e peso *P* então, para qualquer vértice *v* no componente que contém *x*, existe uma árvore geradora de peso *P* e raiz *v*.

Algoritmo de Prim

O algoritmo de Prim usa uma "fila" *de vértices*, com prioridade ditada por uma chave a ser discutida à frente. No início de cada iteração temos dois tipos de vértices, os PRETOs e os BRANCOs, e uma árvore com raiz *x* que "cobre" o conjunto dos vértices PRETOs.

Há também arestas da árvore ligando vértices PRETOs a vértices BRANCOs, mas essas arestas são provisórias. Cada iteração acrescenta à árvore a aresta mais leve dentre as que ligam um vértice PRETO a um vértice BRANCO. Vamos supor que os parâmetros de entrada são o grafo *G* (número de vértices, lista de adjacências e pesos) cujos vértices são identificados por 1,2,...,n.

```
MST_Prim(G, x)
 1.  para u ← 1 até n faça
 2.        cor[u] ← BRANCO
 3.        pred[u] ← NIL
 4.        chave[u] ← ∞
 5.  Q ← Crie-Fila-Vazia()
 6.  para u ← 1 até n faça
 7.        Insira-na-Fila(u,Q)
 8.  chave[x] ← 0
 9.  pred[x] ← x
10.  enquanto Q ≠ Ø faça
11.        u ← Retire-Mínimo(Q)
12.        para cada v em Adj[u]faça
13.              se cor[v] = BRANCO e w(u,v) < chave[v]
14.                  então chave[v] ← w(u,v)
15.                        Refaça-Fila(v,Q)
16.                        pred[v]←u
17.        cor[u] ← PRETO
18.  devolve pred
```

O comando Crie-Fila-Vazia( )cria uma "fila" vazia com prioridades ditadas por *chave*. Na implementação tudo se passa como se os vértices estivessem nessa fila *sempre em ordem crescente* da *chave*. O comando Retire-Mínimo(Q) retira de *Q* um vértice com *chave* mínima. A alteração do valor de *chave[v]* na linha 14 pode afetar a estrutura da fila; a função Refaça-Fila( ), conserta as coisas.

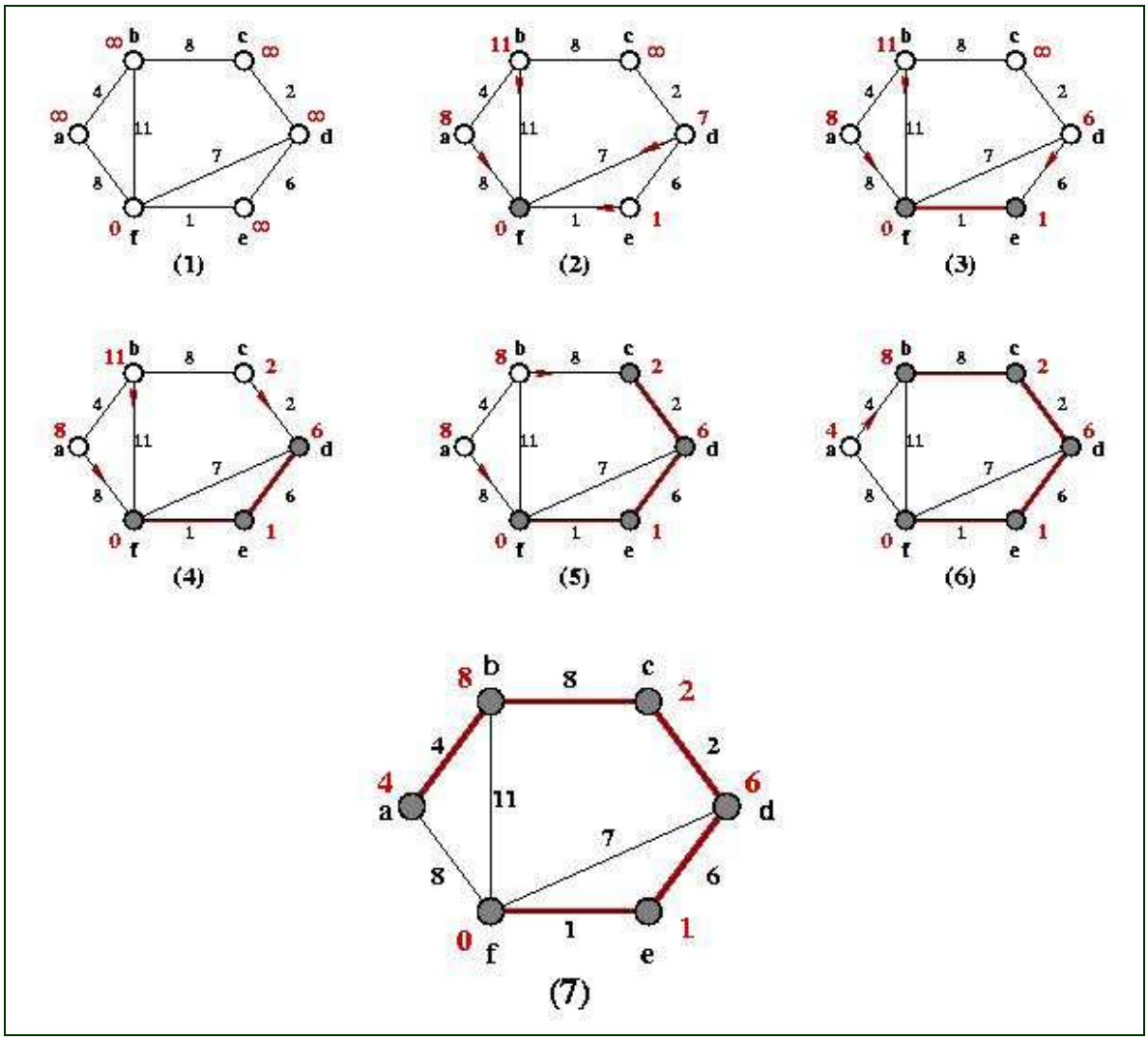
No início de cada iteração:

1. a cor de todo vértice em *Q* é BRANCO e todo vértice de cor BRANCO está em *Q*;
2. para todo vértice *x*, *pred[x] = 0* se e só se *chave[x] = ∞*;
3. para todo vértice *x*, se *pred[x]* não é *0*, então:
    a. *chave[x] = w(pred[x], x);*
    b. a cor de *pred[x]* é PRETO;
    c. o arco *(pred[x], x)* tem peso mínimo dentre todos os arestas que vão de um vértice de cor PRETO a *x*.

A figura a seguir mostra um exemplo de execução do algoritmo MST_Prim. Os números em vermelho (ao lado dos vértices nomeados por a, b, c, ..., f) representam os valores das chaves, enquanto pequenas setas vermelhas indicam os predecessores dos vértices. Vértices que estão fora da fila ou que possuem predecessores "NIL" não têm setas vermelhas. Cada passo na figura pode ser considerado como sendo uma "foto" do grafo exatamente no momento do teste do **enquanto** na linha 10.

Assim, o passo 1 representa o grafo tal como ele se encontra logo após a execução da linha 9, com todos os vértices com chave igual a infinito, com exceção de "f", que é a raiz, e todos os pais iguais a "NIL". O passo 2 mostra o resultado da primeira iteração do laço das linhas 10-17. Nele, os vizinhos da raiz já têm um valor de chave diferente de infinito e o campo pred apontando para a raiz.

Note que enquanto um vértice não sai da fila de prioridade, seu pai não é fixo: o pai do vértice *d* muda entre os passos 2 e 3. Quando um nó é extraído da fila, muda a cor para PRETO e seu pai passa a ser definitivo.



Exemplo de execução do algoritmo de Prim

Algoritmo de Kruskal

O algoritmo de Kruskal começa com uma floresta de árvores formada por apenas um vértice e procura, a cada passo, uma aresta que, além de conectar duas árvores distintas da floresta, possua peso mínimo.

Suponha que, numa determinada iteração, o algoritmo escolheu a aresta (u,v) para ser inserida no conjunto A e que a aresta (u,v) conecte a árvore C_1 à árvore C_2. Note que, dentre todas as arestas que conectam duas componentes distintas nesta iteração, (u,v) é uma das que possui o menor peso, pois ela foi escolhida assim.

