Algoritmos e Estruturas de Dados II

Capítulo 17: Aplicações de DFS

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG – Campus Timóteo

Fevereiro de 2026

Contents

Introdução: O Poder da DFS

Na aula anterior estudamos a DFS como algoritmo de travessia. Nesta aula veremos que, com pequenas modificações, a DFS resolve problemas fundamentais de grafos — todos em $O(V + E)$.

As quatro aplicações desta aula são:

1 **Ordenação Topológica** — escalonar tarefas com dependências.

2 **Detecção de Ciclos** — verificar se um dígrafo é DAG.

3 **Componentes Conexos e Flood Fill** — contar regiões isoladas.

4 **Verificação de Bipartição** — 2-coloração de grafos.

• Teoria

Em todas as aplicações, o que muda é *o que fazemos* ao entrar e sair de um vértice durante a DFS. A estrutura central ($O(V + E)$, pilha de recursão, vetor `visited[]`) permanece a mesma.

Ordenação Topológica

Motivação e Definição

Imagine que você precisa executar tarefas com dependências: a tarefa B só pode começar após A terminar. Em Computação, esse cenário aparece em:

- **Build systems:** O `Makefile` compila os módulos na ordem certa.
- **Gerenciadores de pacotes:** O `apt` instala dependências antes do pacote principal.
- **Currículos acadêmicos:** Cálculo I antes de Cálculo II.
- **Compiladores:** Processar declarações antes dos usos.

• Teoria

Definição. Dado um dígrafo $G = (V, E)$, uma **ordenação topológica** é uma sequência linear $v_1, v_2, \ldots, v_{|V|}$ dos vértices tal que, para toda aresta $(u \to v) \in E$, u aparece antes de v na sequência.

Requisito fundamental: A ordenação topológica só existe se G for um **DAG** (Directed Acyclic Graph). Se existir ciclo (A depende de B e B depende de A), é impossível ordenar.

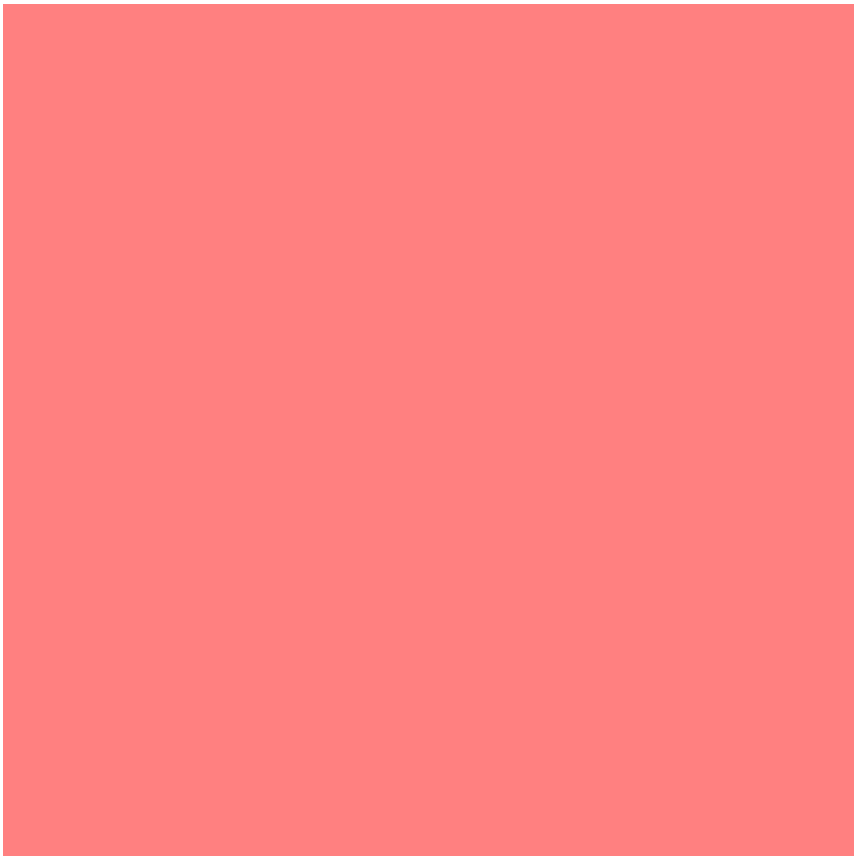

Figure 1: DAG de dependências e uma ordenação topológica válida.

