Digrafos

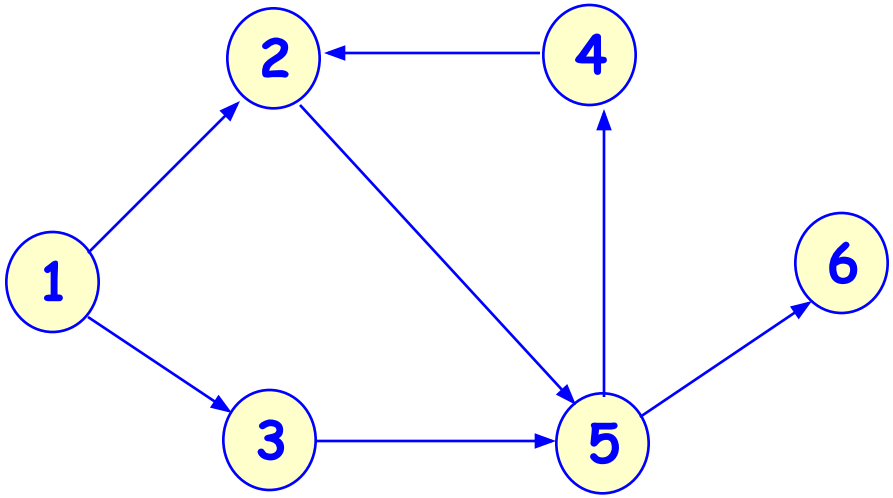
Notas de aula da disciplina IME 04-11312
(Otimização em Grafos)

Paulo Eustáquio Duarte Pinto
(pauloedp at ime.uerj.br)

setembro/2021

Definições Básicas: Digrafos

Um digrafo é um par D(V,E).
V é um conjunto de vértices.
E é um conjunto de pares ordenados de vértices.

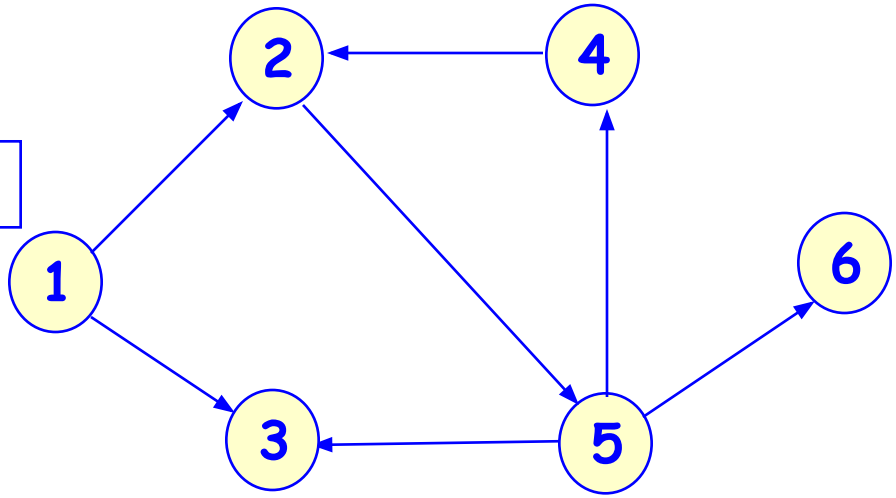


V = {1, 2, 3, 4, 5, 6}
E = { (1, 2), (1, 3), (2, 5), (3, 5), (4, 2), (5, 4), (5, 6) }

A representação computacional é análoga à de grafos, só que cada aresta é representada apenas uma vez.

Buscas em Digrafos

Buscas em Profundidade/Largura

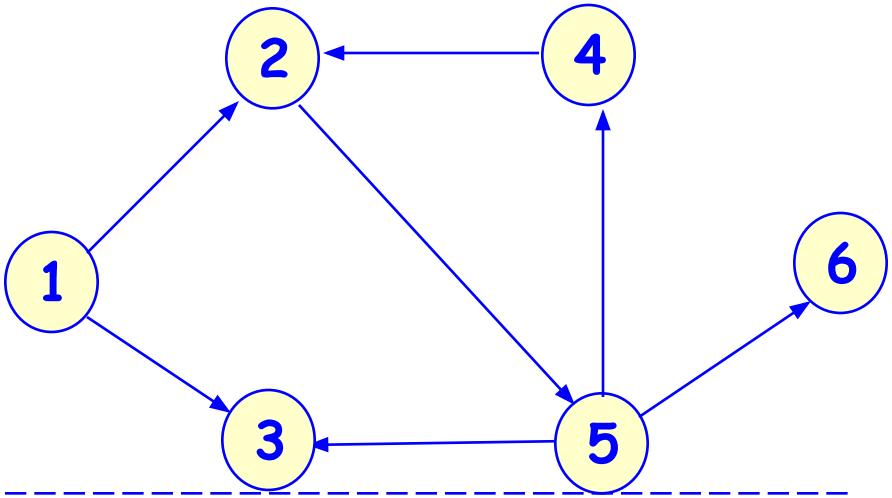


Usam-se os mesmos algoritmos mostrados para grafos simples.
Analogamente às buscas em grafos simples, também temos árvores (florestas) de Profundidade e de Largura.
Mas novos tipos de arestas surgem, as arestas de avanço e de cruzamento. As arestas são, então:

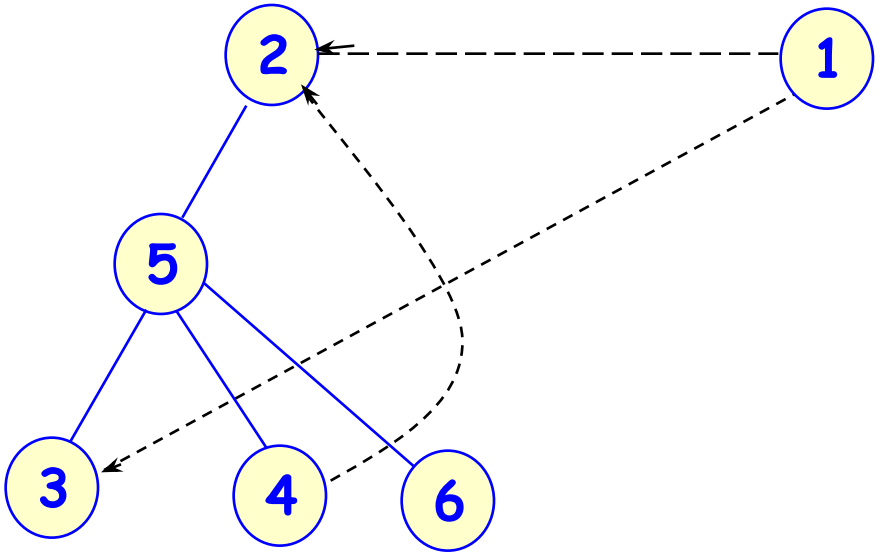
- arestas de árvore
- arestas de retorno
- arestas de avanço
- arestas de cruzamento

Buscas em Digrafos

Busca em Profundidade

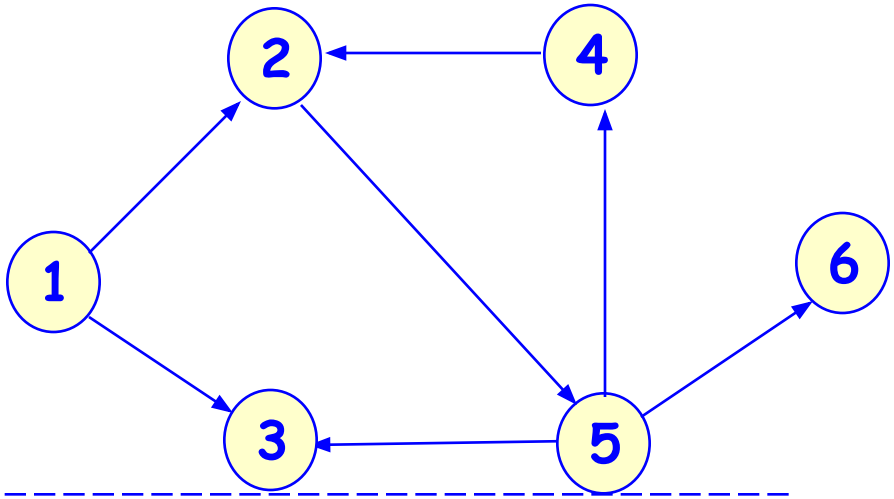


Floresta de Profundidade do digrafo

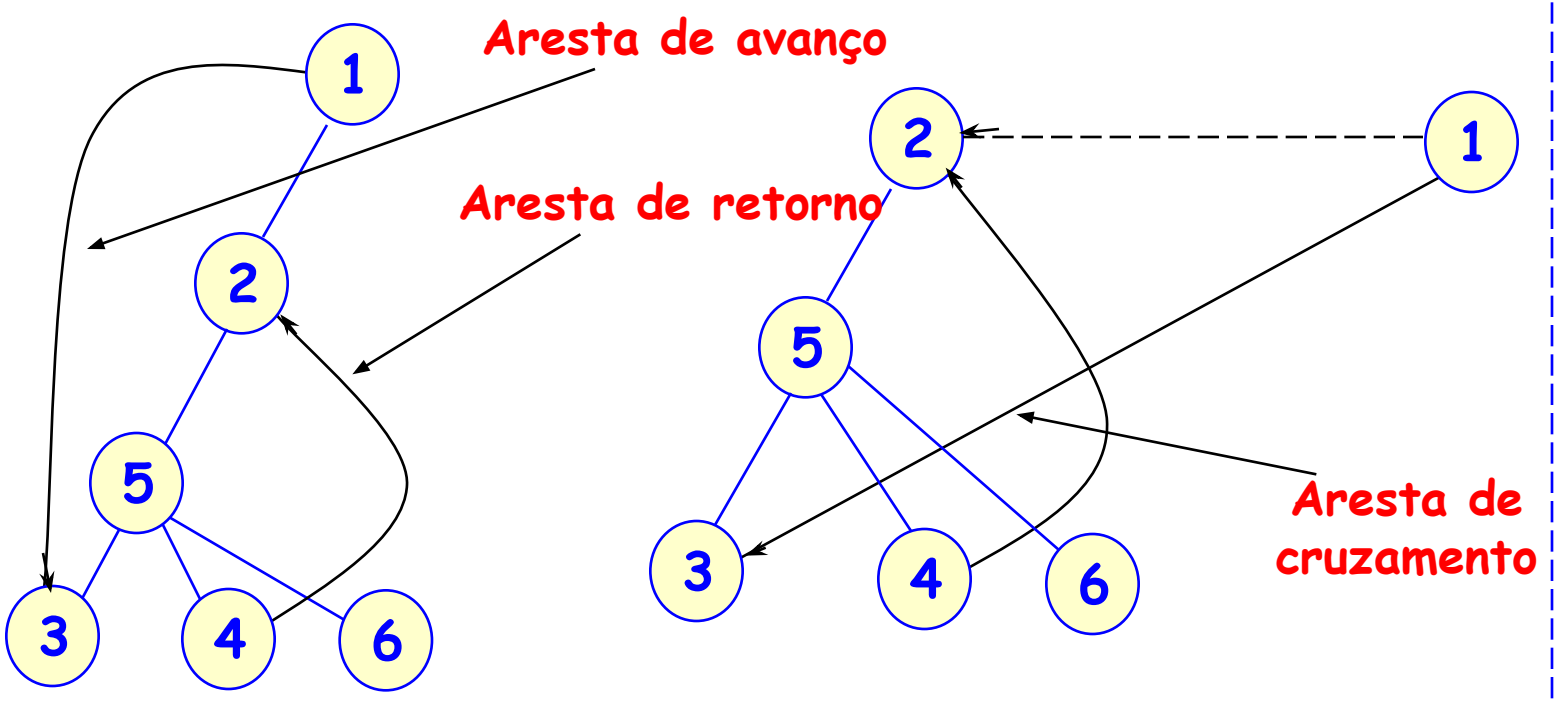


Buscas em Digrafos

Busca em Profundidade



Florestas de Profundidade do digrafo



Digrafos - Busca em Profundidade (DFS) - MA

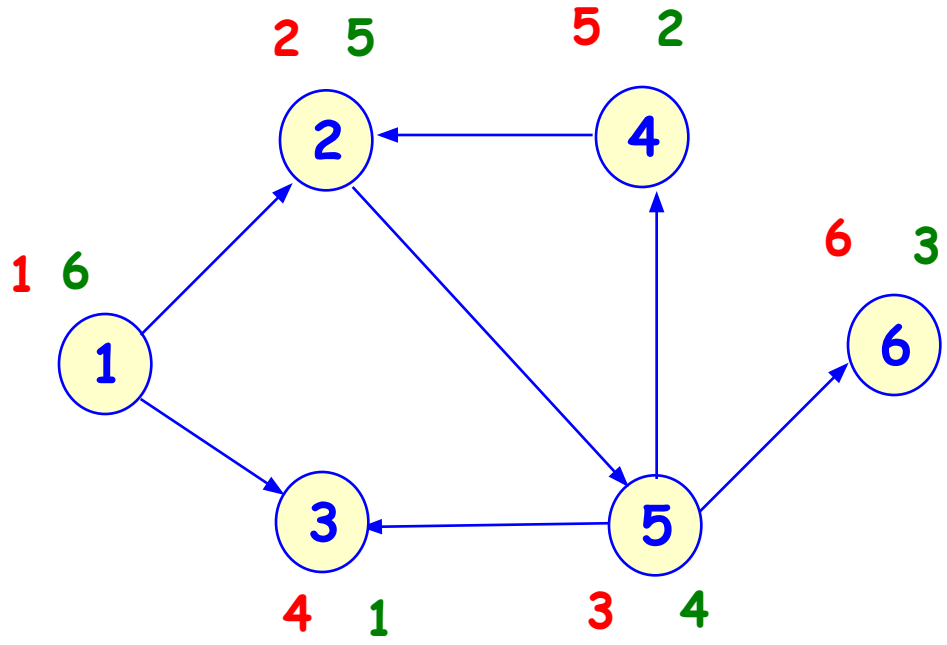
```
BP(u,v):
    se (u ≠ v):
        escrever ('árvore:',u,v)
    pre[v] ← ++cpre
    para w ← 1 até n incl.:
        se  E[v,w] = 1:
            se pre[w] = 0:
                BP(v,w)
            senão se pos[w]=0:
                escrever ('retorno', v, w)
            senão se pre[v] < pre[w]:
                escrever ('avanço', v,w)
            senão:
                escrever ('cruzamento', v, w)
    pos[v] ← ++cpos

pre[*] ← 0;   cpre ← 0;   pos[*] ← 0;   cpos ← 0;
para  i ← 1  até  n incl.:
    se  pre[i] = 0:
        BP(i,i)
```

Complexidade: $O(n^2)$

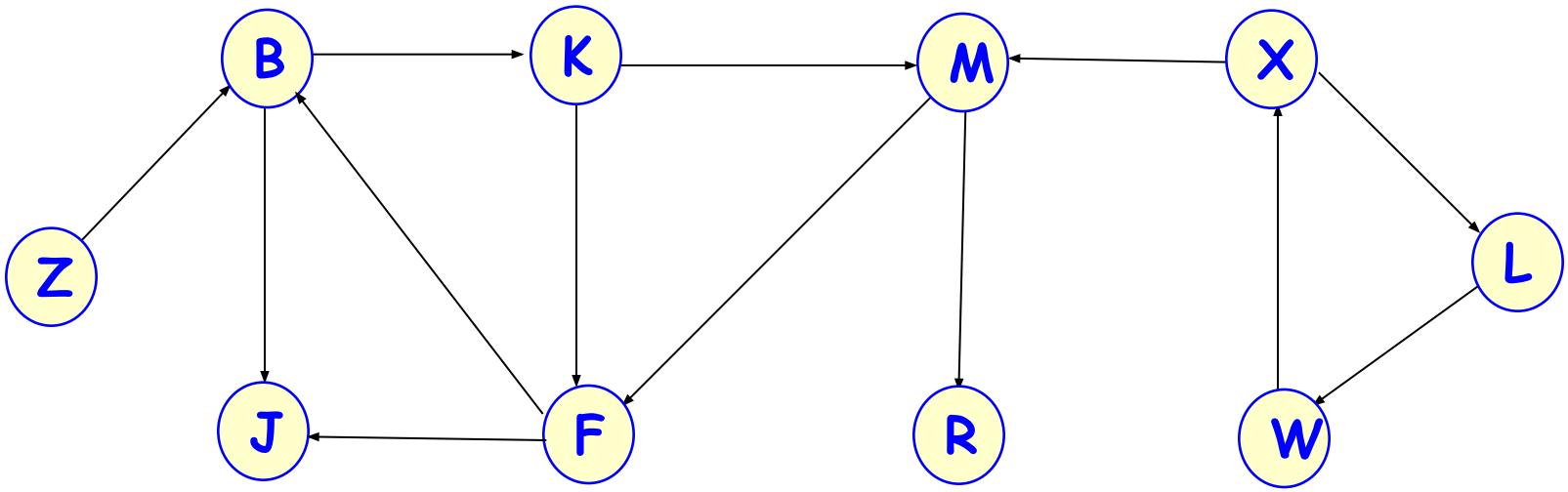
Buscas em Digrafos

Busca em Profundidade



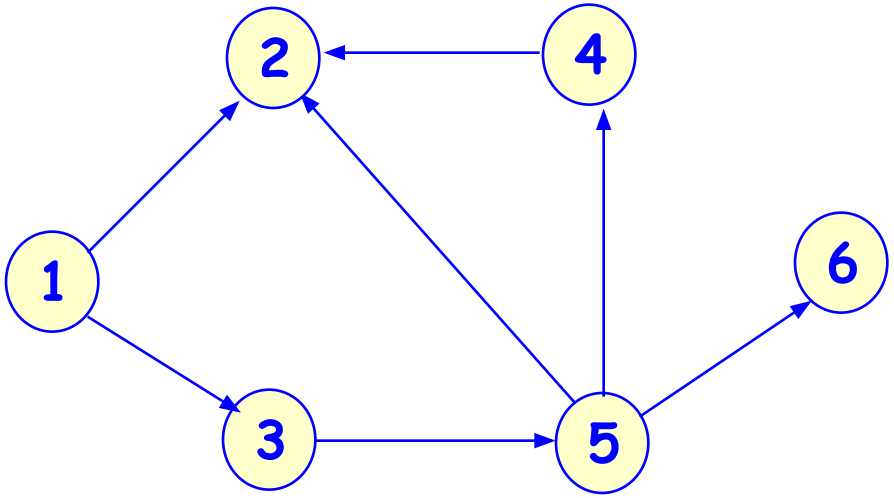
Valores de pre e pos

Busca em Digrafos



EXS8: Mostrar a árvore de profundidade da busca começando em um vértice com a letra mais próxima de seu nome.

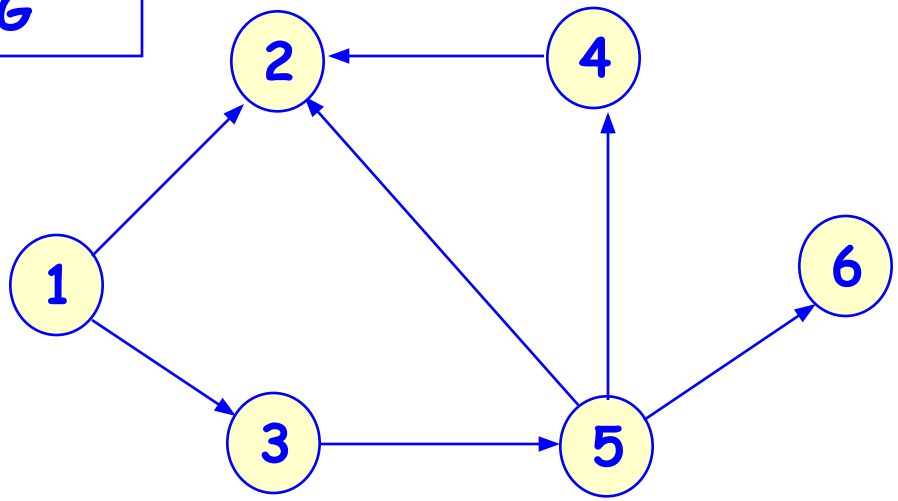
DAG – Digrafos Acíclicos



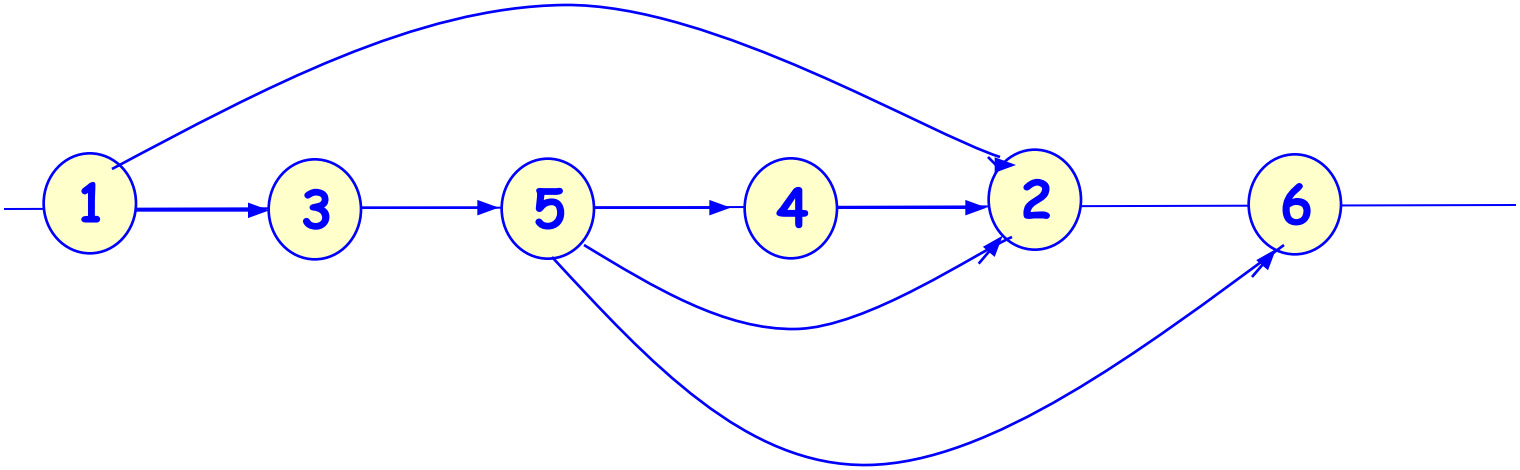
Em um digrafo acíclico existem pelo menos uma fonte (vértice com graus de entrada 0) e um sumidouro (vértice com grau de saída 0).