Logo, nesta iteração, para qualquer corte que separe u de v, (u,v) é uma aresta leve (simplesmente pelo fato de não haver outra aresta com peso menor separando duas componentes nesta iteração). Assim, (u,v) certamente é uma aresta leve que conecta C_1 a uma outra componente conexa (no caso, C_2) da floresta determinada por A, o que significa que (u,v), pelo corolário 23.2, é uma aresta segura para A.

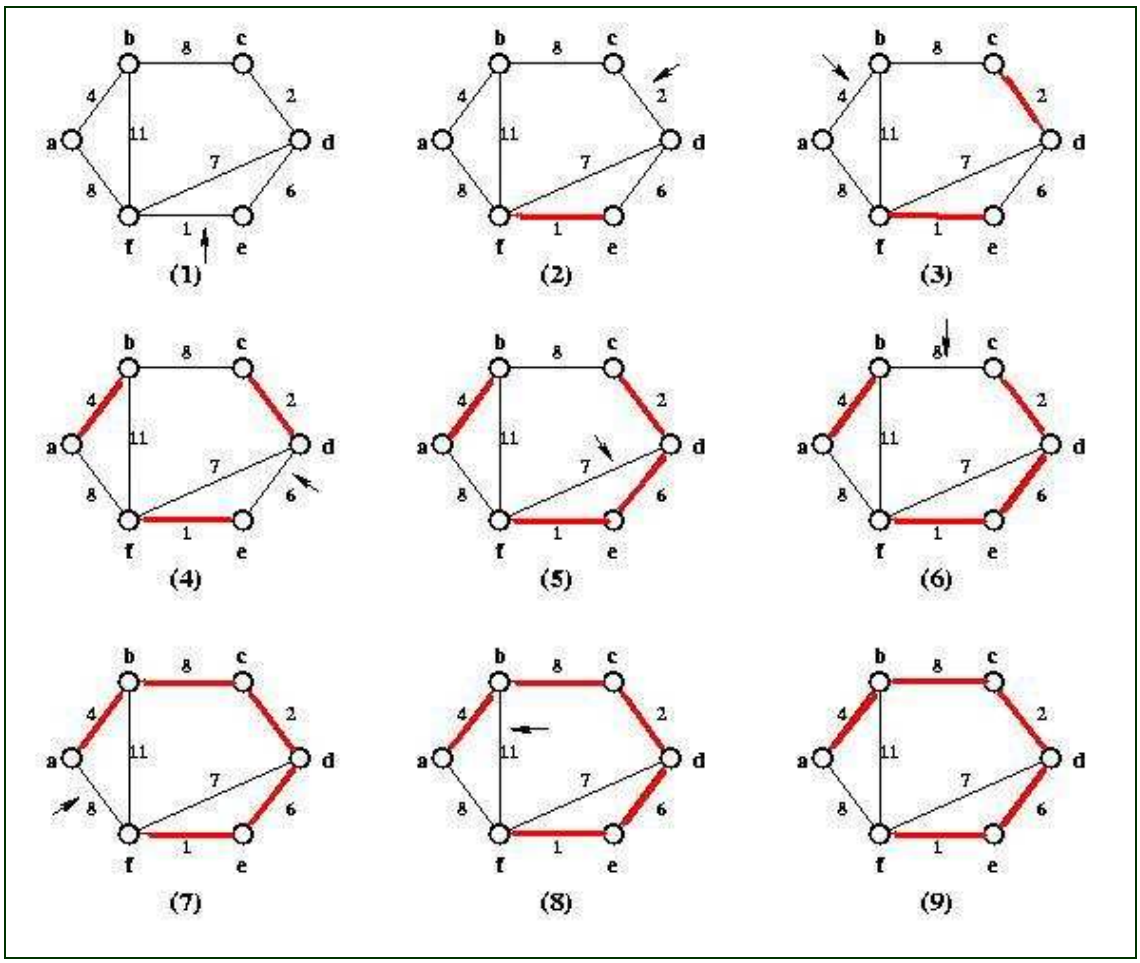
A implementação do algoritmo de Kruskal utiliza uma estrutura de dados de conjuntos disjuntos para testar se dois vértices pertencem a uma mesma componente conexa, ou não, no procedimento Conjunto(). A princípio, o conjunto de arestas é ordenado em ordem crescente de peso. As arestas são verificadas uma a uma, em ordem, e o algoritmo executa uma chamada da função Conjunto( ) para cada vértice adjacente a elas.

Se os vértices não pertencerem ao mesmo conjunto, o que significa que a aresta une vértices de componentes distintas, a aresta é inserida em A e os conjuntos de u e v são unidos com uma chamada do procedimento Une_Arvore. Vejamos uma versão do algoritmo de Kruskal :

```
MST_Kruskal(G,w)
1.  A ← ∅
2.  para u ← 1 até n faça Construa_Conjunto(u)
3.  Ordene(E)                    {arestas de E em ordem crescente de peso (w)}
4.  para cada aresta (u,v) faça  {tomadas em ordem crescente de peso (w)}
5.        se Conjunto(u) ≠ Conjunto(v)
6.              então A ← A ∪ {(u,v)}
7.                    Une_Árvores(u,v)
8.  devolve A
```

A figura a seguir exemplifica uma execução do algoritmo de Kruskal.

Note que as arestas são "visitadas" em ordem crescente de peso (no caso: 1, 2, 4, 6, 7, 8, 8 e 11) e, para que uma aresta seja inserida no conjunto A (aqui representado pelas arestas vermelhas), os seus vértices adjacentes devem pertencer a componentes diferentes.



Exemplo de execução do algoritmo de Kruskal

Note que nos passos 5, 7 e 8, as arestas verificadas acabam não sendo inseridas no conjunto *A*.

Na figura, em cada passo, a aresta indicada é analisada (linha 6 do algoritmo) e o fato de uma aresta ser inserida ou não só é percebido no passo seguinte.

Podemos entender os passos 1 a 8 na figura como "fotos" do grafo exatamente no momento em que o teste da linha 6 é executado.

A "foto" 9 foi tirada depois do término do algoritmo.

Algoritmo de Boruvka

Algoritmo também conhecido como Baruvka, a idéia geral desse algoritmo consiste em começar por ligar cada vértice ao seu vizinho mais próximo, criando, no máximo, |V|/2 árvores. Depois, ligar cada árvore à outra árvore que lhe estiver mais próxima, até que todos os vértices do grafo estejam inseridos na árvore.

O algoritmo de Boruvka compreende os seguintes passos:

1. Para cada vértice escolher o seu arco com peso mínimo. Deste passo poderão resultar vários subgrafos.
2. Caso o passo 1 dê origem a grafos não conectados, considere cada subgrafo gerado no passo anterior como um vértice do grafo final. Para cada um dos subgrafos gerados execute-se de novo o passo 1.
3. Parar quando todos os vértices estiverem conectados entre si formando assim a MST.

```
Boruvka(G)
1.  Seja G' uma floresta inicial com n sub-árvores T_j,   j = 1,...,n de G
2.  enquanto (G' não for uma árvore) {
3.      para (cada T_j ∈ G') {
4.          ache a menor aresta (u, v) incidente em T_j tal que u ∈ T_j e v ∉ T_j
5.          G' = G' + (u, v)
6.      }
7.  }
8.  retorne G'
```

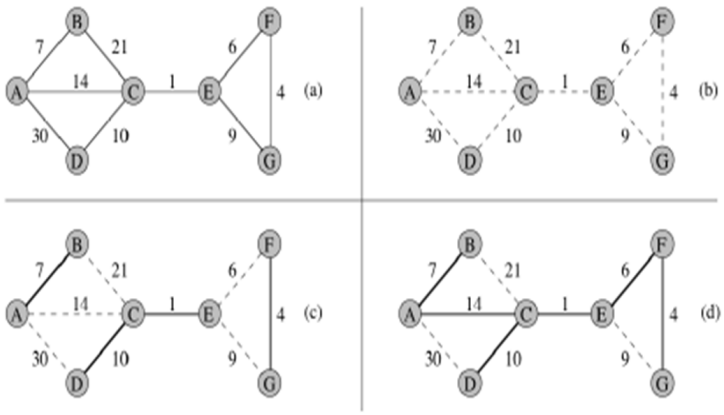
A idéia básica do algoritmo é contrair simultâneamente as arestas de peso mínimo incidente em cada vértice de G.

