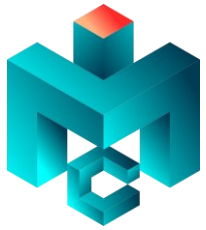
UNIVERSIDADE FEDERAL DE ITAJUBÁ
Instituto de Matemática e Computação



SMAC03 – Grafos

3. Ordenação Topológica e Coloração de Grafos

Rafael Frinhani
frinhani@unifei.edu.br

2º Semestre de 2025

OBJETIVOS

Apresentar os conceitos, algoritmos e aplicações de ordenação topológica e de coloração de grafos.

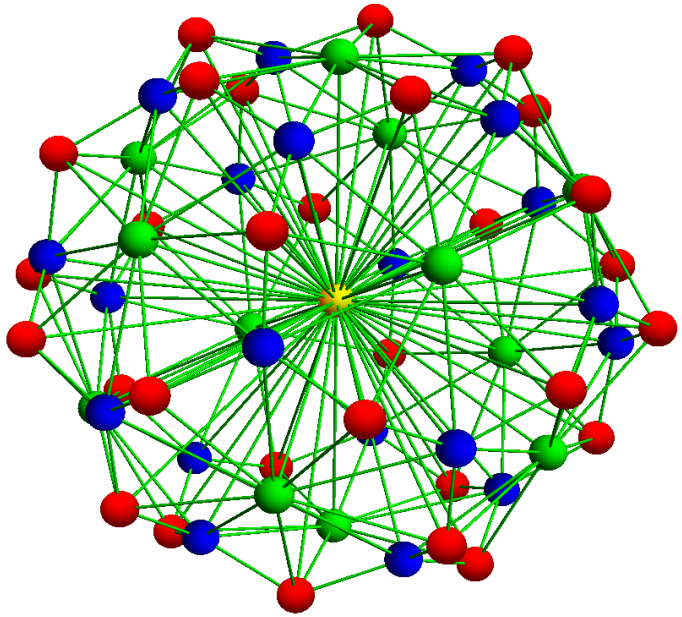
AGENDA

3.1. Ordenação Topológica

Contextualização, Histórico, Aplicações, Verificação de grafos acíclicos direcionados (DAG), algoritmo baseado na busca em profundidade (DFS), algoritmo de Kahn.

3.2. Coloração de Grafos

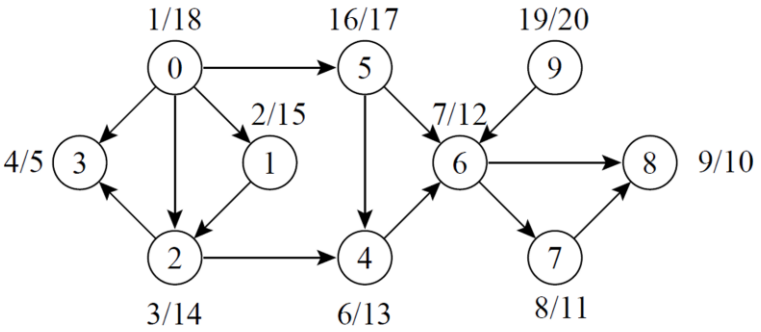
Definição, Coloração de Vértices, Número Cromático, algoritmos para coloração de vértices e verificação de grafo Bipartido. Coloração de arestas, Índice Cromático, Teorema de Vizing. Aplicações.



Contextualização

Uma Ordenação Topológica de um grafo acíclico direcionado (*Directed Acyclic Graph*, DAG) é uma ordenação linear de seus vértices, na qual cada vértice aparece antes de seus descendentes.

Corresponde a ordenação na qual cada vértice precede os vértices que formam seu fecho transitivo direto. Os vértices são organizados em uma linha horizontal de modo que todos os arcos sigam da esquerda para a direita.



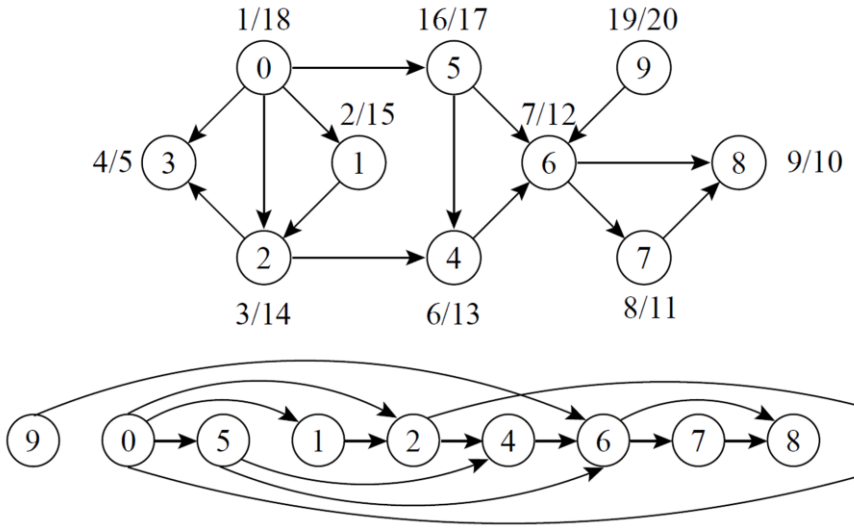
Contextualização

Uma Ordenação Topológica de um grafo acíclico direcionado (*Directed Acyclic Graph*, DAG) é uma ordenação linear de seus vértices, na qual cada vértice aparece antes de seus descendentes.

Corresponde a ordenação na qual cada vértice precede os vértices que formam seu fecho transitivo direto. Os vértices são organizados em uma linha horizontal de modo que todos os arcos sigam da esquerda para a direita.

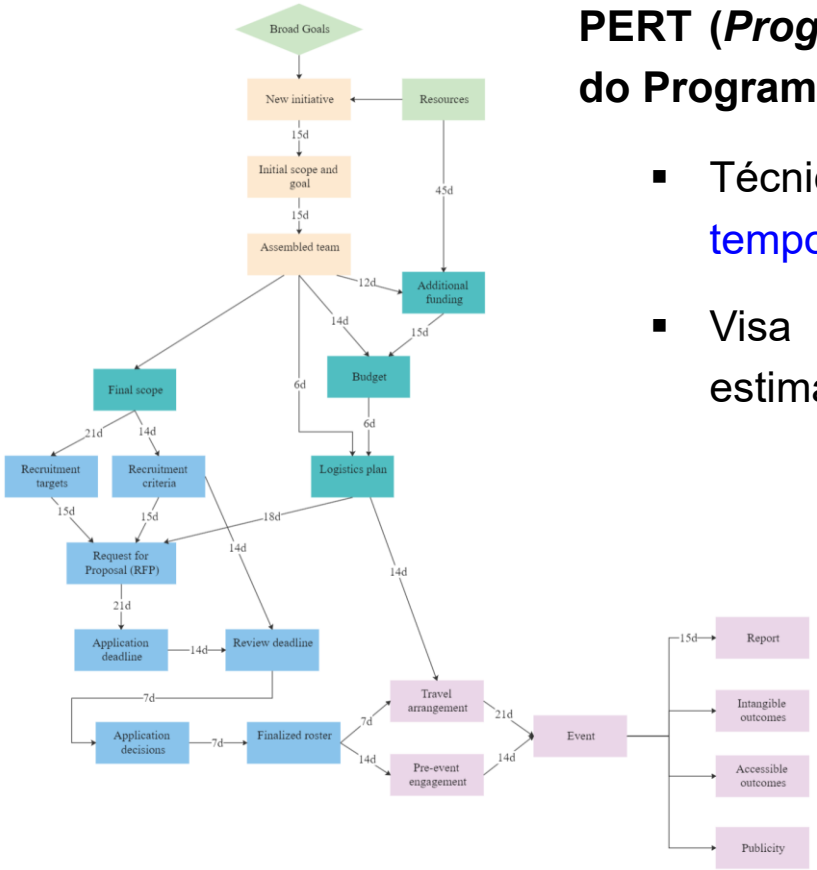
Cada DAG possui uma ou mais ordenações topológicas.

Em grafos simples ou dígrafos com ciclo não é possível estabelecer uma relação de precedência entre os vértices, impossibilitando a ordenação topológica.



Histórico

Algoritmos de ordenação topológica começaram a ser estudados na década de 60 no contexto da técnica PERT/CPM.

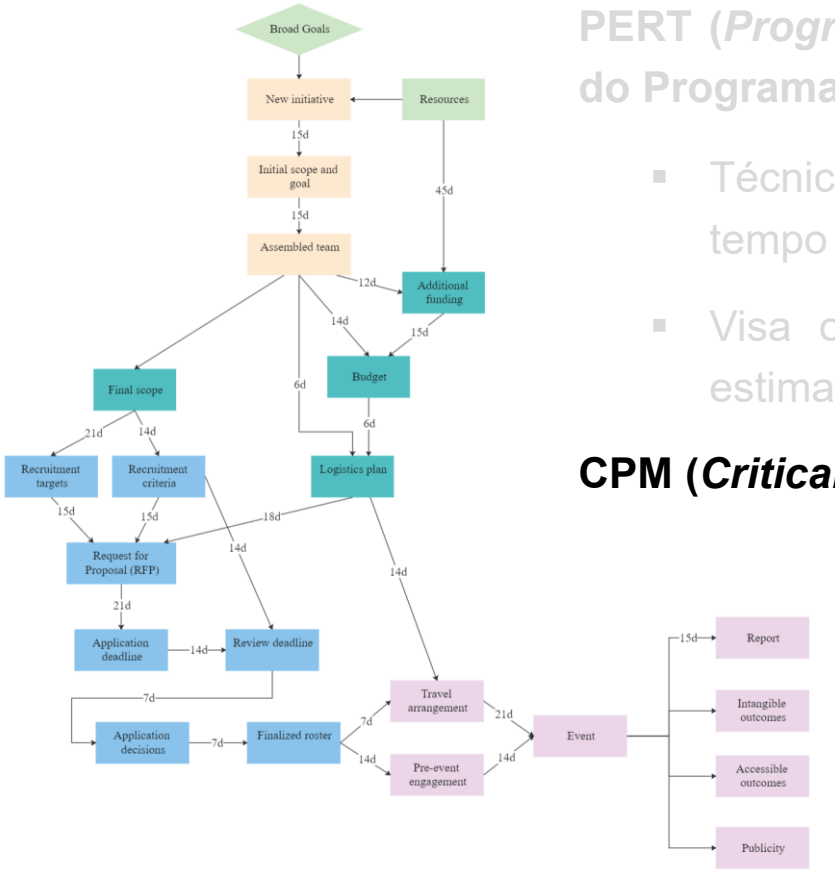


PERT (*Program Evaluation and Review Technique*, Avaliação do Programa e Técnica de Revisão):

- Técnica orientada a evento que usa três estimativas de tempo (otimista, esperada e pessimista) para cada tarefa.
- Visa o planejamento e controle de tempo, não visa a estimativa de tempo.

Histórico

Algoritmos de ordenação topológica começaram a ser estudados na década de 60 no contexto da técnica PERT/CPM.



PERT (*Program Evaluation and Review Technique*, Avaliação do Programa e Técnica de Revisão):

- Técnica orientada a evento que usa três estimativas de tempo (otimista, esperada e pessimista) para cada tarefa.
- Visa o planejamento e controle de tempo, não visa a estimativa de tempo.

CPM (*Critical Path Method,* Método do Caminho Crítico):

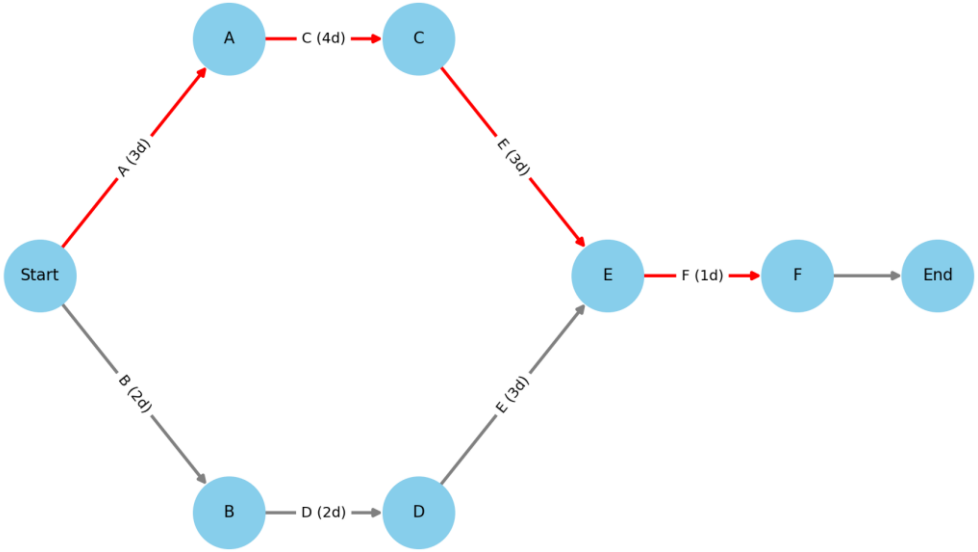
- Método de controle de tempo e custos. A duração das atividades é bem definida.
- Dada uma sequência de atividades, visa identificar quais delas impactam na duração final do projeto se sofrer alteração de duração.

PERT/CPM – Exemplo

O diagrama ajuda a visualizar o **caminho crítico**, dado pela sequência de atividades com a maior duração total, que determina o tempo mínimo de conclusão do projeto.

Cada nó representa um evento (início ou término de uma atividade).

A partir do nó "*Start*", há duas atividades iniciais: A e B. As atividades convergem ao longo do tempo até o nó final (End).

