AED2 - Algoritmos e Estr. de Dados II

Aula 16: Busca em Profundidade (DFS)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

Motivação: Explorando sem Mapa

Problema: Você está num labirinto sem mapa. Como explorar *sistematicamente* sem perder nenhum caminho?

Estratégia de Teseu (mitologia grega):

- Use um **novelo de lã** para marcar onde esteve
- Vá sempre para frente até **não haver mais onde ir**
- **Retroceda** (backtrack) até a última bifurcação
- Tente o próximo caminho inexplorado

DFS na prática

Detectar ciclos, ordenar tarefas com dependências, encontrar componentes fortemente conexos — tudo com $O(V + E)$.

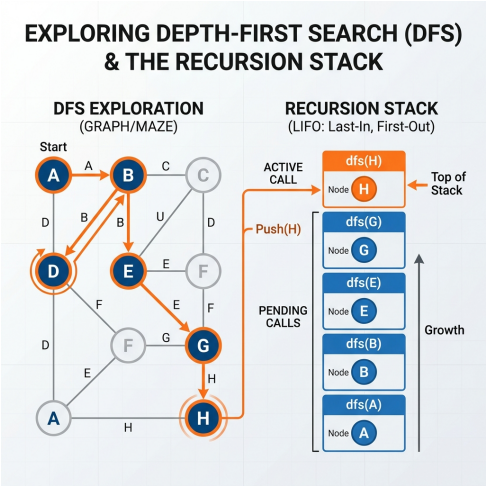


Figure: DFS e a pilha de recursão.

DFS: A Ideia em 3 Etapas

1. **Visita** o vértice atual u (marca como visitado)
2. **Explora** cada vizinho v ainda não visitado, chamando DFS(v) recursivamente
3. **Retorna** (backtrack) quando todos os vizinhos de u já foram visitados

Diferença fundamental da BFS

- BFS usa **Fila** $\Rightarrow$ expande por níveis
- DFS usa **Pilha** (recursão) $\Rightarrow$ mergulha fundo primeiro

Pseudocódigo

DFS(G, s):
  Para todo v: cor[v] $\leftarrow$ BRANCO
  DFS-VISIT(G, s)

DFS-Visit(G, u):
  cor[u] $\leftarrow$ CINZA
  para $v \in$ Adj[u]:
    se cor[v] = BRANCO:
      DFS-VISIT(G, v)
  cor[u] $\leftarrow$ PRETO

As 3 Cores dos Vértices

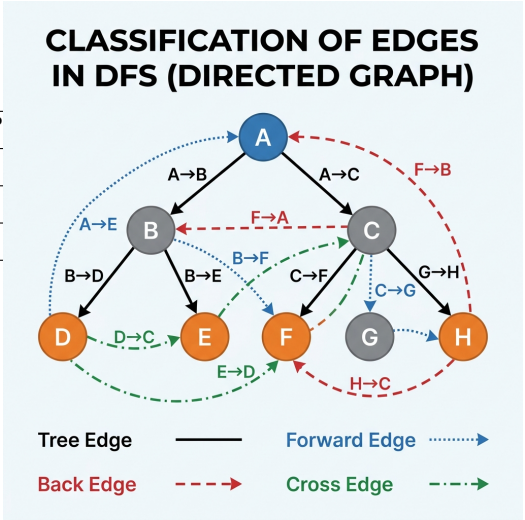
Para entender o que DFS está fazendo, usamos 3 estados:

Cor	Significado	Está na pilha?
BRANCO	Não descoberto	Não
CINZA	Em processamento	**Sim**
PRETO	Finalizado	Não

Um vértice fica **CINZA** desde o momento em que é *descoberto* até o momento em que todos os seus *descendentes* são processados.

Invariante importante

No instante em que u está CINZA, todos os vértices CINZA formam um **caminho** da raiz até



Exemplo: Grafo com 6 Vértices

Grafo não-direcionado: vértices $\{0, 1, 2, 3, 4, 5\}$

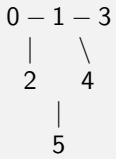
Arestas: $0-1,\ 0-2,\ 1-3,\ 1-4,\ 2-5$

Lista de Adjacência:

- 0: [1, 2] 1: [0, 3, 4]
- 2: [0, 5] 3: [1]
- 4: [1] 5: [2]

DFS a partir do vértice 0
(visita vizinhos em ordem crescente)

Estrutura do Grafo



Rastreamento da DFS — Passo a Passo

Passo	Ação	Pilha de recursão	Visitados
1	Entra em 0 (CINZA)	[0]	{0}
2	Entra em 1 (CINZA)	[0, 1]	{0,1}
3	Entra em 3 (CINZA)	[0, 1, 3]	{0,1,3}
4	3 finalizado (PRETO)	[0, 1]	{0,1,3}
5	Entra em 4 (CINZA)	[0, 1, 4]	{0,1,3,4}
6	4 finalizado (PRETO)	[0, 1]	{0,1,3,4}
7	1 finalizado (PRETO)	[0]	{0,1,3,4}
8	Entra em 2 (CINZA)	[0, 2]	{0,1,3,4,2}
9	Entra em 5 (CINZA)	[0, 2, 5]	{0,1,3,4,2,5}
10	5 finalizado; 2 finalizado; 0 finalizado	[]	todos

