Depth-First Search: A Decrease and Conquer Approach

Sumanth

June 18, 2025

Abstract

Depth-First Search (DFS) is a fundamental **decrease and conquer** graph traversal algorithm that explores as far as possible along each branch before backtracking, using a stack (implicit via recursion or explicit). It is essential for solving graph-related problems like cycle detection and topological sorting. This document details the algorithm's formulation, pseudocode, complexity analysis, practical applications, and implementation considerations, emphasizing its foundational role in graph algorithms, akin to BFS, and its versatility in various graph structures.

1 Introduction

Depth-First Search (DFS) is a **decrease and conquer** algorithm for traversing or searching a graph, prioritizing depth by exploring as far as possible along a branch before backtracking. Using a stack (typically implicit through recursion), DFS is well-suited for tasks like cycle detection, topological sorting, and finding connected components. Its simplicity and adaptability make it a cornerstone of graph algorithms, complementing BFS in educational and practical contexts.

2 Problem Formulation

Given a graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, and a source vertex s, traverse all reachable vertices in G starting from s. Optionally, compute:

- The traversal order of vertices.
- Properties like discovery/finish times, cycles, or topological order.

The output includes the order of visited vertices and, if required, additional graph properties.

2.1 Key Idea

DFS explores the graph by diving deeply into each branch using a stack to track vertices. Starting from a source vertex, it visits an unvisited neighbor, recursively explores that neighbor's unvisited neighbors, and backtracks when no unvisited neighbors remain. The algorithm marks vertices as visited to avoid cycles and can track parent vertices or timestamps for advanced applications.

3 Depth-First Search Algorithm

The algorithm uses recursion for simplicity, with a visited array to prevent revisiting vertices. Below is the pseudocode, including parent tracking:

Algorithm 1 Depth-First Search Algorithm

Input: Graph $G = (V, E)$, source vertex s
Output: Visited vertices, parents parent
Initialize visited$[v] \leftarrow$ false for all $v \in V$
Initialize parent$[v] \leftarrow -1$ for all $v \in V$
Procedure DFS-Visit(u):
visited$[u] \leftarrow$ true
Output u ▷ Process vertex
for each neighbor v of u in G **do**
    if visited$[v]$ = false **then**
        parent$[v] \leftarrow u$
        DFS-VISIT(v)
    end if
end for
Procedure DFS(s):
DFS-VISIT(s)
Call DFS(s)
Return parent

3.1 Handling Disconnected Graphs

To traverse all vertices in a disconnected graph, iterate over all vertices and call `DFS-Visit` for each unvisited vertex:

for each vertex $v \in V$ **do**
    if visited$[v]$ = false **then**
        DFS-VISIT(v)
    end if
end for

3.2 Cycle Detection

DFS detects cycles by checking if an unvisited neighbor v of vertex u is already in the recursion stack (for directed graphs) or visited but not the parent of u (for undirected graphs).

3.3 Topological Sorting

For a directed acyclic graph (DAG), DFS can produce a topological order by appending each vertex to a list after its recursive exploration completes (in reverse finish order).

4 Complexity Analysis

- **Time Complexity**: $O(n+m)$, where $n = |V|$ and $m = |E|$, as each vertex and edge is processed at most once.
- **Space Complexity**: $O(n)$, for the recursion stack, visited array, and parent array.

5 Practical Applications

DFS is used in:

- **Cycle Detection**: Identifying cycles in graphs (e.g., deadlock detection in systems).
- **Topological Sorting**: Scheduling tasks with dependencies (e.g., course prerequisites).
- **Connected Components**: Finding all reachable vertices in undirected graphs.
- **Path Finding**: Exploring possible paths in mazes or game environments.
- **Strongly Connected Components**: Analyzing directed graphs (e.g., Kosaraju's algorithm).

6 Example

Consider an undirected graph with vertices $\{1,2,3,4\}$ and edges $\{(1,2),(1,3),(2,4),(3,4)\}$, starting from vertex 1:

Vertex	1	2	4	3
Parent	-1	1	2	1

DFS visit order: $1,2,4,3$. The traversal explores $1 \to 2 \to 4$, backtracks, then explores $1 \to 3$.

7 Limitations and Extensions

- **Limitations**:
  - Not suitable for shortest path computation in unweighted graphs (use BFS instead).
  - Risk of stack overflow for very deep graphs in recursive implementations.
- **Extensions**:
  - **Iterative DFS**: Use an explicit stack to avoid recursion for deep graphs.
  - **Timestamps**: Track discovery and finish times for advanced applications like articulation points.
  - **Parallel DFS**: Distribute exploration for large graphs using parallel processing.

8 Implementation Considerations

- **Graph Representation**: Use an adjacency list for sparse graphs ($O(n+m)$) or adjacency matrix for dense graphs ($O(n^2)$).
- **Recursion vs. Iteration**: Prefer recursion for simplicity, but use an explicit stack for deep graphs to prevent stack overflow.
- **Visited Tracking**: Use a boolean array or set to avoid revisiting vertices in cyclic graphs.
- **Path Reconstruction**: Maintain a parent array for path recovery or cycle detection.
- **Input Validation**: Ensure the source vertex is valid and handle disconnected graphs by iterating over all vertices.

9 References

- GeeksForGeeks: Depth-First Search
- Wikipedia: Depth-First Search