Algoritmo via DFS (Pós-Ordem)

Intuição: Na DFS, um vértice u é *finalizado* (fica PRETO) somente após todos os seus descendentes terem sido finalizados. Logo, u deve aparecer *depois* de seus descendentes na lista — mas queremos u *antes* de suas dependências. Basta **inverter** a ordem de finalização.

Algoritmo:

1 Execute a DFS completa.

2 Quando um vértice u for finalizado, empilhe-o.

3 A ordenação topológica é a pilha desempilhada (pós-ordem reversa).

```
import java.util.*;

public class TopoSortDFS {
    static boolean[] visitado;
    static Deque<Integer> pilha = new ArrayDeque<>();
    static List<List<Integer>> adj;

    static void dfsTopo(int u) {
        visitado[u] = true;
```

```
        for (int v : adj.get(u))
            if (!visitado[v]) dfsTopo(v);
        pilha.push(u);  // entra na pilha ao SAIR (pos-ordem)
    }

    static List<Integer> topoSort(int V) {
        visitado = new boolean[V];
        for (int i = 0; i < V; i++)
            if (!visitado[i]) dfsTopo(i);
        return new ArrayList<>(pilha);
    }
}
```

Rastreamento no exemplo: DAG com vértices $\{0, 1, 2, 3, 4\}$ e arestas $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$, $3 \to 4$.

Ação	Pilha de recursão	Pilha de saída
dfsTopo(0) entra	[0]	[]
dfsTopo(1) entra	[0,1]	[]
dfsTopo(3) entra	[0,1,3]	[]
dfsTopo(4) entra	[0,1,3,4]	[]
4 finaliza, push(4)	[0,1,3]	[4]
3 finaliza, push(3)	[0,1]	[4,3]
1 finaliza, push(1)	[0]	[4,3,1]
dfsTopo(2) entra	[0,2]	[4,3,1]
3 já visitado	[0,2]	[4,3,1]
2 finaliza, push(2)	[0]	[4,3,1,2]
0 finaliza, push(0)	[]	[4,3,1,2,0]

Desempilhando: $0 \to 2 \to 1 \to 3 \to 4$ — ordem topológica válida!

Algoritmo de Kahn (BFS por Grau de Entrada)

O algoritmo de Kahn é uma alternativa iterativa que usa BFS e detecta ciclos naturalmente.

Intuição: Um vértice sem dependências (grau de entrada 0) pode ser executado imediatamente. Após executá-lo, removemos suas contribuições aos vizinhos — algum vizinho pode ter seu grau zerado e entrar na fila.

```
public List<Integer> kahn(int V, List<List<Integer>> adj) {
    // 1. Calcular grau de entrada de todos
    int[] inDegree = new int[V];
    for (int u = 0; u < V; u++)
        for (int v : adj.get(u)) inDegree[v]++;

    // 2. Enfileirar vértices sem dependências
    Queue<Integer> fila = new LinkedList<>();
    for (int i = 0; i < V; i++)
```

```
        if (inDegree[i] == 0) fila.add(i);

    // 3. Processar
    List<Integer> ordem = new ArrayList<>();
    while (!fila.isEmpty()) {
        int u = fila.poll();
        ordem.add(u);
        for (int v : adj.get(u)) {
            inDegree[v]--;                  // remove dependencia de u
            if (inDegree[v] == 0)
                fila.add(v);                // v liberado!
        }
    }

    // 4. Verificar ciclo
    if (ordem.size() != V)
        throw new RuntimeException("DAG invalido: ciclo detectado!");
    return ordem;
}
```