A contração pode criar múltiplas arestas entre alguns pares de vértice, entretanto, somente a aresta de peso mínimo do conjunto de múltiplas arestas precisa ser representada;

A cada iteração o número de vértices do grafo remanescente é reduzido pelo menos em um fator de 2:

- cada aresta incide em 2 vértices diferentes;
- o número de arestas marcadas por iteração é no mínimo $n/2$;

Vejamos uma figura que ilustra o funcionamento:



Dado o grafo G em (a), começamos em (b) a construção de G', montando a floresta com subárvores isoladas. Em (c) temos as ligações A-B, C – E (o menor peso entre as arestas), D-C e F-G.Finalizando em (d) temos a junção das subárvores completando a MST G'.

Exercícios

1 - Encontre uma árvore geradora mínima (MST) no grafo a seguir, descrito por 12 arestas e respectivos pesos, representados por (u-v, peso): (0-6, 51) (0-1, 32) (0-2, 29) (4-3, 34) (5-3, 18) (7-4, 46) (5-4, 40) (0-5, 60) (6-4, 51) (7-0, 31) (7-6, 25) (7-1, 21)

2 - Supondo que os custos das arestas de um grafo são dois a dois distintos (ou seja, não há duas arestas com o mesmo custo). Mostre que o grafo tem uma única MST.

3 - Considere o grafo cujos vértices são pontos no plano:

Vértices	0	1	2	3	4	5
Coordenadas	(1,3)	(2,1)	(6,5)	(3,4)	(3,7)	(5,3)

Suponha que as arestas do grafo são: 3-5 5-2 3-4 5-1 0-3 0-4 4-2 2-3; e o custo de cada aresta u-v é igual ao comprimento do segmento de reta que liga os pontos v e w no plano. Exiba uma MST desse grafo.

4 - Considerando que os custos das arestas de um grafo conexo são distintos dois a dois. É verdade que, em todo ciclo não-trivial, a aresta de custo mínimo pertence à (única) árvore geradora mínima do grafo?

5 - Suponha dado um grafo com pesos nos arestas. Suponha que um vetor de predecessores *pred* representa uma árvore geradora de peso *P*. Escreva um algoritmo que receba *pred* e um vértice *s* e devolva o vetor de predecessores de uma árvore geradora com raiz *s* e peso *P*.

6 - Até que ponto o consumo de tempo do algoritmo de Prim depende do primeiro vértice escolhido pelo algoritmo? Analise ao algoritmo e responda: *vale a pena escolher esse vértice aleatoriamente?Por que?*

7 – Suponha que todos os pesos de arestas em um grafo sejam inteiros no intervalo 1 a |V|. Com que rapidez é possível executar o algoritmo de Kruskal? E se os pesos de arestas forem inteiros no intervalo de 1 a W para alguma constante W?

8 - Escreva uma implementação do algoritmo de Boruvka que começa por colocar as arestas do grafo em ordem crescente de custos.

Caminhos Mínimos

Imaginemos a seguinte situação: um motorista deseja encontrar o caminho, mais curto possível, entre as cidades de Medina/MG e Bom Jesus/SC.

Caso ele receba um mapa das estradas de rodagem do Brasil, no qual a distância entre cada par adjacente de cidades está exposta, como poderíamos determinar uma rota mais curta entre as cidades desejadas?

Uma maneira possível é enumerar todas as rotas possíveis que levam de Medina à Bom Jesus, e então selecionar a menor. É fácil notar que até mesmo se deixarmos de lado as rotas que contêm ciclos, haverá milhões de possibilidades, a maioria das quais simplesmente não valerá a pena considerar, como uma rota que inclua ir de Medina à Brasília e daí para Bom Jesus ampliando a distância a percorrer em centenas de quilômetros.

Vamos modelar o problema em um dígrafo ponderado onde as ligações entre cidades são arcos com pesos que representam distâncias, conservação, pedágios, etc., expressos em um custo para transitar entre elas.

Estudando o problema dos *caminhos de custo mínimo* podemos chegar a um resultado eficiente e com menor esforço!

Caminhos de custo mínimo

Em um problema de caminhos de custo mínimo, ou "caminhos mais curtos" (shortest paths), consideramos um dígrafo G = (V, E), com uma função peso w(u,v): E → R, que mapeia arestas em pesos correspondentes.

O peso do caminho $p = (v_0, v_1, \ldots, v_k)$ corresponde à soma dos pesos das arestas que o constituem:

$$w(p) = \Sigma_i w(v_{i-1}, v_i)$$

Definimos o caminho de menor peso entre u e v por:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\Rightarrow} v\} & \text{Se existir uma rota de } u \text{ para } v \\ \infty & \text{Caso contrário} \end{cases}$$

Um caminho menor entre os vértices *u* e *v* é então definido como qualquer rota *p* com um peso $w(p) = \delta(u,v)$.

O *problema do menor caminho de fonte única,* denominação usual na literatura, consiste em determinar um menor caminho entre um vértice de origem $s \in V$ e todos os vértices de *V*.

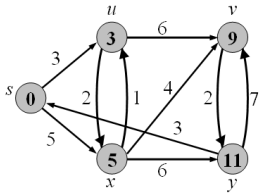
Existem algumas variantes deste problema, são elas:

- Menor caminho com destino único: encontrar um caminho mais curto para um vértice destino *v*
- Menor caminho para um par: encontrar um caminho mais curto para um determinado par de vértices *u* e *v*
- Menor caminho para todos os pares: encontrar um caminho mais curto de *u* para *v*, para todos e quaisquer pares *u* e *v*

Em algumas instâncias do *problema de menor caminho com uma única origem*, podem existir arestas cujos pesos possuem ***valor negativo***.

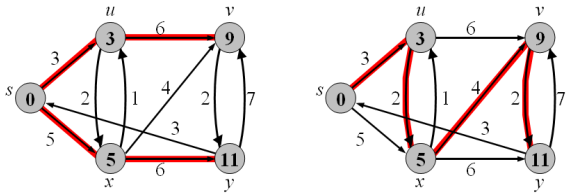
Se o dígrafo G = (V, E) não tiver ciclo com ***peso negativo*** alcançável a partir da fonte s, então o peso do caminho δ(s, v) permanece bem definido para todo v ∈ V, mesmo tendo o valor negativo. Se o dígrafo contiver algum ciclo negativo acessível a partir de s, os pesos de caminhos mais curtos não são bem definidos, pois nenhum caminho desde s poderá ser o mais curto já que sempre haverá outro "mais curto" depois de atravessar o ciclo. Em geral os algoritmos que *buscam caminhos mínimos* em dígrafos que contêm arcos com pesos negativos, detectam e relatam a existência de tais ciclos.

Exemplo de como achar o *caminho mais curto* em um dígrafo, que só tem pesos positivos nos arcos:



Como poderíamos obter a árvore de caminhamentos mínimos partindo da origem s?

Dois possíveis resultados:



Algumas definições:

- Em cada vértice *v*, mantemos um atributo *d[v]*, que é o limite superior do peso de um menor caminho da origem *s* até o vértice *v*, por exemplo, d[s] = 0, d[y] = 11.
- Chamamos *d[v]* de estimativa de menor caminho
- Iniciamos as estimativas de menor caminho e predecessores através do seguinte procedimento:

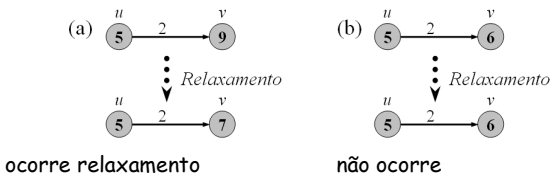
```
IniciaOrigemÚnica(G,s)
1  para cada vértice v em G faça
2        d[v] ← ∞
3        pred[v] ← NIL
4  d[s] ← 0
```

O algoritmo consiste em testar se é possível identificar um menor caminho para um vértice *v* passando pelo vértice *u* (processo de relaxamento de uma aresta (u,v)), e caso afirmativo atualizar *d[v]* e *pred[v].*

O passo de relaxamento pode reduzir a estimativa de menor caminho, e atualizar o predecessor de um determinado vértice, o procedimento a seguir realiza o relaxamento:

```
Relaxa(u,v,w)
1  se d[v] > d[u] + w(u,v)
2     então  d[v] ← d[u] + w(u,v)
3            pred[v] ← u
```

Exemplos:



ocorre relaxamento não ocorre

Algoritmo de Dijkstra

A experiência mostra que não adianta preocupar-se apenas com os caminhos de *s* a um *determinado* vértice u: é preciso considerar também os caminhos que vão de *s a cada um dos demais vértices* do grafo. Ou seja, é preciso encontrar uma *árvore geradora* com raiz s que tenha a propriedade: para cada vértice $v \in V$, o único caminho de *s* a *v* na árvore é um caminho de peso mínimo (dentre os que começam em *s* e terminam em *v*).

