AED2 - Algoritmos e Estr. de Dados II

Aula 17: Aplicações de DFS

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

O Poder da DFS: Além da Travessia

Na Aula 16 aprendemos a *executar* a DFS. Hoje usamos ela para **resolver problemas reais**.

Aplicações desta aula

1. **Ordenação Topológica** — escalonar tarefas com dependências
2. **Detecção de Ciclos** — verificar DAGs
3. **Componentes Conexos** — Flood Fill
4. **Verificação de Bipartição** — 2-coloração

Todas com complexidade $O(V + E)$ — a mesma da DFS base.

Pré-requisito

Revise as 3 cores da DFS:
Branco = não visitado
Cinza = na pilha de recursão
Preto = finalizado

Complexidade

Todas as aplicações desta aula:
$O(V + E)$

Ordenação Topológica: Motivação

Problema: Você tem tarefas com dependências. Em que ordem executá-las?

Exemplos reais:

- Disciplinas com pré-requisitos
- Build systems (Makefile, Gradle)
- Compilação de módulos interdependentes
- Instalação de pacotes de software (apt, npm)

Modelagem

Cada tarefa = vértice; "A depende de B" = aresta $B \to A$.

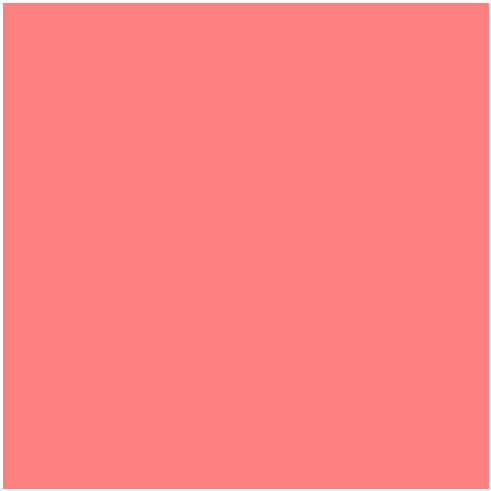

Figure: DAG de dependências e uma ordem topológica válida.

Ordenação Topológica: Exemplo

DAG de disciplinas: AED1 $\rightarrow$ AED2, Calc1 $\rightarrow$ Calc2, AED1 $\rightarrow$ Prog2, Prog2 $\rightarrow$ AED2

Ordem válida 1: AED1, Calc1, Prog2, Calc2, AED2

Ordem válida 2: Calc1, AED1, Calc2, Prog2, AED2

Observação

A ordenação topológica pode **não ser única**. Qualquer ordem que respeite as setas é válida.

Dois algoritmos

1. **DFS + pós-ordem:** elegante, recursivo
2. **Kahn (BFS):** iterativo, detecta ciclo facilmente

Ambos: $O(V + E)$.

Algoritmo 1: DFS com Pós-Ordem

Ideia: Um vértice *u* só entra na lista *depois* que todos os seus descendentes foram processados.

```
Deque<Integer> pilha = new ArrayDeque<>();
boolean[] visitado = new boolean[V];

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

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

Rastreamento

DAG: $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$

dfsTopo(0)	entra
dfsTopo(1)	entra
dfsTopo(3)	entra
3 finaliza	push(3)
1 finaliza	push(1)
dfsTopo(2)	entra
3 já visitado	—
2 finaliza	push(2)
0 finaliza	push(0)

Pilha $\to$ lista: $[0, 2, 1, 3]$

Algoritmo 2: Kahn (BFS por Grau de Entrada)

```
List<Integer> kahn(int V, List<List<Integer>> adj) {
    int[] inDegree = new int[V];
    for (int u = 0; u < V; u++)
        for (int v : adj.get(u)) inDegree[v]++;

    Queue<Integer> fila = new LinkedList<>();
    for (int i = 0; i < V; i++)
        if (inDegree[i] == 0) fila.add(i); // sem dependencias

    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
            if (inDegree[v] == 0) fila.add(v);
        }
    }
    if (ordem.size() != V)
        throw new RuntimeException("Ciclo detectado!");
    return ordem;
}
```

Detecção de Ciclos em Dígrafos

Um dígrafo possui ciclo $\Leftrightarrow$ a DFS encontra uma **Back Edge** (aresta para vértice CINZA).

```
// 0=BRANCO, 1=CINZA, 2=PRETO
int[] cor;

boolean temCiclo(int u) {
    cor[u] = 1; // CINZA: na pilha
    for (int v : adj.get(u)) {
        if (cor[v] == 1) return true; // back
            edge!
        if (cor[v] == 0 && temCiclo(v)) return
            true;
    }
    cor[u] = 2; // PRETO: finalizado
    return false;
}

