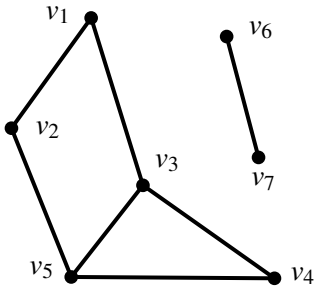
DFS and Connectivity of Graphs

Recall that the *explore* function starting from a given vertex v finds all vertices in G reachable from v. Below we give two examples of running *explore*.



input graph: $G = (V, E)$

$v_1 \rightarrow (v_2,\ v_3)$
$v_2 \rightarrow (v_1,\ v_5)$
$v_3 \rightarrow (v_1,\ v_4, v_5)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow (v_2,\ v_3,\ v_4)$
$v_6 \rightarrow (v_7)$
$v_7 \rightarrow (v_6)$

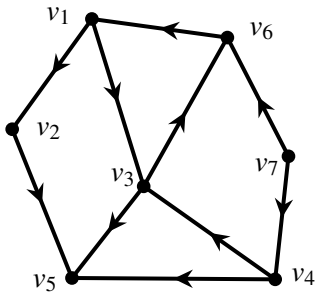
adjacency list of G

final array of *visited*

explore $(G,\ v_1)$

1	1	1	1	1	0	0

Figure 1: Running *explore* (G, v_1) on an undirected graph.



input graph: $G = (V, E)$

$v_1 \rightarrow (v_2, v_3)$
$v_2 \rightarrow (v_5)$
$v_3 \rightarrow (v_5,\ v_6)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow ()$
$v_6 \rightarrow (v_1)$
$v_7 \rightarrow (v_4, v_6)$

adjacency list of G

final array of *visited*

explore $(G,\ v_1)$

1	1	1	0	1	1	0

Figure 2: Running *explore* (G, v_1) on an directed graph.

We now define "connected" and "connected component" to formally reveal the connectivity-structure of graphs. Let $u, v \in V$. We say u and v are *connected* if and only if there exists a path from u to v *and* there exists a path from v to u. We note that this definition applies to both directed and undirected graph. In undirected graph, the existence of a path from u to v implies the existence of a path from v to u. However, this is not necessarily true in directed graphs. For example, in Figure 2, there exists a path from v_1 to v_5 but there is no path from v_5 to v_1 (so they are not connected).

Let $G = (V, E)$. Let $V_1 \subset V$. We say V_1 is a *connected component* of G, if and only if (1), for *every pair of* $u, v \in V_1$, u and v are connected, and (2), V_1 is *maximal*, i.e., there does not exists vertex $w \in V \setminus V_1$ such that $V_1 \cup \{w\}$ satisfies condition (1). For example, in Figure 2, $\{v_1, v_3, v_6\}$ is a connected component; $\{v_2\}$

is a connected component; $\{v_1, v_3\}$ is not a connected component (as it is not maximal, i.e., does not satisfy condition 2).

The explore algorithm identifies all vertices that can be reached from v_i. Hence, in the case of undirected graphs, these vertices (including v_i) are pairwise reachable, i.e., they form a connected component of G (summarized below). In other words, $explore(G, v_i)$ identifies the connected component of G that includes v_i.

Fact 1. For undirected graphs, after $explore(G, v_i)$, the vertices that are marked by *visited*, i.e., $\{v_j \mid visited[j] = 1\}$ forms a connected component of G that includes v_i.

However, the above fact does not apply to directed graph: Figure 2 gives such an example, where $\{v_1, v_2, v_3, v_5, v_6\}$ does not form a connected component. Note: in directed graphs $\{v_j \mid visited[j] = 1\}$ are still those vertices that are reachable from v_i; it's just that they may not be a connected component of G.

How to identify *all* connected components of an undirected graph? We can run above explore algorithm multiple times, each of which starts from an un-explored vertex, until all vertices are explored. To keep track of which vertices are in which connected component, we will introduce variable *num-cc* to store the index of current connected component. We redefine the behavior of *visited* array: $visited[j] = 0$ still represents that v_j has not yet been explored; $visited[j] = k$, $k \geq 1$, represents that v_j has been explored and v_j is in the k-th connected component.