▷ Exemplo

Rastreamento no mesmo exemplo ($0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$, $3 \to 4$):

Passo	Processa	inDegree[] atualizado	Fila
Início	—	[0,1,1,2,1]	[0]
1	0	inD[1]→0, inD[2]→0	[1,2]
2	1	inD[3]→1	[2]
3	2	inD[3]→0	[3]
4	3	inD[4]→0	[4]
5	4	—	[]

Resultado: $[0, 1, 2, 3, 4]$ — ordem topológica válida.

Comparação DFS vs Kahn

Critério	DFS (pós-ordem)	Kahn (BFS)
Estilo	Recursivo	Iterativo
Detecta ciclo	Via 3 cores (extra)	**Automaticamente** (`size < V`)
Ordem produzida	Pós-ordem reversa	Ordem de grau de entrada
Risco Stack Overflow	Sim (grafos lineares)	Não
Complexidade	$O(V + E)$	$O(V + E)$

Detecção de Ciclos

Em Grafos Direcionados

• Teoria

Teorema. Um dígrafo possui ciclo ⇔ a DFS encontra uma **Back Edge** — aresta $(u \to v)$ onde v está atualmente na pilha de recursão (cor CINZA).

```
// 0=BRANCO (nao visitado), 1=CINZA (na pilha), 2=PRETO (finalizado)
int[] cor;

boolean temCicloDir(int u) {
    cor[u] = 1;  // CINZA: entrou na pilha
    for (int v : adj.get(u)) {
        if (cor[v] == 1) return true;          // back edge -> ciclo!
        if (cor[v] == 0 && temCicloDir(v)) return true;
    }
    cor[u] = 2;  // PRETO: saiu da pilha
    return false;
}

boolean grafoPossuiCiclo(int V) {
    cor = new int[V]; // todos BRANCO
    for (int i = 0; i < V; i++)
        if (cor[i] == 0 && temCicloDir(i)) return true;
    return false;
}
```

Por que só CINZA indica ciclo? Uma aresta para um vértice PRETO é uma *Forward Edge* (atalho para descendente) ou *Cross Edge* (para outro ramo) — ambas inofensivas. Apenas aresta para vértice CINZA (ainda na pilha atual) cria um ciclo real.

Em Grafos Não Direcionados

Em grafos não direcionados, qualquer aresta para um vértice já visitado — exceto o pai imediato — constitui um ciclo.

```
boolean temCicloNaoDir(int u, int pai) {
    visitado[u] = true;
    for (int v : adj.get(u)) {
        if (!visitado[v]) {
            if (temCicloNaoDir(v, u)) return true;
        } else if (v != pai) {
            return true;  // aresta para nao-pai visitado -> ciclo
        }
    }
    return false;
}
```

Componentes Conexos e Flood Fill

Componentes Conexos

Em um grafo não-direcionado, um **componente conexo** é um subgrafo maximal em que existe caminho entre qualquer par de vértices. A DFS descobre exatamente um componente por chamada inicial.

```
int[] comp;    // comp[v] = ID do componente de v
boolean[] visitado;

void dfs(int u, int id) {
    visitado[u] = true;
    comp[u] = id;
    for (int v : adj.get(u))
        if (!visitado[v]) dfs(v, id);
}

int contarComponentes(int V) {
    visitado = new boolean[V];
    comp = new int[V];
    Arrays.fill(comp, -1);
    int count = 0;
    for (int i = 0; i < V; i++) {
        if (!visitado[i]) {
            dfs(i, count);
            count++;
        }
    }
    return count;
}
```

▷ Exemplo

Exemplo. Grafo com vértices $\{0, \ldots, 5\}$ e arestas $\{0\text{-}1, 1\text{-}2, 3\text{-}4\}$:

- `dfs(0, 0)`: visita $\{0, 1, 2\}$ — comp[0]=comp[1]=comp[2]=0
- `dfs(3, 1)`: visita $\{3, 4\}$ — comp[3]=comp[4]=1
- `dfs(5, 2)`: visita $\{5\}$ — comp[5]=2

Total: **3 componentes**. Para verificar se u e v estão conectados: `comp[u] == comp[v]` em $O(1)$.