BP em um DAG:
Não existem arestas de retorno.

Ordenação Topológica num DAG



Uma ordenação topológica é uma ordenação dos vértices tal que se v_i vem antes de v_j, então não existe aresta de v_j para v_i.

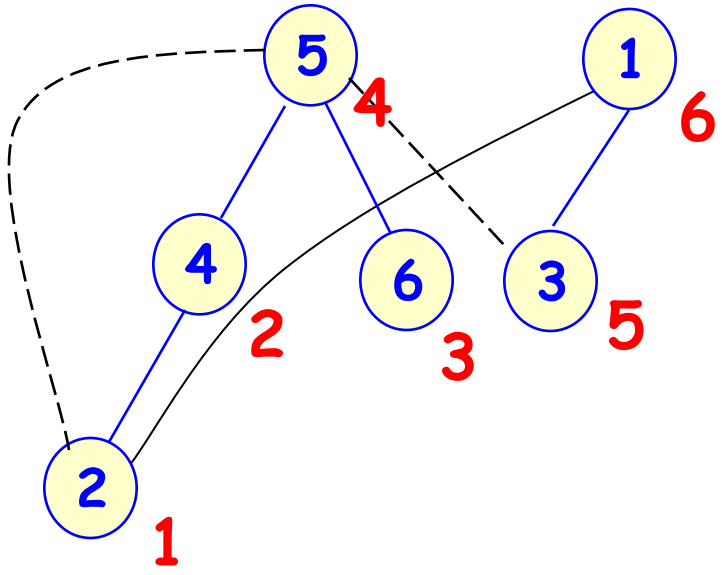


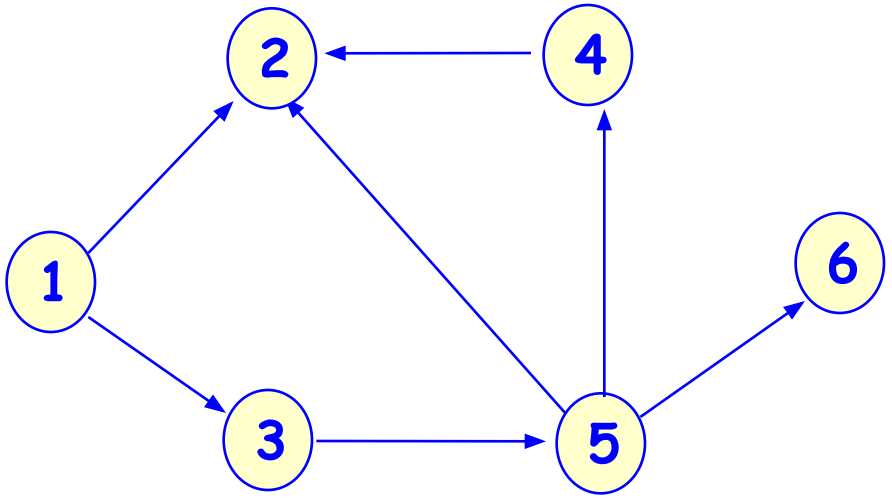
Corresponde a colocar os vértices em uma linha tal que todas as arestas estejam orientadas da esquerda para a direita.

Ordenação Topológica num DAG con DFS

A idéia é numerar os vértices de acordo com a saída da busca em profundidade e considerar o reverso dessa numeração;

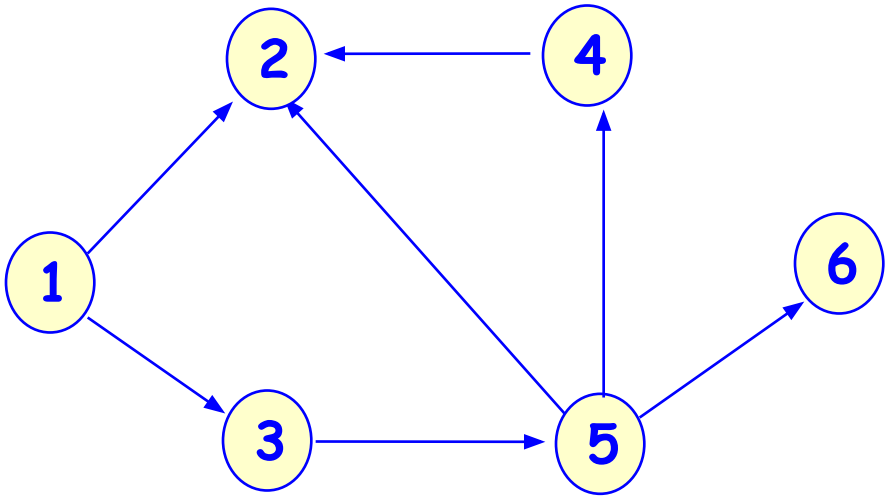
Floresta de Profundidade do digrafo e ordem de saída da busca

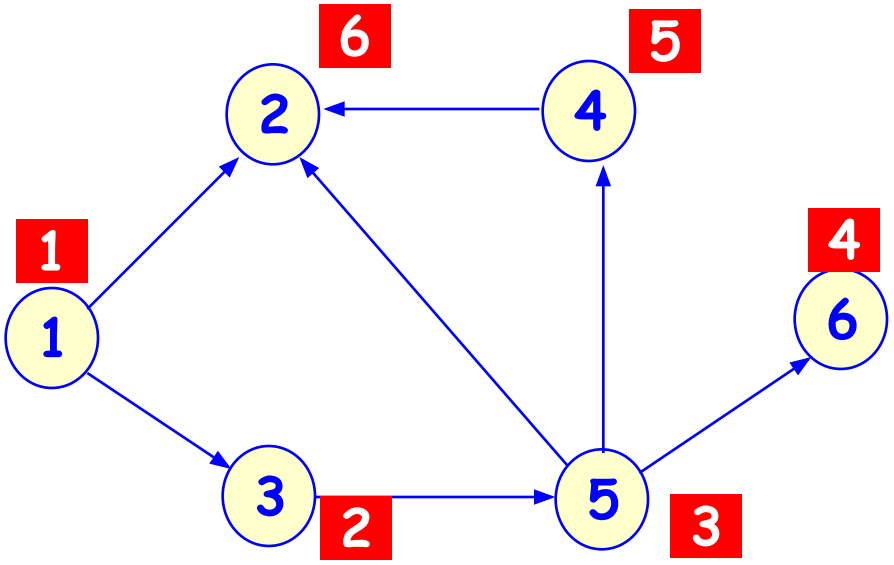




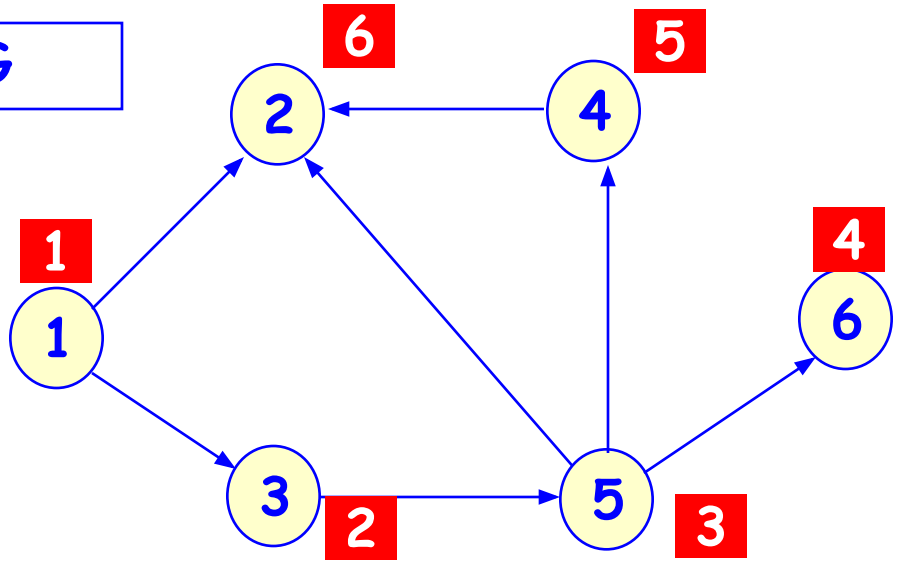
	1	2	3	4	5	6
ot	1	3	5	6	4	2

Ordenação Topológica num DAG





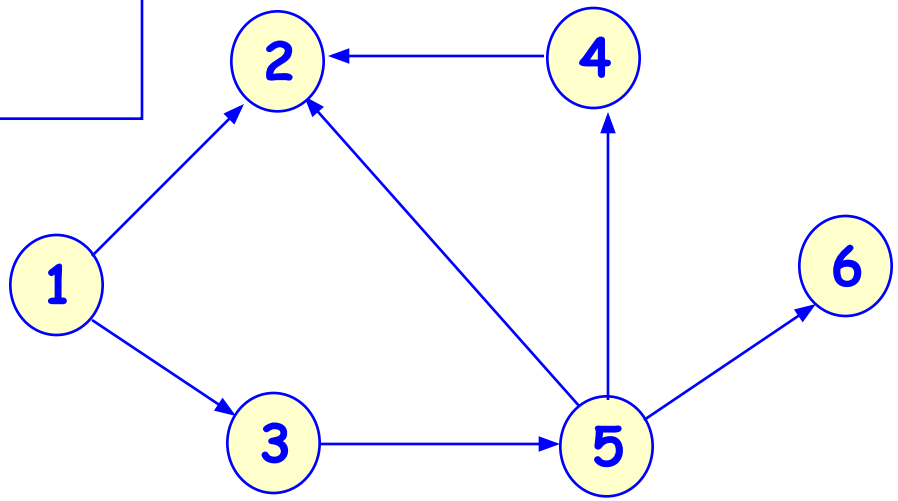
Ordenação Topológica num DAG




```
BP(u,v);
    Marcar v
    para vizinhos w de v:
        se w não marcado:
            BP(v,w)
     ot[os--] ← v

os ← n
Desmarcar vértices
para i ← 1 até n incl.:
    se i não marcado:
        BP(i,i)
```

Ordenação Topológica num DAG - Solução II



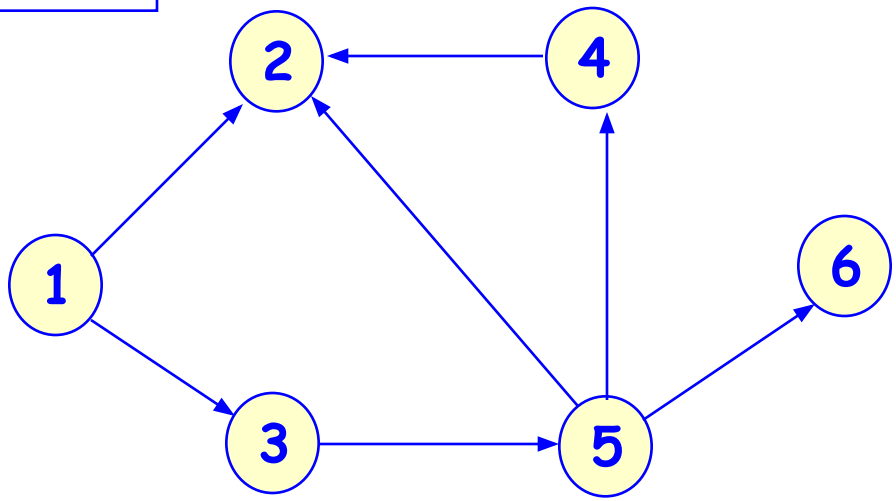
Uma ordenação topológica é uma ordenação dos vértices tal que se v_i vem antes de v_j, então não existe aresta de v_j para v_i.

Ordenação Topológica:

```
esvaziar Q
enfilar vértices com grau de entrada 0
enquanto (Q não vazia):
    v ← vértice da frente de Q
    colocar v na ordenação
    para vizinhos w de v:
        diminuir 1 do grau de entrada(w)
        se grau de entrada(w) = 0:
            Enfilar (w)
    desenfilar v
```

Ordenação Topológica num DAG - Solução II

Situação da Fila da Ordenação topológica:

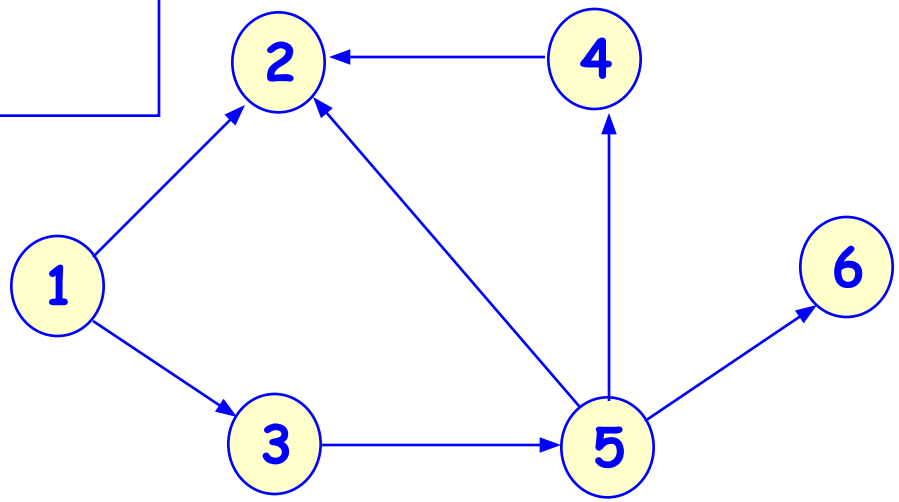


Demonstração da sequência de entrada na Fila:

v	GE (grau de entrada)
1	0
2	3 → 2 → 2 → 1 → 0
3	1 → 0
4	1 → 1 → 1 → 0
5	1 → 1 → 0
6	1 → 1 → 1 → 0

Fila: 1 3 5 4 6 2

Ordenação Topológica num DAG - Solução II



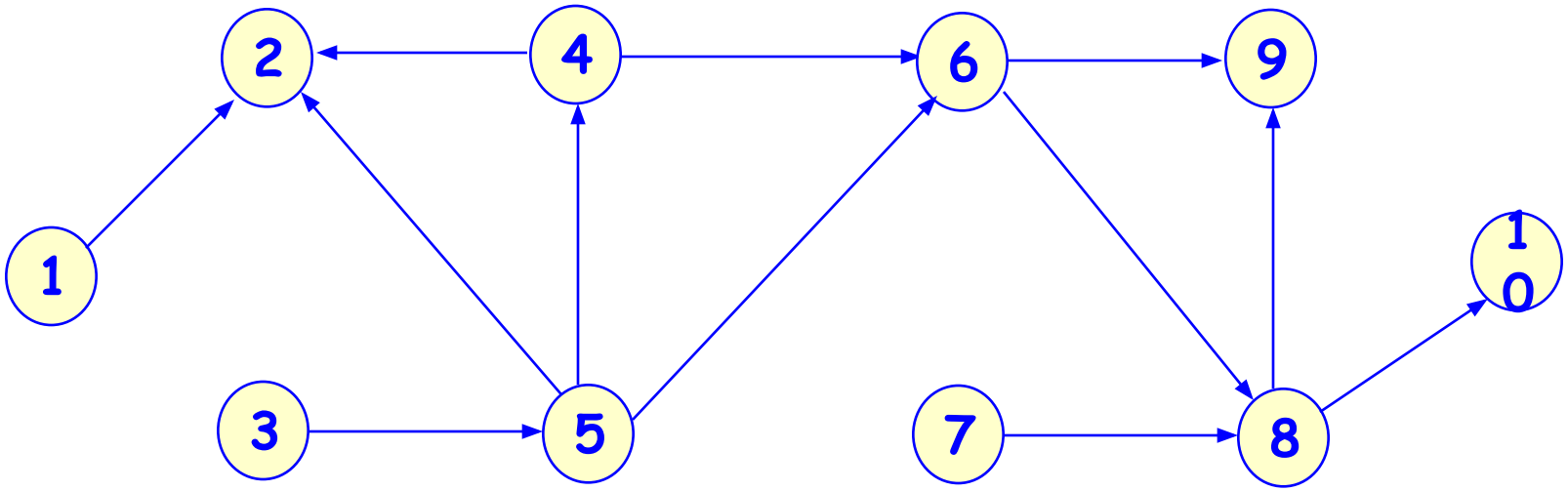
Situação da Fila da Ordenação topológica:

Graus Entrada					
1	2	3	4	5	6
0	3	1	1	1	1
0	2	0	1	1	1
0	2	0	1	0	1
0	1	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Fila					
1					
1	3				
1	3	5			
1	3	5	4	6	
1	3	5	4	6	2
1	3	5	4	6	2
1	3	5	4	6	2

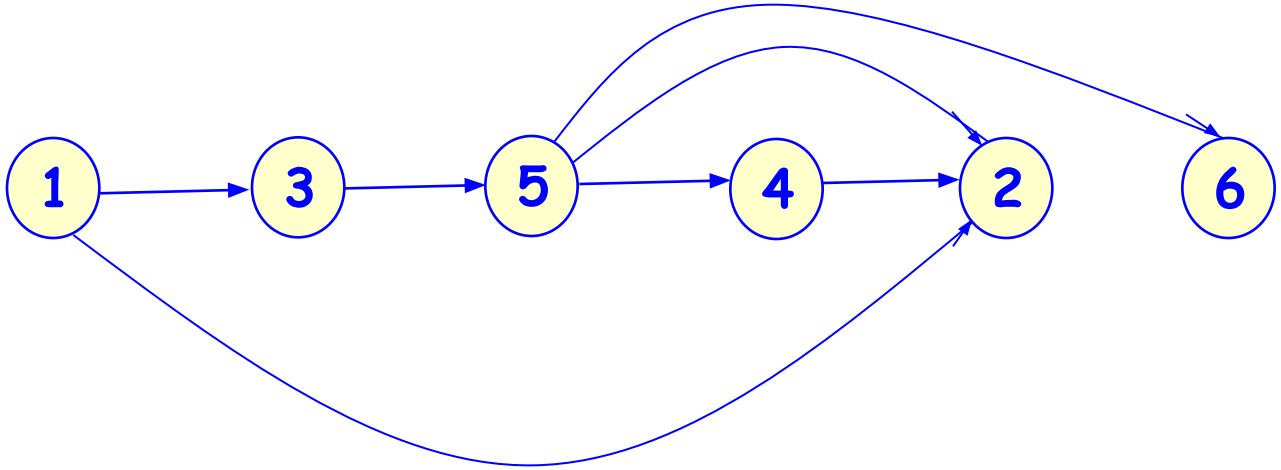
Ordenação Topológica num DAG

EXS10: Mostrar uma ordenação topológica para o grafo abaixo, usando as duas soluções:



Digrafos - Ordenação Topológica num DAG
Caminho máximo em um DAG

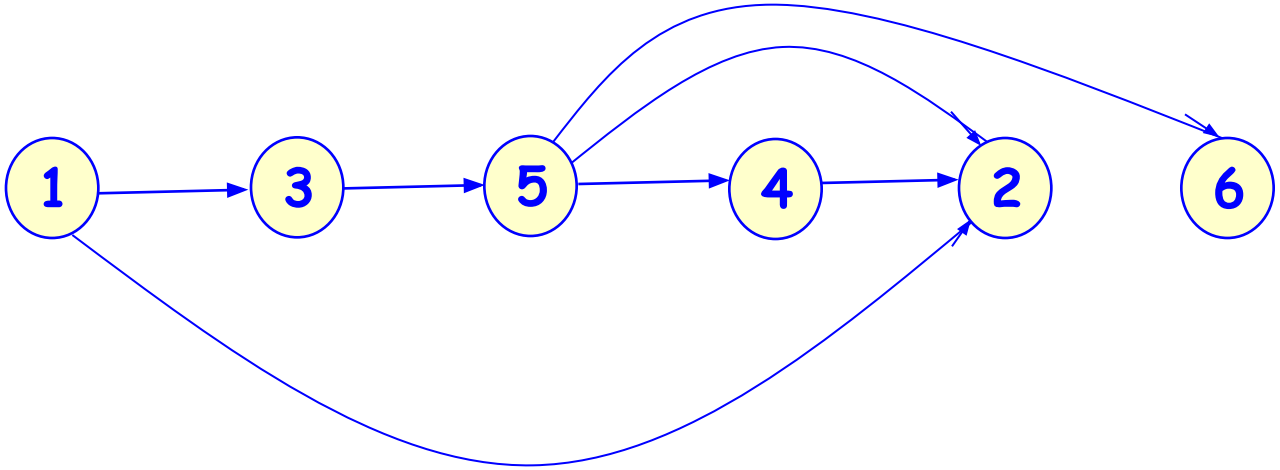
A partir da ordenação topológica pode-se obter o maior caminho em um digrafo, determinando gulosamente, na ordem inversa da ordenação, qual a distância máxima de cada vértice a um sumidouro do digrafo.