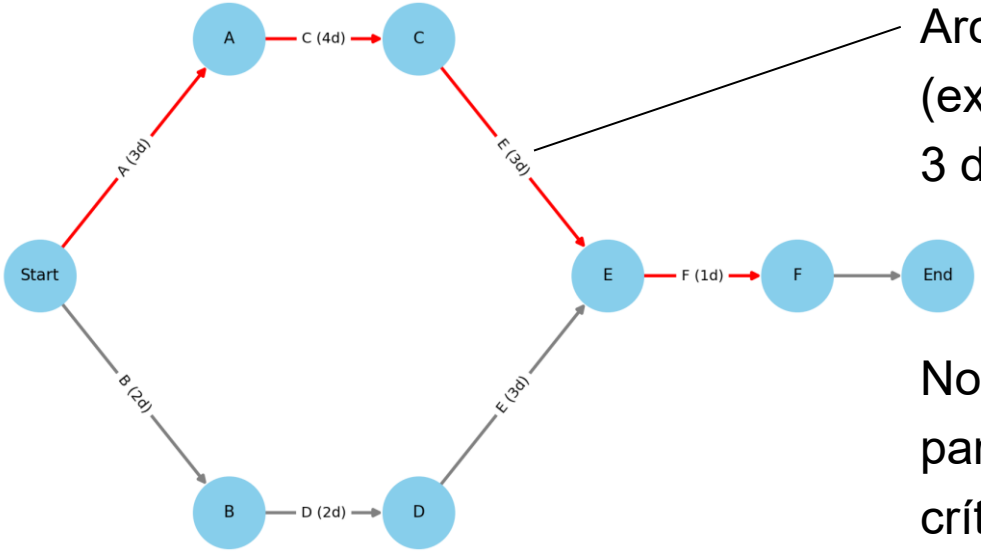


PERT/CPM – Exemplo

O diagrama ajuda a visualizar o **caminho crítico**, dado pela sequência de atividades com a maior duração total, que determina o tempo mínimo de conclusão do projeto.

Cada nó representa um evento (início ou término de uma atividade).

A partir do nó "*Start*", há duas atividades iniciais: A e B. As atividades convergem ao longo do tempo até o nó final (End).



Arcos indicam atividades e suas durações (ex. A(3d) indica que a atividade A requer 3 dias para ser concluída).

No exemplo, mesmo com a execução em paralelo dos dois caminhos, o caminho crítico é o mais custoso (11 dias).

Aplicações

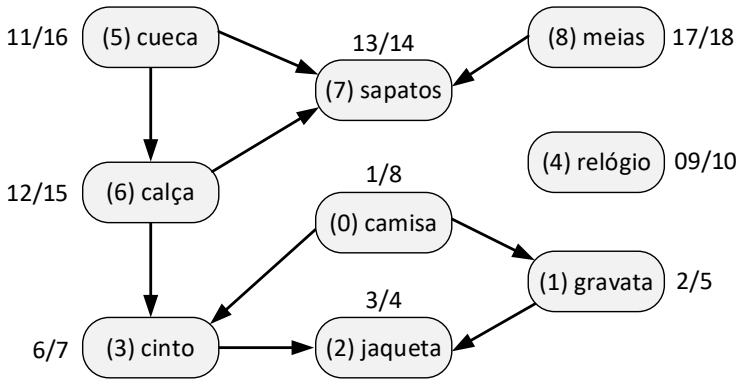
Pode ser usada para indicar a precedência de execução entre eventos:

- Roteiros de instalação de pacotes de *software*
- Hierarquia de herança entre classes em orientação a objetos
- Pré-requisitos entre disciplinas
- *make* (*makefile* de linguagens de programação)
- confecção de dicionários
- Organização de bancos de dados
- Sistemas geográficos
- Alocação de projetos (*Project Scheduling*)

Aplicações

Pode ser usada para indicar a precedência de execução entre eventos:

Exemplo: Processo de como uma pessoa se veste pela manhã.

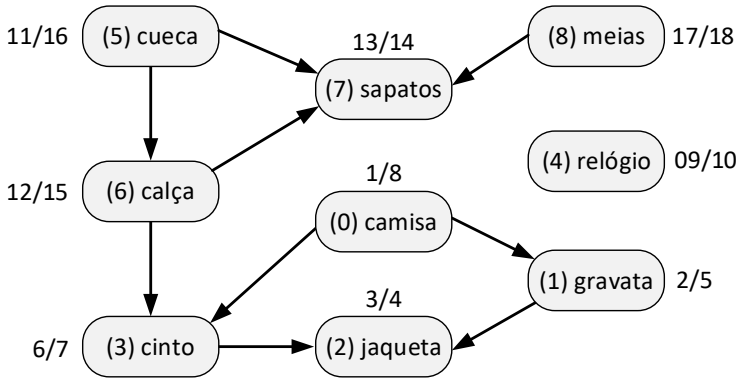


- Uma aresta orientada (u, v) no DAG indica que a peça de roupa u deve ser vestida antes da peça v.
- Algumas peças realmente devem ser vestidas antes de outras (ex. meias antes dos sapatos), outras em qualquer ordem (ex. meias e calças).

Aplicações

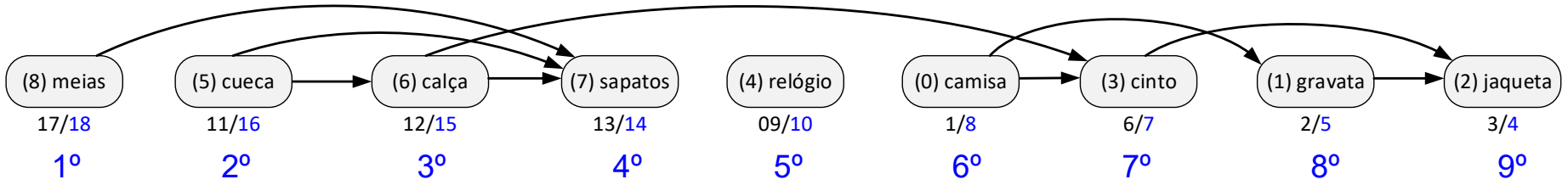
Pode ser usada para indicar a precedência de execução entre eventos:

Exemplo: Processo de como uma pessoa se veste pela manhã.



- Uma aresta orientada (u, v) no DAG indica que a peça de roupa u deve ser vestida antes da peça v.
- Algumas peças realmente devem ser vestidas antes de outras (ex. meias antes dos sapatos), outras em qualquer ordem (ex. meias e calças).
- **Solução:** Executar a DFS para obter os tempos de término. Ordenar os vértices na ordem decrescente do tempo de término.

- Uma ordenação topológica desse DAG fornece a seguinte ordem para o processo:



Classificação de Arestas

Inicialmente é necessário verificar se o grafo é um DAG. A classificação de arestas permite verificar se o grafo possui algum ciclo (aresta tipo ***back*** ou retorno).

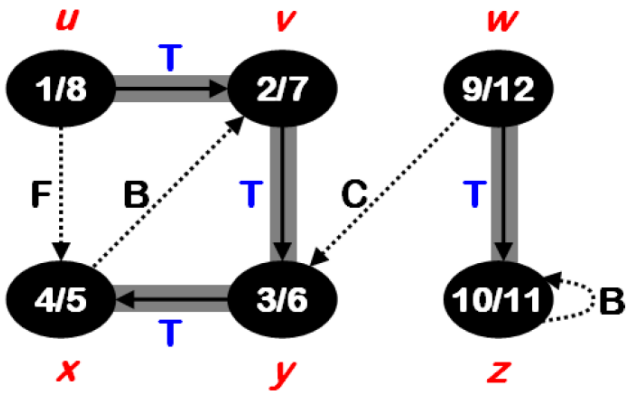
Classificação de Arestas

Inicialmente é necessário verificar se o grafo é um DAG. A classificação de arestas permite verificar se o grafo possui algum ciclo (aresta tipo ***back*** ou retorno).

A aresta pode ser classificada conforme a cor do vértice que ela incide:

white : aresta **T (*tree*)**

- Identificação inicial de todo vértice não-descoberto.



Classificação de Arestas

Inicialmente é necessário verificar se o grafo é um DAG. A classificação de arestas permite verificar se o grafo possui algum ciclo (aresta tipo ***back*** ou retorno).

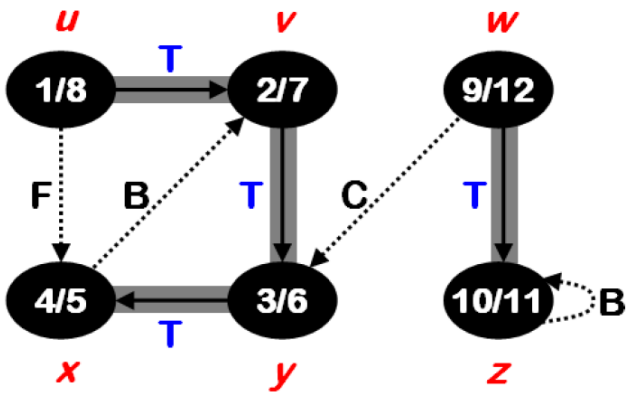
A aresta pode ser classificada conforme a cor do vértice que ela incide:

white : aresta **T** (***tree***)

- Identificação inicial de todo vértice não-descoberto.

gray : aresta **B** (***back***)

- Vértices cinza (*gray*) formam uma sequência linear de descendentes que correspondem à pilha de invocações ativas ao procedimento DFS.
- Se um vértice cinza encontra outro vértice cinza então foi encontrado um ancestral.



Classificação de Arestas

Inicialmente é necessário verificar se o grafo é um DAG. A classificação de arestas permite verificar se o grafo possui algum ciclo (aresta tipo ***back*** ou retorno).

A aresta pode ser classificada conforme a cor do vértice que ela incide:

white : aresta **T** (***tree***)

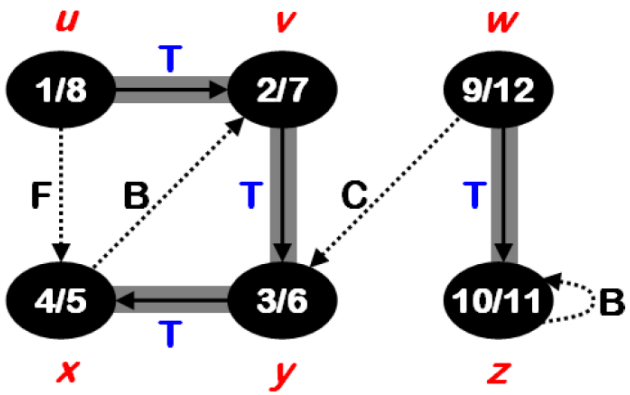
- Identificação inicial de todo vértice não-descoberto.

gray : aresta **B** (***back***)

- Vértices cinza (*gray*) formam uma sequência linear de descendentes que correspondem à pilha de invocações ativas ao procedimento DFS.
- Se um vértice cinza encontra outro vértice cinza então foi encontrado um ancestral.

black : aresta **F** (***forward***) ou **C** (***cross***)

- Possibilidade restante.
- Uma aresta (u, v) é:
  - **F** (forward) se $d[u] < d[v]$
  - **C** (cross) se $d[u] > d[v]$.



Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

ESTRUTURAS

- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u (matriz de adjacências).
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

ESTRUTURAS

- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u (matriz de adjacências).
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

classificaArestasDFS(G)

- **Linhas 1–5 (Inicialização):** Construção e inicialização das estruturas, define a cor branca para todos os vértices e a variável global que armazena o tempo.
- **Linhas 6–9 (*Loop*):** Para cada vértice ainda não descoberto (linhas 6-7) faz a pesquisa em profundidade (linha 8). Vértice v torna-se raiz de uma nova árvore (caso exista) na floresta de pesquisa da DFS.

Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |  if cor[v] == 'branco' then
8  |  |  visitaDFS(v);
9  return tipoAresta
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |  if cor[u] == 'branco' then
5  |  |  tipoAresta[v][u] = 'tree';
6  |  |  visitaDFS(u);
7  |  else if cor[u] == 'cinza' then
8  |  |  tipoAresta[v][u] = 'back';
9  |  else
10 |  |  if tempoD[v] < tempoD[u] then
11 |  |  |  tipoAresta[v][u] = 'forward';
12 |  |  else
13 |  |  |  tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

ESTRUTURAS

- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u (matriz de adjacências).
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

classificaArestasDFS(G)

- **Linhas 1–5 (Inicialização):** Construção e inicialização das estruturas, define a cor branca para todos os vértices e a variável global que armazena o tempo.
- **Linhas 6–9 (*Loop*):** Para cada vértice ainda não descoberto (linhas 6-7) faz a pesquisa em profundidade (linha 8). Vértice v torna-se raiz de uma nova árvore (caso exista) na floresta de pesquisa da DFS.

visitaDFS(v)

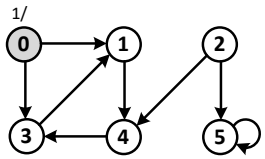
- **Linhas 1–2:** Vértice v é descoberto e torna-se cinza, é armazenado o tempo.
- **Linhas 3–16 (*Loop*):** Para cada vértice u adjacente a v, se u for branco a aresta é do tipo *Tree* (4–5) e a função é executada recursivamente (6). Caso seja cinza a busca retornou a um vértice já visitado a aresta é portanto do tipo *Back* (7–9).

  Se as condições anteriores não forem atendidas, se tempo de descoberta de v é menor que o de u a aresta é tipo *Forward* (10–12), ou *Cross* caso contrário (13–15).

  Após a análise de todos seus adjacentes v é marcado como preto e tem seu tempo de término armazenado.