boolean grafoPossuiCiclo(int V) {
    cor = new int[V]; // 0 por padrao
    for (int i = 0; i < V; i++)
        if (cor[i] == 0 && temCiclo(i)) return
            true;
```

Por que CINZA, não PRETO?

Aresta para vértice **PRETO** é uma *Forward* ou *Cross Edge* — não é ciclo. Apenas aresta para vértice **CINZA** (ainda na pilha) é uma *Back Edge* = **ciclo**.

Grafo não-direcionado

Para grafos não-dir., qualquer vértice visitado que *não seja o pai imediato* indica ciclo.

Alternativa

Kahn: se resultado tiver $< V$ vértices, há ciclo.

Componentes Conexos: Motivação

Problema: Dado um grafo não-direcionado, quantos subgrafos conexos existem?

Exemplos reais:

- Quantas ilhas num mapa de grid?
- Quantas redes de computadores independentes?
- Em redes sociais: grupos sem contato entre si
- Flood Fill: colorir região de uma imagem (balde do Paint)

Ideia

Cada chamada de DFS (ou BFS) a partir de um vértice não visitado descobre **exatamente um** componente conexo.

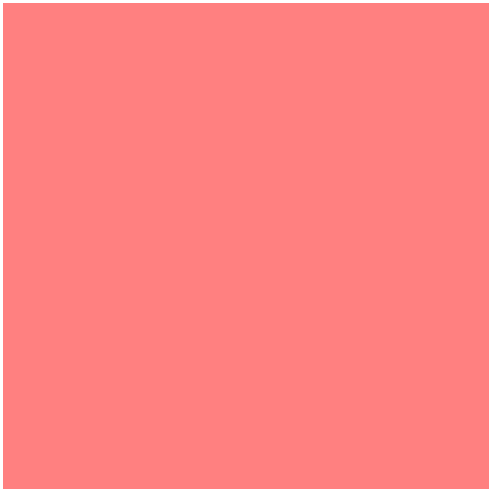

Figure: Grafo com 3 componentes conexos.

Componentes Conexos: Implementação

```
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];
    int count = 0;
    for (int i = 0; i < V; i++) {
        if (!visitado[i]) {
            dfs(i, count);  // todo o componente recebe ID 'count'
            count++;
        }
    }
    return count;
}
```

Flood Fill em Grid

Problema: Dado um grid de '.' (terra) e 'W' (água), conte o número de ilhas.

Exemplo (grid 4×5):

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

Resposta: **3 ilhas**

```java
int[] dr = {-1, 1, 0,  0};
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 &&
```

```java
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;
}
```

Atenção ao Stack Overflow

Grafos Bipartidos: Conceito

Definição: Um grafo é **bipartido** se seus vértices podem ser divididos em dois conjuntos A e B tal que toda aresta conecta um vértice de A a um de B.

Exemplos reais:

- Alunos $\times$ Disciplinas matriculadas
- Candidatos $\times$ Vagas de emprego
- Times $\times$ Partidas jogadas

Teorema

Um grafo é bipartido $\Leftrightarrow$ não contém **ciclo de comprimento ímpar**.

Verificação: tente colorir com 2 cores (BFS ou DFS). Conflito de cores = não bipartido.

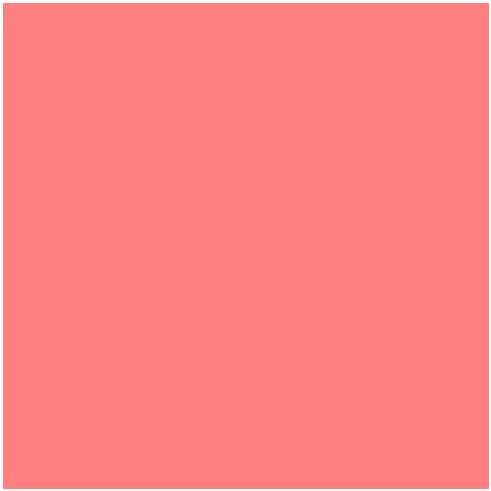

Figure: Grafo bipartido: 2 grupos, arestas só entre grupos.

Verificação de Bipartição: Implementação

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

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;
    return true;
}

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]) {
```

Resumo das Aplicações da DFS

Problema	Algoritmo	Estrutura-chave	Complexidade
Ordenação Topológica	DFS pós-ordem	Pilha de saída	$O(V+E)$
Ordenação Topológica	Kahn (BFS)	Fila + inDegree[]	$O(V+E)$
Detecção de ciclo (dir.)	DFS 3 cores	cor[] Branco/Cinza/Preto	$O(V+E)$
Componentes Conexos	DFS/BFS	comp[]	$O(V+E)$
Flood Fill (grid)	DFS/BFS	visited[][]	$O(R \times C)$
Bipartição	BFS 2 cores	color[]	$O(V+E)$

Regra de ouro

Todas essas aplicações são variações da DFS/BFS base. O que muda é **o que fazemos** quando entramos ou saímos de um vértice.

Próximas aulas

- Dijkstra: menor caminho com pesos
- MST: Kruskal e Prim
- SCC: Kosaraju/Tarjan

Exercícios

1. **[Fácil]** Dado o DAG: $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$, $3 \to 4$. Execute o algoritmo de Kahn e mostre o estado da fila e de `inDegree[]` a cada passo.

2. **[Médio]** Dado o dígrafo: $0 \to 1$, $1 \to 2$, $2 \to 3$, $3 \to 1$. Execute a DFS com 3 cores e identifique qual aresta é a *Back Edge*. O algoritmo de Kahn consegue detectar o ciclo? Como?

3. **[Médio]** Implemente em Java um método que, dado um grafo não-direcionado, retorne uma lista de listas onde cada sublista contém os vértices de um componente conexo.

4. **[Difícil]** Dada uma grid $N \times M$ com células `0` (vazia) e `1` (bloqueada), implemente Flood Fill para determinar o tamanho da maior região conexa de células `0`.