No digrafo acima, da direita para a esquerda, determinamos, gulosamente, as distâncias máximas de cada vértice a um sumidouro.

ot	1	3	5	4	2	6
Dm	4	3	2	1	0	0

Digrafos - Ordenação Topológica num DAG
Caminho máximo em um DAG



ot	1	3	5	4	2	6
Dm	0	0	0	0	0	0
	0	0	1	0	0	0
	1	0	1	1	0	0
	1	0	2	1	0	0
	1	3	2	1	0	0
	4	3	2	1	0	0
	4	3	2	1	0	0

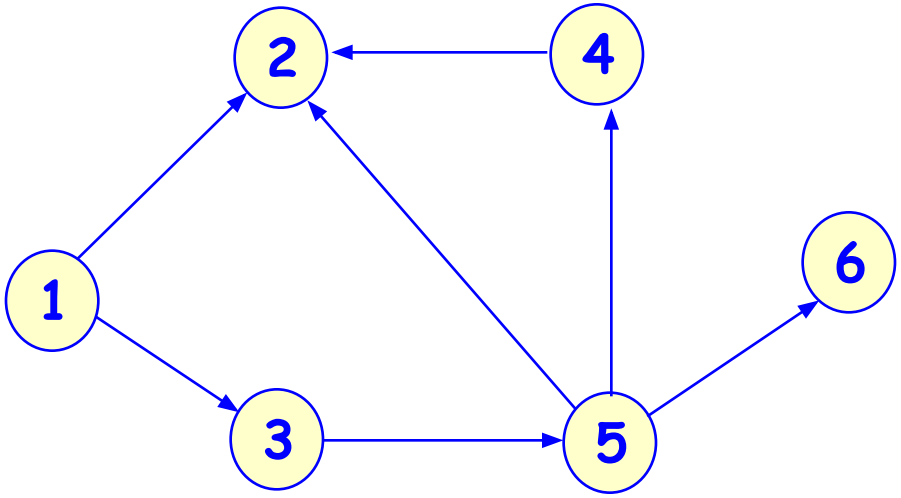
Digrafos - Ordenação Topológica num DAG - maior caminho

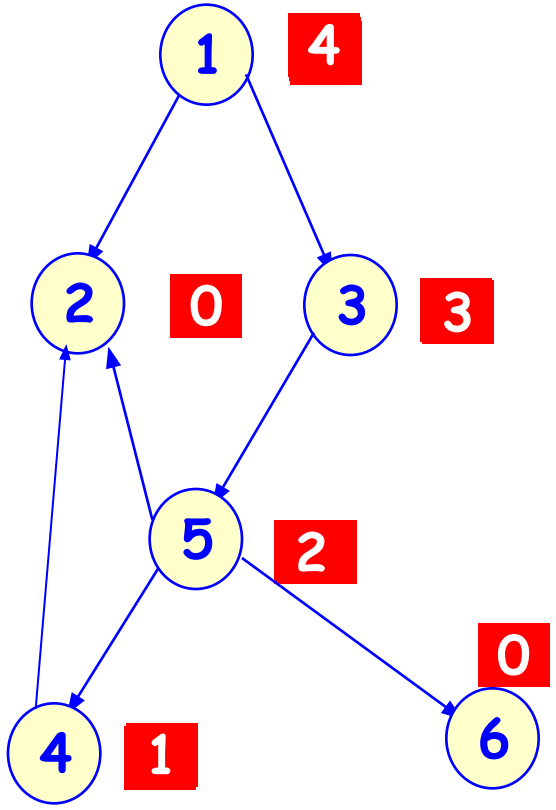
Ordenação Topológica com maior caminho:

```
Imprmc (u):
    escreve (u)
    para w sucessor de u:
        se Dm[w] = (Dm[u]-1):
            Imprmc(w);   parar;
OrdTop(u,v):
    marcar v
    para vizinhos w  de   v:
        se  w  não  marcado:
            OrdTop(v,w)
        Dm[v] ← max (Dm[v], Dm[w]+1)
    ot[os--] ← v; mc ← max (mc, Dm[v]);

os ← n;  mc ← 0;  desmarcar vértices;   Dm[*] ← 0;
para  i ← 1 até  n incl.:
    se  i não marcado:
        OrdTop(i,i)
para  i ← 1 até  n incl.:
    se  Dm[i] = mc:
        Imprmc (i)
```

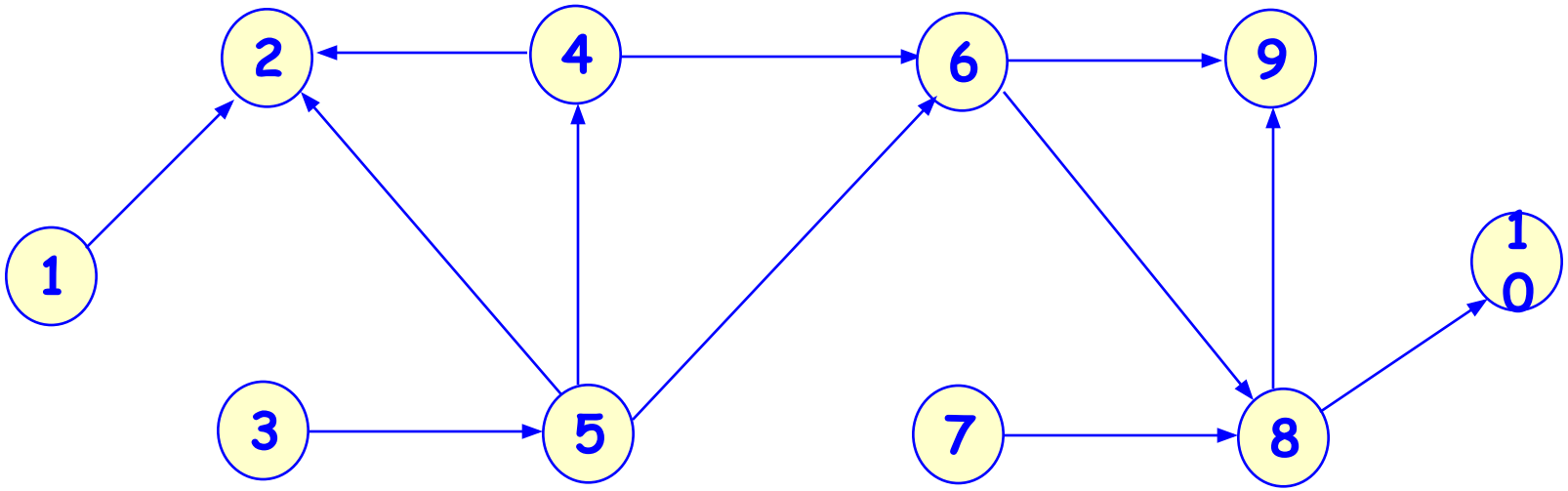
Digrafos - Ordenação Topológica num DAG - maior caminho





Digrafos - Ordenação Topológica num DAG

EXS11: Aplicar o algoritmo de determinação do maior caminho ao digrafo:



Digrafos - Ordenação Topológica num DAG

Aplicação: Dado um conjunto de caixotes, todos de mesma altura e dimensões l x c, determinar a pilha de maior altura, colocando um caixote totalmente sobre o outro:

	l	c
1	5	4
2	8	3
3	9	5
4	7	1
5	10	4
6	12	5
7	6	6
8	4	3

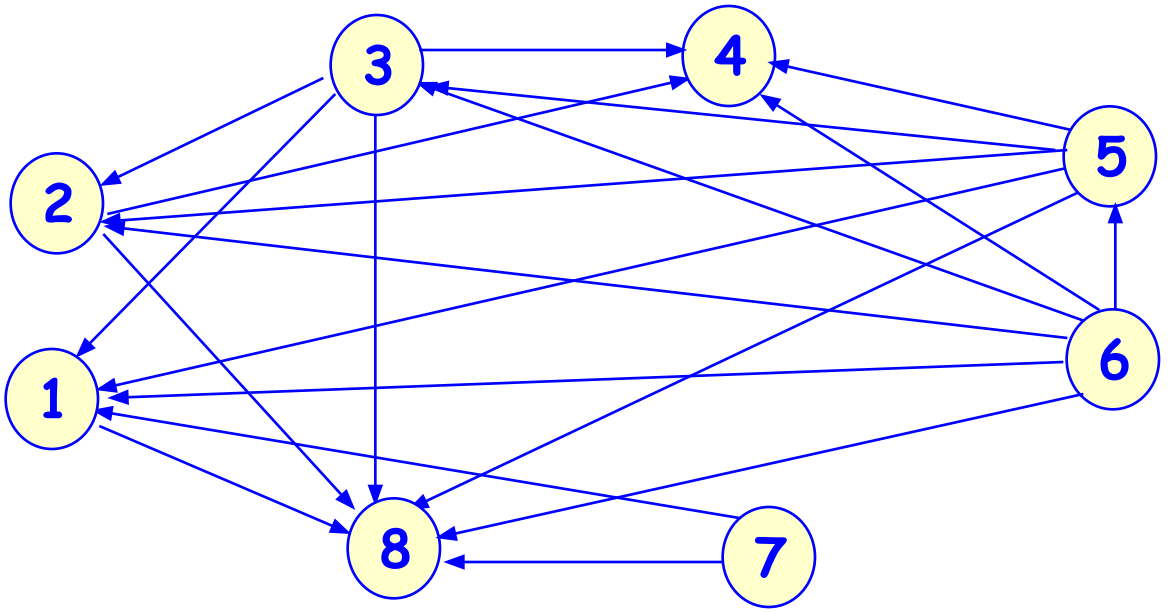
Solução:

Criar um DAG representando as relações de sobreposição, determinar uma ordenação topológica e encontrar o maior caminho no DAG, a partir da ordenação, de trás para frente.

Ordenação Topológica num DAG

Aplicação Dado um conjunto de caixotes, todos de mesma altura e dimensões l x c, determinar a pilha de maior altura, colocando um caixote totalmente sobre o outro:

	l	c
1	5	4
2	8	3
3	9	5
4	7	1
5	10	5
6	12	5
7	6	6
8	4	3



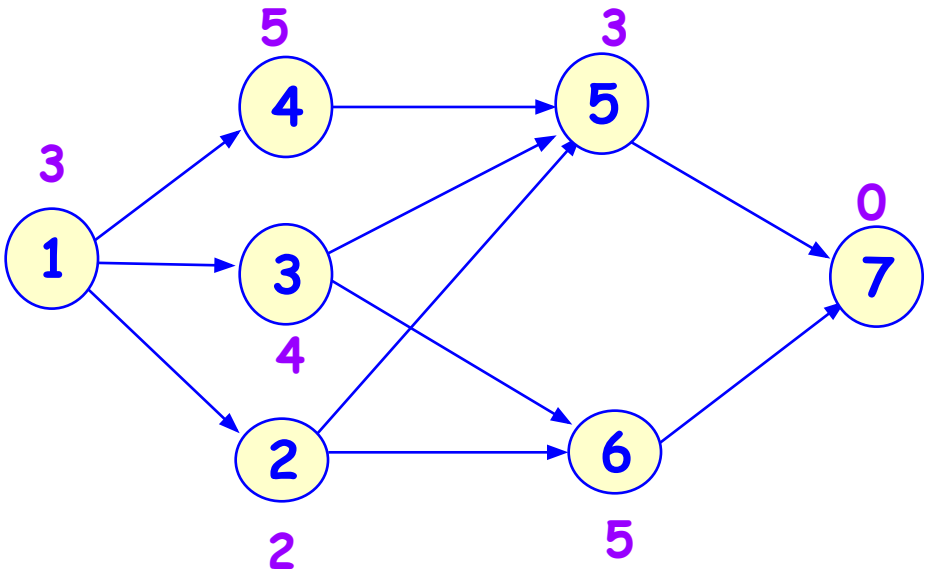
Orden.Top.	6	7	5	3	2	1	4	8
Maior cam.	5	3	4	3	2	2	1	1

Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

Há várias ferramentas para controle de projetos, onde a rede de atividades é descrita por um DAG (?!!), onde os vértices são as tarefas a serem realizadas, as arestas as dependências entre as tarefas e, para cada tarefa, é dada sua duração. O método CPM responde a várias perguntas do tipo:

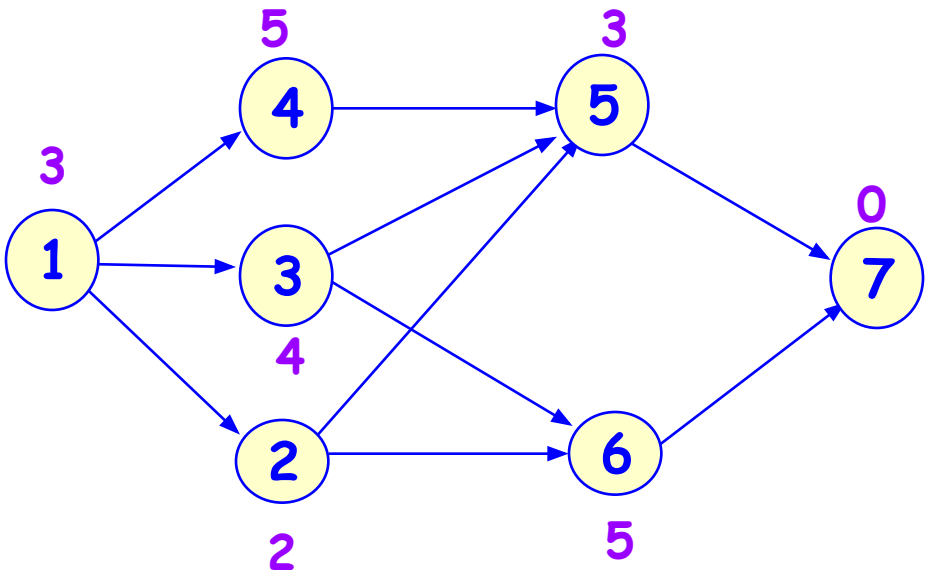
a) Qual a duração mínima do projeto?
b) Quando cada tarefa deve ser iniciada para não atrasar o projeto?



Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

Para o DAG criado, determinam-se TMC (tempos mais cedo), TMT (tempos mais tarde) e F (Folga). É criado um vértice artificial, com duração 0, para indicar o fim do projeto.



Passo 1: Faz-se a ordenação topológica: 1 - 4 - 2 - 3 - 6 - 5-7

Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

TMC (tempos mais cedo para começar as atividades) são calculados usando diretamente a ordenação topológica:

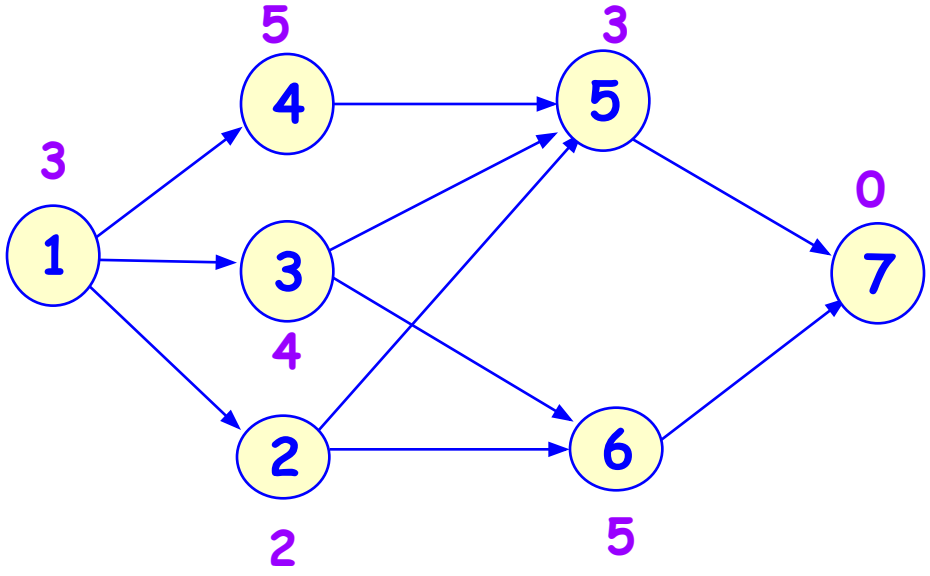
para os vértices v fonte:

TMC(v) = 1

para os demais vértices v:

TMC(v) = max(TMC(z) + C(z)), para z dos quais v é vizinho.

Obs: para os vértices sumidouros, só há interesse em calcular tempos de término, que devem ser considerados como um dia a menos que os de começo.



v	TMC
1	1
4	1+3= 4
2	1+3 = 4
3	1+3 = 4
6	max(4+2, 4+4) = 8
5	max(4+2, 4+4, 4+5) = 9
7	max(8+5, 9+3) = 13

Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

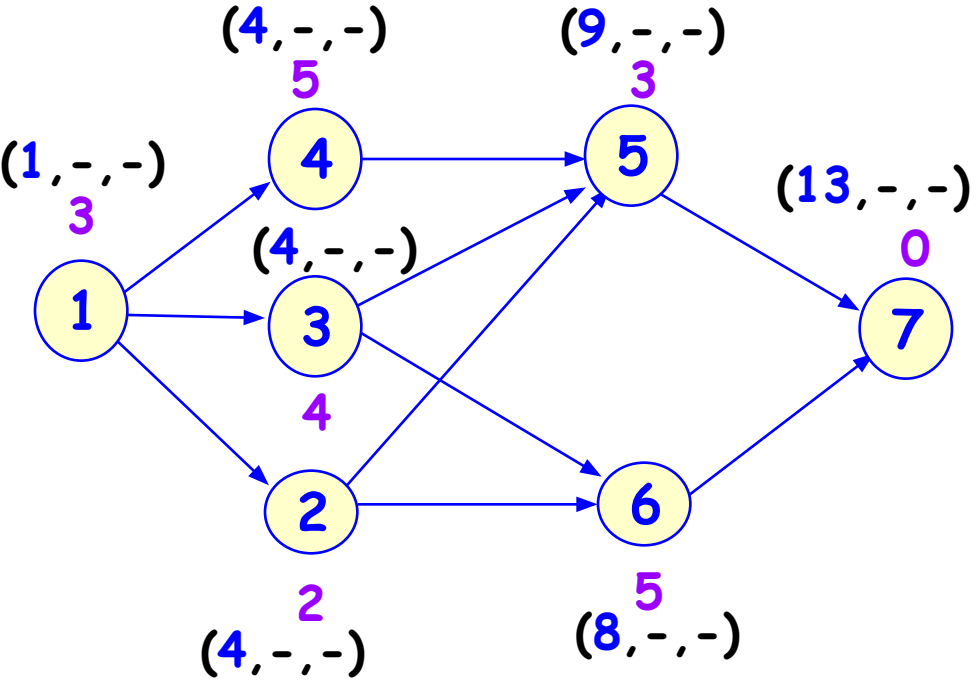
TMT (tempos mais tarde para começar as atividades) são calculados usando inversamente a ordenação topológica:

Para o vértices s sumidouro:

TMT(s) = TMC(s)

Para os demais vértices v:

TMT(v) = min(TMT(z)) – C(v), para vizinhos z de v.