Algoritmo – Classificação de Arestas (DFS)

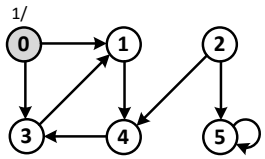
```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

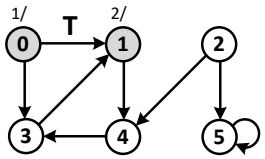



```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```






```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

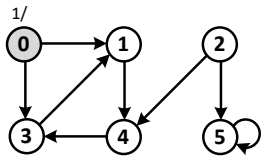
Algoritmo – Classificação de Arestas (DFS)

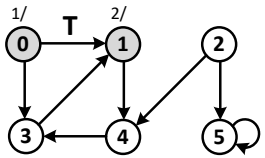
classificaArestasDFS(G)

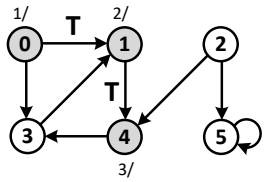
```
1 cor = [];
2 tipoAresta = [][];
3 tempoD = [];
4 tempoT = [];
5 tempo = 0;
6 for cada vertice v de G do
7 |   if cor[v] == 'branco' then
8 |   |   visitaDFS(v);
9 return tipoAresta
```

visitaDFS(v)

```
1 cor[v] = 'cinza';
2 tempoD[v] = tempo + 1;
3 for cada adjacente u do vertice v do
4 |   if cor[u] == 'branco' then
5 |   |   tipoAresta[v][u] = 'tree';
6 |   |   visitaDFS(u);
7 |   else if cor[u] == 'cinza' then
8 |   |   tipoAresta[v][u] = 'back';
9 |   else
10|   |   if tempoD[v] < tempoD[u] then
11|   |   |   tipoAresta[v][u] = 'forward';
12|   |   else
13|   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



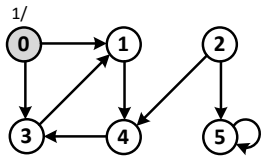


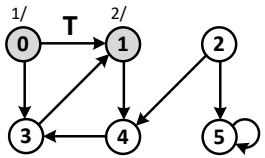


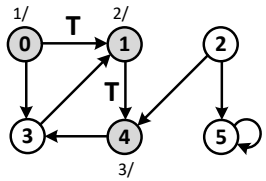
Algoritmo – Classificação de Arestas (DFS)

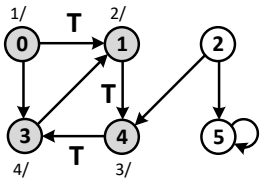
```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```









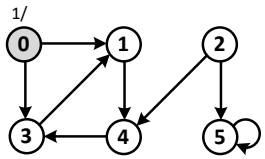
Algoritmo – Classificação de Arestas (DFS)

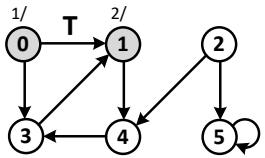
classificaArestasDFS(G)

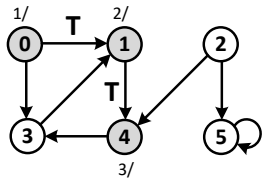
```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

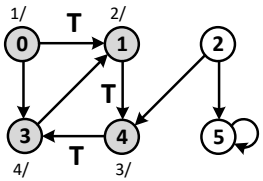
visitaDFS(v)

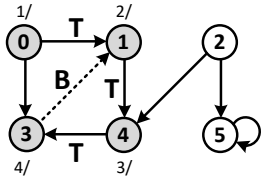
```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```











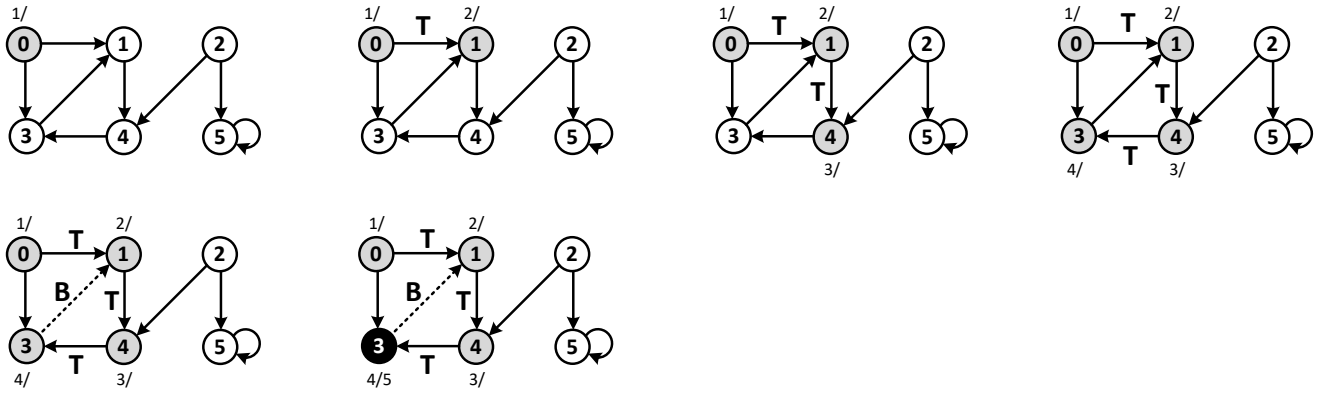
Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



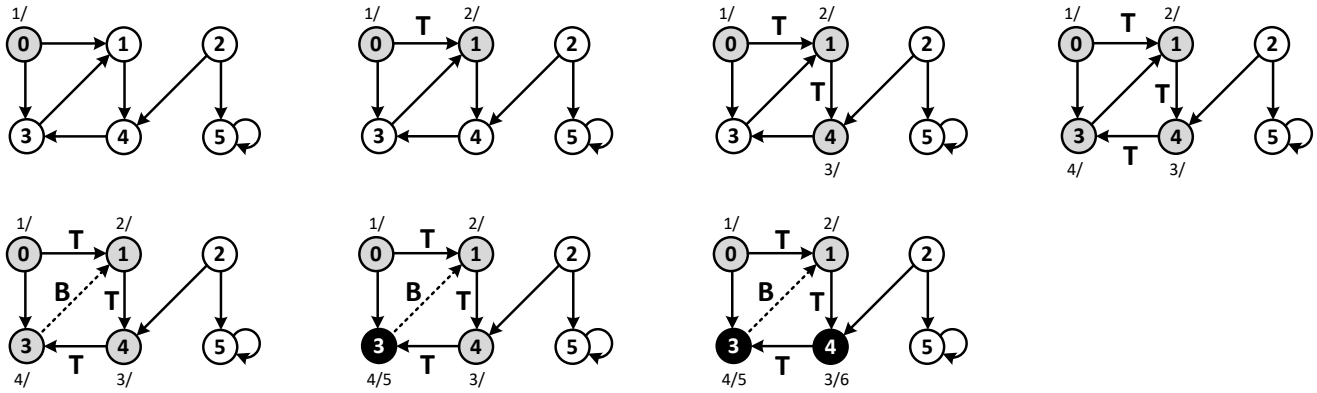
Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



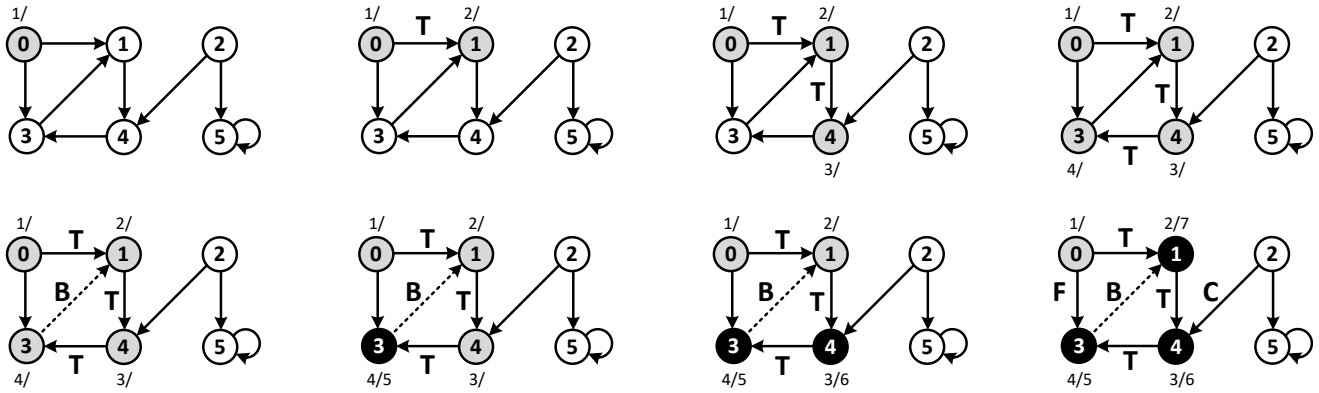
Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



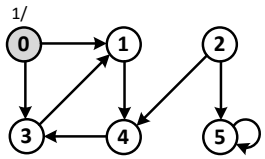
Algoritmo – Classificação de Arestas (DFS)

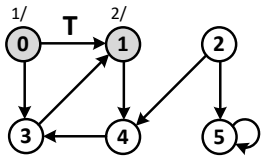
classificaArestasDFS(G)

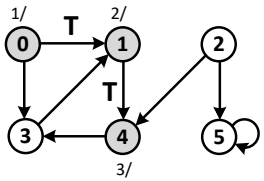
```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

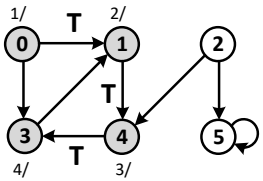
visitaDFS(v)

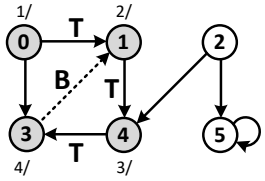
```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

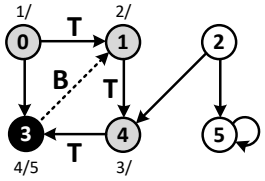


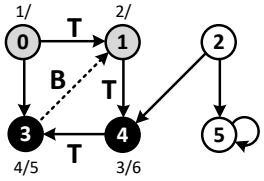


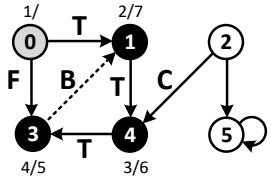


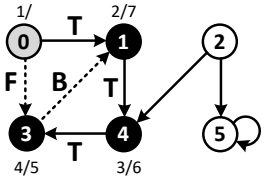












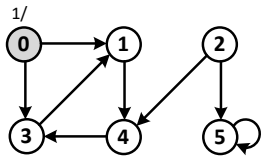
Algoritmo – Classificação de Arestas (DFS)

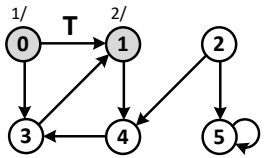
classificaArestasDFS(G)

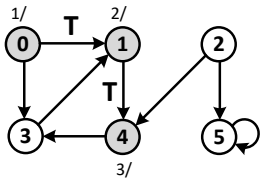
```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

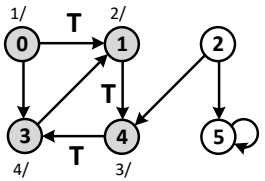
visitaDFS(v)

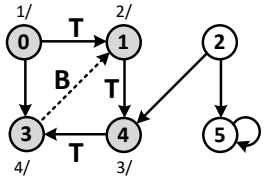
```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

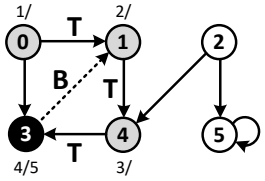


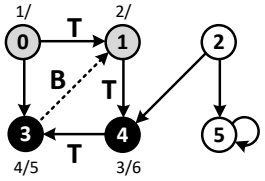


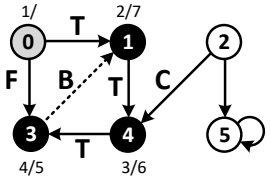


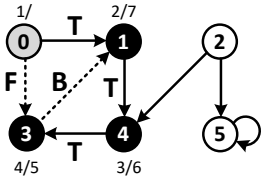


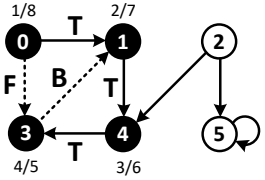












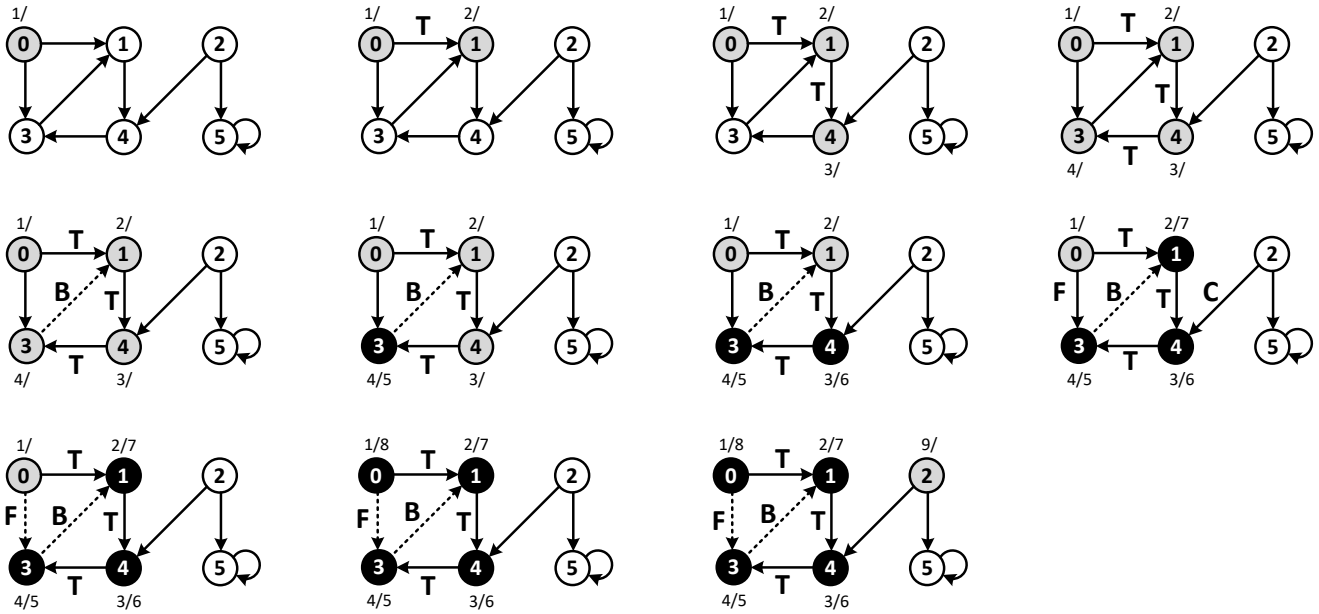
Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

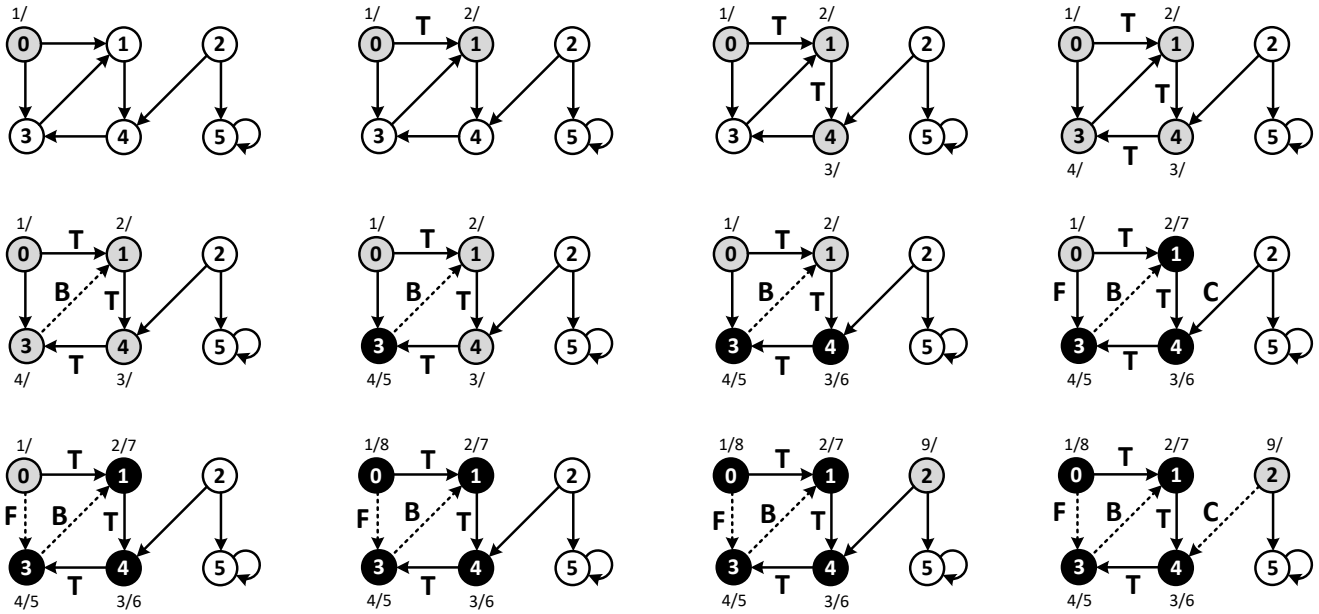
```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



