

## Depth-First Search and Related Graph Algorithms

**Reading:** *CLRS* Chapter 22, Sections B.4–B.5; *CLR* Sections 5.4–5.5, Chapter 23

---

### Graphs

A **graph** is a pair  $(V, E)$  of **vertices**  $V$  and edges  $E \in V \times V$  (all pairs of vertices). A graph is **directed** iff the each edge  $(a, b)$  is interpreted as going from  $a$  to  $b$ . It is **undirected** if  $(a, b)$  and  $(b, a)$  are considered equivalent edges. In a directed graph  $a$  is called the **source vertex** of the edge  $(a, b)$  and  $b$  is called the **target vertex** of the edge  $(a, b)$ . In both directed/undirected graphs, for any vertex  $v$ , a possible edge is  $(v, v)$  (a **self-edge**).

*Example:*

*Notes:*

- A directed graph can be transformed to an undirected graph by “erasing arrows” on all edges and removing duplicate edges.
- An undirected graph can be transformed to a directed graph by adding an edge  $(b, a)$  for every edge  $(a, b)$ .
- The round-trip transformation undirected  $\rightarrow$  directed  $\rightarrow$  undirected loses no information, but the round-trip transformation directed  $\rightarrow$  undirected  $\rightarrow$  directed can lose information.
- We will only consider graphs where there is at most one edge between any two vertices, but more general graphs can have multiple edges between two vertices (distinguished by label). They can even have **hyperedges** that span three or more vertices.

A **subgraph** of  $G = (V, E)$  is a graph  $G' = (V', E')$  where  $V' \subseteq V$  and  $E' \subseteq E$ .

---

## Paths and Graph Classifications

A **path** in a graph  $(V, E)$  is an ordered list of vertices  $[v_0, v_1, \dots, v_n]$  such that each  $v_i \in V$  and each  $(v_{i-1}, v_i) \in E$ . Such a path has **length**  $n$ . The singleton list  $[v_0]$  is a length 0 path. The list  $[v, v]$  is only a path iff  $E$  contains the self-edge  $(v, v)$ .

A path  $[v_0, v_1, \dots, v_n]$  is a **cycle** iff  $n \geq 1$  and  $v_0 = v_n$ . A cycle is **simple** iff  $v_1, \dots, v_n$  are distinct and no edge is repeated. (The latter condition prevents  $[a, b, a]$  from being considered a cycle in an undirected graph.) A graph is **acyclic** iff it contains no simple cycles.

An undirected graph is **connected** if there is a path between any two vertices.

A directed acyclic graph is also known as a **DAG**. A connected acyclic undirected graph is a **tree**. An (not necessarily connected) acyclic undirected graph is known as a **forest** because it is a collection of trees.

---

## Graph Representations

There are many ways to represent a graph:

---

## Depth-First Search (DFS)

*Idea:* Explore graph from a given vertex by first exploring all children of that vertex. To avoid looping through cycles, mark each vertex upon first visiting it; do not explore children of a previously visited vertex.

▷ *Induce a depth-first forest on a graph.*

```
DFS(G)
  for v in vertices[G] do
    color[v] ← white ▷ white means “unexplored”
    parent[v] ← nil
  for v in vertices[G] do
    if color[v] = white then
      DFS-Visit(v)
```

▷ *Induce a depth-first tree on a graph starting at v.*

```
DFS-Visit(v)
  color[v] ← gray ▷ gray means “frontier”
  for a in Adj[v] do
    if color[a] = white then
      parent[a] = v
      DFS-Visit(a)
  color[v] ← black black means “processed”
```

*Notes:*

- The edges  $(\text{parent}[v], v)$  form the **depth-first forest** created by depth-first search. The adjacency list of the vertices in the forest is implicit in this algorithm, but could be made explicit.
- The structure of the depth-first forest and its component trees therein depends on the order of vertices in `vertices[G]` and `Adj[v]`.
- DFS effectively uses an implicit stack to process frontier vertices.

*Analysis:*

- `DFS-Visit` called exactly once on each vertex:  $\Theta(V)$ .
- Each directed edge is explored exactly once in `for` loop within `DFS-Visit`:  $\Theta(E)$ .
- Total:  $\Theta(V + E)$ .

---

## Edge Classification

Edges can be classified by depth-first search as follows:

- **Tree edges** are edges ( $\text{parent}[v], v$ ) forming depth-first forest;
- **Back edges** connect vertex to an ancestor in a depth-first tree;
- **Forward edges** are non-tree edges connecting vertex to a descendent in a depth-first tree;
- **Cross edges** are non-tree edges connecting (1) two vertices in a tree that are not in an ancestor/descendant relationship or (2) two vertices from different trees.

Can mark edges by type during DFS by noting color of vertex when first encountered:

- *white (unexplored)* indicates a tree edge;
- *gray (frontier)* indicates a back edge;
- *black (processed)* indicates a forward or cross edge (can use timestamps – see below – to distinguish these).

---

## Simple DFS-based Algorithms

*Cycle Detection:* Any cycle in a directed or undirected graph must pass through a back edge found during DFS. A tree is acyclic if no back edges are found during DFS. Running time is running time of DFS =  $\Theta(V + E)$ .