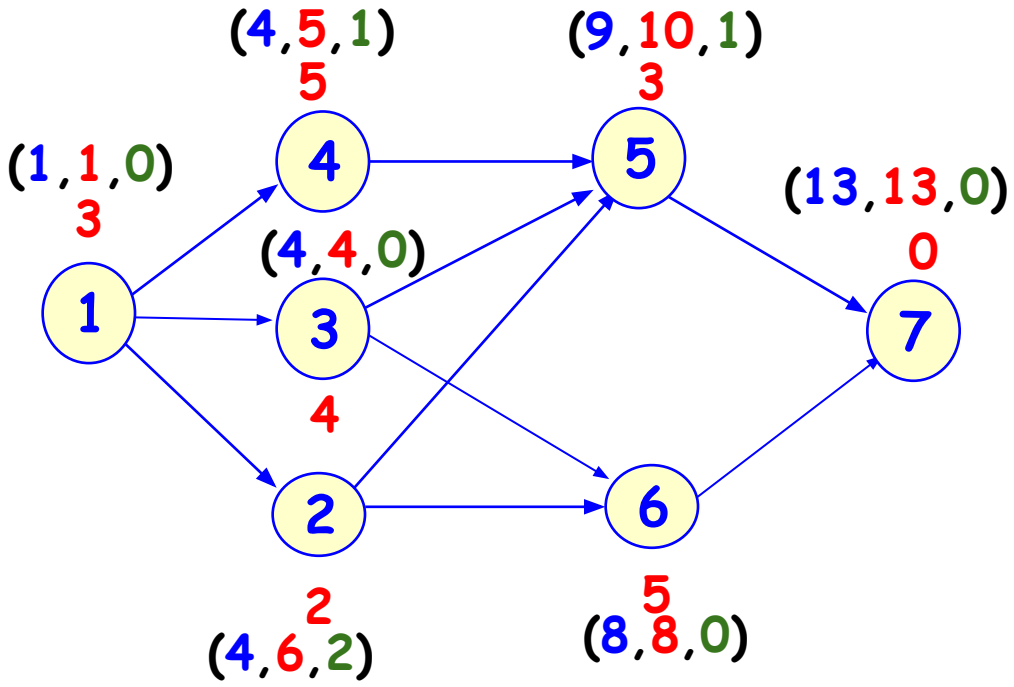


v	TMT
7	13
5	13-3=10
6	13-5=8
3	min(10, 8)-4 = 4
2	min(10, 8)-2 = 6
4	10-5 = 5
1	min(6, 4, 5) - 3 = 1

Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

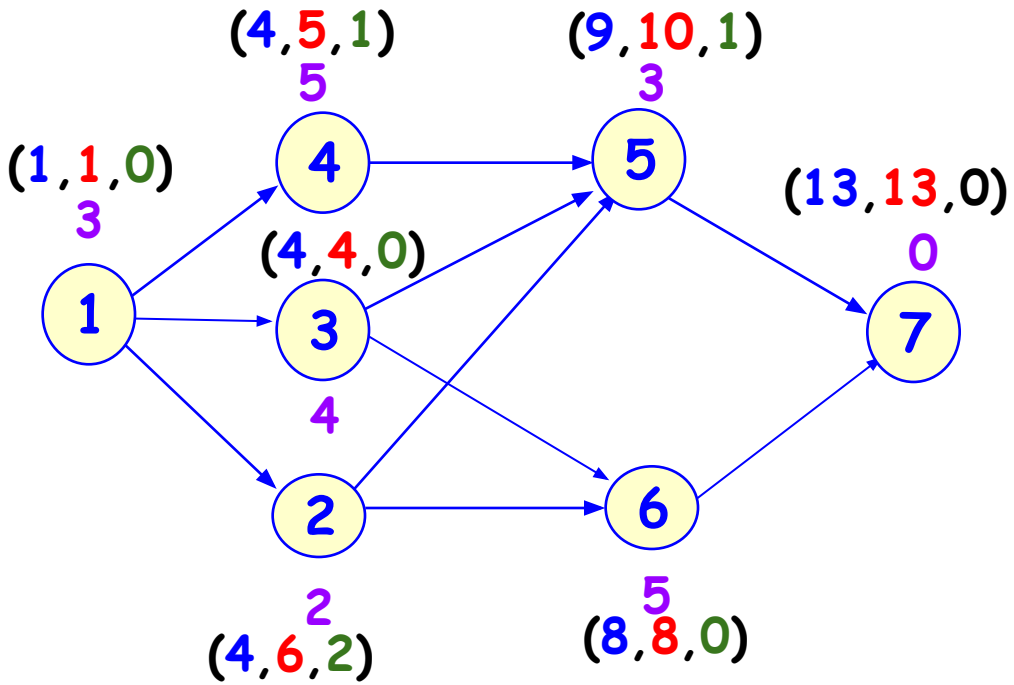
As Folgas são dadas pelas diferenças entre os TMT e os TMC.



Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

O "Caminho crítico" é formado pelos vértices v com Folga = 0.



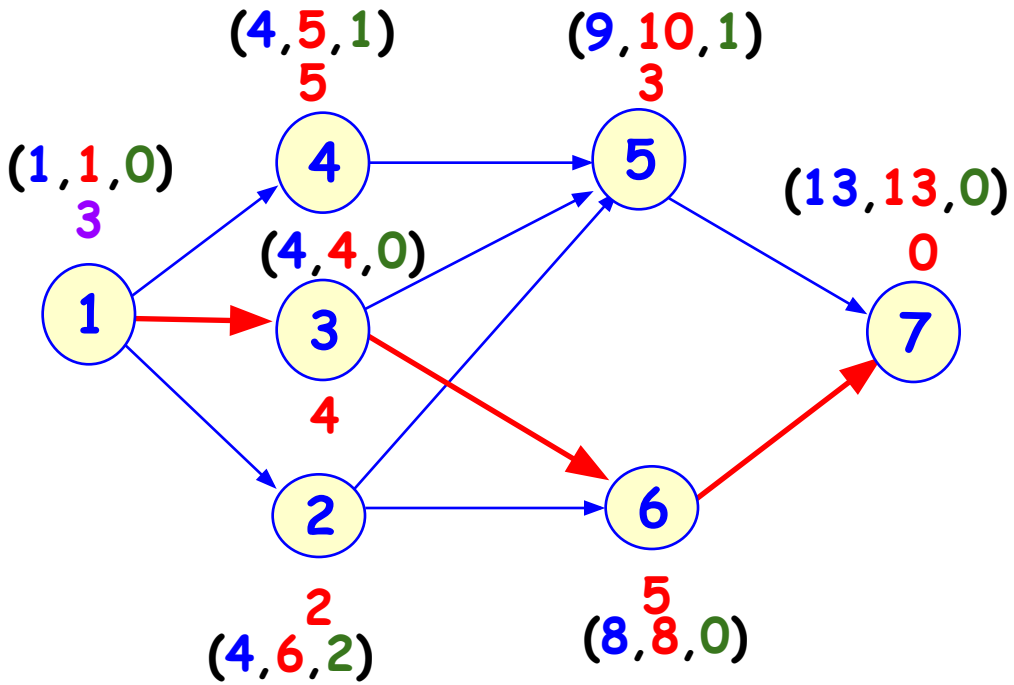
Neste projeto, o caminho crítico é dado por 1 – 3 – 6 – 7. A duração mínima do projeto é 12 (13 – 1).

Obs: O "Caminho crítico" é um subgrafo do digrafo original, não necessariamente um caminho.

Digrafos - Ordenação Topológica num DAG

Aplicação: controle de projetos usando CPM (Critical Path Method)

O "Caminho crítico" é formado pelos vértices v com TMC(v) = TMT(v). A folga é a diferença entre esses valores.



Neste projeto, o caminho crítico é dado por 1 – 3 – 6 – 7. A duração mínima do projeto é 12 (13 – 1).

Obs: O "Caminho crítico" é um subgrafo do digrafo original, não necessariamente um caminho.

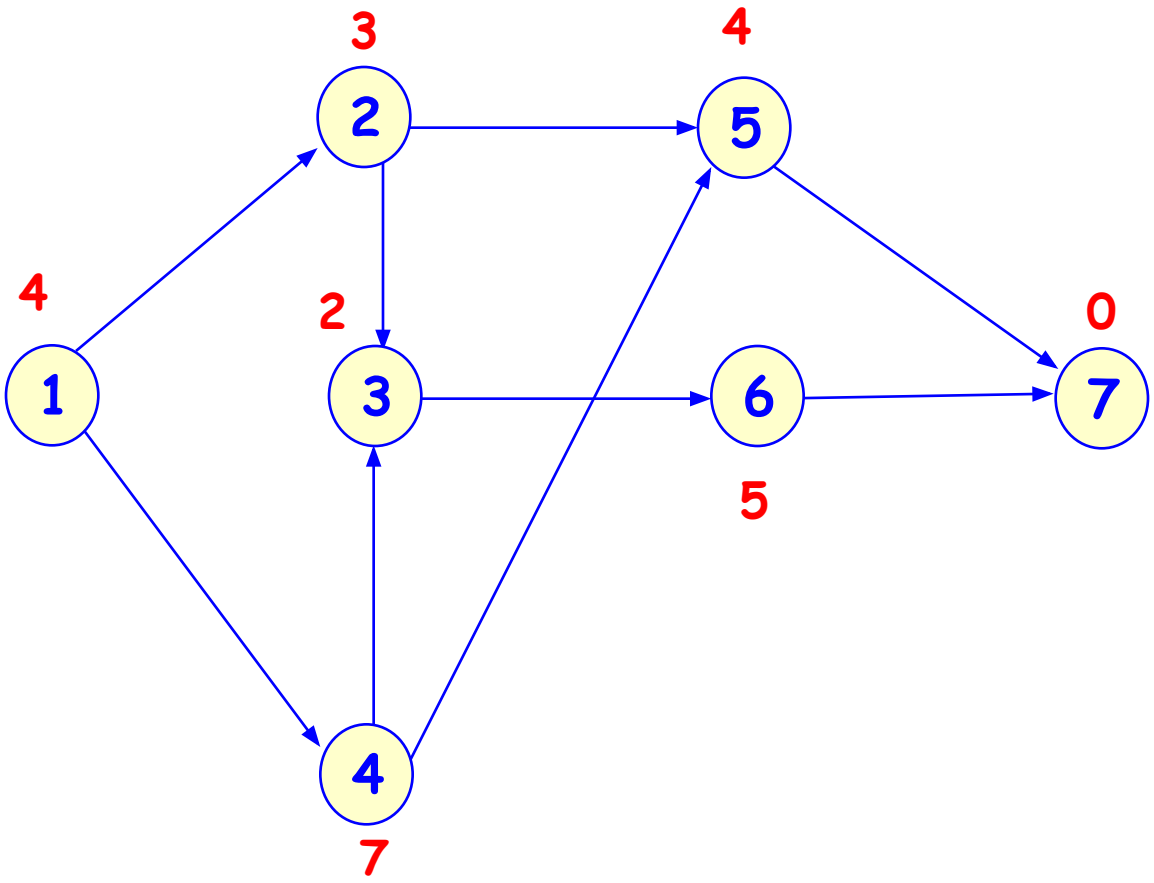
Digrafos - CPM

```
PreencheTMC():
     para i ← 1 até n incl.:
          v ← ot[i]
          TMC[v]] ← 1
          para z ← 1 até n incl.:
               se  v é vizinho de z:
                    TMC[v] ← max(TMC[v], TMC[z]+C[z])
PreencheTMT():
     para i ← n..1 incl.:
          v ← ot[i]
          TMT[v] ← Infinito
          para z ← 1 até n incl.:
               se  z é vizinho de v:
               TMT[v] ← min(TMT[v], TMT[z])
          se  TMT[v] = Infinito:
               TMT[v] ← TMC[v]
          senão:
               TMT[v] ← TMT[v]-C[v]
          Folga[v] ← TMT[v] - TMC[v]

OrdTop();
PreencheTMC();
PreencheTMT();
```

Digrafos - Ordenação Topológica num DAG

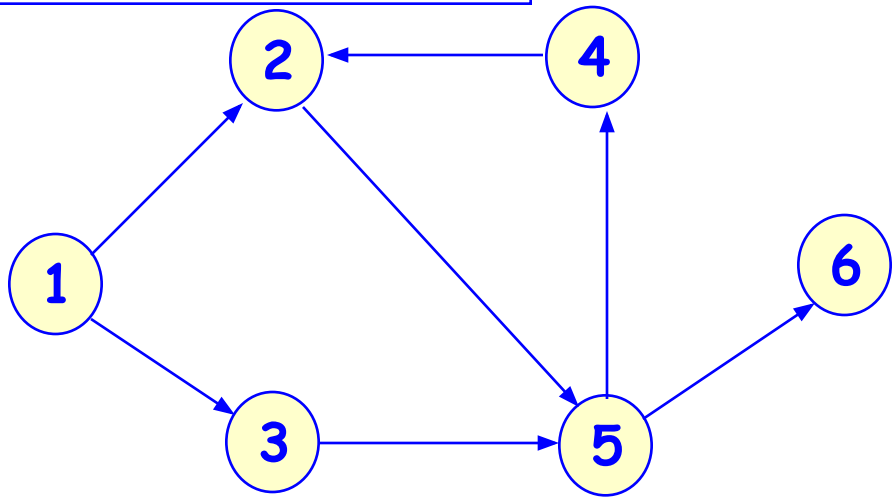
EXS12: Mostrar, em três etapas, a determinação do Caminho Crítico na rede de atividades abaixo:



Digrafos - Componentes fortemente conexos

Conectividade em digrafos:

A conectividade em digrafos muda, em relação a grafo simples, porque não temos simetria. No exemplo mostrado, o vértice 1 alcança todos os demais e não é alcançado por nenhum deles.



Componentes fortemente conexos (cfc):

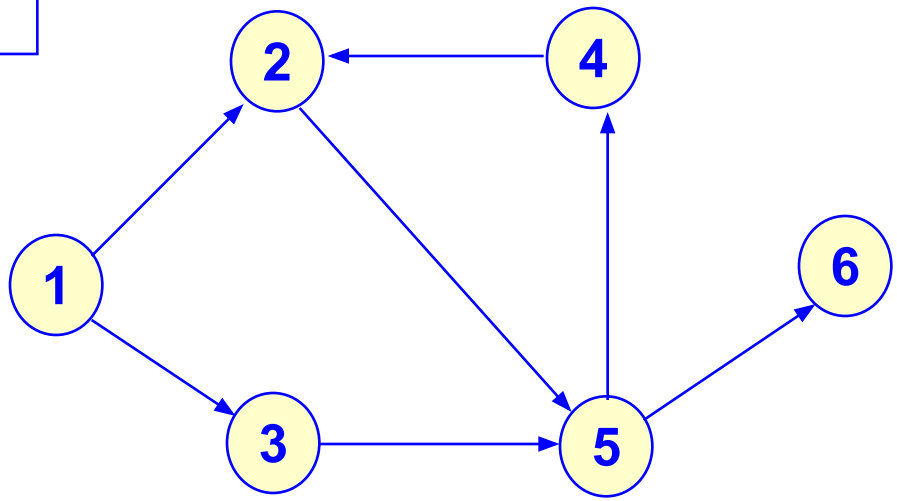
Subdigrafos maximais tal que todos vértices de cada componente alcançam os demais vértices desse componente.

No digrafo do exemplo, os 4 componentes fortemente conexos são: {1}, {3}, {2, 4, 5}, {6}.

Há vários algoritmos lineares para se determinar os cfc. Apresentaremos dois algoritmos: Tarjan e Kosaraju.

Alcançabilidade em digrafos

A Alcançabilidade em um digrafo é bem diferente daquela em grafos simples.
Um digrafo é **fortemente conexo** quando há caminhos de u para v e de v para u, para qualquer par de vértices u, v.



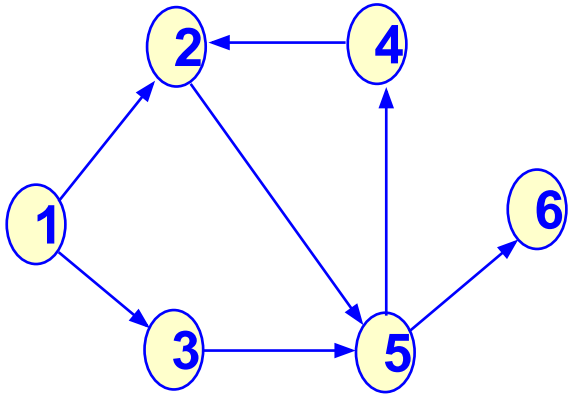
A potência p da matriz de adjacências mostra todos os caminhos de tamanho p.

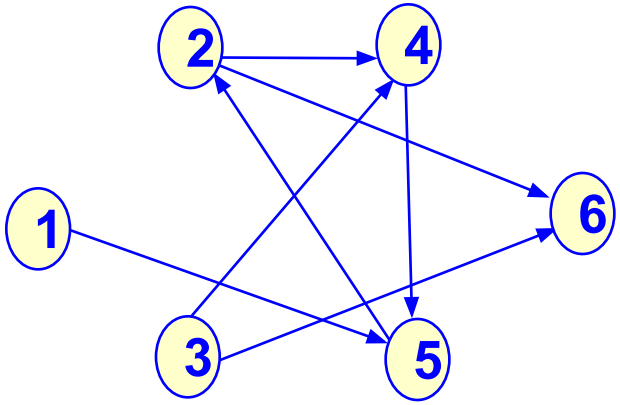
E =

0	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	1	0	1
0	0	0	0	0	0

Alcançabilidade em digrafos

Elevando-se a matriz de adjacências à potência 2, obtem-se um digrafo relativo aos caminhos de tamanho 2.





$E =$

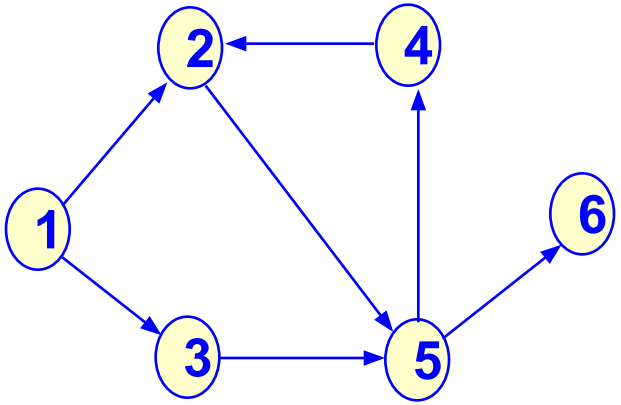
0	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	1	0	1
0	0	0	0	0	0

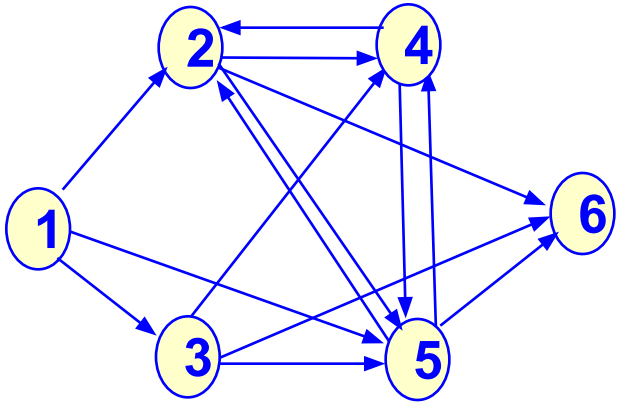
$E^2 =$

0	0	0	0	1	0
0	0	0	1	0	1
0	0	0	1	0	1
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	0	0	0

Alcançabilidade em digrafos

Preenchendo a diagonal com 1's e elevando-se a matriz de adjacências à potência 2, obtem-se um digrafo relativo aos caminhos de tamanhos 1 e 2.





$E' =$

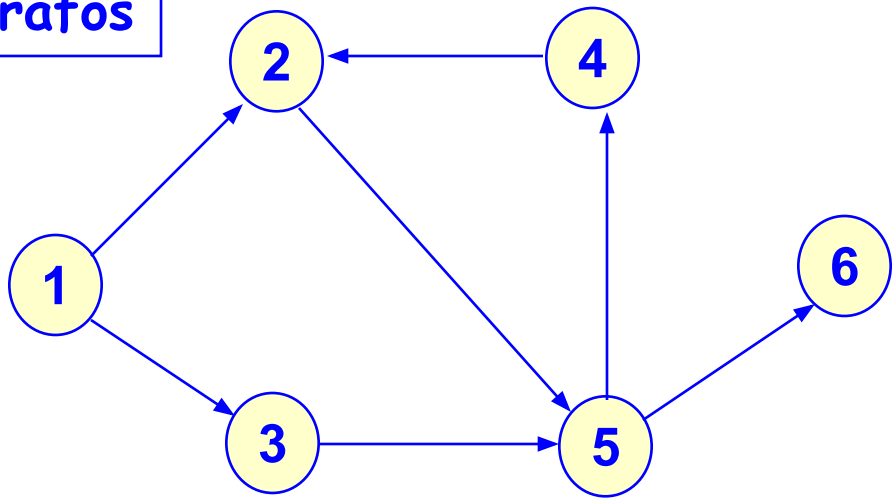
1	1	1	0	0	0
0	1	0	0	1	0
0	0	1	0	1	0
0	1	0	1	0	0
0	0	0	1	1	1
0	0	0	0	0	1

$E'^2 =$

1	1	1	0	1	0
0	1	0	1	1	1
0	0	1	1	1	1
0	1	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

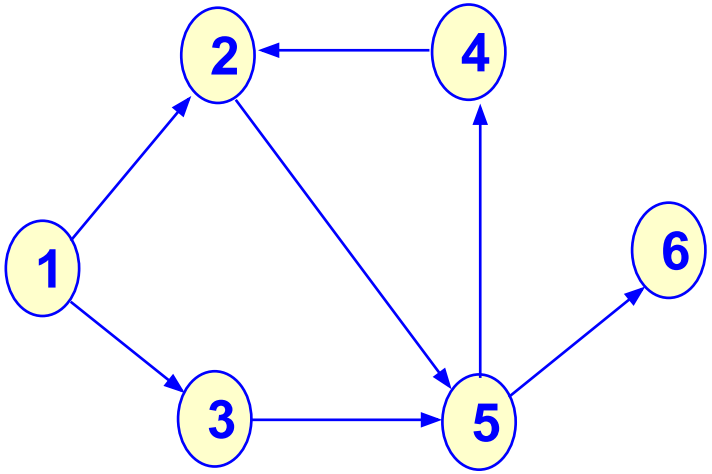
Fechamento transitivo em digrafos

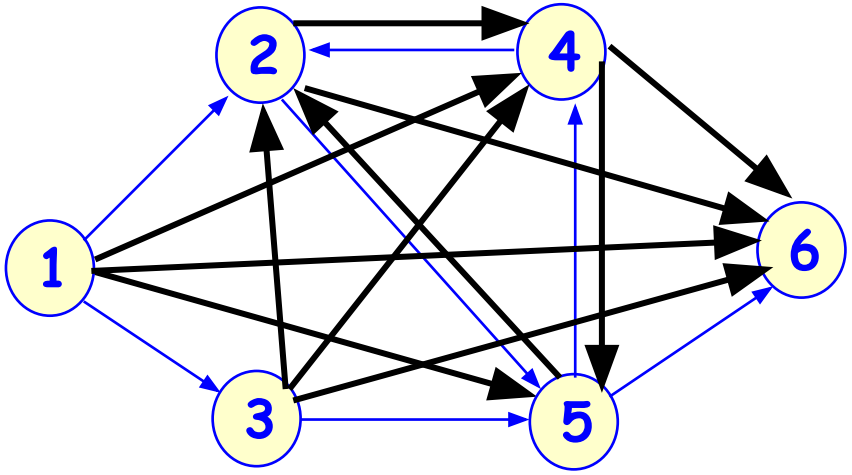
O fechamento transitivo de um digrafo dá a alcançabilidade de todos os vértices.
Corresponde a elevar a matriz modificado à potência n.