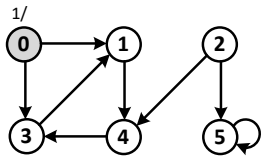
Algoritmo – Classificação de Arestas (DFS)

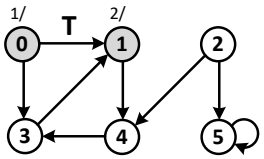
classificaArestasDFS(G)

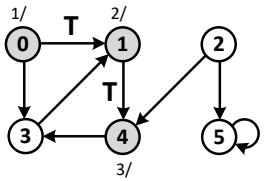
```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

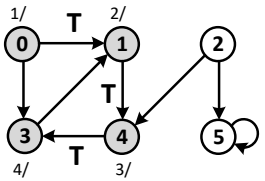
visitaDFS(v)

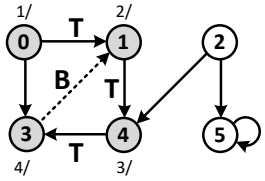
```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```

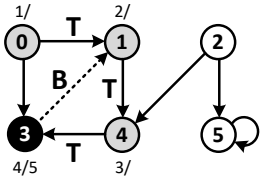


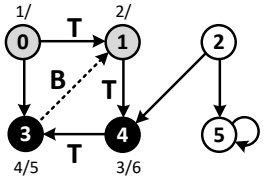


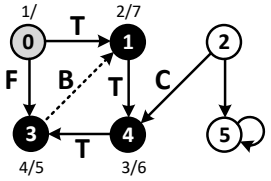


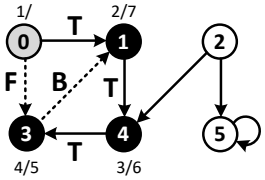


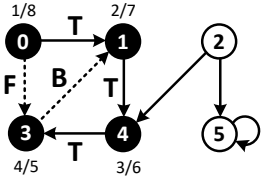


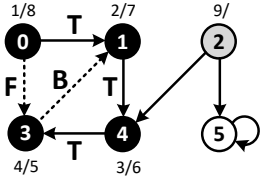


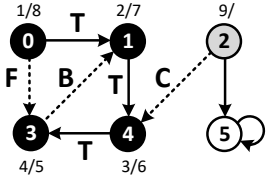


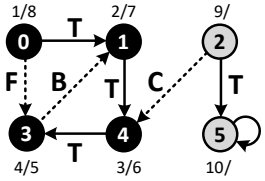








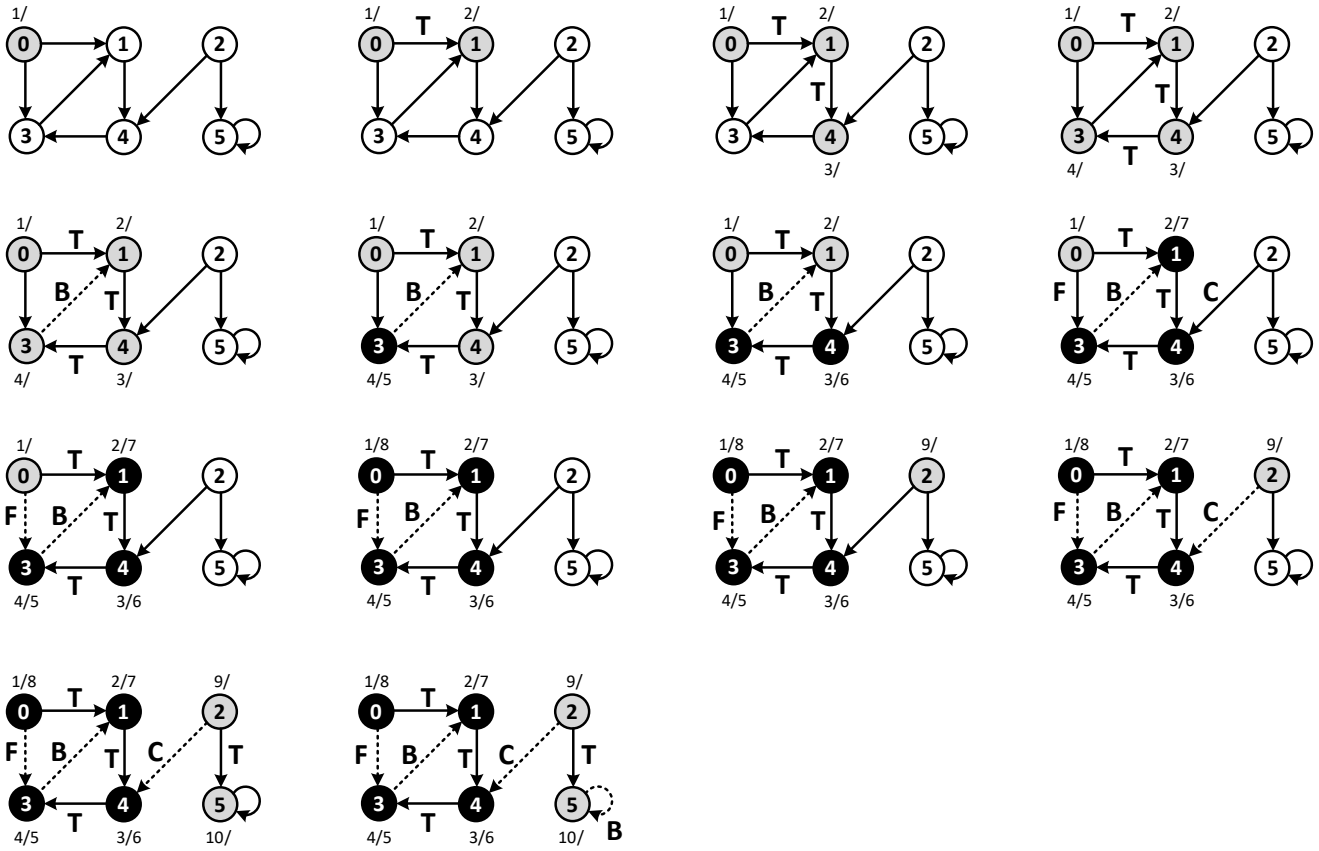




Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

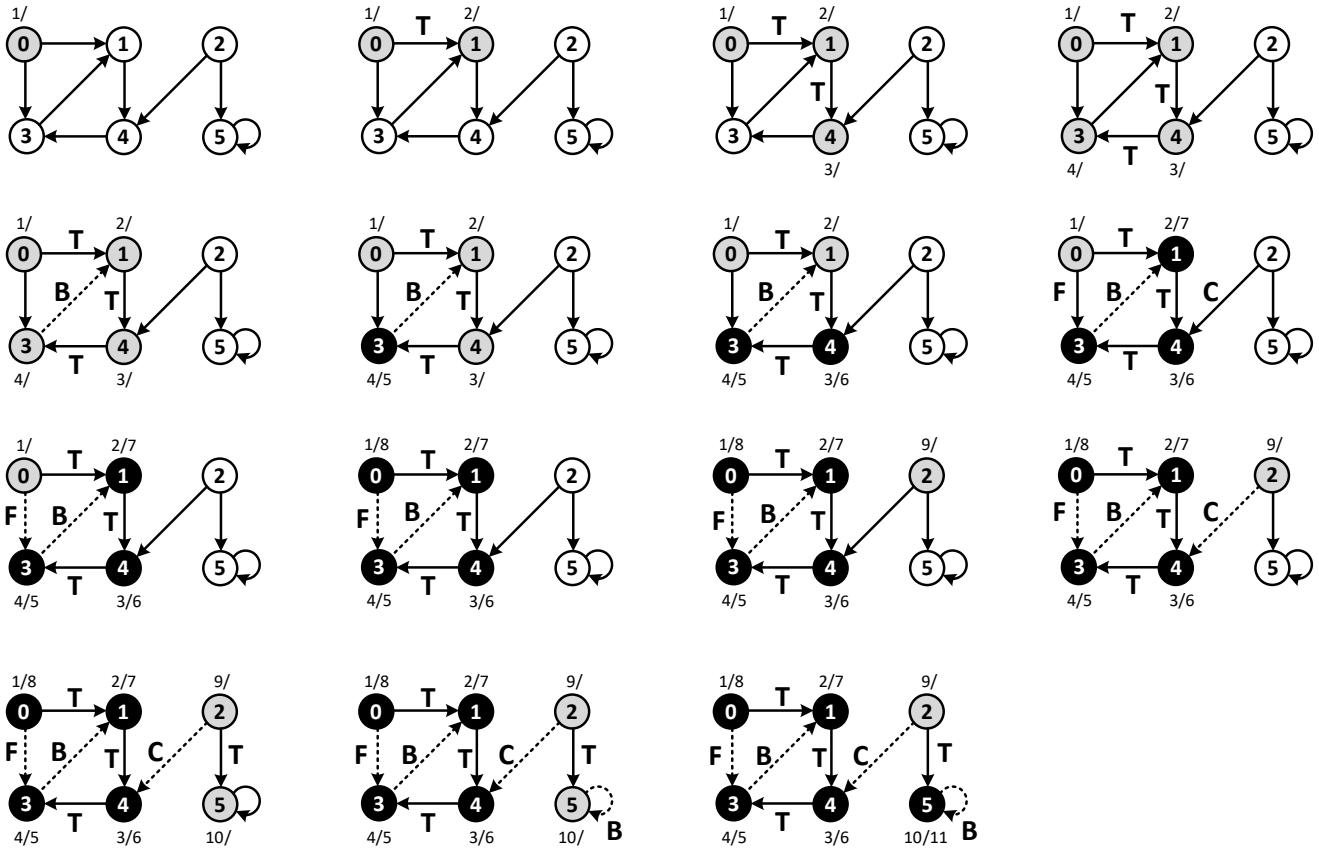
```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



Algoritmo – Classificação de Arestas (DFS)

```
classificaArestasDFS(G)
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