Ordem de visita (pré-ordem): $0 \to 1 \to 3 \to 4 \to 2 \to 5$

Implementação Recursiva em Java

```java
import java.util.*;

public class DFS {
    static boolean[] visited;
    static List<List<Integer>> adj;

    // Chama dfsVisit para cada vertice nao visitado
    public static void dfs(int V) {
        visited = new boolean[V];
        for (int i = 0; i < V; i++)
            if (!visited[i]) dfsVisit(i);  // grafos desconexos!
    }

    public static void dfsVisit(int u) {
        visited[u] = true;
        System.out.print(u + " ");          // pre-ordem

        for (int v : adj.get(u)) {
            if (!visited[v]) {
                dfsVisit(v);                // recursao (backtrack automatico)
            }
        }
```

DFS Iterativa — Pilha Explícita

```java
public static void dfsIterativa(int start) {
    Deque<Integer> pilha = new ArrayDeque<>();
    boolean[] visited = new boolean[adj.size()];

    pilha.push(start);
    visited[start] = true;

    while (!pilha.isEmpty()) {
        int u = pilha.pop();
        System.out.print(u + " ");

        for (int v : adj.get(u)) {
            if (!visited[v]) {
                visited[v] = true;
                pilha.push(v);
            }
        }
    }
}
```

Atenção à ordem de visita

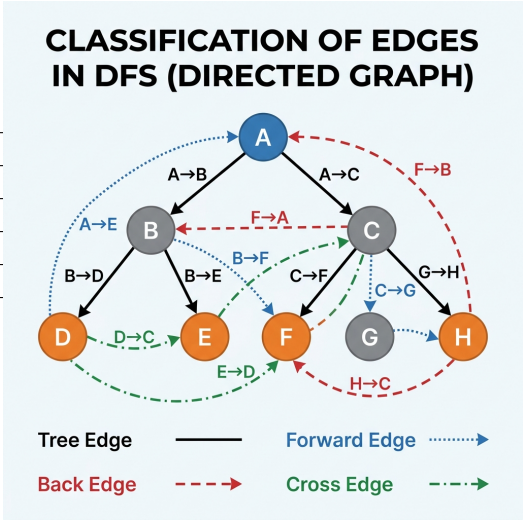
Classificação de Arestas na DFS

Durante a DFS, ao processar a aresta (u, v):

Tipo	**Condição de v**	**Ocorre em**
Tree Edge	BRANCO	Sempre
Back Edge	CINZA	Dir. e Não-dir.
Forward Edge	PRETO, desc.	Só Direcionado
Cross Edge	PRETO, não-desc.	Só Direcionado

Relevância

- **Back Edge** $\Rightarrow$ existe **ciclo**
- **Tree Edges** $\Rightarrow$ formam a árvore DFS
- Grafos não-dir.: só Tree e Back edges



Detectando Ciclos com DFS

Grafo Direcionado: ciclo $\Leftrightarrow$ existe Back Edge (aresta para vértice CINZA).

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

boolean temCiclo(int u) {
    cor[u] = 1; // CINZA
    for (int v : adj.get(u)) {
        if (cor[v] == 1)
            return true;  // Back Edge!
        if (cor[v] == 0)
            if (temCiclo(v)) return true;
    }
    cor[u] = 2; // PRETO
    return false;
}
```

Grafo Não-Direcionado: ciclo $\Leftrightarrow$ encontra vértice já visitado que *não* seja o pai imediato.

Exemplo com ciclo

Grafo: $0 \to 1 \to 2 \to 0$

Execução:

- Entra 0 (CINZA)
- Entra 1 (CINZA)
- Entra 2 (CINZA)
- Vizinho de 2: vértice 0
- cor[0] = 1 (CINZA) $\Rightarrow$ **CICLO!**

Sem ciclo $\Rightarrow$ DAG

Um grafo direcionado sem ciclos é chamado de **DAG** (Directed Acyclic Graph) e permite ordenação topológica.

Ordenação Topológica via DFS

Definição: Em um DAG, ordenação topológica é uma sequência linear dos vértices tal que toda aresta $(u \to v)$ tem u antes de v.

Exemplo — Dependências de disciplinas:

- Cálculo I $\to$ Cálculo II
- AED I $\to$ AED II
- AED I $\to$ Cálculo I (pré-req)

Algoritmo: DFS + pilha de saída

1. Execute DFS completa
2. Quando um vértice fica **PRETO**, empilhe-o
3. A ordem topológica é a pilha invertida (pós-ordem reversa)

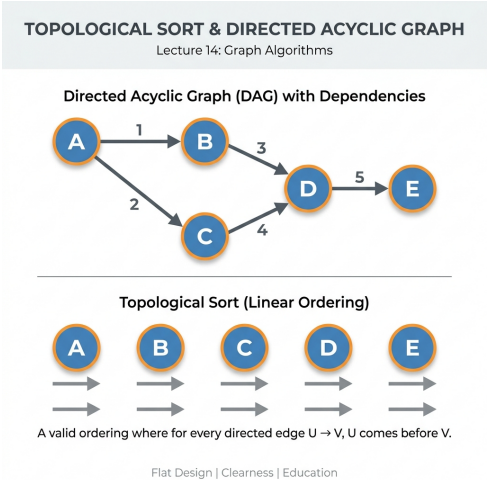


Figure: DAG e ordenação topológica.

Implementação da Ordenação Topológica

