Directed Acyclic Graphs

Let $G = (V, E)$ be a directed graph. If a vertex $v \in V$ does not have any in-edges (i.e., *in-degree* is 0), we call it *source vertex*; if a vertex $v \in V$ does not have any out-edges (i.e., *out-degree* is 0), we call it *sink vertex*.

A directed graph may contain multiple source vertices or sink vertices, or may not have any source vertex or sink vertex. (Can you give such examples?)

Claim 1. A DAG $G = (V, E)$ always has source vertex and sink vertex.

Proof. Let's prove it by contradiction. Assume that G does not contain any source. First, G must not contain self-loop as otherwise G won't be a DAG. Let v be any vertex in V. As v is not a source, we know that there exists some vertex u points to v, i.e., $(u, v) \in E$. Now since u is not a source then there must exist another vertex w such that $(w, u) \in E$. Notice that $w \neq v$ as otherwise there will be a cycle: $v = w \rightarrow u \rightarrow v$. This means that w is a new vertex. Again as w is not a source, there must exist another *new* vertex points to it. This process can be extended infinitely following the fact and assumption that G is a DAG and all vertices not are sources, but this is not possible as the number of vertices is limited. The existence of sink can be proved symmetrically. □

Following above fact, we can design an algorithm to find a linearization of a DAG: it iteratively finds source vertex and removes it and its adjacent edges.

```
Algorithm find-linearization (G = (V, E))
    init X as empty list;
    while (|X| < |V|)
        arbitrarily find a source vertex u of G;
        add u to the end of X;
        remove u and its adjacent edges;
    end while;
end algorithm;
```

This algorithm is correct. First, when a vertex u is added to X, it is a source vertex of the current graph, which means that $\{w \mid (w, u) \in E\}$ is either empty or all of them have been added to X. Second, X will include all vertices. This is because, a source always exists in a DAG (as we just proved). The above algorithm is more a framework, as how we update the graph is not given specifically, and which affects the running time.

Above algorithm also gives a constructive proof that, a DAG can always be linearized. This completes the proof stated in previous lecture: a directed graph is a DAG if and only if it can be linearized.

Claim 2. Let X be any linearization of a DAG. Then the first vertex of X is a source vertex and the last vertex of of X is a sink vertex.

Proof. Let v_1 be the first vertex of X. Suppose that v_1 is not a source of X. By definition of (not being a) source vertex, there exists $(u, v_1) \in E$. Then by the definition of linearization, we know that u will be before v_1 in X, contradicting to the fact that v_1 is the first element of X. The other side can be proved symmetrically. □

We can use above algorithm to constructively prove following statement.

Claim 3. Let $G = (V, E)$ be a DAG. A vertex $u \in V$ is a source if and only if there exists a linearization X of G such that u is the first vertex in X.

Proof. We first prove that, if u is a source, then there exists a linearization where u is the first vertex of X. We prove it by showing that, we can construct such a linearization X. We can use above algorithm, and in its first step, we simply pick u. The correctness of above algorithm explains the rest. The other side is exactly Claim 2, which we have proved. □

Connectivity of Directed Graphs

For a directed graph $G = (V, E)$, its structure of connectivity can be represented as a new directed graph, called *meta-graph*, denoted as $G_M = (V_M, E_M)$. Each of the vertices of the meta-graph corresponds to a connected component of G, and two vertices $C_i, C_j \in V_M$ are connected by edge $(C_i, C_j) \in E_M$ if and only if there exists edge $(u, v) \in E$ such that $u \in C_i$ and $v \in C_j$. An example of meta-graph is given below.

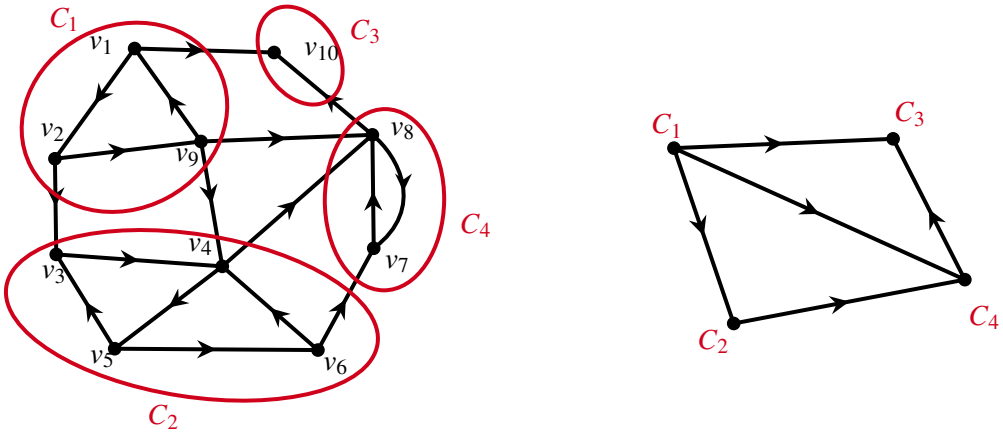


Figure 1: Example of meta-graph.

