Connectivity of Graphs

A *path* (also called *walk* in some literature) in a graph G is a sequence of vertices and edges, where each edge is incident to its preceding and succeeding vertices. Note that paths may contain duplicate vertices or edges. We say a path is *simple*, if it does not contain repeated vertex. If there exists a path from u to v, then we also say u can reach v, or v is reachable from u, or v can be reached from u. These definitions applies to both directed and undirected graphs. Clearly, in undirected graphs, u can reach v is equivalent to that v can reach u. But this is not the case for directed graphs.

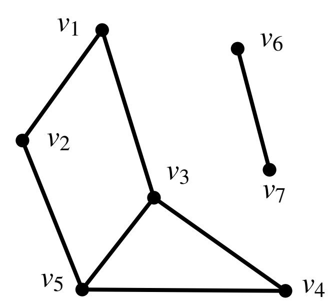
One basic procedure in graphs is to find the set of vertices that are reachable from a given vertex. We will use an array, called *visited*, of size $|V|$, to store the vertices that are reachable from the given vertex v_i: $visited[j] = 1$ if and only if there exists a path from v_i to v_j. This array will be initialized as 0 for all entries. The following recursive algorithm, named *explore*, finds all vertices that are reachable from v_i and stores these vertices in *visited* array properly.

function explore $(G = (V,E), v_i \in V)$
    $visited[i] = 1$;
    for each v_j where $(v_i, v_j) \in E$
        if $(visited[j] = 0)$: explore (G, v_j);
    end for;
end algorithm;

In above algorithm, we can assume that G is represented/stored with adjacency list. In this case, the tranverse of v_j can be done by simply tranversing the list associated with v_i in the adjacency list.

The time complexity of explore function is $\Theta(|E|)$, as it may traverse all lists in the adjacency list at most once, and we have showed that the total size of all lists is $\Theta(|E|)$.

Below we give two examples of running *explore* (Figure 1 and Figure 2).



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

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

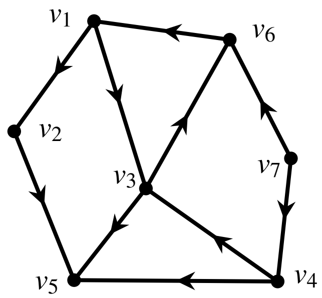
adjacency list of G

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.

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



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

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

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.

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 reachable from a given vertex v_i. Hence, in the case of undirected graphs, these vertices (including v_i) are pairwise reachable, and these vertices are also maximal (as otherwise the explore function will find them). 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.

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.

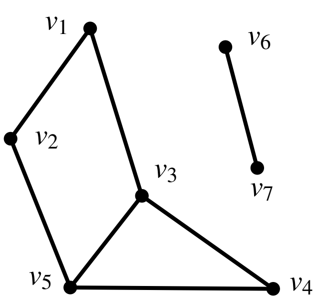
How to identify *all* connected components of an undirected graph? We can run above explore algorithm multiple times, each of which starts from an un-explored vertex, until all vertices are explored. To keep track of which vertices are in which connected component, we will introduce variable *num-cc* 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 $1 \le i \le |V|$;
    for $i = 1 \to |V|$
        if ($visited[i] = 0$)
            num-cc = num-cc + 1;
            explore (G, v_i);
        end if;
    end for;
end algorithm;

function explore ($G = (V, E), v_i \in V$)
    $visited[i] =$ num-cc;
    for each v_j where $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore (G, v_j);
    end for;
end algorithm;

Below we gave examples of running DFS on an undirected graph.



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

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

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.

DFS runs in $\Theta(|E| + |V|)$ time. This is because, each vertex is explored exactly once, and the all lists in the adjacency list are visited once.

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.