0	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	1	0	1
0	0	0	0	0	0

Fechamento transitivo – representado do lado direito



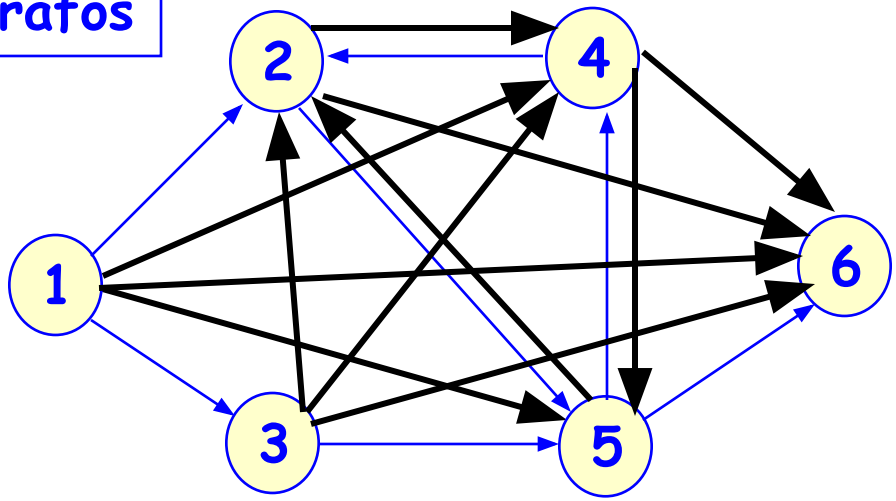


0	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	1	0	1
0	0	0	0	0	0

1	1	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
0	1	0	1	1	1
0	1	0	1	1	1
0	0	0	0	0	1

Fechamento transitivo em digrafos

O fechamento transitivo é equivalente a se adicionar todas as arestas entre vértices alcançáveis + autoloops.

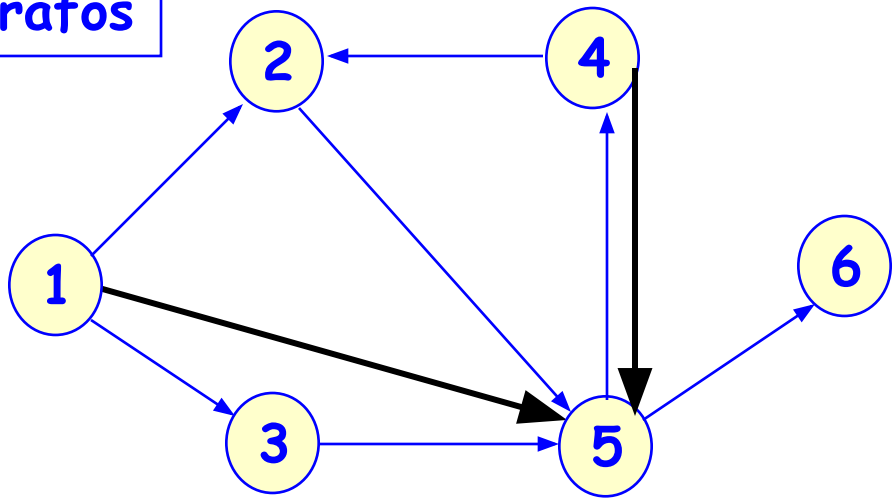


0	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	1	0
0	1	0	0	0	0
0	0	0	1	0	1
0	0	0	0	0	0

1	1	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
0	1	0	1	1	1
0	1	0	1	1	1
0	0	0	0	0	1

Fechamento transitivo em digrafos

Um algoritmo com complexidade $O(n^3)$.



Algoritmo de Warshall

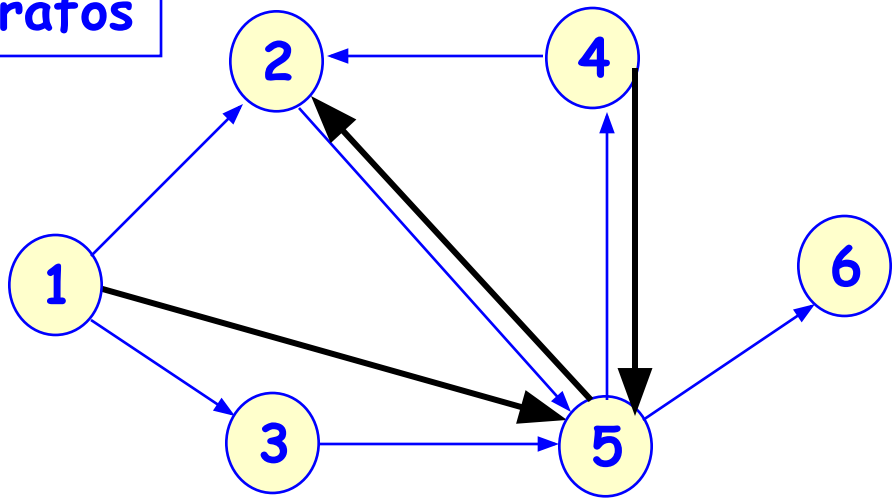
```
para i ← 1 até n incl.:
    A[i, i] ← 1
para k ← 1 até n incl.:
    para i ← 1 até n incl.:
        se A[i, k] = 1:
            para j ← 1 até n incl.:
                se A[k, j] = 1:
                    A[i, j] ← 1
```

k= 1

k= 2

Fechamento transitivo em digrafos

Um algoritmo com complexidade $O(n^3)$.



Algoritmo de Warshall

```
para i ← 1 até n incl.:
    A[i, i] ← 1
para k ← 1 até n incl.:
    para i ← 1 até n incl.:
        se A[i, k] = 1:
            para j ← 1 até n incl.:
                se A[k, j] = 1:
                    A[i, j] ← 1
```

k= 1

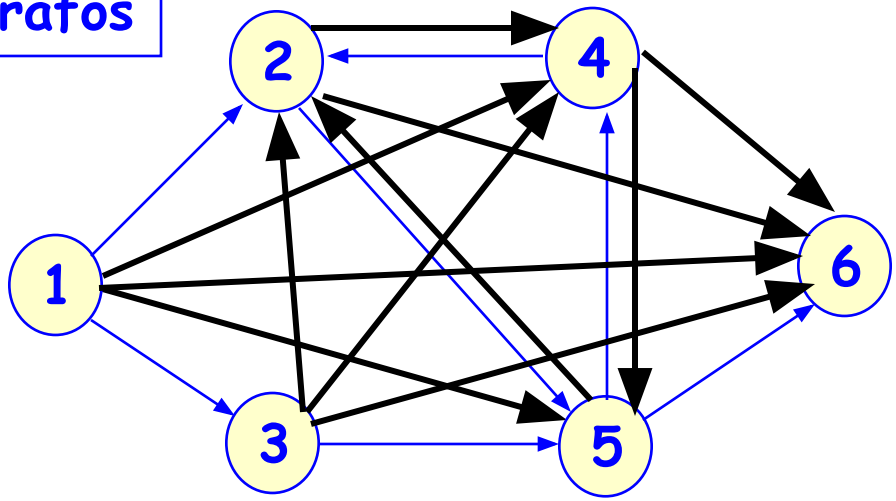
k= 2

k= 3

k= 4

Fechamento transitivo em digrafos

Um algoritmo com complexidade $O(n^3)$.



Algoritmo de Warshall

```
para i ← 1 até n incl.:
    A[i, i] ← 1
para k ← 1 até n incl.:
    para i ← 1 até n incl.:
        se A[i, k] = 1:
            para j ← 1 até n incl.:
                se A[k, j] = 1:
                    A[i, j] ← 1
```

k= 1

k= 2

k= 3

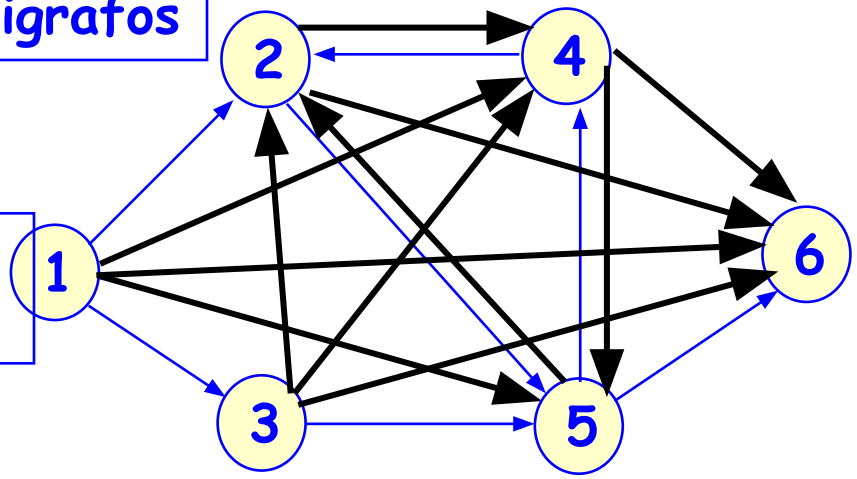
k= 4

k= 5

k= 6

Fechamento transitivo em digrafos

Um algoritmo com complexidade $O(n^3)$.

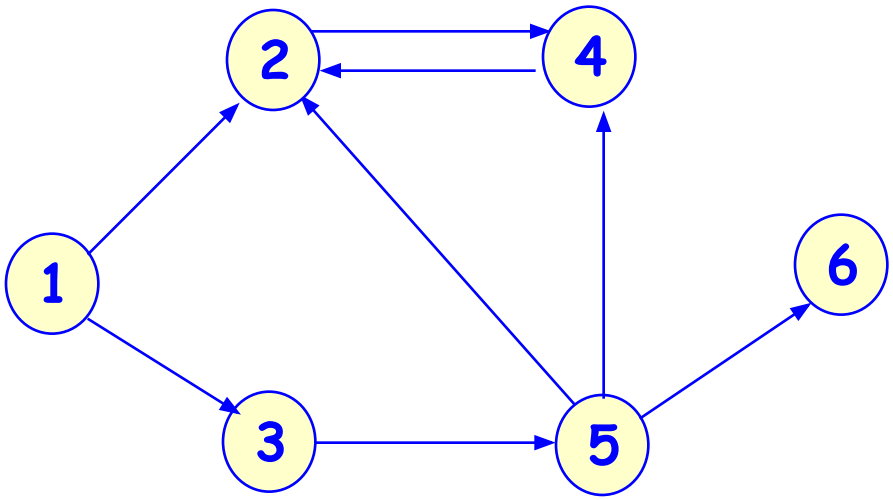


Algoritmo de Warshall – Porque funciona?

```
para i ← 1 até n incl.:
      A[i, i] ← 1
para k ← 1 até n incl.;
      para i ← 1 até n incl.:
            se A[i, k] = 1:
                  para j ← 1 até n incl.:
                        se A[k, j] = 1:
                              A[i, j] ← 1
```

A cada passo do loop em k, determina caminhos entre i e j que só utilizam vértices até o índice k, excetuando-se i e j.

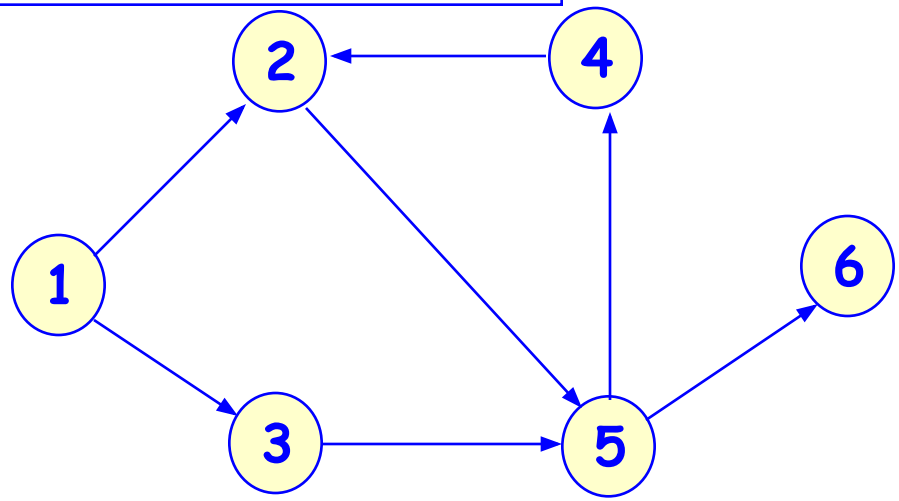
Fechamento transitivo em digrafos



EXS9: Mostrar o digrafo do fechamento transitivo e a matriz do fechamento, para o digrafo acima

Digrafos - Componentes fortemente conexos

Algoritmo de Tarjan

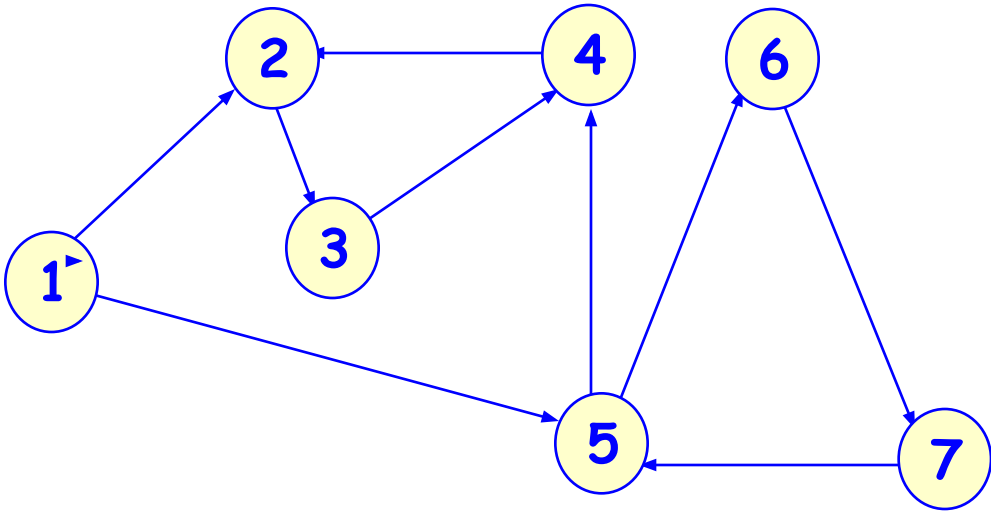


Algoritmo p/ determinação dos componentes forte/ conexos:

1. BP, determinação de low e pre e empilhamento.

2. Retirada da pilha quando low = pre

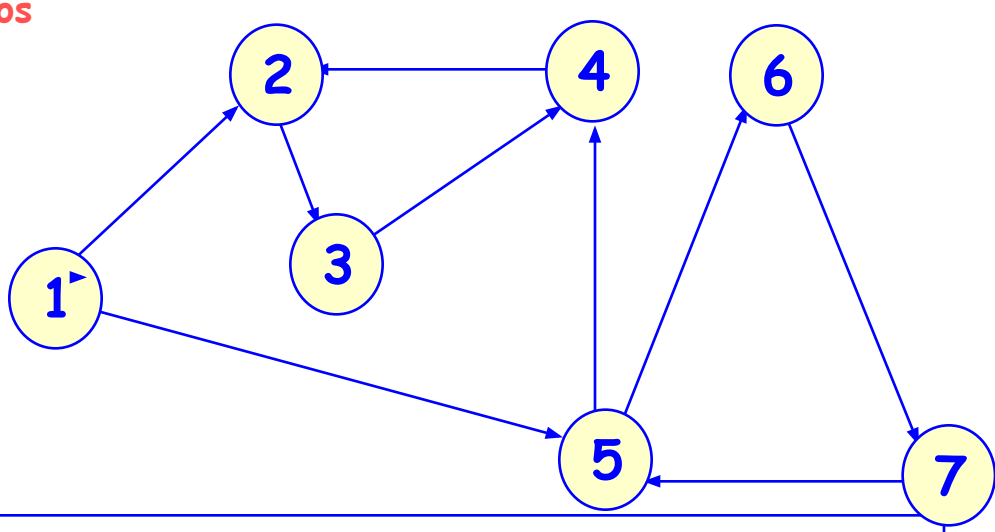
Digrafos -CFC- Tarjan



Externamente:

low[*] ← pre[*] ← vis[*] ←0;
esvaziar pilha; cpre ← 0;
para i ← 1 até n incl.:
    se pre[i] = 0:
        CFC(i)

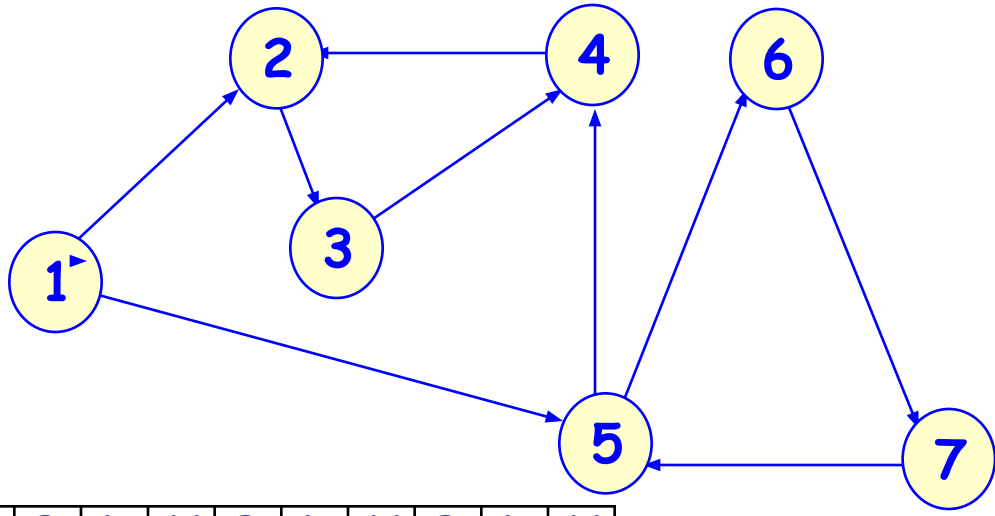
Digrafos -CFC- Tarjan




```
CFC (v):
    pre[v] ← ++cpre;  low[v] ← cpre; vis[v] ← 1; PUSH(v);
    para vizinhos w de v:
        se  pre[w]=0:
            CFC(w)
         se  vis[w]=1:
            low[v] ← min(low[v], low[w])

    se  low[v] = pre[v]:
        escrever ('Novo componente:')
        enquanto (1):
            p ← POP();  escrever (p);  vis[p] ← 0;
            se  p = v:
                parar loop
```

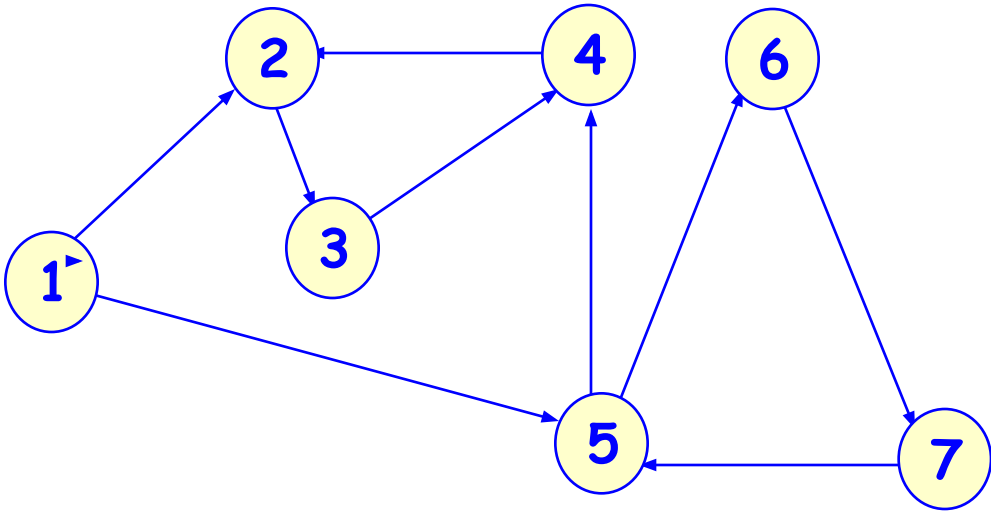
Digrafos - CFC - Tarjan



	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V
v																					
1	1	1	1																		
2	1	1	1	2	2	1															
3	1	1	1	2	2	1	3	3	1												
4	1	1	1	2	2	1	3	3	1	4	4	1									
4	1	1	1	2	2	1	3	3	1	4	2	1									
3	1	1	1	2	2	1	3	2	1	4	2	1									
2	1	1	1	2	2	0	3	2	0	4	2	0									
5	1	1	1										5	5	1						
6	1	1	1										5	5	1	6	6	1			
7	1	1	1										5	5	1	6	6	1	7	7	1