*Connected Components:* A **connected component** of an *undirected* graph is a maximal set of vertices such that for any two vertices  $a$  and  $b$  in the set, there is a path from  $a$  to  $b$ . An undirected graph is said to be **connected** if it consists of a single connected component.

The connected components of a graph can be computed by DFS: each tree in the resulting depth-first forest is a connected component of the graph. Running time is running time of DFS =  $\Theta(V + E)$ .

---

## Timestamps

Can extend the simple DFS above to timestamp each discovery and finish step using a global clock. (Changes to the previous algorithm are in bold italic.)

▷ *Induce a depth-first forest on a graph.*

```
DFS(G)
  for v in vertices[G] do
    color[v] ← white ▷ white means “unexplored”
    parent[v] ← nil
  time ← 0 ▷ Assume time is a global variable
  for v in vertices[G] do
    if color[v] = white then
      DFS-Visit(v)
```

▷ *Induce a depth-first tree on a graph starting at v.*

```
DFS-Visit(v)
  color[v] ← gray ▷ gray means “frontier”
  time ← time + 1
  discovery[v] ← time
  for a in Adj[v] do
    if color[a] = white then
      parent[a] = v
      DFS-Visit(a)
  color[v] ← black black means “processed”
  time ← time + 1
```

*Note:* Timestamps range between 1 and  $2 \cdot |V|$

*Distinguishing Forward and Cross-Edges:* If  $b$  is already colored black when  $(a, b)$  is discovered, then  $(a, b)$  is a forward edge if  $\text{discovery}[a] \leq \text{discovery}[b]$ , and a cross-edge if  $\text{discovery}[a] > \text{discovery}[b]$ .

*Parenthesis Theorem:* For two vertices  $a$  and  $b$  in a depth-first forest of  $G$ , exactly one of the following three holds:

1. The intervals  $(\text{discovery}[a], \text{finish}[a])$  and  $(\text{discovery}[b], \text{finish}[b])$  are disjoint. (True when  $a$  and  $b$  do not have an ancestor or descendant relationship.)
2. The interval  $(\text{discovery}[a], \text{finish}[a])$  is nested within  $(\text{discovery}[b], \text{finish}[b])$ . (True when  $a$  is a descendant of  $b$  in depth-first forest.)
3. The interval  $(\text{discovery}[b], \text{finish}[b])$  is nested within  $(\text{discovery}[a], \text{finish}[a])$ . (True when  $b$  is a descendant of  $a$  in depth-first forest.)

*White-path Theorem* In a depth-first forest of  $G$ , vertex  $d$  is a descendant of ancestor  $a$  iff at time  $\text{discovery}[a]$ ,  $d$  can be reached from  $a$  along a path consisting entirely of white (unexplored) vertices.

---

## Topological Sort

A **directed acyclic graph (DAG)** is a directed graph without cycles. A topological sort of a DAG  $G = (V, E)$  is a linear ordering of vertices in  $V$  consistent with the following partial order:  $a < b$  iff  $(a, b) \in E$ . In other words, each vertex in a topological sort must precede all its descendants in the DAG and must follow all of its ancestors.

*Example:*

*Approach 1:*

Modify DFS so that when it finishes processing a vertex, it prepends it to the front of an initially-empty global list. Since processing of a vertex is finished only when all descendants are finished, each vertex precedes all its descendants in the list. The running time is  $\Theta(V + E)$  since DFS takes  $\Theta(V + E)$  time.

*Approach 2:*

The **in-degree** of a vertex  $v$  in a directed graph is the number of edges whose target is  $v$ .

```
Topological-Sort-2 (G)
  L ← []
  while not Empty(vertices[G]) do
    v ← Find-Vertex-With-Indegree-0(G)
    L ← L ++ [v]
    for e in Out-Edges(G, v) ▷ All edges in G whose source is v.
      G ← Remove-Edge(e, G)
    Remove-Vertex(v, G)
  return L
```

Each vertex clearly follows all its ancestors. The running time can be made  $O(V + E)$  (left as an exercise: see *CLRS* Exercise 22.4-5/*CLR* Exercise 23.4-5.)

---

## Strongly Connected Components

A **strongly connected component** of a *directed* graph is a maximal set of vertices such that for any two vertices  $a$  and  $b$  in the set, there is a path from  $a$  to  $b$  *and* from  $b$  to  $a$ . In other words, in a strongly connected component, there is a path from every member of the set to every other member of the set.

The **transpose** of a graph  $G$ , written  $G^T$ , is a graph with the same vertices as  $G$  in which the directions of all edges have been reversed.

### Strongly-Connected-Components( $G$ )

1. Call  $\text{DFS}(G)$  to compute  $\text{finish}[v]$  for each vertex  $v$  in  $G$ .
2. Call  $\text{Modified-DFS}(G^T)$ , where the main loop of  $\text{Modified-DFS}$  processes vertices in order of decreasing finish times.
3. Each tree in depth-first forest of  $\text{Modified-DFS}(G^T)$  is a strongly connected component of  $G$ .

Why does this work? Based on notion of **forefather** detailed in *CLRS 22.5/CLR 23.5*.

*Example:*

Running time of **Strongly-Connected-Components** is the running time of two calls to  $\text{DFS} = \Theta(V + E)$ .