```java
import java.util.*;

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

    public static void dfsTopoVisit(int u) {
        visited[u] = true;
        for (int v : adj.get(u))
            if (!visited[v]) dfsTopoVisit(v);
        pilha.push(u);  // empilha ao finalizar (pos-ordem)
    }

    public static List<Integer> topoSort(int V) {
        visited = new boolean[V];
        for (int i = 0; i < V; i++)
            if (!visited[i]) dfsTopoVisit(i);

        List<Integer> ordem = new ArrayList<>(pilha);
        return ordem;
    }
```

Contando Componentes Conexos

Problema: Dado um grafo, quantos subgrafos conexos ele possui?

Estratégia: DFS a partir de cada vértice não visitado. Cada chamada descobre um componente.

```
1  int contarComponentes(int V) {
2      visited = new boolean[V];
3      int count = 0;
4      for (int i = 0; i < V; i++) {
5          if (!visited[i]) {
6              dfsVisit(i);  // visita 1
                   componente
7              count++;
8          }
9      }
10     return count;
11 }
```

Exemplo

Vértices: {0,1,2,3,4}
Arestas: 0-1, 1-2, 3-4

DFS(0): visita {0,1,2}
DFS(3): visita {3,4}

$\Rightarrow$ **2 componentes**

Variação útil

Guardar o `comp[]` de cada vértice permite responder "u e v estão no mesmo componente?" em $O(1)$.

Aplicações Clássicas da DFS

1. Detecção de Ciclos

Grafos direcionados e não-direcionados. Base para verificar se um DAG é válido.

2. Ordenação Topológica

Escalonamento de tarefas, compiladores (dependências), build systems (Make, Gradle).

3. Componentes Fortemente Conexos

Algoritmos de Kosaraju e Tarjan — ambos baseados em DFS. Aplicação: análise de redes.

4. Pontes e Articulações (Tarjan)

Identificar arestas cuja remoção desconecta o grafo. Crítico em redes de telecomunicações.

5. Resolver Labirintos / Backtracking

DFS + desfazer escolha. Base de N-Rainhas, Sudoku, geração de labirintos.

6. Checagem de Bipartição

Colorir o grafo com 2 cores durante DFS. Conflito de cor $\Rightarrow$ não bipartido.

DFS vs BFS — Resumo Comparativo

Critério	DFS	BFS
Estrutura	Pilha / Recursão	Fila
Caminho mínimo (sem peso)	**Não**	**Sim**
Detecção de ciclos	**Sim**	Sim
Ordenação Topológica	**Sim**	**Não**
SCCs (Kosaraju/Tarjan)	**Sim**	**Não**
Pontes e Articulações	**Sim**	**Não**
Risco de Stack Overflow	**Sim (recursiva)**	Não
Ideal para grafos profundos	**Melhor**	Ruim
Ideal para grafos largos	Ruim	**Melhor**

"DFS = exploração em profundidade, estrutura do grafo. BFS = distâncias mínimas."

Exercícios

1. **[Fácil]** Dado o grafo: $0-1$, $0-2$, $1-3$, $2-3$, $3-4$. Execute DFS a partir do vértice 0 e mostre: (a) a pilha de recursão a cada passo; (b) a ordem de visita em pré-ordem; (c) a ordem de finalização (pós-ordem).

2. **[Médio]** Implemente em Java um método que determine se um grafo **direcionado** possui ciclo, usando a coloração branco/cinza/preto.

3. **[Médio]** Dado um DAG com V vértices, implemente a ordenação topológica via DFS. Teste com: $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$. Qual a saída esperada?

4. **[Difícil]** Implemente o algoritmo de **Kosaraju** para encontrar componentes fortemente conexos em $O(V + E)$. (Dica: requer 2 passagens de DFS — uma no grafo original e outra no transposto.)

Resumo da Aula

O que aprendemos

- DFS explora em profundidade usando **pilha** (recursão)
- As 3 cores (Branco/Cinza/Preto) revelam a estrutura
- **Back Edge** $\Rightarrow$ ciclo no grafo direcionado
- **Pós-ordem reversa** $\Rightarrow$ ordenação topológica
- Complexidade $O(V + E)$ em tempo e espaço
- DFS iterativa evita Stack Overflow em grafos grandes

Próximas aulas

- Dijkstra: menor caminho com pesos
- Componentes Fortemente Conexos
- Árvore Geradora Mínima (Kruskal / Prim)

Lembre-se

"DFS: pilha, profundidade, estrutura. Se há Back Edge, há ciclo."