C1: 4 3 2

Digrafos - CFC - Tarjan



	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V	P	L	V
7	1	1	1										5	5	1	6	6	1	7	5	1
6	1	1	1										5	5	1	6	5	1	7	5	1
5	1	1	1										5	5	0	6	5	0	7	5	0
1	1	1	1																		

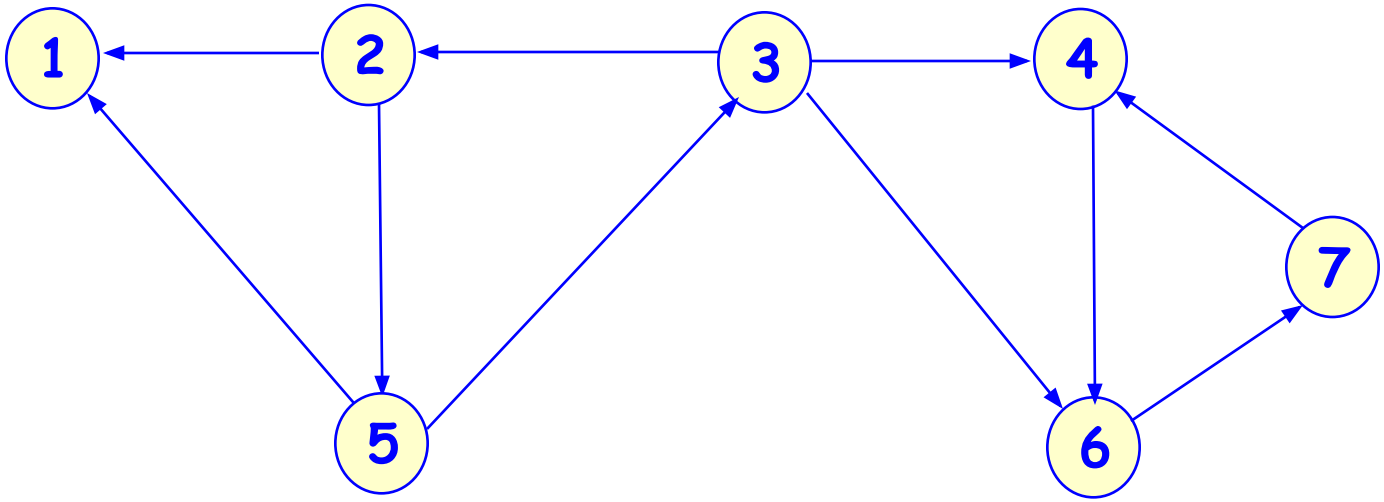
C2:7 6 5

C3:1

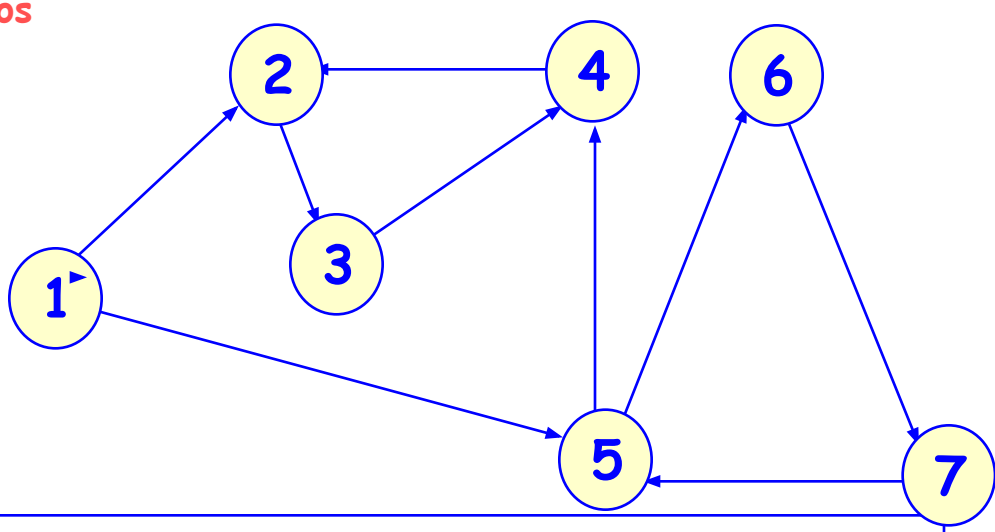
Componentes:{1}, {2, 3, 4}, {5, 6, 7}

Digrafos - Componentes Fortemente Conexos

Exercício: Aplicar o algoritmo de Tarjan ao digrafo abaixo:



Digrafos -CFC- Tarjan

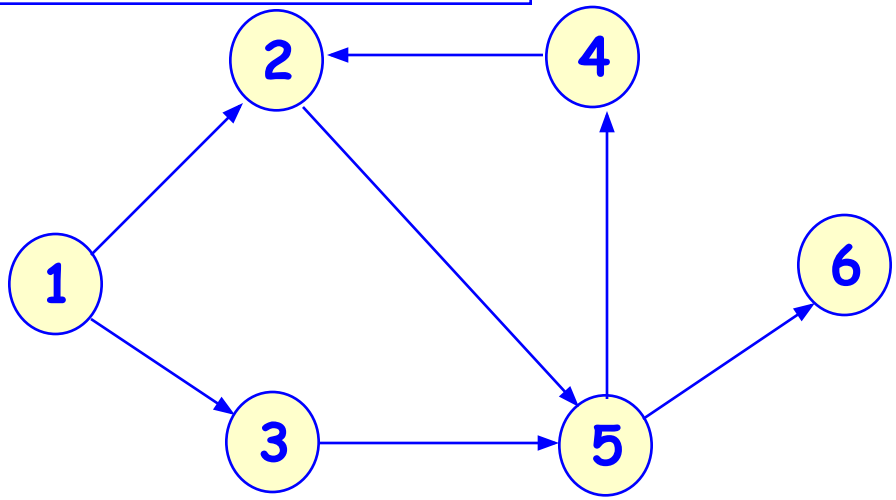


Porque o algoritmo de Tarjan funciona?

Todos os vértices de um CGC são descendentes, na árvore de profundidade, do primeiro vértice v do componente que entra na busca. Consequentemente, todos os vértices w do componente terão low[w] = low[v] = pre[v]. Se v tiver como descendentes apenas vértices do componente, todos estarão na pilha no momento da saída de v na pilha e o componente será relatado corretamente. Se v tiver apenas um outro componente como descendente, cujo primeiro vértice a entrar na busca foi u, então os vértices desse componente eentrarão na busca e na pilha imediatamente após u. Quando u sair da busca, os vértices correspondente também saem da pilha, restando apenas os vértices do componente de v. Portanto, quando v sair da busca estarão na pilha apenas os vértices do componente. E Assim sucessivamente...

Digrafos - Componentes fortemente conexos

Algoritmo de Kosaraju:
Faz BP no digrafo e no digrafo transposto.O digrafo transposto é o digrafo original com todas as arestas invertidas. Daí obtem-se os componentes fortemente conexos(cfc).



Esboço do algoritmo de Kosaraju:

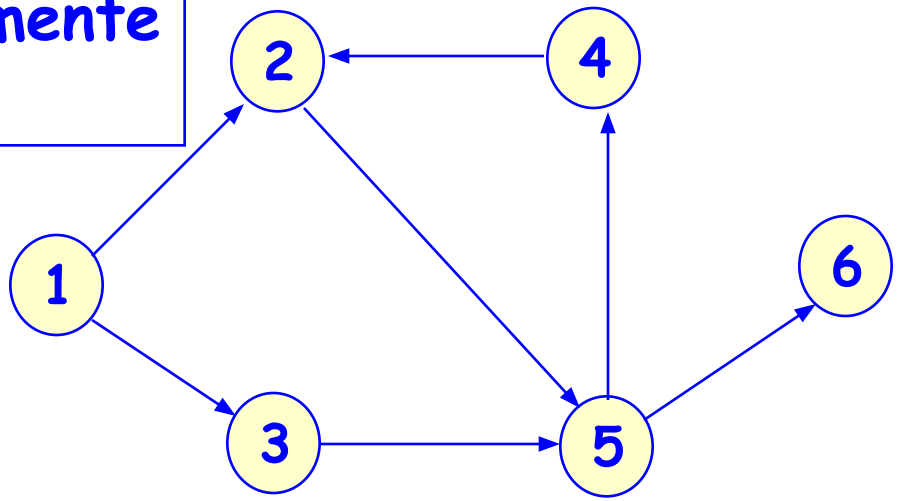
1. "Ordenação Topológica" no digrafo, obtendo vetor ot.

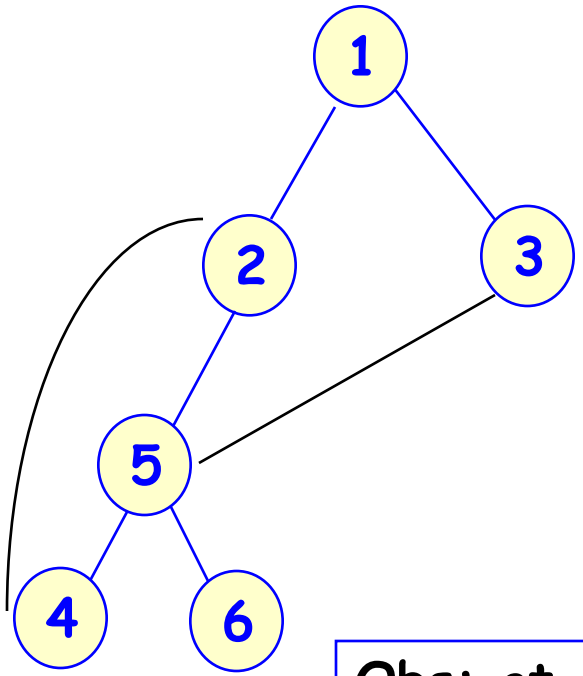
2. Busca em Profundidade no digrafo transposto(D^T), obedecendo a ordem de ot.

3. Cada árvore de profundidade obtida é um cfc.

Grafos – Componentes fortemente conexos

1ª etapa: OT no digrafo.

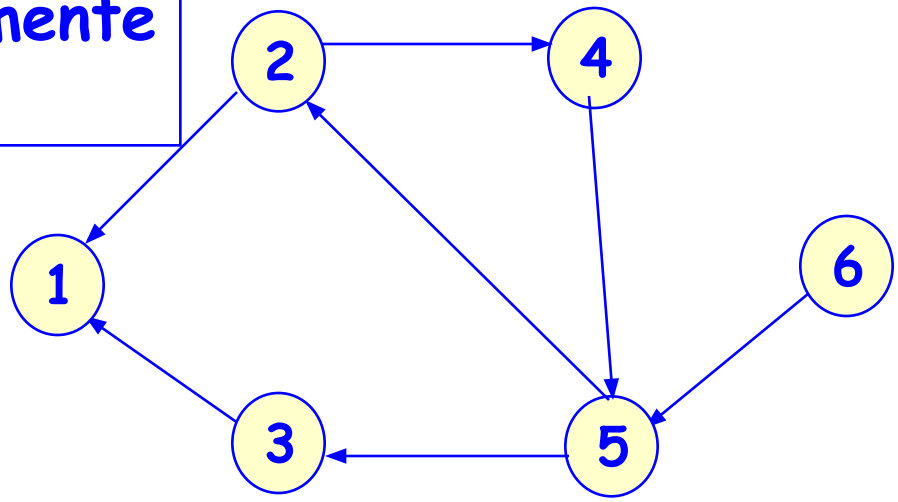




	ot
1	1
2	3
3	2
4	5
5	6
6	4

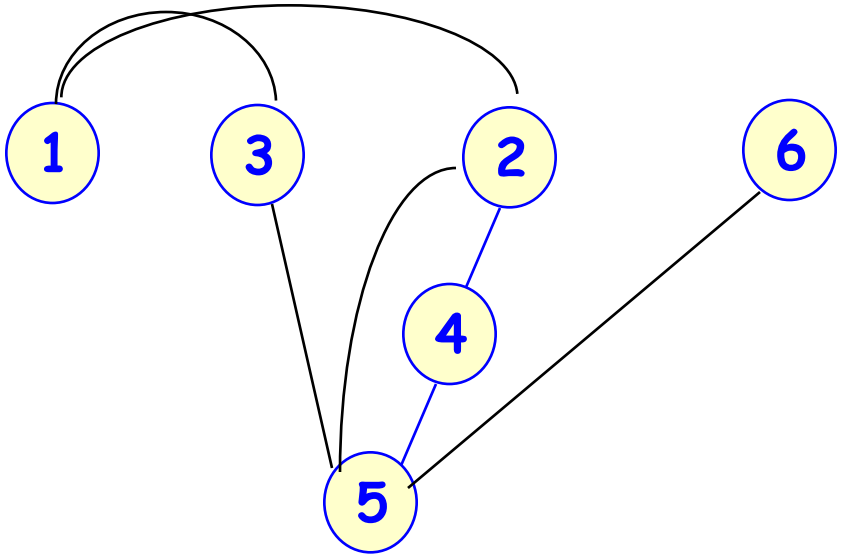
Obs: ot é o resultado da "ordenação topológica". (vetor de vértices com ordem inversa da saída na busca.

Grafos - Componentes fortemente conexos



2ª etapa: BP no digrafo transposto, seguindo a ordem de ot.

	ot
1	1
2	3
3	2
4	5
5	6
6	4

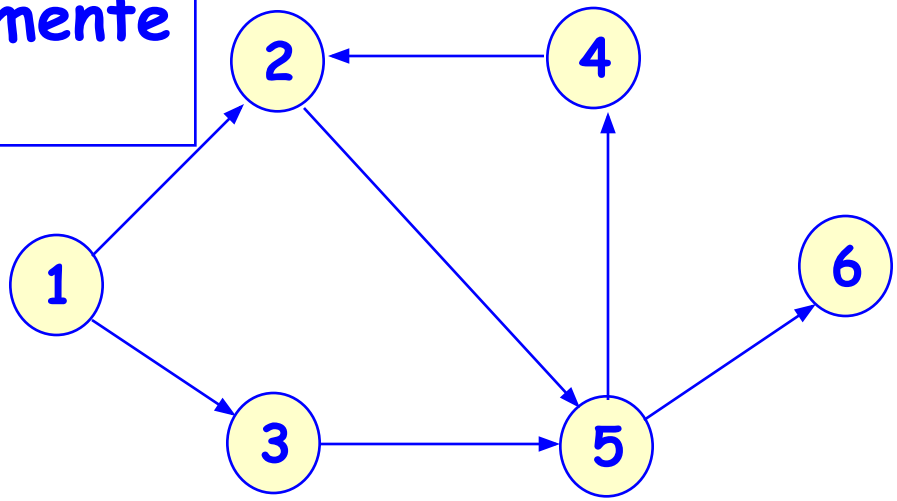


Componentes: {1}, {3}, {2, 4, 5} , {6}

Obs: O digrafo transposto não precisa ser criado quando a representação é por matriz de adjacência. Basta percorrer a coluna da matriz.

Grafos - Componentes fortemente conexos

Ord Top. e BP no digrafo transposto.

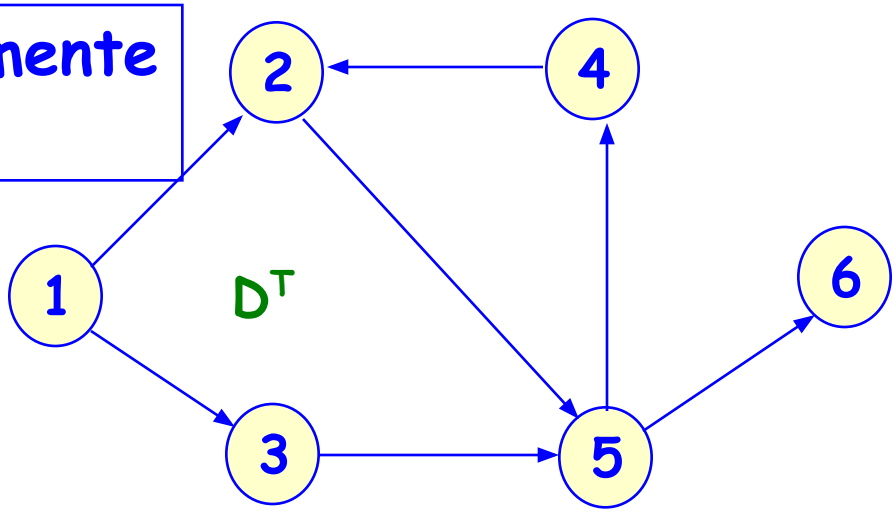



```
OT(u,v):
    marcar v
    para vizinhos w de v:
        se w não marcado:
            OT(v,w)
    ot[os--] ← v

BPT(u,v):
    desmarcar v;  escrever (v);
    para w vizinho de v no digrafo transposto:
        se w marcado:
            BPT(v,w)
```

Grafos - Componentes fortemente conexos

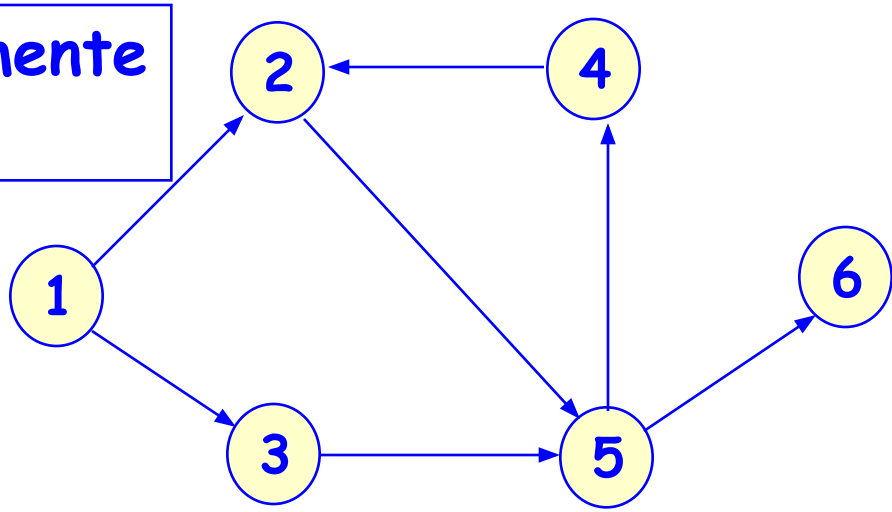
Externamente:




```
os ← n
comp ← 0
desmarcar vértices
para  i ← 1  até  n incl.:
    se  i não marcado:
        OT(i,i)
para  i ← 1  até  n incl.:
    se  ot[i] marcado:
        escrever ('Componente ', ++comp)
        BPT(ot[i],ot[i])
```

Complexidade: $O(n+m)$ ou $O(n^2)$

Grafos - Componentes fortemente conexos



Porque o algoritmo de Kosaraju funciona?