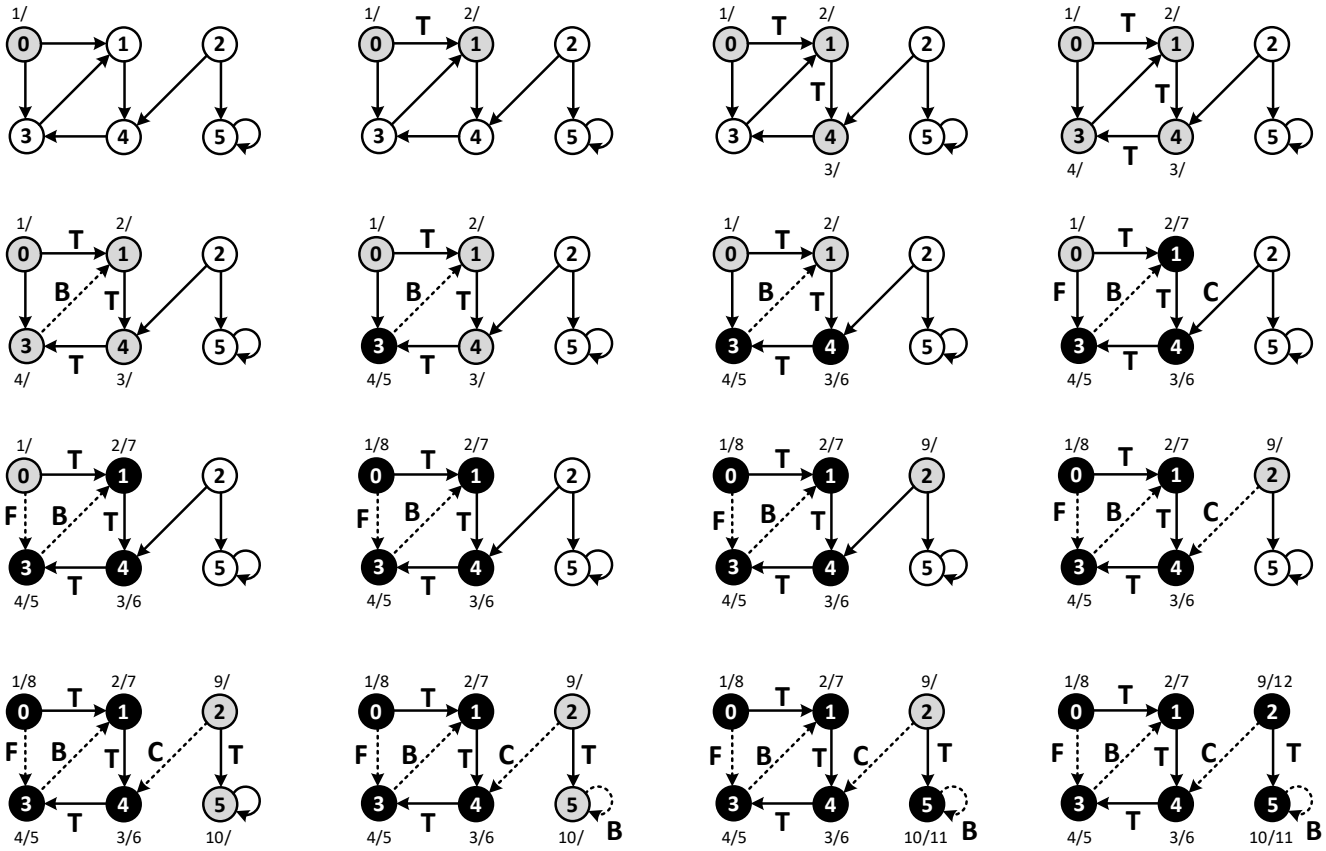
Algoritmo – Classificação de Arestas (DFS)

classificaArestasDFS(G)

```
1  cor = [];
2  tipoAresta = [][];
3  tempoD = [];
4  tempoT = [];
5  tempo = 0;
6  for cada vertice v de G do
7  |   if cor[v] == 'branco' then
8  |   |   visitaDFS(v);
9  return tipoAresta
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |   if cor[u] == 'branco' then
5  |   |   tipoAresta[v][u] = 'tree';
6  |   |   visitaDFS(u);
7  |   else if cor[u] == 'cinza' then
8  |   |   tipoAresta[v][u] = 'back';
9  |   else
10 |   |   if tempoD[v] < tempoD[u] then
11 |   |   |   tipoAresta[v][u] = 'forward';
12 |   |   else
13 |   |   |   tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
```



Algoritmo baseado na DFS

ordenacaoTopologica(G)

```
1  L = [];
2  cor = [];
3  tipoAresta = [][];
4  tempoD = [];
5  tempoT = [];
6  tempo = 0;
7  for cada vertice v de G do
8  |  if cor[v] == 'branco' then
9  |  |  visitaDFS(v);
10 return L
```

visitaDFS(v)

```
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |  if cor[u] == 'branco' then
5  |  |  tipoAresta[v][u] = 'tree';
6  |  |  visitaDFS(u);
7  |  else if cor[u] == 'cinza' then
8  |  |  tipoAresta[v][u] = 'back';
9  |  else
10 |  |  if tempoD[v] < tempoD[u] then
11 |  |  |  tipoAresta[v][u] = 'forward';
12 |  |  else
13 |  |  |  tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
16 L.insere(v)
```

ESTRUTURAS

- L : Lista que conterá os *ids* dos vértices da ordenação topológica.
- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u.
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

Algoritmo baseado na DFS

```
ordenacaoTopologica(G)
1  L = [];
2  cor = [];
3  tipoAresta = [][];
4  tempoD = [];
5  tempoT = [];
6  tempo = 0;
7  for cada vertice v de G do
8      if cor[v] == 'branco' then
9          visitaDFS(v);
10 return L
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4      if cor[u] == 'branco' then
5          tipoAresta[v][u] = 'tree';
6          visitaDFS(u);
7      else if cor[u] == 'cinza' then
8          tipoAresta[v][u] = 'back';
9      else
10         if tempoD[v] < tempoD[u] then
11             tipoAresta[v][u] = 'forward';
12         else
13             tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
16 L.insere(v)
```

ESTRUTURAS

- L : Lista que conterá os ids dos vértices da ordenação topológica.
- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u.
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

---

ordenacaoTopologica(G)

- Mesma estrutura da função *classificaArestasDFS(G)*, mas requer a inclusão da linha 1 para inicialização na lista L.
- Inclusão da linha 12 que retorna a lista L ao final da execução. **OBS.** Os vértices ordenados topologicamente aparecem na ordem inversa de seus tempos de término.

Algoritmo baseado na DFS

```
ordenacaoTopologica(G)
1  L = [];
2  cor = [];
3  tipoAresta = [][];
4  tempoD = [];
5  tempoT = [];
6  tempo = 0;
7  for cada vertice v de G do
8  |  if cor[v] == 'branco' then
9  |  |  visitaDFS(v);
10 return L
```

```
visitaDFS(v)
1  cor[v] = 'cinza';
2  tempoD[v] = tempo + 1;
3  for cada adjacente u do vertice v do
4  |  if cor[u] == 'branco' then
5  |  |  tipoAresta[v][u] = 'tree';
6  |  |  visitaDFS(u);
7  |  else if cor[u] == 'cinza' then
8  |  |  tipoAresta[v][u] = 'back';
9  |  else
10 |  |  if tempoD[v] < tempoD[u] then
11 |  |  |  tipoAresta[v][u] = 'forward';
12 |  |  else
13 |  |  |  tipoAresta[v][u] = 'cross';
14 cor[v] = 'preto';
15 tempoT[v] = tempo + 1;
16 L.insere(v)
```

ESTRUTURAS

- L : Lista que conterá os ids dos vértices da ordenação topológica.
- $cor[v]$: Cor do vértice durante a execução do algoritmo.
- $tipoAresta[v][u]$: Tipo da aresta que liga os vértices v e u.
- $tempoD[v]$: Tempo de Descoberta do vértice v.
- $tempoT[v]$: Tempo de Término do vértice v.

---

ordenacaoTopologica(G)

- Mesma estrutura da função *classificaArestasDFS(G)*, mas requer a inclusão da linha 1 para inicialização na lista L.
- Inclusão da linha 12 que retorna a lista L ao final da execução. **OBS.** Os vértices ordenados topologicamente aparecem na ordem inversa de seus tempos de término.

---

visitaDFS(v)

- Inclusão da linha 19. A medida que cada vértice v é terminado, ele deve ser inserido no início da lista L.

Exemplo de execução:
https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html

Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |   S.remove(v);
6  |   L.insere(v);
7  |   for cada arco (v, w) do
8  |   |   E.remove((v, w));
9  |   |   if w não possui mais arcos de entrada then
10 |   |   |   S.insere(w);
11 |   |   end
12 |   end
13 end
14 if E ≠ ∅ then
15 |   return NÃO DAG;
16 end
17 else
18 |   return L;
19 end
```

Tem como princípio encontrar vértices sem arestas de entrada, os quais são inseridos na solução (L) a cada instante.

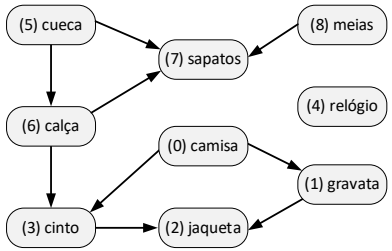
A cada vértice inserido na solução, todos seus arcos correspondentes são "removidos" do grafo.

Terminologia

- L: Lista que conterá os elementos da ordenação topológica;
- S: Conjunto de vértices que não possuem arcos de entrada (*indegree* = 0).

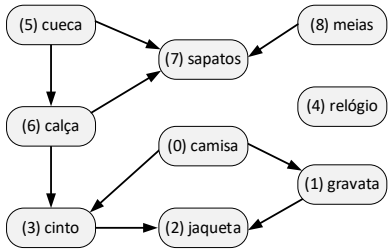
Algoritmo de Kahn

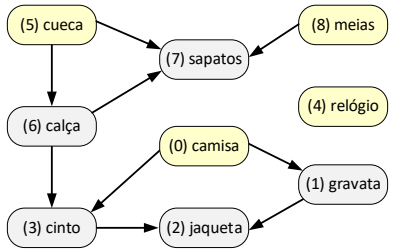
```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```



Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

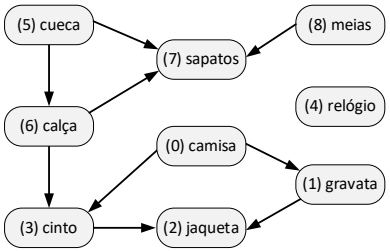


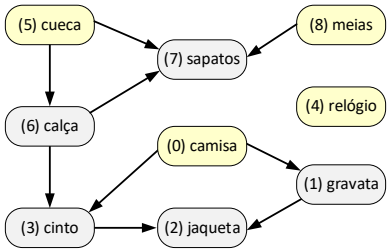


L = { }
S = {0, 4, 5, 8}

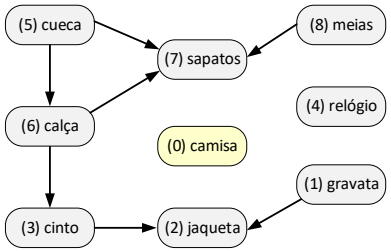
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```





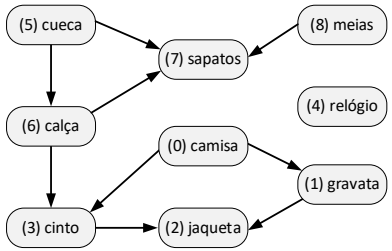
L = { }
S = {0, 4, 5, 8}

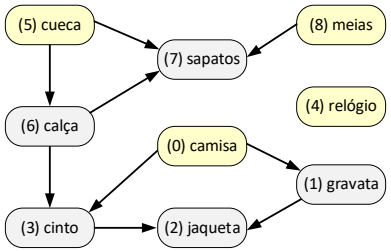


L = {0}
S = {4, 5, 8, 1}

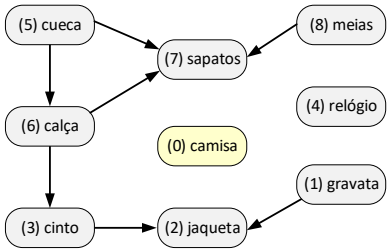
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

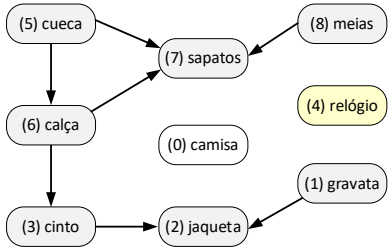




L = { }
S = {0, 4, 5, 8}



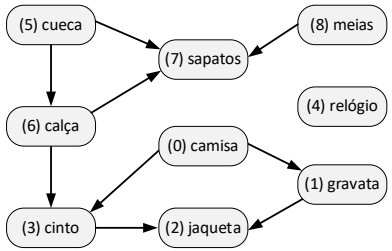
L = {0}
S = {4, 5, 8, 1}

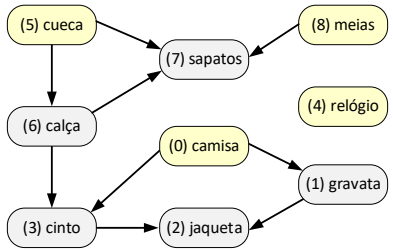


L = {0, 4}
S = {5, 8, 1}

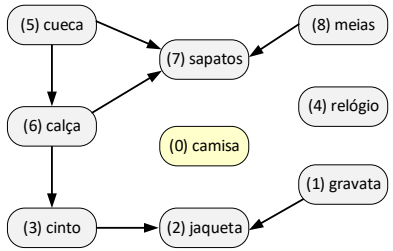
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

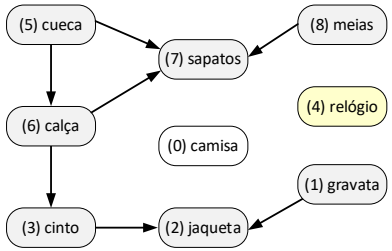




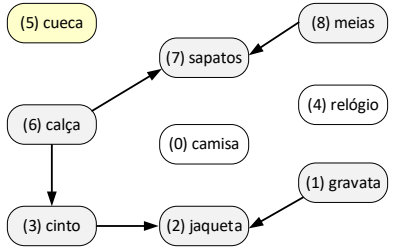
L = { }
S = {0, 4, 5, 8}



L = {0}
S = {4, 5, 8, 1}



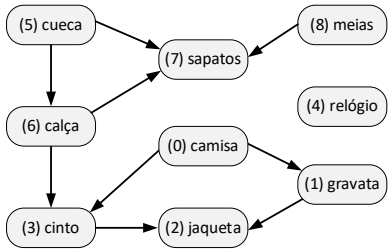
L = {0, 4}
S = {5, 8, 1}

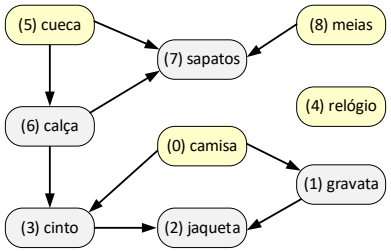


L = {0, 4, 5}
S = {8, 1, 6}

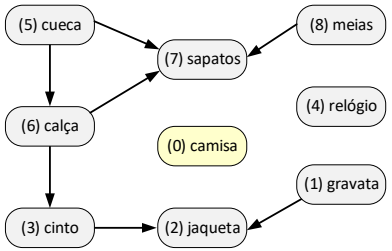
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