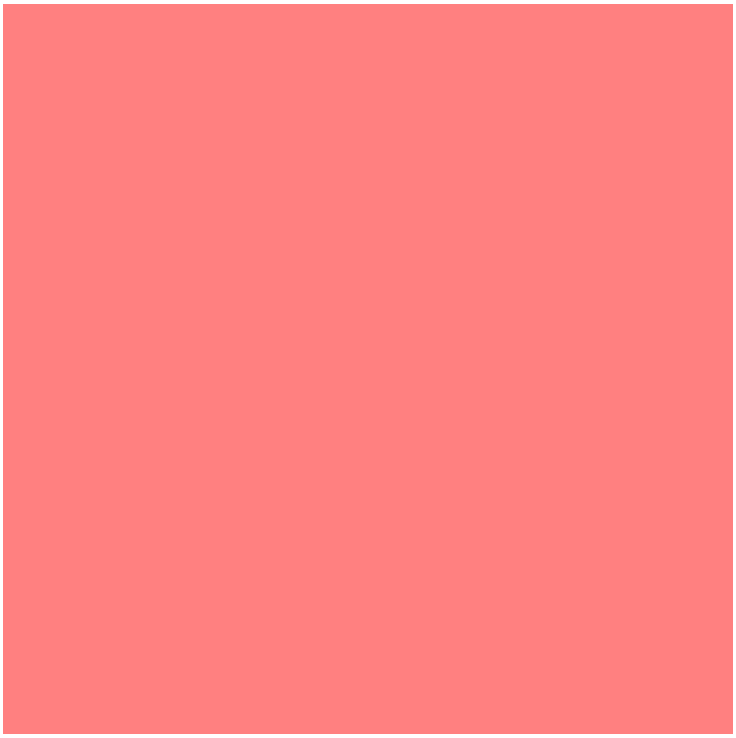

Figure 2: Grafo com 3 componentes conexos, cada um descoberto por uma chamada DFS diferente.

Flood Fill em Grids

O Flood Fill ("balde de tinta") é a aplicação de componentes conexos em matrizes 2D. Cada célula é um vértice; vizinhos são as 4 células adjacentes (cima, baixo, esquerda, direita).

Problema clássico: dado um grid com células `'.'` (terra) e `'W'` (água), contar o número de ilhas.

```
int[] dr = {-1, 1, 0,  0}; // cima, baixo, esquerda, direita
int[] dc = { 0, 0, -1, 1};

void dfsGrid(char[][] g, boolean[][] vis, int r, int c) {
    vis[r][c] = true;
    for (int d = 0; d < 4; d++) {
        int nr = r + dr[d], nc = c + dc[d];
        if (nr >= 0 && nr < g.length && nc >= 0 && nc < g[0].length
                && !vis[nr][nc] && g[nr][nc] == '.') {
            dfsGrid(g, vis, nr, nc);
        }
    }
}

int contarIlhas(char[][] g) {
    int R = g.length, C = g[0].length;
```

```
    boolean[][] vis = new boolean[R][C];
    int ilhas = 0;
    for (int r = 0; r < R; r++)
        for (int c = 0; c < C; c++)
            if (!vis[r][c] && g[r][c] == '.') {
                dfsGrid(g, vis, r, c);
                ilhas++;
            }
    return ilhas;
}
```

▷ Exemplo

Exemplo de grid 4×5:

```
. W W . .
. . W . W
W W . . W
W . W W .
```

Ilhas: {(0,0),(1,0),(1,1)}, {(0,3),(0,4),(1,3),(2,2),(2,3)}, {(1,4) bloqueado}, {(3,1)}, {(3,4)} — **4 ilhas**.