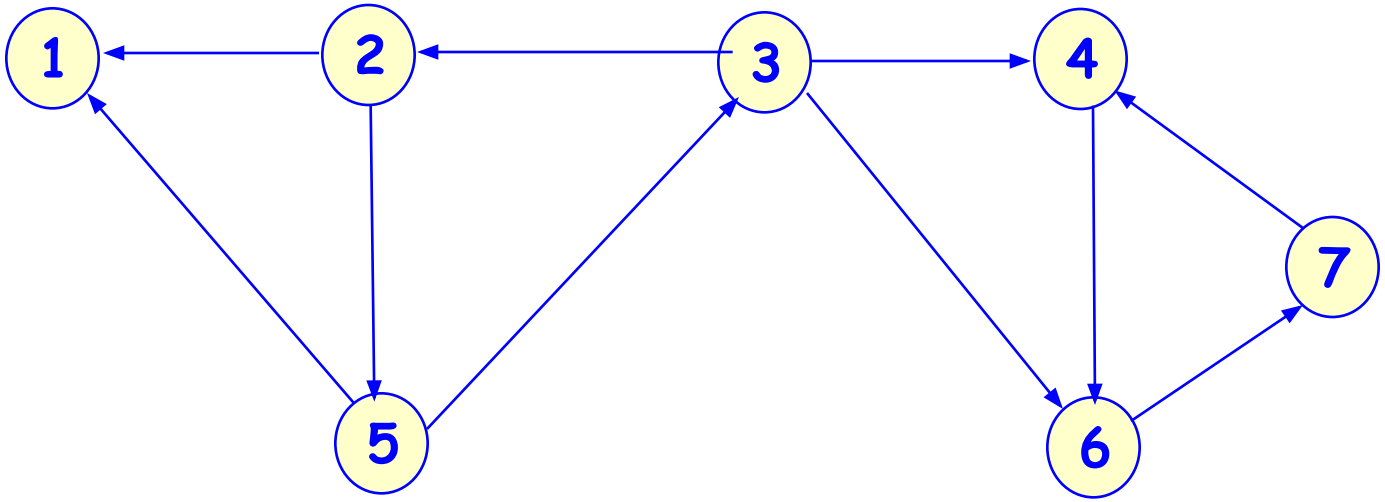
Para provar que o algoritmo é correto, temos que provar que, se os vértices r e s pertencem ao mesmo cfc em D então vão estar na mesma árv. profund. de D^T. Temos tb que provar o inverso, ou seja, se r e s estão na mesma árv. profund de D^T, então pertencem ao mesmo cfc de D.

a) Suponhamos r e s pertencentes a um mesmo cfc, em D. Suponhamos r o vértice com menor ordem de entrada na busca em D^T. Como existe o caminho de s a r em D, então s será descendente de r nessa árvore.

b) Suponhamos s e r na mesma árv. prof. de D^T. Seja t a raiz dessa árvore. Então existe caminho de s para t, em D. Mas também tem que ter caminho de t para s, pois t tem ordem de saida maior que s na busca em D. Então s tem que ser descendente de t nessa árvore, já que existe caminho de s a t. Logo, existe caminho de t para s em D. Portanto, t e s estão no mesmo cfc em D. De forma análoga, t e r estão no mesmo cfc em D. Logo s e r estão no mesmo cfc em D.

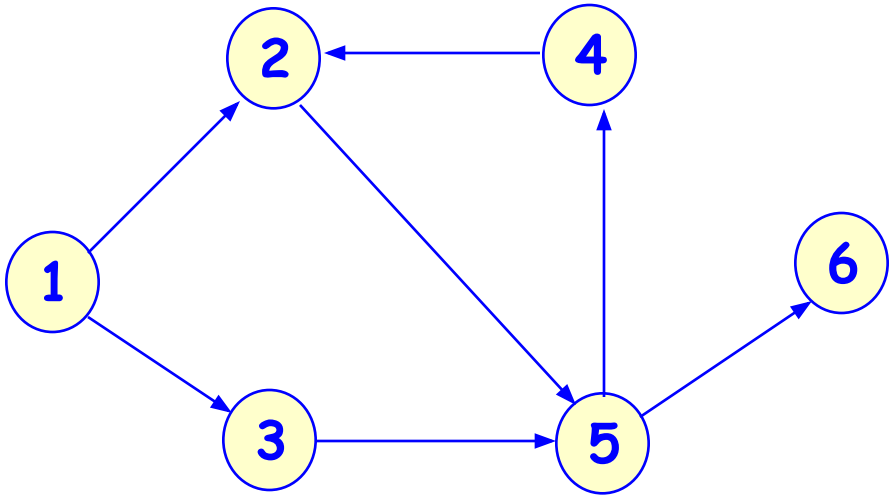
Digrafos - Componentes Fortemente Conexos

Exercício: Aplicar o algoritmo de Kosaraju ao digrafo abaixo:



Grafos – Componentes Fortemente conexos

Problema DOMINÓS:
Dado um grupo de dominós, com a indicação de qual dominó é derrubado por um outro (essas relações estão em um digrafo), determinar o número mínimo de dominós que têm que ser "empurrados" à mão, para que todo o conjunto caia.



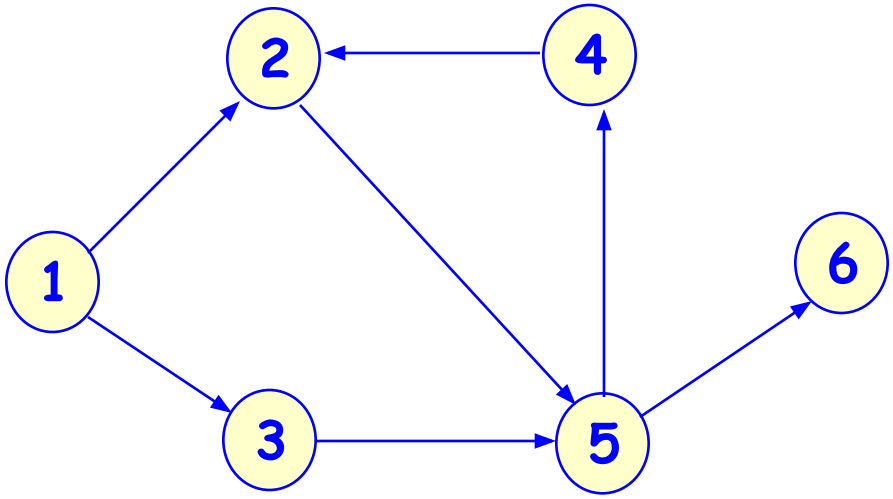
Solução DOMINÓS:

-Determinar cfc.
-Criar DAG relativo aos cfc.
-Determinar fontes no DAG.

Grafos – Componentes Fortemente conexos

Problema MÃO DUPLA (I):

Dado o mapa do trânsito de uma cidade, representado como um digrafo, onde os vértices são as esquinas e as arestas orientadas são os trechos de rua entre esquinas, indicar se é possível atribuir mão dupla a alguns dos trechos de ruas de mão única, tal que toda esquina seja atingível a partir de qualquer outra.



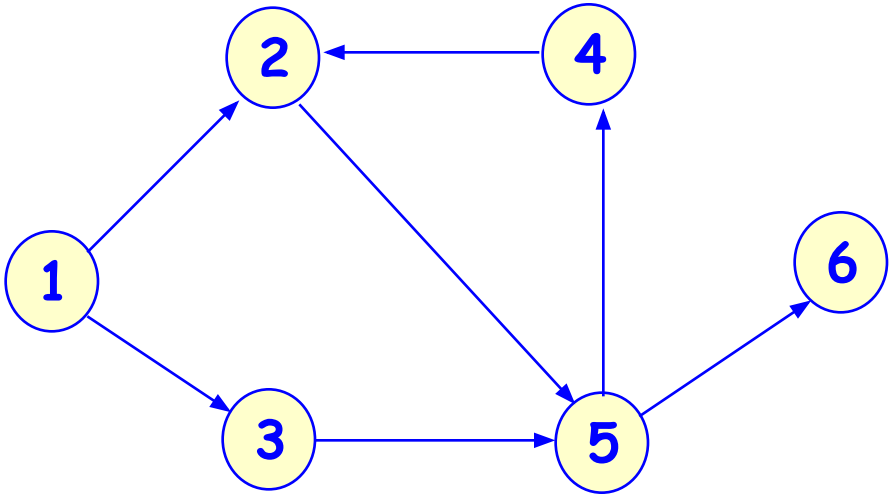
Solução MÃO DUPLA (I):

Basta verificar se o grafo subjacente é conexo.

Grafos – Componentes Fortemente conexos

Problema MÃO DUPLA (II):

Dado o mapa do trânsito de uma cidade, representado como um digrafo, onde os vértices são as esquinas e as arestas orientadas são os trechos de rua entre esquinas, indicar se é possível atribuir mão dupla a alguns dos trechos de ruas de mão única, tal que toda esquina seja atingível a partir de qualquer outra. Quando for, indicar o número mínimo de trechos que devem ser colocados em mão dupla para conseguir isso. Indicar, também, quais são esses trechos.



PROBLEMA SATISFATIBILIDADE

Satisfatibilidade ∈ NP.

Problema Satisfatibilidade:
Dada uma expressão booleana E, na forma normal conjuntiva, E é satisfatível?

Expressão na forma normal conjuntiva:

Sejam $e_1,...e_n$ variávies booleanas. Cada e_i é denominado literal. Uma cláusula é uma disjunção, isto é, uma expressão da forma $e_1 \mid e_2 \mid \ldots e_k$ (só usa | (OU), literais e_i ou sua negação $\neg e_i$). Uma expressão na forma normal conjuntiva é um conjunto de cláusulas ligadas por & (AND).

Ex: $E = (e_1 \mid \neg e_3) \,\&\, (\neg e_1 \mid e_2 \mid e_3)$.

Uma expressão E é satisfatível, quando existe uma atribuição aos literais que torna a expressão verdadeira. No exemplo dado, E é satisfatível, bastando fazer: $e_1 = V$; $e_2 = V$; $e_3 = F$.

PROBLEMAS 2(3)-SAT

2-SAT(3-SAT) ∈ NP.

Problema 2-SAT(3-SAT):
Dada uma expressão booleana E, na forma normal conjuntiva, onde cada cláusula tem exatamente 2(3) literais, E é satisfatível?

Certificado: Uma atribuição de valores para os literais.

(Os problemas são um caso particular de Satisfatibilidade.).

Exercício:

Escrever uma fórmula na FNC com 2 variáveis .booleanas que não seja satisfatível

SUBCLASSES DE Satisfatibilidade TRATÁVEIS

Algumas subclasses de Satisfatibilidade têm algoritmo polinomial:

-2-SAT (a expressão só contém cláusulas Krom, i.e., com no máximo 2 literais)

-Horn-SAT (a expressão só contém cláusulas Horn, i.e., com no máximo um literal positivo)

-Subclasses triviais (p. ex. cláusulas positivas ou negativas, onde todos os literais são ou positivos ou negativos)

ALGORITMOS POLINOMIAIS PARA 2-SAT

Entrada: Uma expressão C na 2FNC

Decisão: Existe uma atribuição que satisfaz C?

Métodos de Solução:

1. **Resolução de cláusulas:** reduz 2 cláusulas do tipo (a,b) (-b,c) a (a,c).

2. **DPLL (Backtracking):** fixa-se uma variável, simplifica-se a fórmula e resolve-se recursivamente o novo problema...

3. **Algoritmo "Aspvall":** gera e analisa um digrafo.

ALGORITMO ASPVALL

Entrada: Uma expressão C na 2FNC

Decisão: Existe ma atribuição que satisfaz C?

1. Criar um digrafo D correspondente à expressão. Para cada literal, dois vértices (positivo e negativo). Para cada cláusula, duas arestas, baseadas na seguinte identidade:

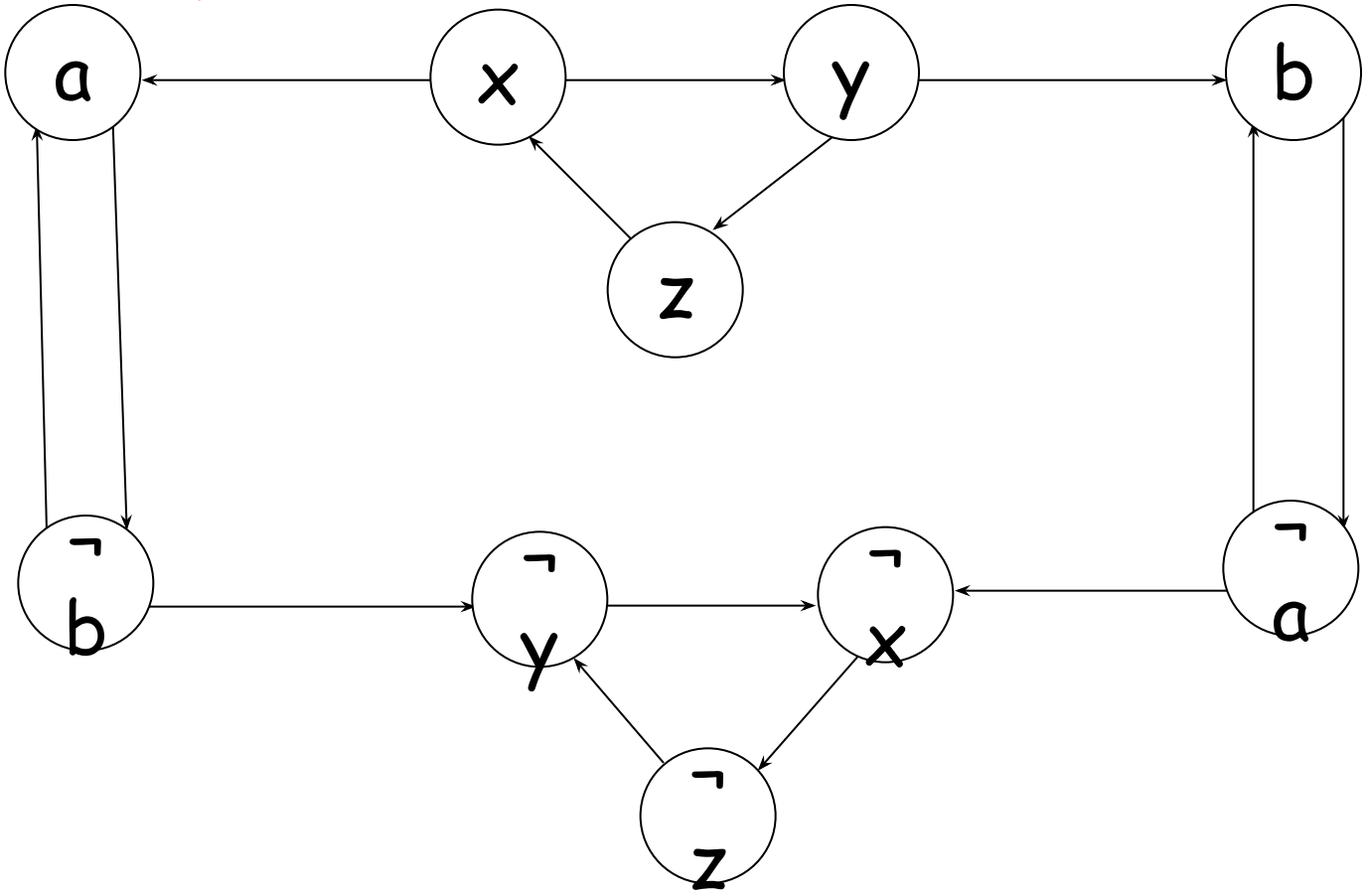
x	y	x ∨ y	-x ⇒ y	-y ⇒ x	(x∨y)⇔(-x⇒y)∧(-y⇒x)
1	1	1	1	1	1
1	0	1	1	1	1
0	1	1	1	1	1
0	0	0	0	0	0

2. Analisar se existe em D ciclo passando pelos 2 vértices de um mesmo literal. Isso significa contradição entre as cláusulas. Então basta verificar se os dois vértices de um mesmo literal estão na mesma componente fortemente conexa de D.

ALGORITMO ASPVALL

Exemplo: $C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (a \vee b)$
$\wedge$
$(\neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg y \vee b)$

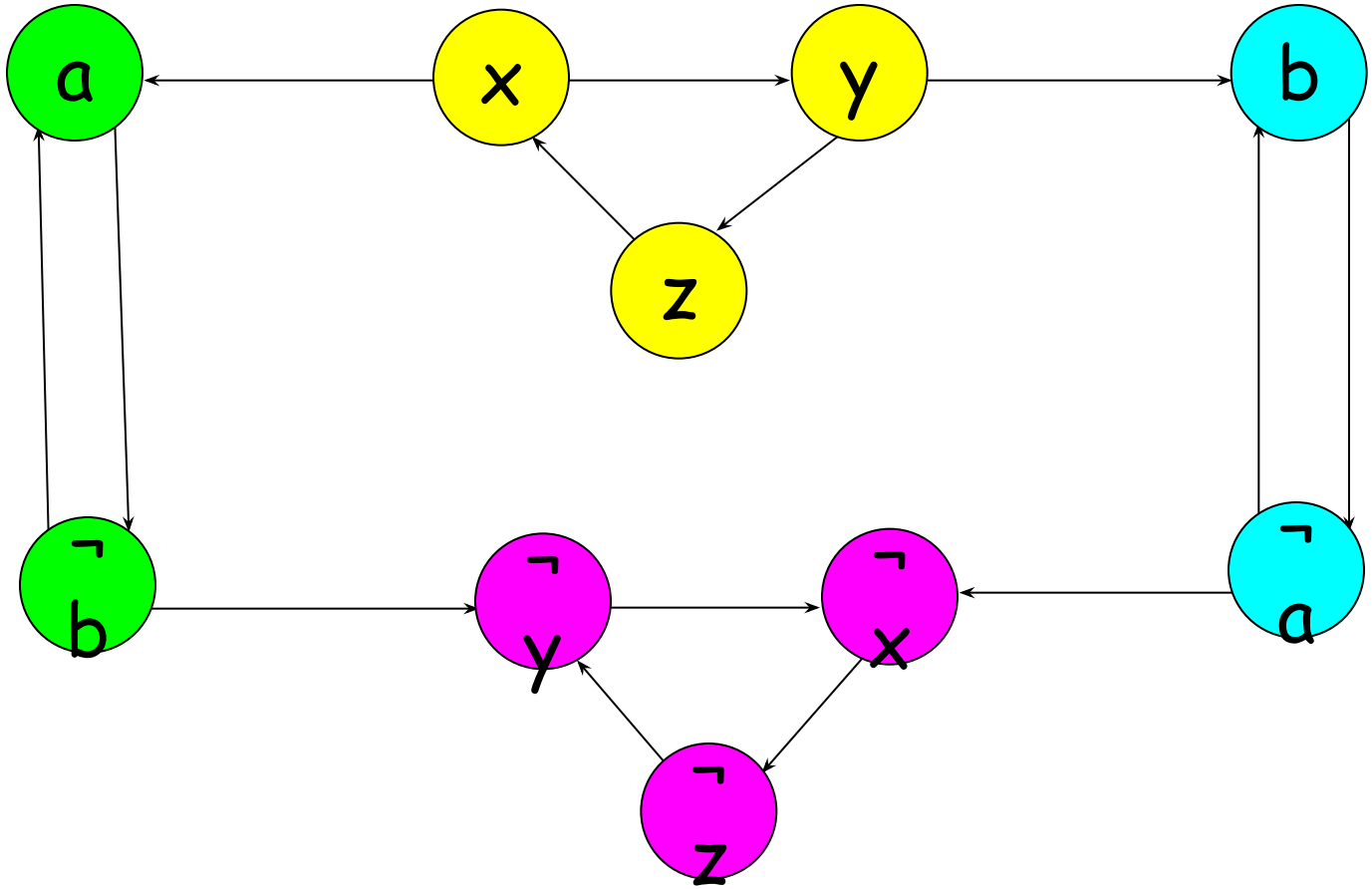
Grafo D de Implicações:



ALGORITMO ASPVALL

Exemplo: $C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg y \vee b)$

Componentes Fortemente Conexos de D:



Conclusão:

C é satisfatí-vel, pois ne-nhum literal e sua negação estão no mesmo cfc!!

Exercício:

:Aplicar o algoritmo de Aspvall à seguinte fórmula

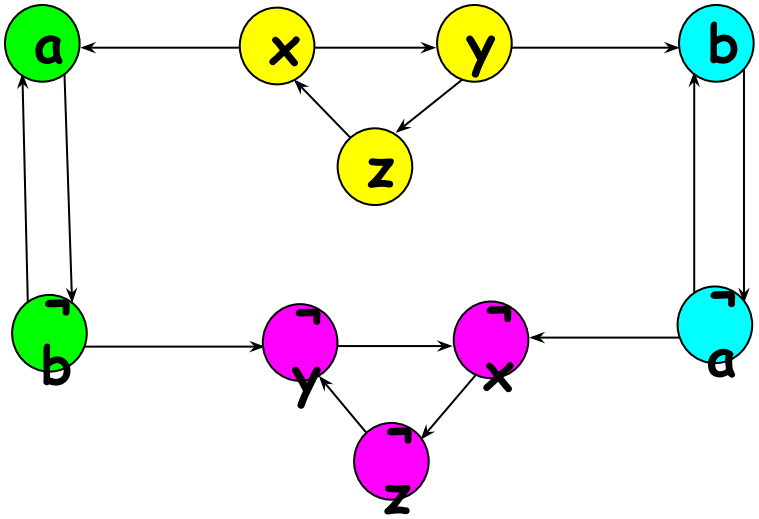
$C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$

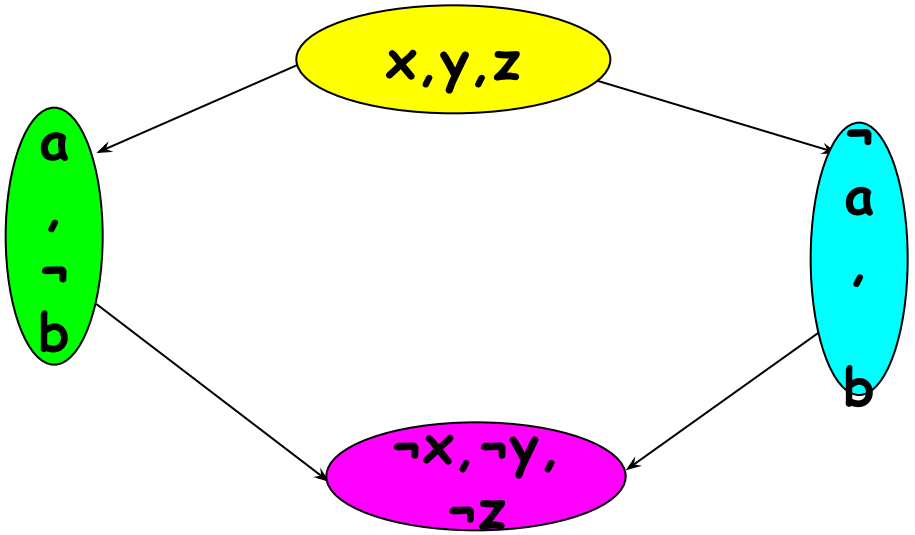
.