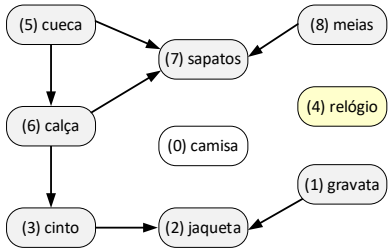




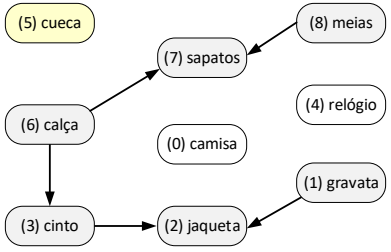
L = { }
S = {0, 4, 5, 8}



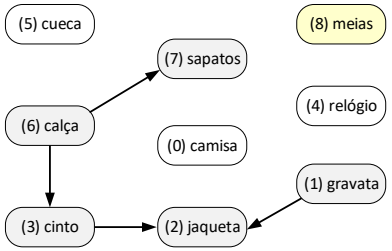
L = {0}
S = {4, 5, 8, 1}



L = {0, 4}
S = {5, 8, 1}



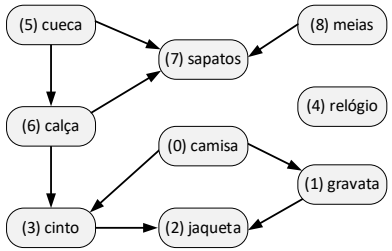
L = {0, 4, 5}
S = {8, 1, 6}

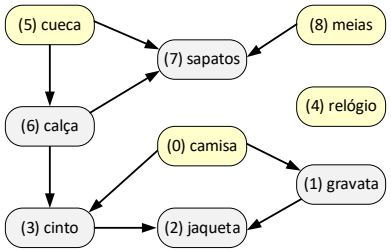


L = {0, 4, 5, 8}
S = {1, 6}

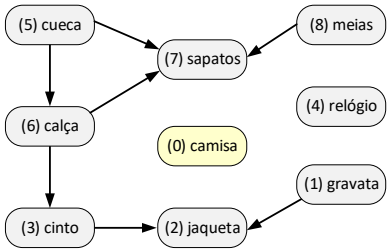
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

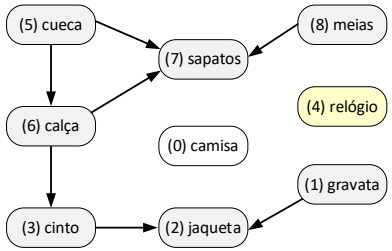




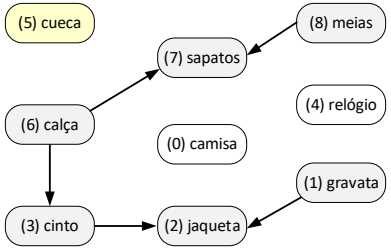
L = { }
S = {0, 4, 5, 8}



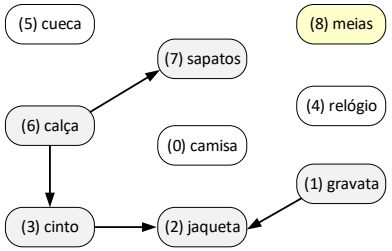
L = {0}
S = {4, 5, 8, 1}



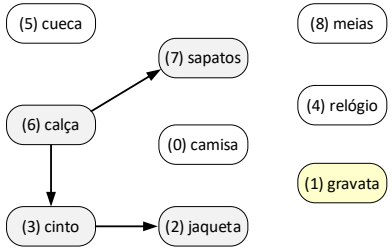
L = {0, 4}
S = {5, 8, 1}



L = {0, 4, 5}
S = {8, 1, 6}



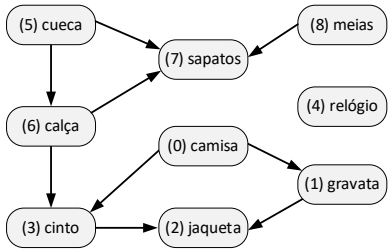
L = {0, 4, 5, 8}
S = {1, 6}

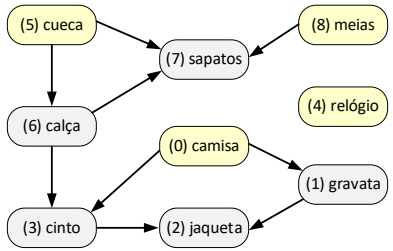


L = {0, 4, 5, 8, 1}
S = {6}

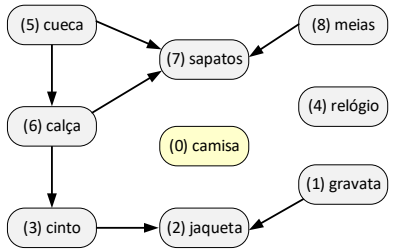
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

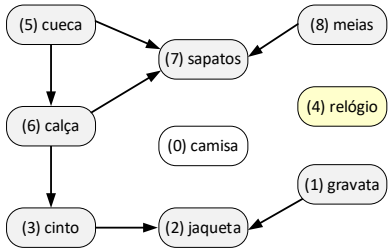




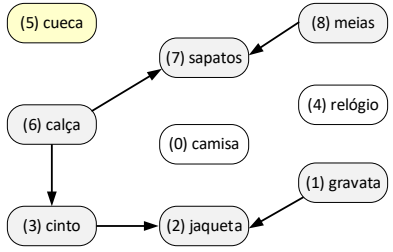
L = { }
S = {0, 4, 5, 8}



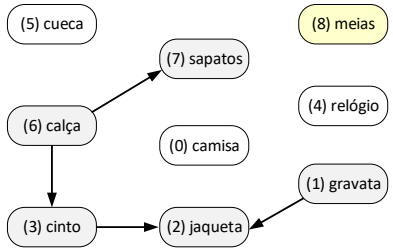
L = {0}
S = {4, 5, 8, 1}



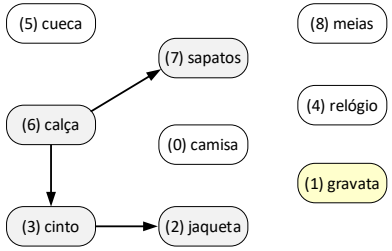
L = {0, 4}
S = {5, 8, 1}



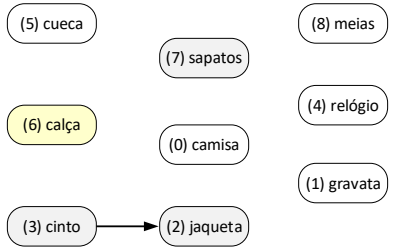
L = {0, 4, 5}
S = {8, 1, 6}



L = {0, 4, 5, 8}
S = {1, 6}



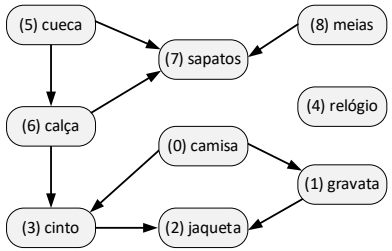
L = {0, 4, 5, 8, 1}
S = {6}

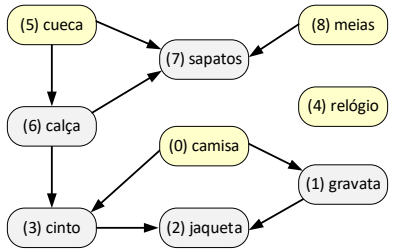


L = {0, 4, 5, 8, 1, 6}
S = {3, 7}

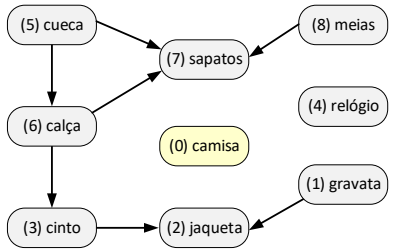
Algoritmo de Kahn

```
ordenacaoTopologicaKahn(G)
1  L = [];
2  S = vértices sem arcos de entrada;
3  E = arcos de G;
4  while S ≠ ∅ do
5  |  S.remove(v);
6  |  L.insere(v);
7  |  for cada arco (v, w) do
8  |  |  E.remove((v, w));
9  |  |  if w não possui mais arcos de entrada then
10 |  |  |  S.insere(w);
11 |  |  end
12 |  end
13 end
14 if E ≠ ∅ then
15 |  return NÃO DAG;
16 end
17 else
18 |  return L;
19 end
```

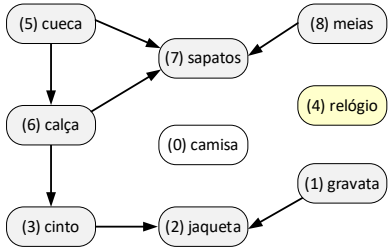




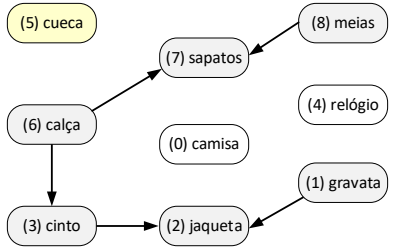
L = { }
S = {0, 4, 5, 8}



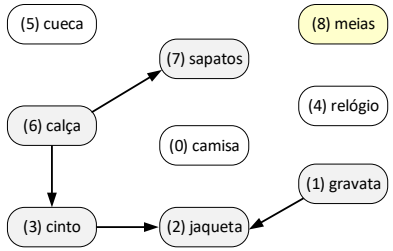
L = {0}
S = {4, 5, 8, 1}



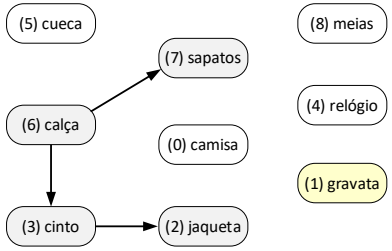
L = {0, 4}
S = {5, 8, 1}



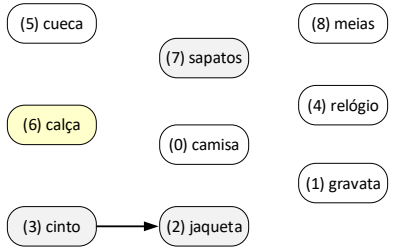
L = {0, 4, 5}
S = {8, 1, 6}



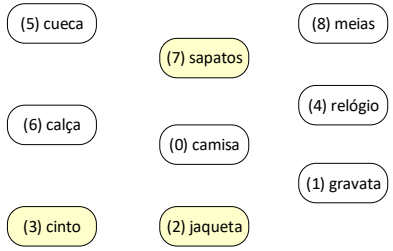
L = {0, 4, 5, 8}
S = {1, 6}



L = {0, 4, 5, 8, 1}
S = {6}



L = {0, 4, 5, 8, 1, 6}
S = {3, 7}

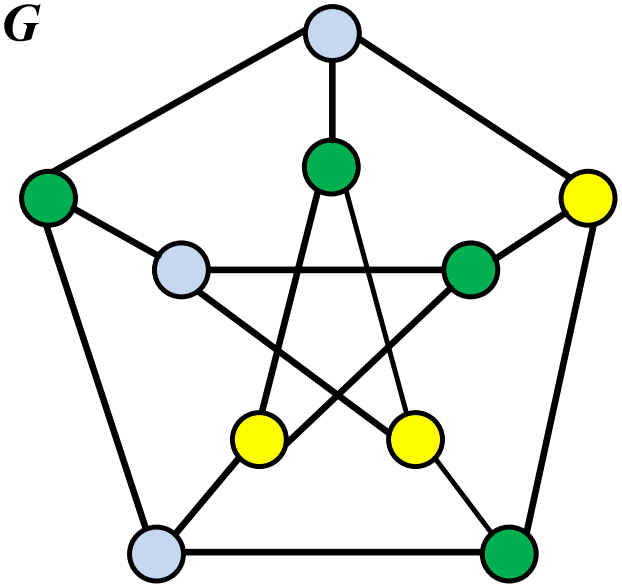


L = {0, 4, 5, 8, 1, 6, 3, 7, 2}
S = { }

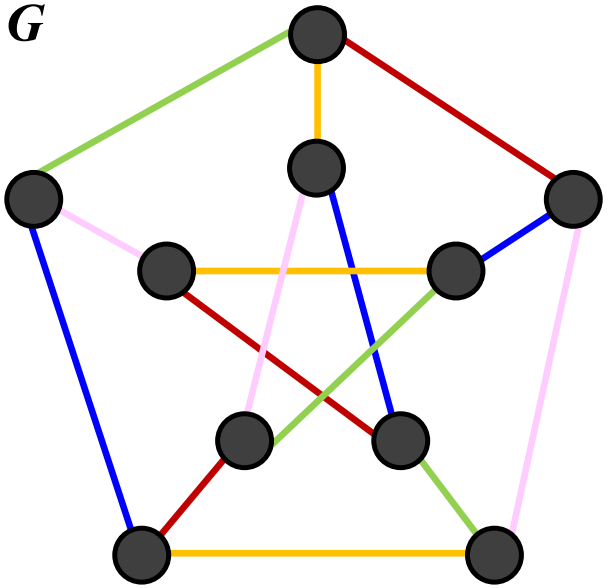
Definição

O problema de coloração de grafos consiste em atribuir cores aos seus elementos (vértices ou arestas) de maneira que seus adjacentes possuam cores diferentes.

Exemplos de coloração de vértices e coloração de arestas de um grafo G.



Coloração de Vértices

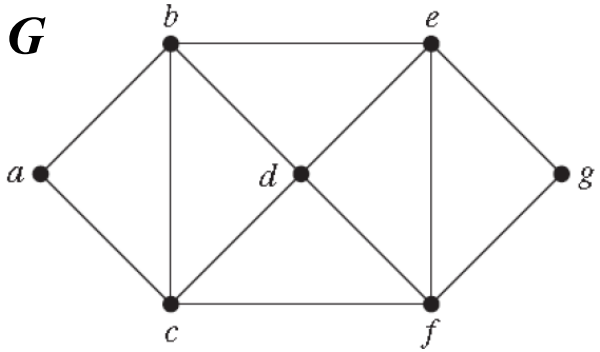


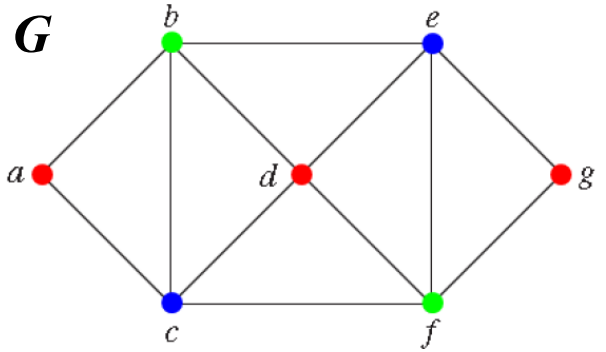
Coloração de Arestas

Coloração de Vértices

Consiste em atribuir cores aos vértices de um grafo de maneira que vértices adjacentes possuam cores diferentes.