• Teoria

Cuidado com Stack Overflow. Em grids grandes ($N \times M > 10^6$ células), a DFS recursiva pode estourar a pilha. Prefira BFS (iterativa) nesses casos.

Verificação de Bipartição

Definição e Teorema

• Teoria

Definição. Um grafo G é **bipartido** se V pode ser particionado em dois conjuntos A e B (disjuntos) tal que toda aresta conecta um vértice de A a um de B.

Teorema. G é bipartido $\Leftrightarrow$ G não contém nenhum ciclo de comprimento ímpar.

Exemplos de grafos bipartidos:

- Alunos × Disciplinas: aresta = matrícula.
- Candidatos × Vagas: aresta = candidatura.
- Times × Jogos: aresta = participação.

A verificação é feita por **2-coloração**: tentamos pintar cada vértice com cor 0 ou 1, de forma que vizinhos nunca tenham a mesma cor.

Implementação por BFS

```
// color[v]: -1=sem cor, 0=grupo A, 1=grupo B
int[] color;

boolean bfsColor(int start) {
    Queue<Integer> fila = new LinkedList<>();
    fila.add(start);
    color[start] = 0;
    while (!fila.isEmpty()) {
        int u = fila.poll();
        for (int v : adj.get(u)) {
            if (color[v] == -1) {
                color[v] = 1 - color[u];  // cor oposta
                fila.add(v);
            } else if (color[v] == color[u]) {
                return false;  // conflito -> nao bipartido
            }
        }
    }
    return true;
}

boolean isBipartite(int V) {
    color = new int[V];
    Arrays.fill(color, -1);
    for (int i = 0; i < V; i++)
        if (color[i] == -1 && !bfsColor(i))
            return false;  // componente nao bipartido
    return true;
}
```

▷ Exemplo

Exemplo 1 — bipartido. Quadrado 0-1-2-3-0:

- color[0]=0, color[1]=1, color[2]=0, color[3]=1.
- Verificar aresta 3 → 0: color[3]=1 ≠ color[0]=0. ✓ **Bipartido**.

Exemplo 2 — não bipartido. Triângulo 0-1-2-0:

- color[0]=0, color[1]=1, color[2]=0.
- Verificar aresta 2 → 0: color[2]=0 = color[0]=0. × **Não bipartido!**

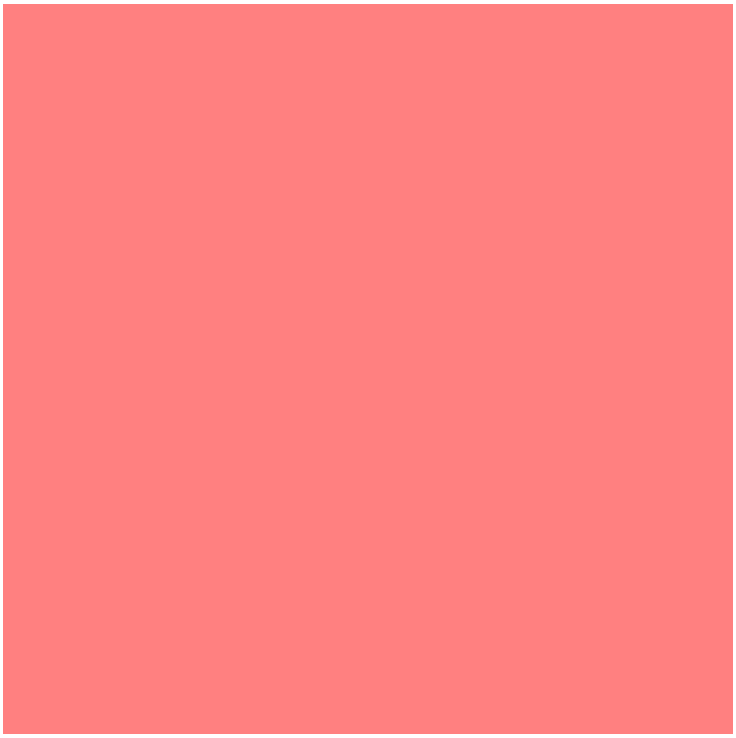

Figure 3: Grafo bipartido: vértices em dois grupos (0 e 1); arestas apenas entre grupos.