ALGORITMO ASPVALL - ATRIBUIÇÃO

Exemplo: $C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg y \vee b)$

Se fundirmos cada cfc, obtemos um DAG D':

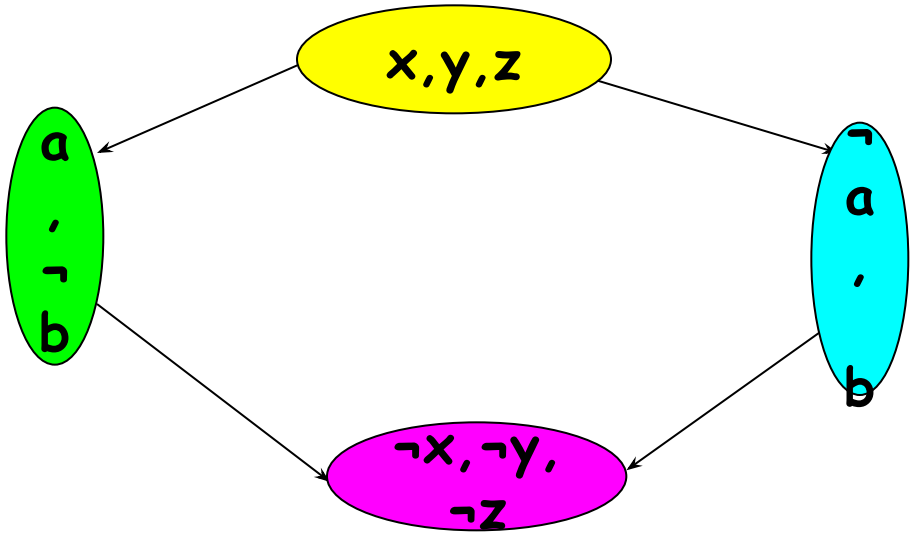




ALGORITMO ASPVALL - ATRIBUIÇÃO

Exemplo: $C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg y \vee b)$

Se fundirmos cada cfc, obtemos um DAG D':



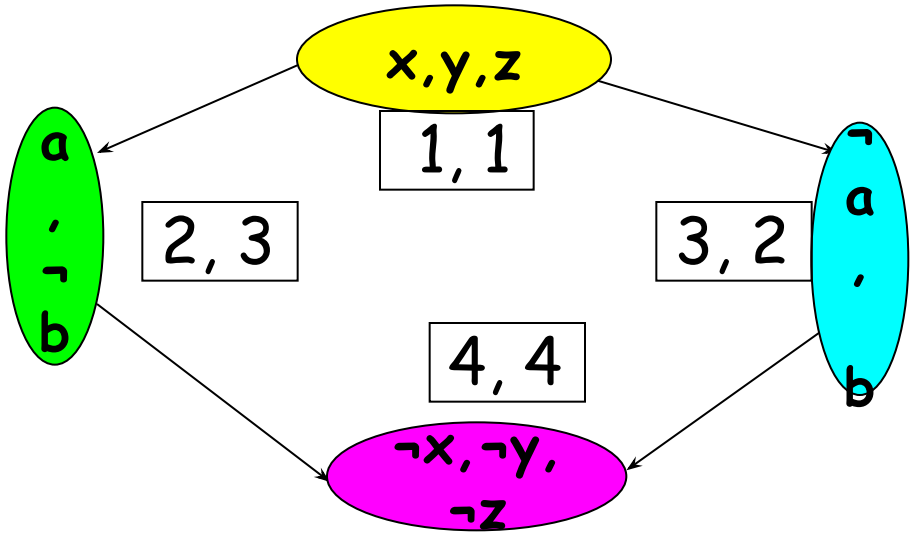
Atribuição:

1. Fazer a ordenação topo-lógica de D'.
2. Dada um literal x, seja f(x) a posição de x na ordenação.
3. Se f(x) > f(¬x) Então
   atr(x) = V
   Senão
   atr(x) = F;

ALGORITMO ASPVALL - ATRIBUIÇÃO

Exemplo: $C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg y \vee b)$

Possíveis ordenações:



Possíveis atribuições:

Alternativa 1:

x,y,z ← F;
a ← V;
b ← F;

Alternativa 2:

x,y,z ← F;
a ← F;
b ← V;

ALGORITMO ASPVALL - IMPLEMENTAÇÃO

A implementação pode ser simplificada, fazendo a determinação dos CFCs e a ordenação topológica simultâneamente, pelo algoritmo de Tarjan. Após a execução desse algoritmo é que se testa se houve conflito ou não. Neg(i) é uma função que retorna o número do vértice correpondente à negação de i.

Externamente:

```
gerar grafo
low[*] ← pre[*] ← vis[*] ← co[*] ←0;   atr[*] ←-1;
esvaziar pilha;   cpre ← 0;  nco ← 0;
para i ← 1..n incl.:
     se pre[i] = 0:

k ← 1
para i ← 1..n incl.:
     se co[i] = co[Neg(i)]:
          k ← 0
se k=1:
     escrever ("S"); Imprimir atr[*];
senão:
     escrever ("N")
```

ALGORITMO ASPVALL - IMPLEMENTAÇÃO

```
CFCOT (v);

    pre[v] ← ++cpre;  low[v] ← cpre;  vis[v] ← 1; PUSH(v);

    para vizinhos w de v:
        se pre[w]=0:
            CFCOT(w)
        se  vis[w]=1:
            low[v] ← min(low[v], low[w]);

    se  low[v] = pre[v]:
        nco++
        enquanto (1):
            p ← POP();  vis[p] ← 0;  co[p] ← nco;
            se  atr[Neg(p)] = -1:
                atr[p] ← 1
            senão:
                atr[p] ← 0
            se  p = v:
                parar loop
```

Exercício:

Indicar as atribuições possíveis de satisfatibilidade da :seguinte fórmula

$C = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$

.

Problema 10319 - Manhattan

Descrição: Dado um grid s (ruas) x a (avenidas) e dados m pares de pontos, quer-se sa-ber se é possível orientar as direções do trânsito em mão única, para que haja sempre um caminho simples para os m pares de pontos dados.

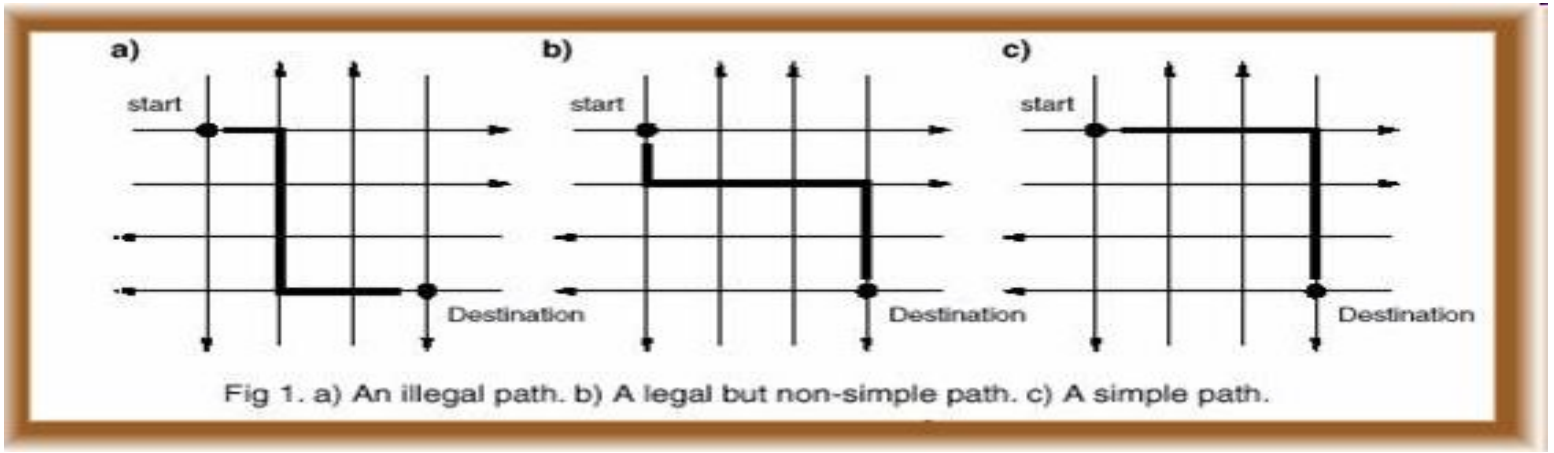


Fig 1. a) An illegal path. b) A legal but non-simple path. c) A simple path.

Solução: criar uma expressão 2FNC para a situação e usar o algoritmo de 2SAT para verificar se a expressão é satisfatível. As cláusulas a serem criadas, para cada um dos m pares (s_1, a_1, s_2, a_2) dados são as seguintes:

a) se o trajeto não é reto, criar o trajeto c_i,associando a c_i uma atribuição para s e para a, de acordo com o trajeto a ser feito (1 para a direita, 0 para a esquerda, 1 p/cima, 0 p/baixo). Para essa condição, criar a cláusula (c_i ou c_i), criar 2 vértices v e w ($\neg v$) e colocar a aresta ($w \rightarrow v$).

...

Problema 10319 - Manhattan

Descrição: Dado um grid s (ruas) x a (avenidas) e dados m pares de pontos, quer-se sa-ber se é possível orientar as direções do trânsito em mão única, para que haja sempre um caminho simples para os m pares de pontos dados.

Solução: ...

b) caso contrário, criar duas condições (c_j e c_k), uma para cada percurso possível, associando a $c_{j,k}$ uma atribuição para s e para a, como no caso anterior. Criar dois vértices para cada condição (v, w (¬v), p, q(¬p)), e criar a cláusula (c_j ou c_k), colocando no grafo as arestas ($w \rightarrow v$ e $q \rightarrow p$).

c) verificar, para cada condição, se há alguma incompatibilidade com condições já criadas anteriormente incompatibilidade entre (c_i e c_j), criar a cláusula ($\neg c_j$ ou $\neg c_k$), e duas arestas no grafo correspondentes.

Finalmente, rodar CFC do digrafo e verificar se há vértice e sua negação no mesmo CFC

Entrada:
7 7 4
1 1 1 6
6 1 6 6
6 6 1 1
4 3 5 1

Cond.	s	vs	a	va	clausulas
c1	1	1	0	-1	(c1 ou c1)
c2	6	1	0	-1	(c2 ou c2)
c3	6	0	1	0	
c4	1	0	6	0	(c3 ou c4) (¬c3 ou ¬c2)(¬c4 ou ¬c1)
c5	4	0	1	1	
c6	5	0	4	1	(c5 ou c6)

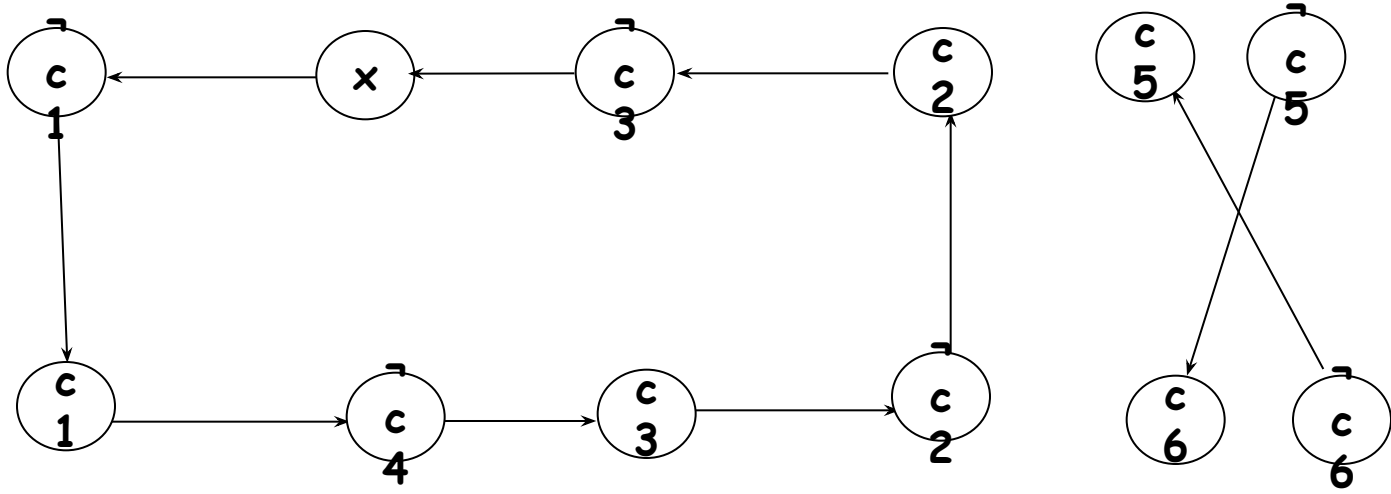
Problema 10319 - Manhattan

Descrição: Dado um grid s (ruas) x a (avenidas) e dados m pares de pontos, quer-se sa-ber se é possível orientar as direções do trânsito em mão única, para que haja sempre um caminho simples para os m pares de pontos dados.

Solução: ...

Entrada:
```
7 7 4
1 1 1 6
6 1 6 6
6 6 1 1
4 3 5 1
```

Cond.	s	vs	a	va	clausulas
c1	1	1	0	-1	(c1 ou c1)
c2	6	1	0	-1	(c2 ou c2)
c3	6	0	1	0	
c4	1	0	6	0	(c3 ou c4) (¬c3 ou ¬c2)(¬c4 ou ¬c1)
c5	4	0	1	1	
c6	5	0	4	1	(c5 ou c6)



Saída: No (c1 e ¬c1 estão nomesmo CFC)

Problema Banquete = (11294 – Wedding)

Descrição: Quer-se arrumar n casais em uma mesa de banquete, com o casal anfitrião na posição 1, os casais sentados um em frente ao outro. São dados m pares de inimigos (diferentes sexos ou não). Do lado da mesa onde está o anfitrião não pode haver nenhum par de inimigos. Indicar como a distribuição deve ser feita.

Solução: ....

Entrada: 10 6 (dez casais, 6 pares de inimigos, anfitrião = 0)

3h 7h
5w 3w
7h 6w
8w 3w
7h 3w
2w 5h

Saída: 1h 2h 3w 4h 5h 6h 7h 8h 9h

Problema Banquete - Exemplo

Exemplo: 3 2
1h 2w
1h 2h

a) o anfitrião senta-se na posição 1 e as expressões referem-se q quem senta do seu lado na mesa.

b) Variáveis: a = homem do casal 1 senta-se do mesmo lado do anfitrião
b = mulher do casal 1 senta-se do mesmo lado do anfitrião
c = homem do casal 2 senta-se do mesmo lado do anfitrião
d = mulher do casal 2 senta-se do mesmo lado do anfitrião

c) Éxpressão que obriga cada componente do casal sentar-se em lados opostos:

$$(a \vee b) \wedge (\neg a \vee \neg b) \wedge (c \vee d) \wedge (\neg c \vee \neg d)$$

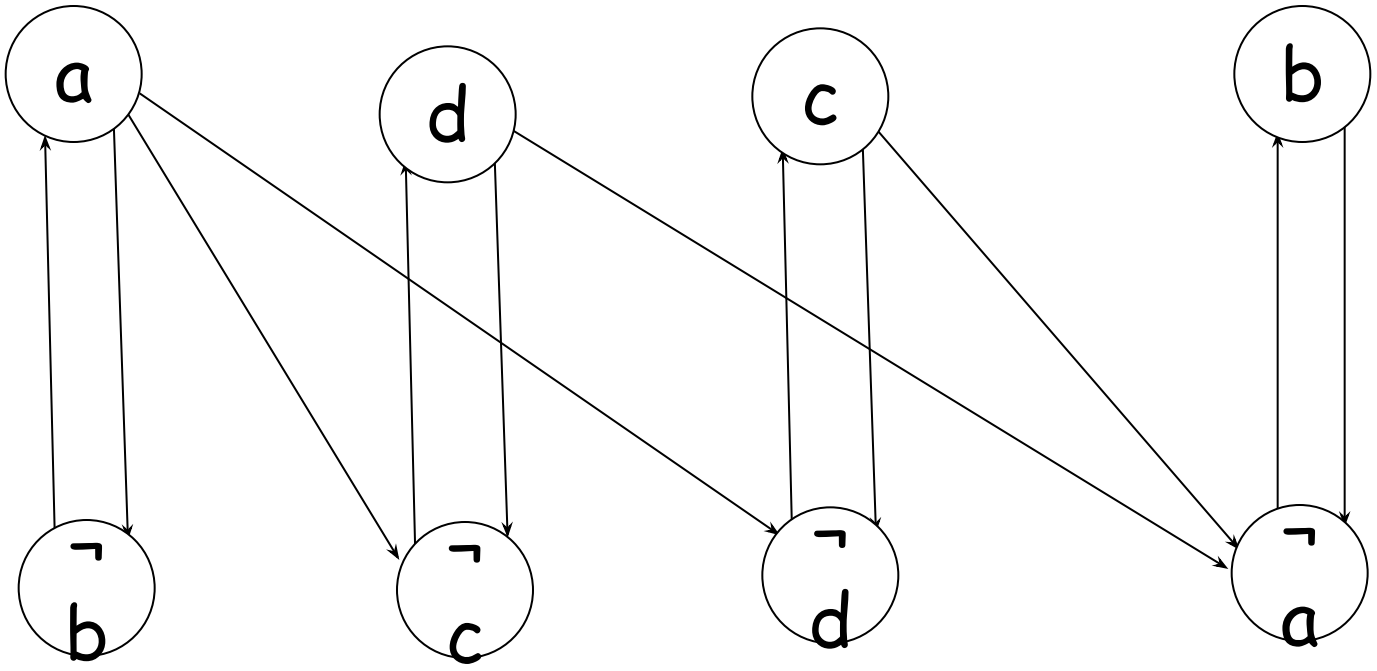
d) Expressão que proibe dois inimigos sentarem-se do lado do anfitrião:

$$(\neg a \vee \neg c) \wedge (\neg a \vee \neg d)$$

e) Expres.final: $(a \vee b) \wedge (\neg a \vee \neg b) \wedge (c \vee d) \wedge (\neg c \vee \neg d) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d)$

Problema Banquete - Exemplo

e) Expres.final: $(a \vee b) \wedge (\neg a \vee \neg b) \wedge (c \vee d) \wedge (\neg c \vee \neg d) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d)$

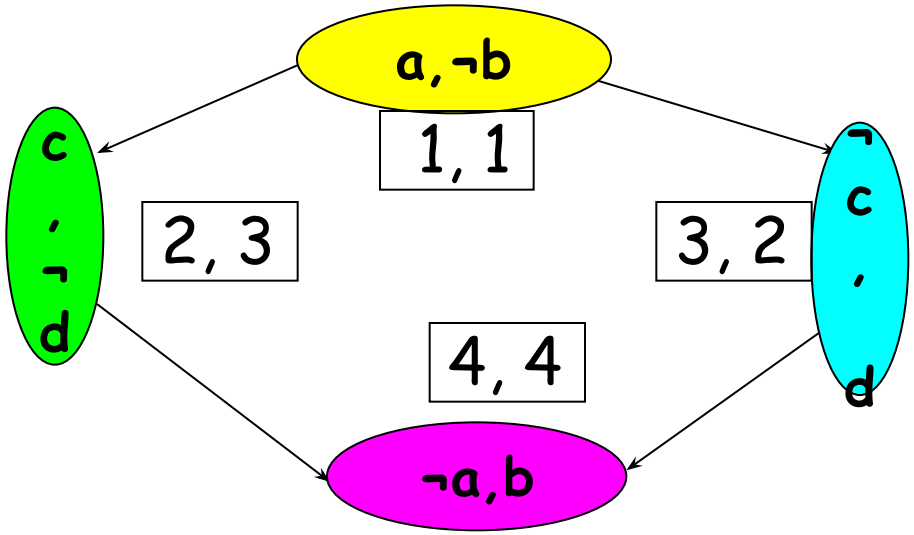


Componentes: u={a, ¬b} v={d,¬c} w={c, ¬d} x={b, ¬a}

Problema Banquete - Exemplo

Expres.final: $(a \vee b) \wedge (\neg a \vee \neg b) \wedge (c \vee d) \wedge (\neg c \vee \neg d) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d)$

Possíveis ordenações:



Possíveis atribuições:

Alternativa 1:

a ← F;
b ← V;
c ← F;
d ← V;

Alternativa 2:

a ← F;
b ← V
c ← V;
d ← F;

Problema 2886 – X-mart

Descrição: Um supermercado quer reduzir os produtos através de uma pesquisa junto aos clientes. Eles indicam 2 produtos para ficarem e dois para sairem. Dados os votos, quer-se saber se é possível satisfazer a todos os clientes (um produto escolhido para ficar fica e um para sair, sai).

Solução: ...

Entrada: 4 4 (clientes e produtos)

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

Saída: n

FIM