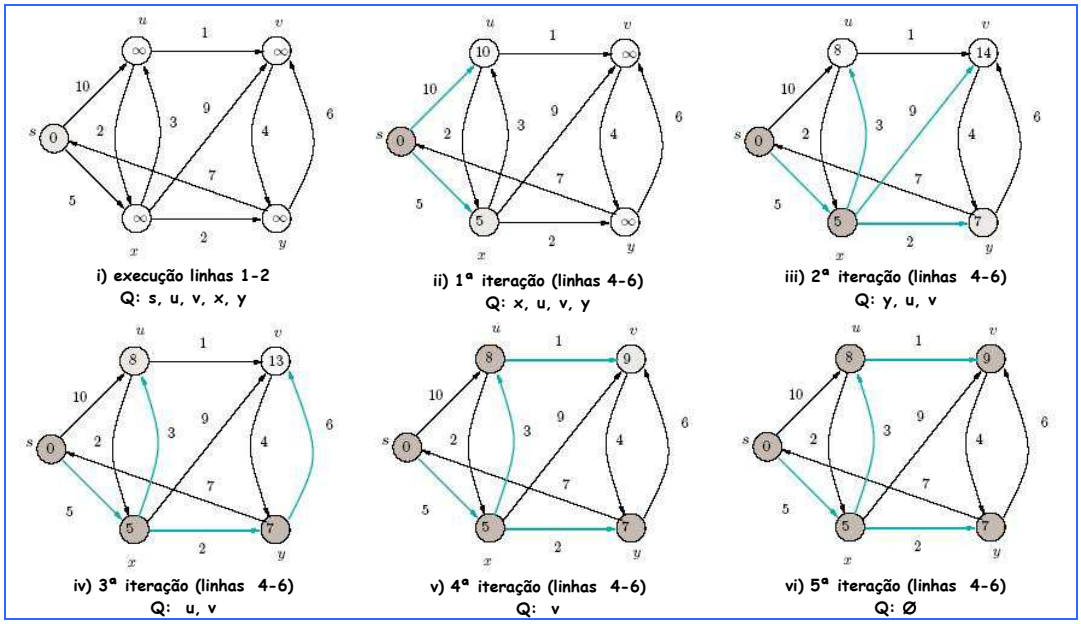
Vemos a seguir o algoritmo de Dijkstra que resolve o *problema de caminhos mínimos com uma única fonte* para um dígrafo G = (V,E), quando não há arcos de peso negativo, ou seja, $w(u, v) \geq 0$ para qualquer arco $(u, v) \in E$.

Este algoritmo mantém um conjunto S de vértices, através de uma fila Q, onde o menor caminho entre eles e a origem s já foi determinado, ou seja, para todos os vértices $v \in S$, temos d[v] = d(s,v). O algoritmo repetidamente escolhe um vértice $u \in V - S$, com o menor caminho estimado, e insere u em S. O algoritmo recebe um dígrafo *G* com vértices 1,2,...,n e arestas definidas por listas de adjacências *Adj*, recebe também um vértice *s* e um peso não-negativo *w(u,v)* para cada arco *(u,v)*.

```
Dijsktra(G, w, s)
1. IniciaOrigemÚnica(G,s)
2. Q ← Cria-Fila( )
3. enquanto Q ≠ Ø faça
4.       u ← Retire-Mínimo(Q)
5.       para cada v em Adj[u]faça
6.          Relaxa(u,v,w)
7. devolve pred
```

O comando **Crie-Fila()** cria uma "fila" com todos os vértices. A prioridade dos vértices na fila é dada pelo vetor *d*. O comando **Retire-Mínimo**(Q) retira de *Q* um vértice *u* para o qual *d[u]* é mínimo. A alteração do valor de d[v] no procedimento **Relaxa( )** pode afetar a estrutura da fila; isso deve ser levado em conta quando a fila for implementada.

As figuras a seguir demonstram a execução do algoritmo em um dígrafo onde a origem é o vértice "s" mais à esquerda. As estimativas de caminhos mais curtos são mostradas dentro dos vértices, e a origem dos arcos verdes indica os predecessores.



Algoritmo de Bellman - Ford

O algoritmo *Bellman-Ford* resolve o *problema do menor caminho com uma única origem* de uma forma mais genérica, incluindo arestas com peso negativo.

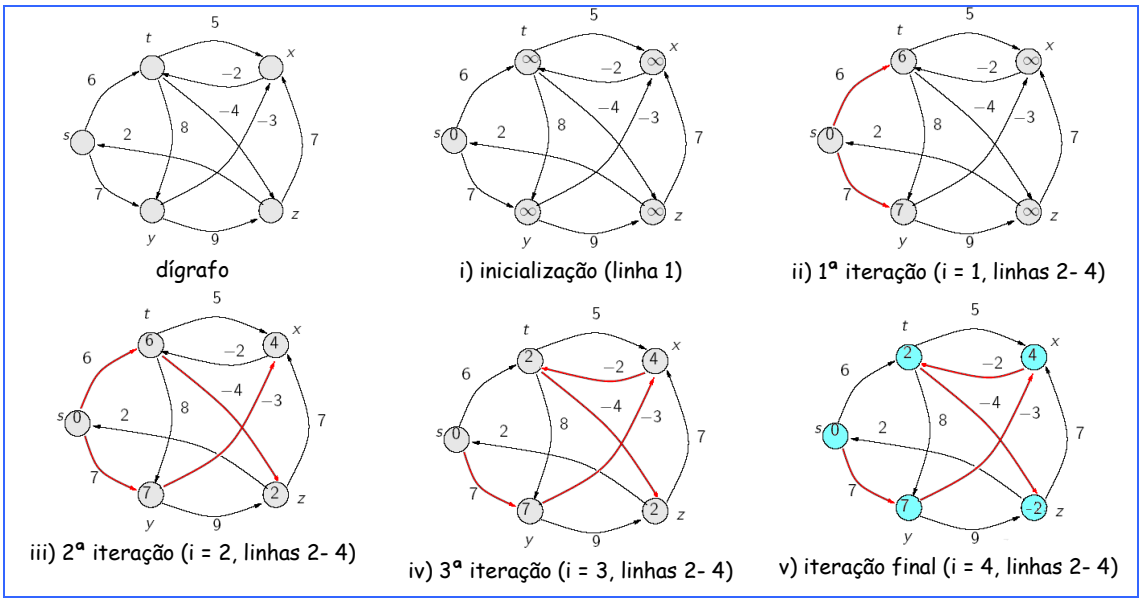
O algoritmo indica se um ciclo de comprimento negativo alcançável a partir de *s* foi ou não encontrado. Em caso afirmativo não há solução e, em caso contrário - onde não há ciclo negativo alcançável a partir de *s* - o algoritmo produz a árvore de caminhos mínimos com raiz em *s* e os seus respectivos comprimentos (pesos).

Da mesma forma que o algoritmo de *Djikstra,* o algoritmo de *Bellman-Ford* também utiliza a técnica de relaxamento de arestas progressivamente diminuindo a estimativa de menor caminho d[*v*] entre os vértices *s* e *v*∈*V* até encontrar o valor real de menor caminho *δ(s, v)*.

```
Bellman-Ford (G, w)
 1. IniciaOrigemÚnica(G,s)
 2. para i ← 1 até |V| -1 faça          (|V| = n vértices)
 3.       para cada (u,v) ∈ E faça      (E = conjunto de arcos)
 4.             Relaxa(u,v,w)
 5. para cada (u,v) ∈ E faça            (verifica se existe ciclo negativo)
 6.    se d[v] > d[u] + w(u,v)
 7.          então devolve 0           (tem ciclo negativo alcançavel a partir de s)
 8. devolve 1                           (não tem ciclo negativo)
```

As figuras a seguir demonstram a execução do algoritmo em um dígrafo onde a origem é o vértice "s" mais à esquerda que contêm arcos de peso negativo. Para este exemplo específico, cada passagem na "linha 3", para relaxar os arcos e, também na "linha 5", para observar se há ciclo negativo, segue a ordem: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t) e (s,y). Em outras simulações podemos adotar, por exemplo, a ordem em o arco foi inserido no dígrafo como forma de estabelecer uma ordem de inspeção no algoritmo.