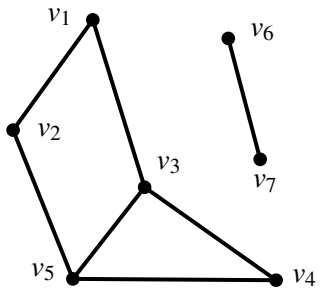
This new algorithm that traverses all vertices and edges of a graph is named as DFS (depth first search). We also slightly changed the explore function, which allows to store which connected component each vertex is in. The pseudo-codes are given below.

```
function DFS (G = (V, E))
    num-cc = 0;
    visited[i] = 0, for all 1 ≤ i ≤ |V|;
    for i = 1 → |V|
        if (visited[i] = 0)
            num-cc = num-cc + 1;
            explore (G, v_i);
        end if;
    end for;
end algorithm;
```

```
function explore (G = (V, E), v_i ∈ V)
    visited[i] = num-cc;
    for any edge (v_i, v_j) ∈ E
        if (visited[j] = 0): explore (G, v_j);
    end for;
end algorithm;
```

Below we gave examples of running DFS on undirected graphs and undirected graphs.

DFS runs in $\Theta(|E| + |V|)$ time. This is because, each vertex is explored exactly once, and each edge is examined exactly once (in the case of directed graphs) or exactly twice (in the case of undirected graphs).



input graph: $G = (V, E)$

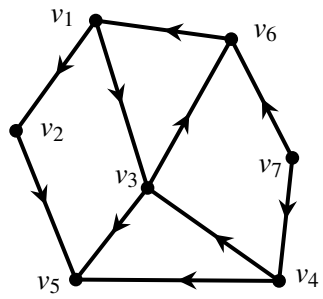
$v_1 \rightarrow (v_2,\ v_3)$
$v_2 \rightarrow (v_1,\ v_5)$
$v_3 \rightarrow (v_1,\ v_4, v_5)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow (v_2,\ v_3,\ v_4)$
$v_6 \rightarrow (v_7)$
$v_7 \rightarrow (v_6)$

adjacency list of G

final array of *visited* after running $DFS(G)$

1	1	1	1	1	2	2

Figure 3: Running $DFS\ (G)$ on an undirected graph.



input graph: $G = (V, E)$

$v_1 \rightarrow (v_2, v_3)$
$v_2 \rightarrow (v_5)$
$v_3 \rightarrow (v_5,\ v_6)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow ()$
$v_6 \rightarrow (v_1)$
$v_7 \rightarrow (v_4, v_6)$

adjacency list of G

final array of *visited* after running $DFS(G)$

1	1	1	2	1	1	3

Figure 4: Running $DFS\ (G)$ on a directed graph.

Fact 2. For undirected graphs, DFS (G) identifies all connected components of G: $\{v_j \mid visited[j] = k\}$ constitutes the k-th connected component of G.

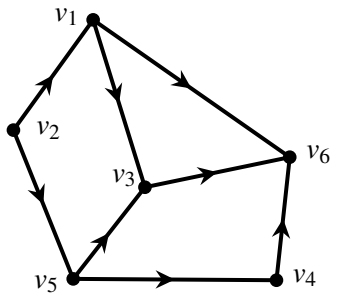
Again, the above fact does not apply to directed graphs: $\{v_j \mid visited[j] = k\}$ are not necessarily form a connected component, although you can still run DFS on directed graphs. In Figure 4, $\{v_j \mid visited[j] = 1\}$ gives such an counter-example. To prepare revealing the connectivity-structure of directed graphs, we first introduce a spsecial class of directed graphs, given below.

Directed Acyclic Graph (DAG)

Definition 1 (DAG)**.** A directed graph $G = (V, E)$ is *acyclic* if and only if G does not contain cycles.

Definition 2 (Linearization / Topological Sorting)**.** Let $G = (V, E)$ be a directed graph. Let X be an ordering of V. If X satisfies: if $(v_i, v_j) \in E$, then v_i is before v_j in X, then we say X is a linearization (or toplogical sorting) of G.

See some examples below.



$(v_2, v_1, v_5, v_3, v_4, v_6)$ linearization? Yes

$(v_2, v_5, v_1, v_4, v_3, v_6)$ linearization? Yes

$(v_2, v_5, v_3, v_4, v_1, v_6)$ linearization? No, see edge (v_1, v_3)

$(v_2, v_1, v_4, v_5, v_3, v_6)$ linearization? No, see edge (v_5, v_4)

Figure 5: Examples of linearization.

If a directed graph G admits a linearization, then we say G can be *linearized*. We now show that linearization is an *equivalent* characterization of DAGs.

Claim 1. A directed graph G can be linearized if and only if G is a DAG.

Proof. Let's first prove that if G can be linearized, then G is a DAG. This is equivalent to proving its contraposition: if G contains a cycle, then G cannot be linearized. Suppose that there exists an cycle $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} \to v_{i_1}$ in G. Then the linearization X must satisfy that v_{i_j} is before $v_{i_{j+1}}$ for all $j = 1, 2, \cdots, k-1$, and that v_{i_k} is before v_{i_1}, in X. Clearly, this is not possible.

The other side of the statement, i.e., if G is a DAG, then G can always be linearized, can be proved constructively. We will design an algorithm (see below), that constructs a linearization for any DAG. □