O número cromático de um grafo G, denotado por $\chi(G)$, corresponde ao menor número de cores necessárias para sua coloração.





coloração de G, com $\chi(G) = 3$

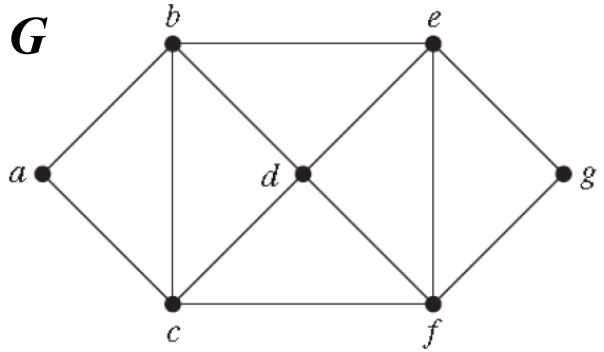
Coloração de Vértices

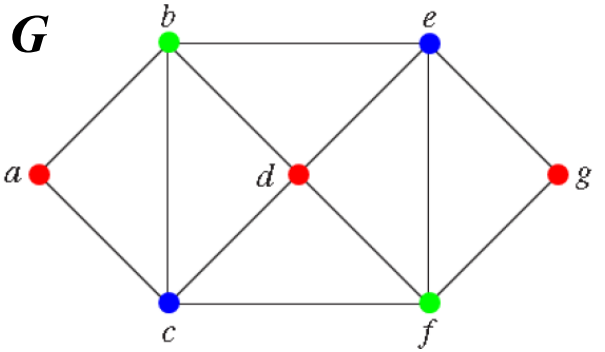
Consiste em atribuir cores aos vértices de um grafo de maneira que vértices adjacentes possuam cores diferentes.

O número cromático de um grafo G, denotado por $\chi(G)$, corresponde ao menor número de cores necessárias para sua coloração.

Teorema Appel & Haken (1976): O número cromático de um grafo G planar é no máximo quatro.

A coloração equivale a particionar o grafo no menor número de conjuntos independentes (nem sempre máximos). Logo, determinar o número cromático de um grafo é um problema NP-Difícil.





coloração de G, com $\chi(G) = 3$

Coloração de Vértices – Aplicações

Problema da Coloração de Mapas

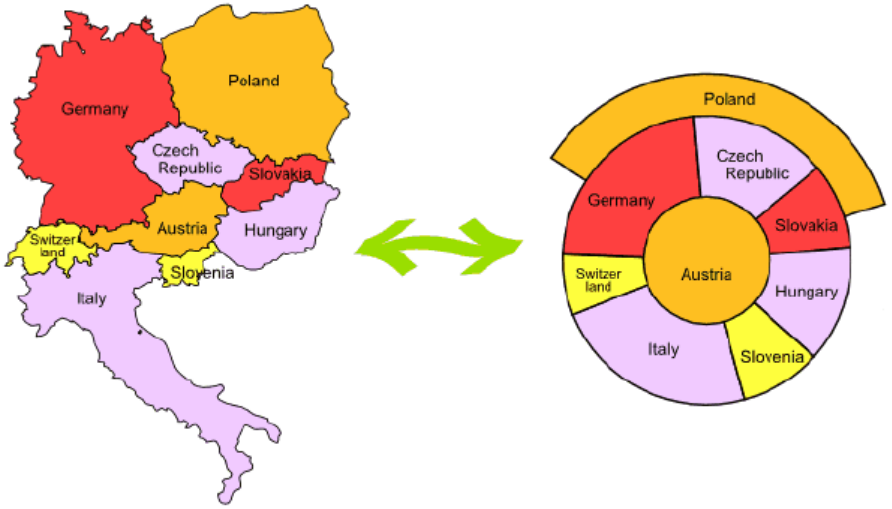
- O objetivo é atribuir o menor número de cores necessárias para colorir um mapa.
- Grafo induzido pelo mapa é planar, sendo dado pela restrição geométrica imposta pelas fronteiras (ex. países, estados, regiões, cidades, bairros, etc).

Coloração de Vértices – Aplicações

Problema da Coloração de Mapas

- O objetivo é atribuir o menor número de cores necessárias para colorir um mapa.
- Grafo induzido pelo mapa é planar, sendo dado pela restrição geométrica imposta pelas fronteiras (ex. países, estados, regiões, cidades, bairros, etc).

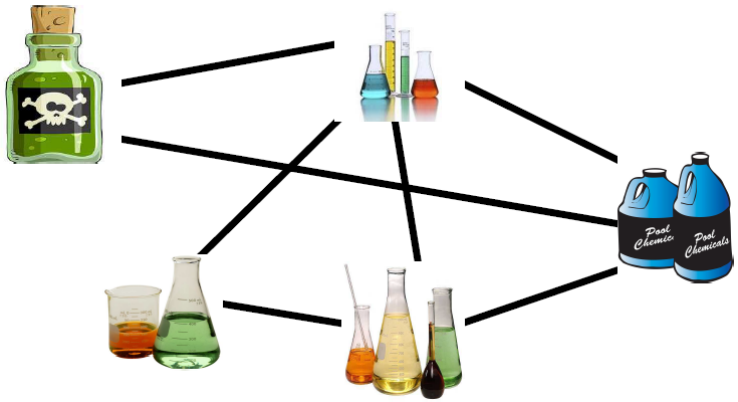
- **Teorema das Quatro Cores:** Quatro cores são suficientes para colorir qualquer mapa.
  - Conjectura de De Morgan em 1852 apresentou várias provas erradas!
  - Provado por força bruta por Appel & Haken (1976), que mostraram que não há mapa para qual 5 cores seja necessário (análise de 2.000 casos por computador)
  - Não muito aceito pela comunidade matemática devido as possibilidades de *bug* no *software*, o qual foi amplamente verificada sua corretude posteriormente.



Coloração de Vértices – Aplicações

Separação de produtos Explosivos

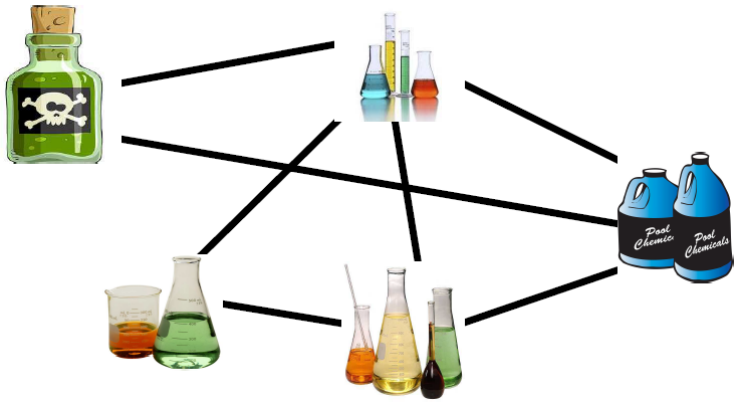
- Os vértices representam produtos químicos usados em um processo.
- Uma aresta liga pares de produtos que podem explodir se combinados.
- O número cromático representa a quantidade de compartimentos para guardar os produtos em segurança.



Coloração de Vértices – Aplicações

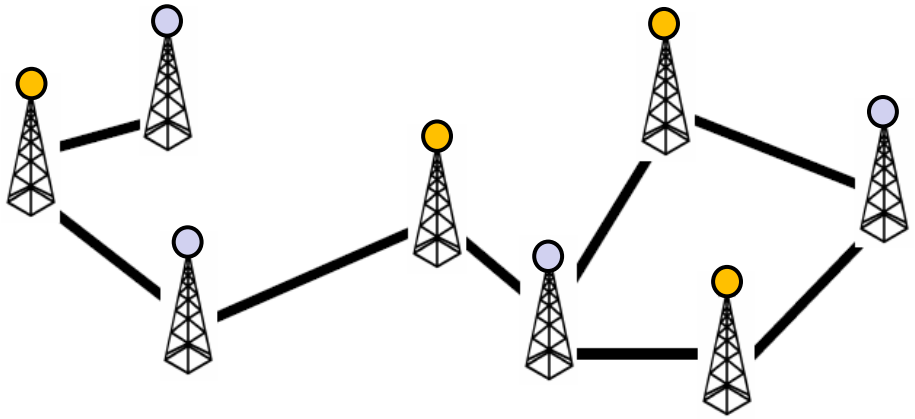
Separação de produtos Explosivos

- Os vértices representam produtos químicos usados em um processo.
- Uma aresta liga pares de produtos que podem explodir se combinados.
- O número cromático representa a quantidade de compartimentos para guardar os produtos em segurança.



Atribuição de Frequências de Rádio

- Vértices representam os transmissores das estações de rádio. Duas estações são adjacentes quando suas áreas de transmissão se sobrepõem, podendo ocasionar em interferências caso adotem a mesma frequência.
- Cada cor contém as estações que podem receber a mesma frequência.



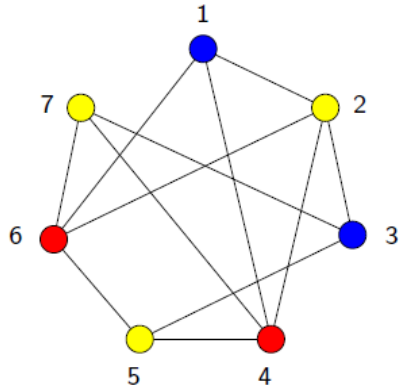
Coloração de Vértices – Aplicações

Programação de Horários

- Dadas as matrículas dos alunos, é necessário determinar os horários das disciplinas para que eles assistam às aulas sem que ocorra conflito de horários.

- Os vértices representam disciplinas sendo que os adjacentes representam aquelas com alunos em comum.

	1	2	3	4	5	6	7
1	—	X	—	X	—	X	—
2		—	X	X	—	X	—
3			—	—	X	—	X
4				—	X	—	X
5					—	X	—
6						—	X
7							—

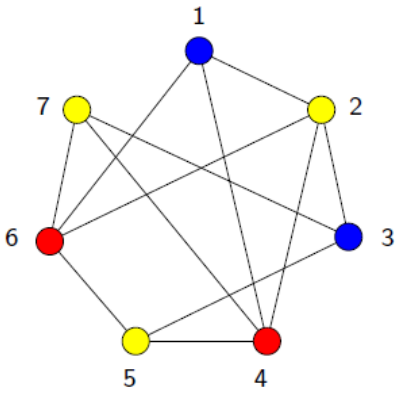


Coloração de Vértices – Aplicações

Programação de Horários

- Dadas as matrículas dos alunos, é necessário determinar os horários das disciplinas para que eles assistam às aulas sem que ocorra conflito de horários.
- Os vértices representam disciplinas sendo que os adjacentes representam aquelas com alunos em comum.

	1	2	3	4	5	6	7
1	—	X	—	X	—	X	—
2		—	X	X	—	X	—
3			—	—	X	—	X
4				—	X	—	X
5					—	X	—
6						—	X
7							—

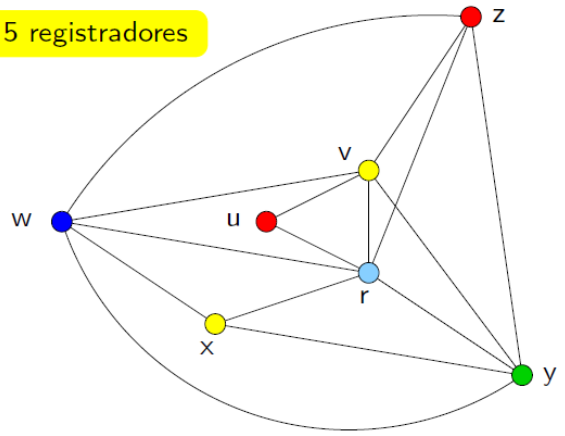


Alocação de Registradores

- Otimização de código. Variáveis mais usadas ficam nos registradores mais rápidos.
- Vértices são as variáveis e arestas conectam as variáveis necessárias ao mesmo tempo.
- $\chi(G)$ indica o mínimo de registradores necessários para evitar *overswapping*.

5 registradores

r: 1 a 6
u: passo 2
v: de 2 a 4
w: 1, 3 e 5
x: 1 e 6
y: 3 a 6
z: 4 e 5



Coloração de Vértices – Aplicações

Sudoku

- O sudoku é uma variação do problema de coloração de vértices.
- Cada célula representa um vértice e existe uma aresta entre dois vértices se eles estão em uma mesma linha, mesma coluna ou mesmo bloco.

	6		1		4		5	
		8	3		5	6		
								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

→

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Coloração de Vértices – Aplicações

Sudoku

- O sudoku é uma variação do problema de coloração de vértices.
- Cada célula representa um vértice e existe uma aresta entre dois vértices se eles estão em uma mesma linha, mesma coluna ou mesmo bloco.