As estimativas de caminhos mais curtos são mostradas dentro dos vértices, e a origem dos arcos "vermelhos" indica os predecessores.



Finalizando, a execução das linhas 5 e 6 para todos os arcos não resulta no diagnóstico da linha 7, isto é, esse dígrafo não tem um ciclo negativo alcançável a partir de *s*.

Algoritmo de Floyd-Warshall

O algoritmo *Floyd-Warshall* resolve o problema de calcular o caminho mais curto entre todos os pares de vértices em um dígrafo $G = (V, E)$.

Arestas com *peso negativo* podem estar presentes, mas assumimos por conveniência que não há ciclos com pesos negativos. (O algoritmo é capaz de detectar tais ocorrências.)

Caminhos mínimos entre todos os pares:

- Consideramos o problema de encontrar o caminho mais curto entre cada par de vértices de um grafo $G = (V, E)$.
- Este problema pode surgir na construção de uma tabela de distâncias entre cidades de um atlas.
- É dado um dígrafo $G = (V, E)$ com função peso das arestas dada por $w : E \rightarrow R$.
- Problema: para cada par $u, v \in V$, encontre o caminho de menor peso de *u* para *v*.

Métodos de solução

- Podemos resolver o "problema de caminhos mínimos entre todos os pares de vértices" resolvendo $|V| = n$ "problemas de caminhos mínimos com uma fonte."
- Se todos os pesos são não negativos, podemos utilizar o algoritmo de Dijkstra em cada vértice.
- Se há arestas com pesos negativos, precisaremos utilizar o algoritmo de Bellman-Ford, que deve ser executado a partir de cada vértice: um processo mais lento se comparado com Dijkstra.
- Temos uma solução mais eficiente: algoritmo de Floyd-Warshall:

Definições

- Comparando com os algoritmos anteriores, que utilizaram lista de adjacência, representaremos o dígrafo por meio de uma matriz de adjacência.
- Por conveniência, assumiremos que os vértices são numerados 1, 2, . . . , $|V| = n$.
- Ciclos com peso negativo podem ocorrer, mas assumiremos que o grafo não contém ciclo negativo.
- A saída tabulada do algoritmo é uma matriz $D = [d_{ij}]$.
- Cada entrada d_{ij} contém o comprimento do caminho mais curto do vértice i ao vértice j, ou seja, $d_{ij} = \delta(i, j)$.
- Computamos não apenas a distância, mas também o caminho. Para tanto, utilizamos uma matriz $\Pi = [\pi_{ij}]$ com os predecessores.
- A entrada $\boldsymbol{\pi_{ij}}$ = NIL se i = j e $\boldsymbol{\pi_{ij}}$ é o predecessor de j em um caminho mais curto de i para j .
- Da mesma forma que nos problemas anteriores, o subdígrafo predecessor G_π é uma árvore de caminhos mínimos a partir de um vértice.
- O subdígrafo predecessor $G_{\pi,i}$ induzido pela i-ésima linha é uma árvore de caminhos mínimos em *G* a partir de i .
- O dígrafo predecessor $G_{\pi,i}$ é definido por:

$$V_{\pi,i} = \{ j \in V : \pi_{i,j} \neq NIL \} \cup \{ i \}$$
$$E_{\pi,i} = \{ (\pi_{i,j}, j) : j \in V_{\pi,j} - \{ i \} \}$$

- Como os vértices são V = {1, 2, ..., n} considere um subconjunto {1, 2, ..., k};
- Para qualquer par de vértices (i,j) em V, considere todos os caminhos de i a j cujos vértices intermediários pertencem ao subconjunto {1, 2, ..., k}, e p como o mais curto de todos eles;
- O algoritmo explora um relacionamento entre o caminho p e os caminhos mais curtos de i a j com todos os vértices intermediários em {1, 2, ..., (k-1)};
- O relacionamento depende de k ser ou não um vértice intermediário do caminho p.

Estrutura:

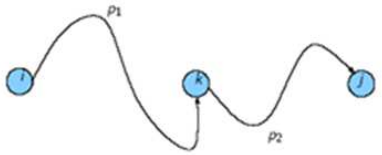
O relacionamento depende se k é um vértice intermediário em p^k_{ij} ou não;

Caso a) $k \notin p^k_{ij}$

- k não é um vértice intermediário do caminho p^k_{ij}.
- Então todos os vértices intermediários em p^k_{ij} são elementos do conjunto $\{1, \ldots, k-1\}$.
- Logo um caminho mais curto $p^k_{ij} \in S^k_{ij}$ é também um elemento de S^{k-1}_{ij}.
- Ou seja, $p^k_{ij} \in S^{k-1}_{ij} \subset S^k_{ij} \Rightarrow p^k_{ij} = p^{k-1}_{ij}$.

Caso b) $k \in p^k_{ij}$

- Se k é um vértice intermediário de p^k_{ij}, então podemos quebrar p^k_{ij} em $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$, como mostra a figura.



- Todos os vértices intermediários de p_1 são vértices do conjunto $\{1, \ldots, k-1\}$, ou seja, $p_1 = p^{k-1}_{ik} \in S^{k-1}_{ik}$.
- Todos os vértices intermediários de p_2 são vértices do conjunto $\{1, \ldots, k-1\}$, ou seja, $p_2 = p^{k-1}_{kj} \in S^{k-1}_{kj}$.
- Sabemos que p_1 é um caminho simples, tal que $p_1 = p^{k-1}_{ik} \in S^{k-1}_{ik}$, pois k não é vértice intermediário de p_1.
- Sabemos que p_2 é um caminho simples, tal que $p_2 = p^{k-1}_{kj} \in S^{k-1}_{kj}$, pois k não é vértice intermediário de p_2.
- Portanto $p^k_{ij} = p^{k-1}_{ik} \cup p^{k-1}_{kj}$.

Algoritmo:

Floyd-Warshall (G)

1. **n** ← |V|
2. $D^0 \leftarrow G$ (iteração zero = matriz de adjacência de G)
3. **para** k ← 1 até n **faça**
4. **para** i ← 1 até n **faça**
5. **para** j ← 1 até n **faça**
6. $d^k_{i,j} \leftarrow$ Mínimo{ $d^{k-1}_{i,j}$, $d^{k-1}_{ik} + d^{k-1}_{k,j}$ }
7. **devolve** D^n .

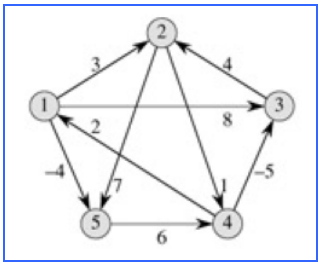
Construindo do caminho:

- Há uma variedade de métodos para construção do caminho mais curto no algoritmo de Floyd-Warshall.
- Um jeito é computar a matriz D^n de distâncias e depois construir a matriz predecessora Π a partir de D^n.
- Outra maneira é computar a matriz predecessora Π *"on-line"*, da mesma forma que o algoritmo Floyd-Warshall calcula as matrizes D^k.
- Podemos computar $\Pi^0, \Pi^1, \ldots, \Pi^n$, onde $\Pi = \Pi^n$.
- Π^k_{ij} é definida como a matriz predecessora do vértice j no caminho mais curto a partir de i, tendo como vértices intermediários os elementos do conjunto $\{1, 2, \ldots, k\}$.
- Podemos dar uma fórmula recursiva para Π^k_{ij}.
- Quando $k = 0$, um caminho mais curto de i para j não tem vértices intermediários, logo: $\pi^0_{ij} = \begin{cases} nil & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$
- Para $k \geqslant 1$, se tomamos o caminho $i \rightsquigarrow k \rightsquigarrow j$, com $k \neq j$, então o predecessor de j é o mesmo que o predecessor de i escolhido no caminho mais curto de k para j, com vértices intermediários do conjunto $\{1, \ldots, k-1\}$.
- Ou seja, o predecessor de j no caminho p^{k-1}_{ki}.
- Formalmente, temos: $\pi^k_{ij} = \begin{cases} \pi^{k-1}_{ij} & \text{if } d^{k-1}_{ij} \leqslant d^{k-1}_{ik} + d^{k-1}_{kj} \\ \pi^{k-1}_{kj} & \text{if } d^{k-1}_{ij} > d^{k-1}_{ik} + d^{k-1}_{kj} \end{cases}$

Obtidas as matrizes $\mathbf{D}^n$ e Π^n podemos imprimir um caminho mais curto desde o vértice i até o vértice j, onde i, j ∈ V.

```
MostraCaminhoMaisCurto( Π, i, j )
1. se i = j
2.     então escreve i
3.     senão se π_ij = NIL
4.             então escreve i"não há caminho de" i "para" j
5.             senão MostraCaminhoMaisCurto( Π, i, π_ij )
6.                 escreve j.
```

Exemplo: considere o dígrafo abaixo, que contêm pesos positivos e negativos e não tem ciclo negativo



Aplicando o algoritmo de Floyd-Warshall obtemos a sequência de matrizes D^k e Π^k:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

E se queremos verificar o caminho mais curto entre os vértices:

- "3" e "4" obtemos como MostraCaminhoMaisCurto(Π, 3, 4): 3 → 2 → 4 .
- Ou, entre "5" e "2" aplicamos MostraCaminhoMaisCurto(Π, 5, 2): 5 → 4 → 3 → 2 .