Resumo Comparativo

Problema	**Algoritmo**	**Estrutura extra**	**Complexidade**
Ord. Topológica	DFS pós-ordem	Pilha de saída	$O(V+E)$
Ord. Topológica	Kahn (BFS)	Fila + `inDegree[]`	$O(V+E)$
Ciclo (direcionado)	DFS 3 cores	`cor[]` (B/C/P)	$O(V+E)$
Ciclo (não-dir.)	DFS + pai	`visitado[]` + pai	$O(V+E)$
Componentes Conexos	DFS/BFS	`comp[]`	$O(V+E)$
Flood Fill (grid)	DFS/BFS	`visited[][]`	$O(R \cdot C)$
Bipartição	BFS 2 cores	`color[]`	$O(V+E)$

Exercícios

Exercícios Conceituais

1 Explique a diferença entre a ordenação topológica por DFS (pós-ordem) e o algoritmo de Kahn. Em qual cenário você preferiria cada um?

2 Por que um grafo direcionado com ciclos não admite ordenação topológica? Dê um exemplo concreto onde um ciclo torna o escalonamento impossível.

3 Explique por que, na detecção de ciclos em dígrafos, uma aresta para um vértice PRETO não indica ciclo, mas uma aresta para um vértice CINZA indica.

4 Um grafo bipartido pode ter ciclos? Se sim, quais tipos são permitidos? Justifique usando o teorema do ciclo ímpar.

Exercícios Analíticos

1 Dado o DAG: $A \to B$, $A \to C$, $B \to D$, $C \to D$, $D \to E$.

- Execute o algoritmo de Kahn passo a passo, mostrando o estado de `inDegree[]` e da fila a cada iteração.
- Execute a DFS pós-ordem e mostre a pilha de saída a cada finalização.
- As duas ordens produzidas são iguais? Devem ser?

2 Dado o dígrafo: $0 \to 1$, $1 \to 2$, $2 \to 0$, $2 \to 3$.

- Execute a DFS com 3 cores a partir do vértice 0.
- Identifique a *Back Edge* que indica o ciclo.
- Execute o algoritmo de Kahn. O que acontece?

3 Dado o grafo não-direcionado com arestas $\{0\text{-}1, 1\text{-}2, 2\text{-}3, 3\text{-}4, 0\text{-}4, 1\text{-}3\}$:

- Execute a verificação de bipartição por BFS a partir do vértice 0.
- O grafo é bipartido? Justifique pelo teorema do ciclo ímpar.

Exercícios de Programação

1 Implemente as duas versões de ordenação topológica (DFS e Kahn) e compare as saídas para:

- DAG linear: $0 \to 1 \to 2 \to 3 \to 4$.
- DAG com múltiplos caminhos: $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$.
- Dígrafo com ciclo: $0 \to 1 \to 2 \to 0$.

2 Implemente Flood Fill em um grid `char[][]`:

- Versão DFS recursiva.
- Versão BFS iterativa.
- Compare o desempenho em grids de 100×100 e 1000×1000.
- Modifique para retornar também o *tamanho* de cada componente.

3 **Desafio:** Implemente um sistema de resolução de dependências similar ao de um gerenciador de pacotes:

- Leia um conjunto de pacotes e suas dependências.
- Use o algoritmo de Kahn para determinar a ordem de instalação.
- Se houver ciclo, reporte quais pacotes formam a dependência circular.
- Exiba os pacotes que podem ser instalados em paralelo (mesmo nível de dependência).