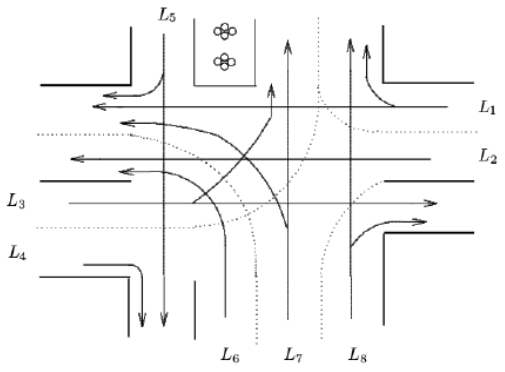
	6		1		4		5	
		8	3		5	6		
								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

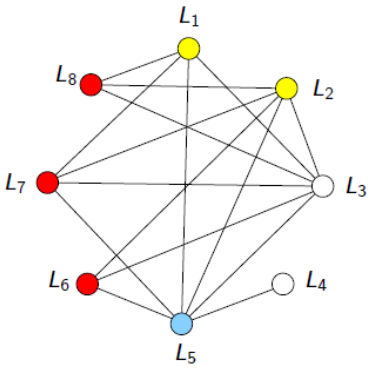
→

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Problema do Semáforo

- Um cruzamento de trânsito pode ser modelado em um grafo, no qual os vértices representam o fluxo de uma rua para outra.
- Dois vértices são ligados por arestas caso os respectivos fluxos não possam estar ativos ao mesmo tempo.



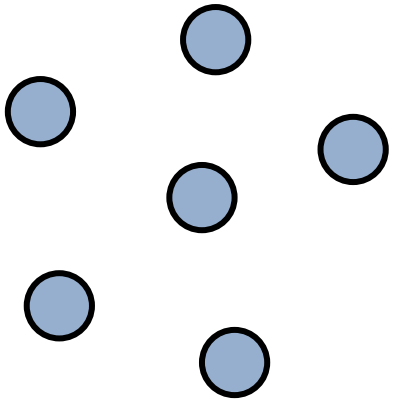


Propriedades do Número Cromático

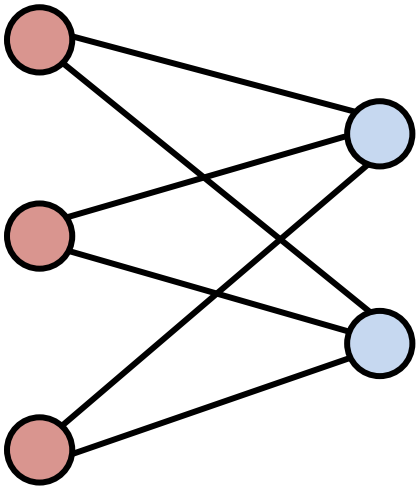
$\chi(G) = 1$ se e somente se G é completamente desconexo.

$\chi(G) = 2$ se G é um grafo bipartido.

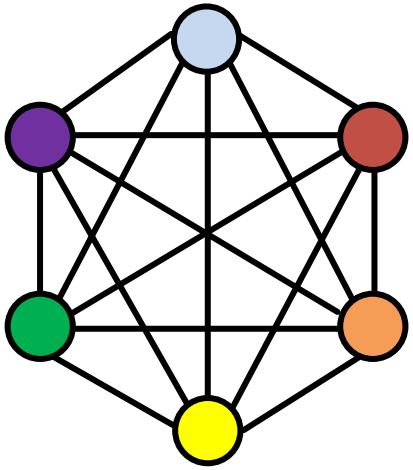
$\chi(G) = n$ se G é um grafo completo de ordem n.



$\chi(G) = 1$



$\chi(G) = 2$



$\chi(G) = 6$

Propriedades do Número Cromático

$\chi(G) = 1$ se e somente se G é completamente desconexo.

$\chi(G) = 2$ se G é um grafo bipartido.

$\chi(G) = n$ se G é um grafo completo de ordem n.

$\chi(G) \leq 4$ para qualquer grafo planar, conforme o teorema das Quatro Cores de Appel & Haken (1976)

Definições:

- **Conjectura:** proposição que é consistente com dados conhecidos, mas nunca foi verificada ou mostrada ser falsa. Sinônimo de hipótese.

- **Teorema:** Afirmação que pode ser provada como verdadeira via outras afirmações já demonstradas (ex. teoremas), juntamente com afirmações anteriormente aceitas (ex. axiomas). A prova é o processo de demonstrar que um teorema está correto.

Algoritmo – Coloração de Vértices

Heurística conhecida como algoritmo de coloração sequencial, produz uma coloração viável de um grafo, embora nem sempre ótima (número cromático).

No início de cada iteração existe uma coloração válida porém incompleta que usa as cores codificadas por 0, 1, 2 ... k−1. Deve-se priorizar as cores de menor índice.

A cada interação:

```
coloracaoSequencial(G, v)
1 idCor = 0;
2 Iniciar pelo vértice incolor v;
3 if alguma idCor não é usada por nenhum adjacente de v
    then
4 |   cor[v] = idCor;
5 end
6 else
7 |   idCor = idCor + 1;
8 |   cor[v] = idCor;
9 end
```

Algoritmo – Coloração de Vértices

Heurística conhecida como algoritmo de coloração sequencial, produz uma coloração viável de um grafo, embora nem sempre ótima (número cromático).

No início de cada iteração existe uma coloração válida porém incompleta que usa as cores codificadas por 0, 1, 2 ... $k-1$. Deve-se priorizar as cores de menor índice.

A cada interação:

coloracaoSequencial(G, v)

```
1 idCor = 0;
2 Iniciar pelo vértice incolor v;
3 if alguma idCor não é usada por nenhum adjacente de v
    then
4 |   cor[v] = idCor;
5 end
6 else
7 |   idCor = idCor + 1;
8 |   cor[v] = idCor;
9 end
```

No geral, cada iteração tem mais de uma cor disponível para o vértice v.

O algoritmo escolhe qualquer uma das cores sem se preocupar com as consequências que essa escolha terá em iterações futuras (*greedy*).

Algoritmo – Grafo Bipartido

Os vértices de um grafo bipartido podem ser particionados em dois conjuntos independentes.

A coloração deste grafo necessita apenas de duas cores, uma para cada conjunto independente, logo $\chi(G) = 2$.

Teorema: G é bipartido se e somente se não possui ciclo ímpar.

verificarBipartido(G, v)

1 Iniciar pelo vértice v e colori-lo com a Cor1;
2 Colorir todos os adjacentes de de v com a Cor2;
3 Continuar colorindo todos os adjacentes dos vértices já coloridos usando ou a Cor1 ou a Cor2. Ao atribuir as cores, se encontrar um adjacente colorido com a mesma cor do vértice atual, então G não pode ser colorido com apenas duas cores;

Algoritmo – Grafo Bipartido

Os vértices de um grafo bipartido podem ser particionados em dois conjuntos independentes.

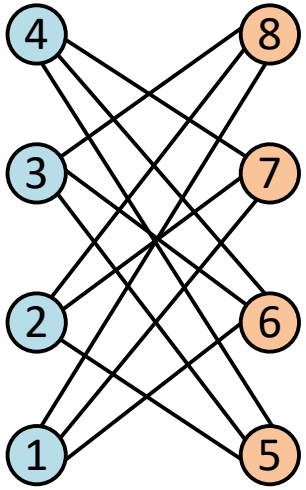
A coloração deste grafo necessita apenas de duas cores, uma para cada conjunto independente, logo $\chi(G) = 2$.

Teorema: G é bipartido se e somente se não possui ciclo ímpar.

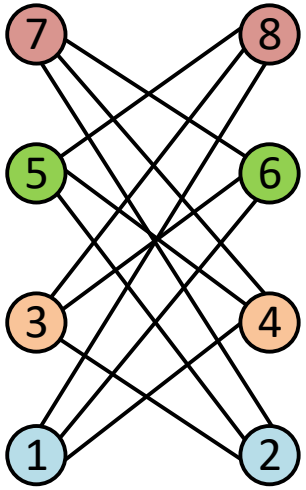
verificarBipartido(G, v)

1 Iniciar pelo vértice v e colori-lo com a Cor1;
2 Colorir todos os adjacentes de de v com a Cor2;
3 Continuar colorindo todos os adjacentes dos vértices já coloridos usando ou a Cor1 ou a Cor2. Ao atribuir as cores, se encontrar um adjacente colorido com a mesma cor do vértice atual, então G não pode ser colorido com apenas duas cores;

Dependendo da ordem em que os vértices são considerados na análise, é possível que uma determinada ordem de vértices leve a uma solução ótima ou não.



$\chi(G) = 2$
(ótimo)



$\chi(G) = 4$

Coloração de Arestas

De maneira análoga a coloração de vértices, a coloração de arestas visa a atribuição de cores diferentes para as arestas incidentes em vértices em comum.

O índice cromático de arestas de um grafo, representado por $\chi'(G)$, corresponde ao menor número de cores que podem ser usadas na coloração das arestas.

Coloração de Arestas

De maneira análoga a coloração de vértices, a coloração de arestas visa a atribuição de cores diferentes para as arestas incidentes em vértices em comum.

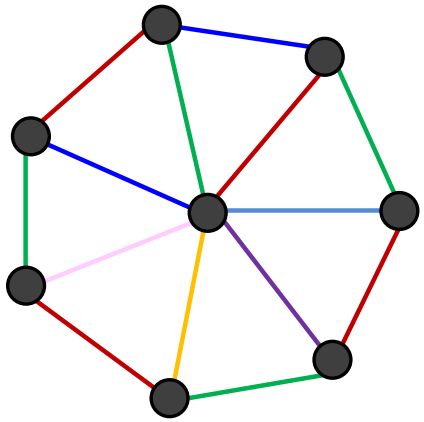
O índice cromático de arestas de um grafo, representado por $\chi'(G)$, corresponde ao menor número de cores que podem ser usadas na coloração das arestas.

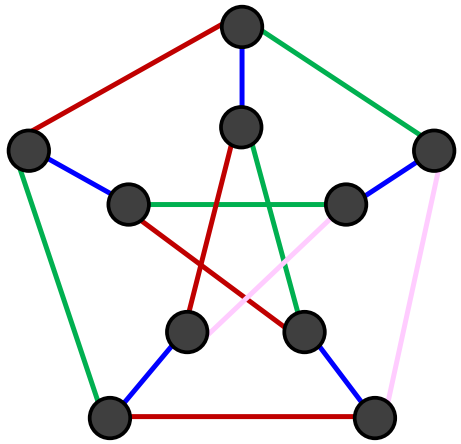
Teorema de Vizing: O limite inferior para $\chi'(G)$ é dado por $\chi'(G) \leq \Delta(G) + 1$, sendo Δ o maior grau de G. O grafo é denominado classe 1 quando $\chi'(G) = \Delta(G)$, os demais casos são classe 2.

$\Delta(G) = 7$

$\chi'(G) = 7$

grafo classe 1





$\Delta(G) = 3$

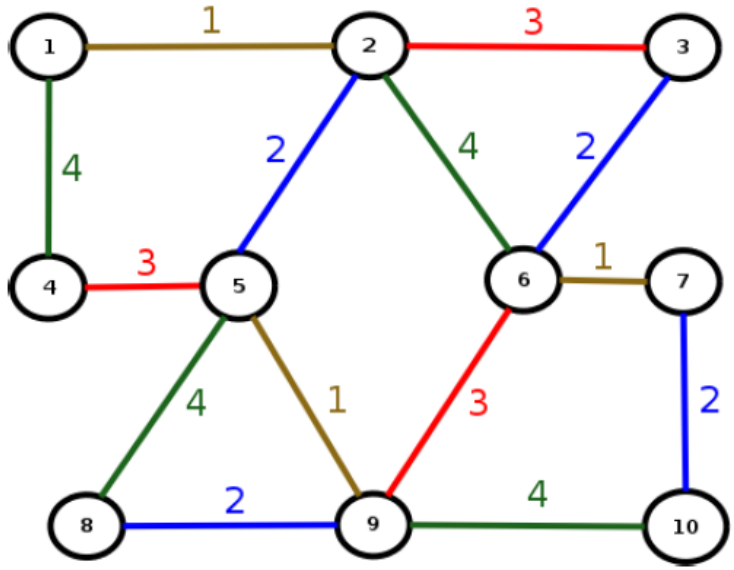
$\chi'(G) = 4$

grafo classe 2

Coloração de Arestas – Exemplo

Organização de reuniões entre duas pessoas considerando um grupo com dez pessoas.

Com objetivo de eliminar os conflitos de horários, todas as entrevistas poderiam ser agendadas em momentos distintos.



Solução ótima para o problema, são necessários quatro horários.

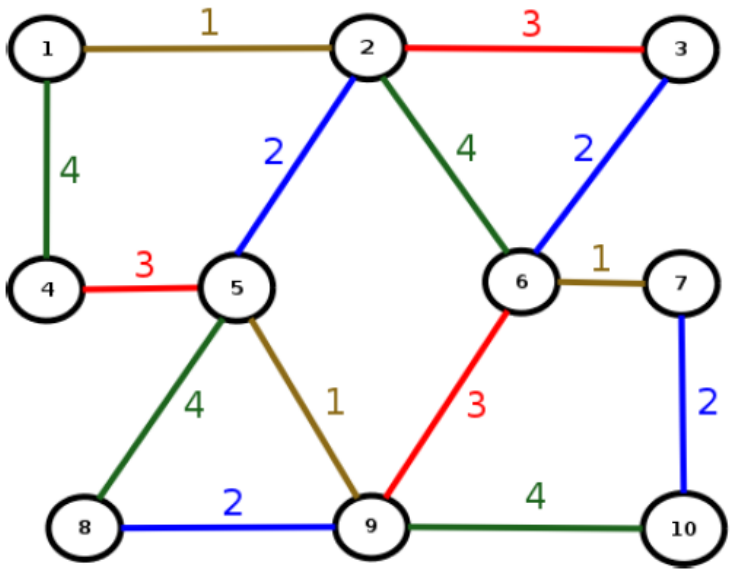
Coloração de Arestas – Exemplo

Organização de reuniões entre duas pessoas considerando um grupo com dez pessoas.

Com objetivo de eliminar os conflitos de horários, todas as entrevistas poderiam ser agendadas em momentos distintos.

Contudo, uma menor quantidade de recursos será desperdiçada (ex. tempo, salas de reuniões, horários, etc) se diversas entrevistas forem realizadas paralelamente.

Vértices representam as pessoas e as arestas as reuniões. Cada cor representa um horário, com todas as reuniões da mesma cor ocorrendo simultaneamente.



Solução ótima para o problema, são necessários quatro horários.

Perguntas? Sugestões?