Exercícios

1) Aplique o algoritmo de Dijkstra ao dígrafo abaixo começando com o vértice 1. No começo de cada iteração, dê o custo de cada vértice. No fim da última iteração, exiba a árvore de caminhos mínimos com origem 1.

arco	0-1	0-4	1-5	2-0	2-3	2-4	4-3	5-0	5-2
custo	1	3	1	1	6	5	1	4	2

2) Suponha dado um grafo com vértices 1,2,...,n e que todas os arcos são da forma *(j, i)*, sendo *i < j*. Suponha que cada arco *(u,v)* tem um peso *w(u,v)*, que pode ser positivo ou negativo.

a) Escreva um algoritmo que encontre um caminho de peso mínimo de um vértice *s* a um vértice *t*. Quanto tempo o seu algoritmo consome?

b) A *altura* de um vértice *u* é o peso de um caminho de peso mínimo de *u* até o vértice 1. (OBS: "de *u* até 1".) Escreva um algoritmo que resolva o seguinte problema: *Encontrar a altura de cada um dos vértices do grafo.*

3) Que acontece se o algoritmo Dijkstra for interrompido quando *Q* tem apenas um elemento?

4) O algoritmo de Dijkstra funciona corretamente sobre grafos não-simétricos (ou é preciso que o grafo seja simétrico)?

5) Mostre que o algoritmo Dijkstra pode produzir resultados errados se o peso *w(u,v) < 0* para alguma aresta *(u,v)*.

6) Considere o dígrafo *G*:

arco	1-2	1-4	2-3	2-4	3-5	4-2	4-3	4-5	5-3	5-1
custo	10	5	1	2	4	3	9	2	6	7

a) Usando o algoritmo de Dijkstra encontre os menores caminhos entre o vértice S e os demais vértices de *G*.

b) Modifique o valor (peso) da aresta (X, U) para -3 e empregue o algoritmo de Bellmann-Ford para encontrar os caminhos entre S e os demais vértices. Existem ciclos negativos?

c) Comparando as execuções nos itens (a) e (b), a estimativa de tempo de execução é maior em qual algoritmo? Por quê?

d) Utilize o algoritmo de Floyd-Warshall e encontre o caminho mais curto para todos os pares de vértices de *G*.

7) Suponha dado um dígrafo com custos não-negativos associados aos vértices (e não aos arcos). O custo de um caminho num tal dígrafo é a soma dos custos dos vértices do caminho. Quero encontrar um caminho de custo mínimo dentre os que começam num vértice *s* e terminam num vértice *t*. Adapte o algoritmo de Dijkstra para resolver esse problema.

8) Escreva um algoritmo que encontre um caminho de peso *máximo* de um vértice *s* a um vértice *t* em um grafo acíclico com pesos nos arcos (os pesos podem ser negativos). Quanto tempo o seu algoritmo consome?

9) Escreva uma função que encontre um caminho de custo mínimo num tabuleiro com n linhas e n colunas. Cada casa do tabuleiro tem um custo não-negativo. O seu caminho deve começar na casa que está no cruzamento da linha 1 com coluna 1 e terminar na casa que está no cruzamento da linha n com a coluna n. O caminho só pode passar de um casa para a casa vizinha na horizontal ou vertical (não na diagonal). O custo de um caminho é a soma dos custos das casas por onde o caminho passa.

Fluxo em Redes

Considere examinar um grafo orientado (dígrafo) e interpretá-lo como uma Rede de Fluxo usando-o para analisar fluxo de materiais a partir de uma origem, onde o material é produzido, até um depósito, onde o material é consumido. A origem produz o material a uma taxa fixa, e o depósito consome o material na mesma taxa. O "fluxo" do material em qualquer ponto no sistema será intuitivamente a taxa na qual o material se move.

Cada aresta orientada pode ser imaginada como um canal, com uma capacidade estabelecida, dada como uma taxa máxima na qual o material pode fluir pelo canal. Vértices são junções de canais, onde o material flui sem acumulação. Isto é, com exceção da origem e do destino, a taxa de entrada e de saída de material no vértice deve ser a mesma. Chamamos essa propriedade de "conservação do fluxo".

Com essas premissas podemos estudar o problema de fluxo máximo, onde desejamos calcular a maior taxa na qual o material pode ser enviado da origem até o depósito.

Vamos usar uma terminologia diferente da adotada na bibliografia para facilitar a apresentação dos conceitos: usamos **"Redes de Fluxo"** ao invés de "Fluxo em Redes" e **"antisimetria"** e lugar de "simetria obliqua".

Fluxo em Rede

Fluxo em Rede corresponde ao estudo de um dígrafo, $G = (V, E)$, em que cada arco $(u,v) \in E$ tem uma *capacidade* $c(u,v) \geq 0$ (não negativa). Se um dado arco não está em E, então supomos que sua capacidade é zero (em geral não se desenham tais arcos nos dígrafos). Na rede de fluxo (ou dígrafo em que estudamos o fluxo em rede) temos dois vértices especiais, *origem* s e um *destino ou alvo* t, e para todo vértice v do dígrafo existe um caminho a partir de s passando por v que chega em t.

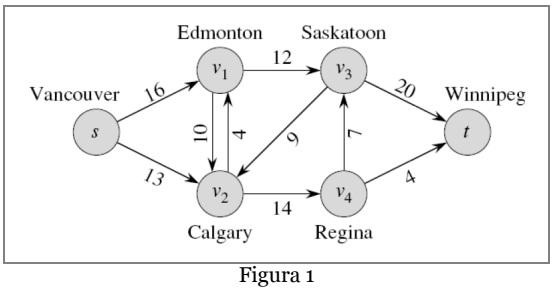


Figura 1

Na figura 1* acima, temos uma rede de fluxo para o *problema de distribuição de produtos*. A fábrica da empresa canadense fica em Vancouver e seu depósito em Winnipeg. Os vértices intermediários indicam as cidades entre a fábrica e o depósito por onde os produtos podem ser transportados. Cada aresta é identificada pela sua capacidade diária, $c(u,v)$, de transporte.

Fluxos

Definimos um *fluxo* como um valor que atribuímos a cada aresta do grafo. De um modo mais formal, dizemos um *fluxo* em $G = (V,E)$ como uma função de valor real $f : V \times V \rightarrow \mathbf{R}$, que satisfaz a três propriedades:

Restrição de capacidade: Para todos $u, v \in V$, exigimos $f(u, v) <= c(u,v)$. Ou seja, um fluxo não pode exceder à capacidade da aresta.

Antisimetria: Para todos $u, v \in V$, exigimos $f(u, v) = -f(v, u)$.

Conservação de fluxo: Para todo $u \in V - \{s,t\}$, exigimos

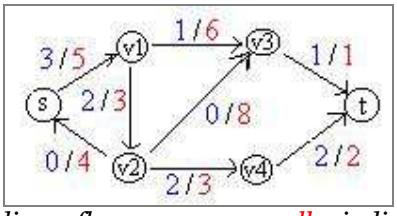
$$\sum_{v \in V} f(u, v) = 0$$

A quantidade $f(u, v)$, é chamada de *fluxo do vértice u* até o vértice *v*.

Pela propriedade de conservação de fluxo, o fluxo total de um vértice é 0 (zero), sendo que esse vértice não pode ser a origem, nem o destino. Observe que o somatório referente à conservação do fluxo é para cada vértice, e não para o grafo.

Exemplifica-se essa propriedade usando a rede da fábrica apresentado na figura anterior considerando que os caminhões que transportam os produtos passam pelas cidades durante o caminho, mas em nenhuma delas eles podem armazenar os produtos. Somente a fábrica e o depósito têm essa capacidade.

Outro exemplo de rede:



Valores em azul indicam fluxos e, em vermelho indicam as capacidades.

Repare que em todos os vértices temos conservação de fluxo.

O *valor* de um *fluxo f é definido como:*

$$|f| = \sum_{v \in V} f(s, v)$$

Ou seja, o *fluxo total* que sai da origem, no exemplo o valor é 3.

Um *fluxo máximo* é um fluxo de valor máximo.

Uma *distribuição* é um caminho possível.

Na figura acima temos uma distribuição que passa pelo vértice $v1$ e vai para o vértice $v3$, e temos outra que passa por $v1$ e segue para $v2$.

Na figura.2* abaixo, temos mais um exemplo com um fluxo f em G com valor $|f| = 19$. São mostrados apenas fluxos positivos. Se $f(u,v) > 0$, a aresta (u,v) é identificada por $f(u,v)/c(u,v)$. Se $f(u,v) \leq 0$, a aresta (u,v) é identificada apenas por sua capacidade.

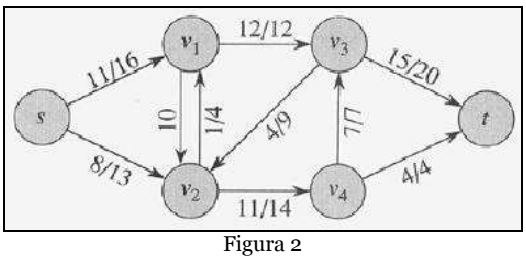


Figura 2

Método de Ford-Fulkerson

O *método de Ford-Fulkerson* tem por objetivo *encontrar um fluxo máximo* para uma rede de fluxos.

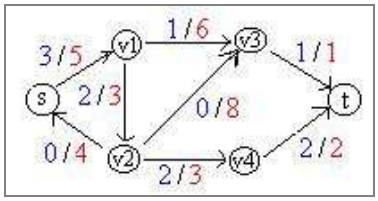
É chamado de método por englobar diversas implementações com diferentes tempos de execução. O método é iterativo, começando com $f(u,v) = 0$ para todo $u, v \in V$ e, um fluxo inicial de valor 0 (zero).

A cada iteração aumentamos o valor do fluxo encontrando um *caminho aumentante*, que podemos imaginar como um caminho de s a t onde podemos *"empurrar"* mais fluxo para ampliar o fluxo ao longo da rede. O processo é repetido até que não sejam encontramos mais caminhos aumentantes.

```
MÉTODO FORD-FULKERSON(G, S, T)
1  iniciar fluxo f como 0
2  enquanto existir um caminho aumentante p faça
3        ampliar fluxo f ao longo de p
4  devolve f
```

Redes residuais

Intuitivamente uma rede residual consiste em arestas que podem admitir mais fluxo. Em nosso exemplo abaixo, a aresta que liga *v1* a *v3* tem *fluxo = 1* e *capacidade = 6*. Essa aresta tem *capacidade residual* de 5, e portanto estaria na rede residual desse grafo.



De um modo mais formal, considere que temos uma rede de fluxo *G = (V, E),* com origem *s* e destino *t*. Seja *f* um fluxo em *G*, e considere um par de vértices *u, v* $\in$ *V.* A quantidade de fluxo adicional que podemos empurrar desde *u* até *v* e que não ultrapasse a capacidade c(u, v) é a capacidade residual de (u, v) dada por:

$$c_f(u, v) = c(u, v) - f(u, v)$$

Quando o fluxo $f(u,v)$ é *negativo*, a *capacidade residual* $c_f(u,v)$ é maior que a capacidade $c(u,v)$.

Dados uma *rede de fluxo G = (V,E)* e um fluxo *f*, a rede residual de *G* induzida por *f* é $G_f=(V,E_f)$, onde $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$.

A figura.3* abaixo, mostra a rede residual gerada a partir da figura.2 anterior.

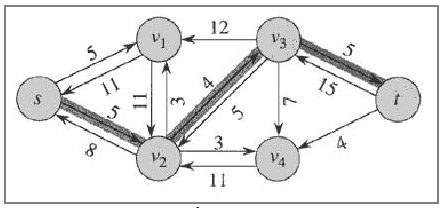


Figura 3

Caminhos aumentantes

Dados uma *rede de fluxo G = (V,E)* e um fluxo *f* , um *caminho aumentante p* é um caminho simples desde *s* até *t* na rede residual G_f constituído por arestas estritamente positivas. Se o fluxo na rede não é máximo, sempre existe um caminho aumentante. O caminho demarcado na figura.3 acima é um *caminho aumentante.*

Cortes de redes de fluxo

De maneira geral um corte de uma rede de fluxo é dividir o dígrafo em duas partições onde os vértices de origem e de destinos ficam separados.

Podemos definir um *corte(S, T)* de uma rede de fluxo *G = (V, E)* como uma partição de *V* em *S* e *T=V-S* tal que *s* $\in S$ e *t* $\in T$.

Se f é um fluxo, então o *fluxo líquido* pelo *corte(S, T)* é definido como *f (S, T)*. A *capacidade do corte(S, T)* é *c(S, T)* dada pela soma das capacidades das arestas que cruzam o corte feito com origem em *S* e chegam em *T*.

Um ***corte mínimo*** de uma rede é um corte cuja capacidade é mínima dentre todos os cortes da rede. A figura.4* abaixo, mostra um *corte(S, T),* onde $S = \{s, v_1, v_2\}$ e $T = \{v_3, v_4, t\}$. Os vértices em *S* são pretos, e os em *T* são brancos. O *fluxo liquido* por *(S,T)* é f*(S, T)* = 19, e a capacidade é *c(S, T)* = 26.

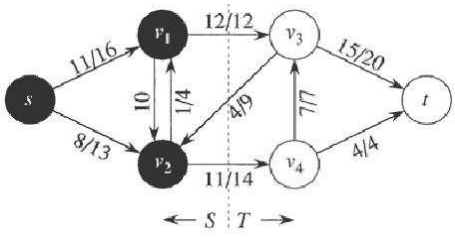


Figura 4

Algoritmo de Ford - Fulkerson

O método de Ford-Fulkerson consiste em procurar por um caminho aumentante *p* e aumentar o fluxo *f* de cada aresta do caminho aumentante levando em consideração a capacidade residual das mesmas. Devemos nos atentar para o fato de que se dois vértices, u e v, não estiverem conectados por uma aresta, deve-se considerar que o fluxo *f*[u,v] é igual a zero.

O algoritmo Ford-Fulkerson consiste numa extensão do método de Ford-Fulkerson mostrado em pseudocódigo anterior. Ele termina quando não for mais possível encontrar um caminho aumentante na rede residual.

```
FORD-FULKERSON(G,s,t)
1  para cada aresta (u,v) ∈ G faça
2       f[u,v] ← 0
3       f[v,u] ← 0
4  enquanto existir um caminho p de s até t na rede residual Gf faça
5     cf(p) ← min{cf(u,v) : (u,v) está no caminho p}
6     para cada aresta (u,v) em p faça
7              f[u,v] ← f[u,v] + cf(p)
8              f[v,u] ← (-f[u,v])
```

O tempo de execução depende da maneira como se determina o caminho aumentante.

Em alguns casos, o algoritmo pode nunca terminar (se as arestas forem números irracionais) ou demorar muito tempo até encontrar o valor de fluxo máximo.

Para que o algoritmo seja executado em tempo polinomial, deve ser utilizada a Busca em Largura (BFS) para se determinar o caminho aumentante.

Se as capacidades das arestas consistirem em números inteiros, no pior caso o caminho será aumentando de uma em uma unidade a cada iteração.

O tempo de execução do algoritmo será $O(E\ |f^*|)$, onde o f^* corresponde ao valor do fluxo máximo encontrado pelo algoritmo.

A implementação do algoritmo deve utilizar uma lista de adjacências para representar a rede G=(V,E). Assim, o tempo para se encontrar um caminho na rede residual será $O(E)$.

Como o laço de repetição da linha 4 é executado em tempo $O(E)$ e ele será executado no máximo $|f^*|$ vezes (o valor do fluxo aumenta pelo menos uma unidade para cada iteração) chega-se a um tempo de execução total igual a $O(E\ |f^*|)$.

Algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp consegue melhorar o limite do algoritmo de Ford-Fulkerson escolhendo sempre a distância do caminho mais curto desde *s* até *t* na rede residual.

Aresta crítica: uma aresta (u,v) em uma rede residual G_f é crítica em um caminho aumentante quando a capacidade residual do caminho *equivale* à capacidade residual da aresta (u,v). Uma aresta crítica pode ser comparada a um "gargalo", pois é a menor do caminho.

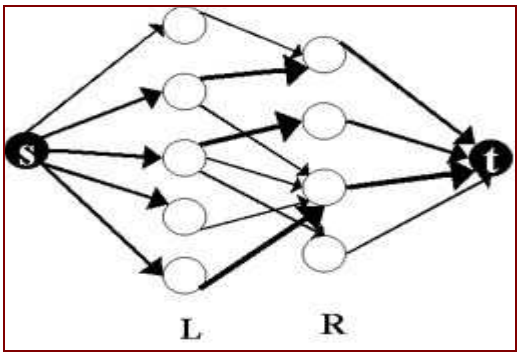
Como cada iteração do algoritmo de Ford-Fulkerson leva tempo $O(E)$ quado se utiliza Busca em Largura na procura do caminho aumentante, o algoritmo de Edmonds-Karp pode ser executado em tempo $O(VE^2)$.

```
EDMONDS-KARP(G, s, t)
1. para cada aresta (u, v) ∈ G faça
2.          f[u, v] ← 0
3.          f[v, u] ← 0
4. BFS*(Gf, s)  modificando BFS(Gf, s) : se u = t anota PRETO e pára a repetição
5. enquanto cor[t] = PRETO faça
6.          cf(p) ←MIN { cf(u, v) : (u, v) está em p }
7.          para cada aresta (u, v) em p faça
8.                  f[u, v] ← f[u, v] + cf(p)
9.                  f[v, u] ← - f[u, v]
```

Considerando um grafo G = (V,E), um emparelhamento consiste em um subconjunto de arestas M contido em E, tais que, para todo vértice v pertencente a V, no máximo uma aresta de M é incidente sobre v. Dessa forma, o emparelhamento bipartido máximo consiste numa escolha de arestas que não se tocam, em grafos bipartidos.

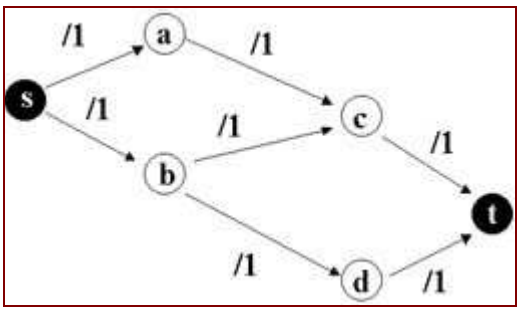
Pode-se visualizar uma aplicação de emparelhamento bipartido máximo a partir do problema da linha de produção, que considera uma correspondência de um conjunto L de máquinas com um conjunto R de tarefas a serem executadas simultaneamente. A presença de uma aresta (u,v) significa que uma determinada máquina u pertencente a L é capaz de executar uma tarefa v pertencente a R. A resolução do problema deverá *fornecer trabalho para o maior número de máquinas possível.*

O problema de se encontrar um emparelhamento máximo em um grafo bipartido não orientado pode ser resolvido pelo *método de Ford-Fulkerson*, em tempo polinomial proporcional ao número de vértices, $|V|$ *e, arestas* $|E|$*:* $T(n) = O(VE)$. Para tal, deve ser construído um fluxo em rede onde os fluxos são equivalentes aos emparelhamentos. A figura abaixo ilustra o problema:

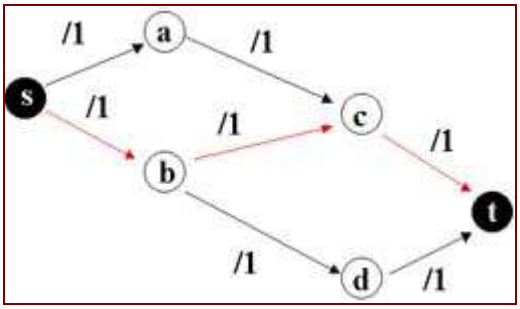


Ao fluxo de rede correspondente ao grafo bipartido são adicionados dois vértices, *s* e *t* (em negrito). Em seguida o vértice *s* é conectado a todos os vértices L e todos os vértices R são conectados ao vértice *t*. Cada aresta tem capacidade unitária. As arestas sombreadas têm um fluxo igual a 1 e as arestas restantes não conduzem nenhum fluxo. As arestas em negrito de L até R compõe uma correspondência máxima do grafo bipartido. Neste caso, tem-se M = 3.

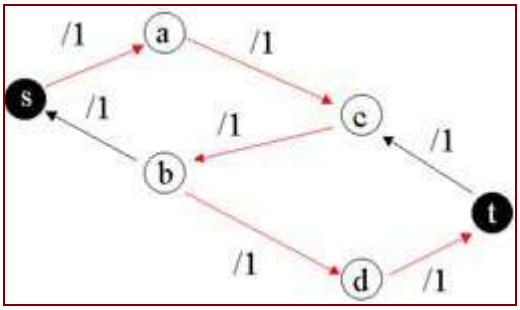
Um exemplo ilustrando um caso em que uma aresta poderia deixar de participar de um emparelhamento. Considere o grafo abaixo:



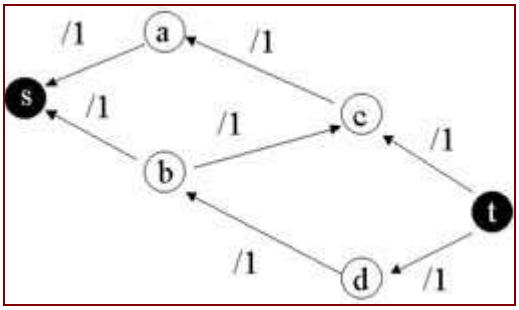
O primeiro caminho aumentante encontrado foi s → b → c → t, marcado em vermelho na próxima figura. A aresta participante do emparelhamento é a aresta (b,c). As arestas (s,b),(b,c),(c,t) terão suas capacidades residuais iguais a zero.



Na figura abaixo, o novo caminho aumentante está representado pelas arestas vermelhas.



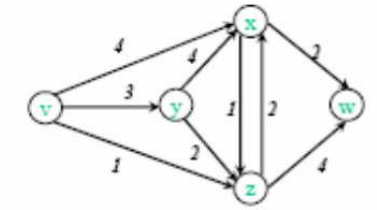
Finalmente, observe que na figura abaixo não é mais possível estabelecer um caminho aumentante. Verifique que as arestas (a,c) e (b,d) são as únicas participantes do emparelhamento, sendo que a aresta (b,c) deixou de participar do emparelhamento, caracterizando o emparelhamento máximo.



Exercícios

1 - Considere a situação modelada pelo grafo ao lado:

Cada arco representa uma rua de mão única. O peso de cada arco indica o maior fluxo possível ao longo da rua (veículos/hora). Qual o maior número possível de veículos que pode viajar de v a w em uma hora, representado pelo grafo ?



2 - A lista ao lado define um fluxo **f** numa rede capacitada com capacidade **c**. Qual o vértice inicial e o final? Existe caminho de aumento para f?

Exiba um fluxo máximo e um corte mínimo.

Diga qual a capacidade do corte.

arco	c	f
0-1	1	0
0-4	2	2
1-5	4	2
2-0	2	1
2-3	2	0
2-4	1	0
4-3	2	2
5-0	2	1
5-2	1	1

3- Considere a rede capacitada ao lado.

Dê uma seqüência de caminhos de aumento que produza um fluxo "com ciclo" (ou seja, um fluxo cuja representação por caminhos e ciclos tenha um ciclo).

Cada um dos caminhos de aumento deve ser simples (ou seja, não pode ter vértices repetidos).

arco	cap
0-1	2
0-2	3
0-3	2
1-3	1
1-4	1
2-1	1
2-5	2
3-4	2
3-5	3
4-2	1
4-5	2

s = 0 t = 5

* figura extraída do livro texto: CORMEN, Thomas H et al. *Algoritmos: teoria e prática.* 2a ed. Rio de Janeiro: Campus, 2002.