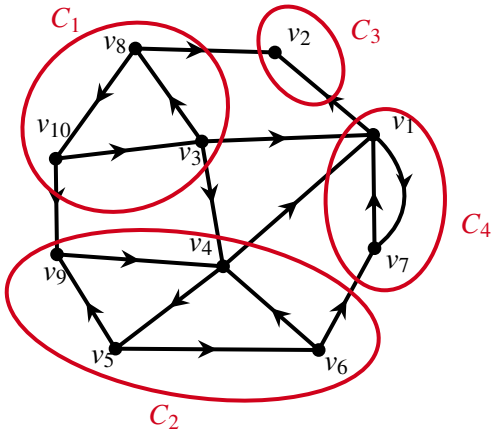
Meta-graph has an important property: it does not contain cycles, i.e., it is a directed acyclic graph (DAG).

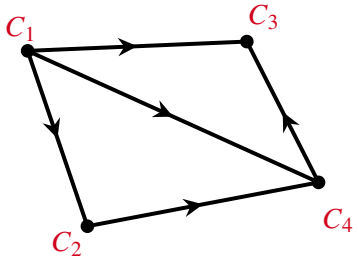
Claim 4. The meta-graph G_M of any directed graph G is a directed acyclic graph.

Proof. Suppose conversely that G_M contains a cycle, $C_1 \to C_2 \to C_k \to C_1$, then the union of the vertices in these connected components form a single connected component, contradicting to the *maximal* property of connected component. □

We now design algorithms to determine the connected components of directed graphs and then to construct the corresponding meta-graph. Recall that the DFS algorithm introduced in Lecture A9 can successfully determine all connected components in an undirected graph. Does this work also work for directed graph? Let's try. See Figure 2. Recall that the (top-layer) of the DFS algorithm is to tranverse all vertices in some ordering, and if the current vertex v is not yet visited, we will explore v. For the example in Figure 2, let's try it with the (natrual) ordering $v_1, v_2, \cdots, v_{10}$. The resulting visited array is given in the figure (please make sure you try it yourself). Unfortunately, it does not give all connected components. In fact, the vertices marked as "1"s are the union of C_3 and C_4, and the vertices marked as "2"s are the union of C_1 and C_2. The reason is quite clear: we start with explore v_1, and during it all vertices reachable from v_1 will be marked as "1" in the visted array. It turns out that v_1 is in connected component C_4, and C_4 can reach C_3 in the

meta-graph. Therefore, all vertices in C_4 and C_3 are reachable from v_1; consequently, the visited array is set as "1" for vertices in C_4 and C_3 during explore v_1. Similarly, we can also see why the vertices marked as "2"s are the union of C_1 and C_2. After exploring v_1, $\{v_1, v_2, v_7\}$ are visited; the next unvisited vertex is v_3 according to above natural ordering, the the algorithm will explore v_3. Again, during it all (unvisited) vertices reachable from v_3 will be marked as "2" in the visted array. It turns out that v_3 is in connected component C_1, and C_1 can reach C_2 in the meta-graph. Therefore, all vertices in C_1 and C_2 are reachable from v_2; consequently, the visited array is set as "2" for vertices in C_1 and C_2.





final array of *visited* after running DFS with ordering $(v_1, v_2, \cdots, v_{10})$:

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	1	2	2	2	2	1	2	2	2

final array of *visited* after running DFS with a special ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$:

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
2	1	4	3	3	3	2	4	3	4

Figure 2: Run the DFS algorithm with an arbitrary ordering (introduced in Lecture A9) and a special ordering (pseudo-code given below) on above example.

How to fix this issue? The idea is to use a *special ordering* of vertices, instead of an arbitrary ordering, in above DFS. This special ordering should allow us to determine connected component one by one. Hence, the first vertex we explore in DFS, should be one vertex in a *sink* component of the meta-graph. In Figure 2, the only sink component is C_3, and since there is only one vertex in C_3, we should start with exploring v_2. Clearly, exploring v_2 will exactly determine the single connected component C_2. Next, after C_3 is determined, the next component we can determine is again a sink component after removing C_3 from the meta-graph. It is C_4. So, the next vertex the DFS should explore must be v_1 or v_7. It does matter which one we pick; let's assume we pick v_7 (and it does not matter where v_1 is in the ordering as long as it is after v_2 and v_7). Clearly, exploring v_2 will exactly determine the single connected component C_4—see the visited array. The next component we can determine is again a sink component after removing C_3 and C_4 from the meta-graph. It is C_2. So, the next vertex the DFS should explore must be in $\{v_4, v_5, v_6, v_9\}$. And it does matter which one we pick neither the ordering of remaining ones as long as they are behind the one we pick. Let's say we pick v_4. Clearly, exploring v_4 will exactly determine the single connected component C_2. Now the only component remaining is C_1 after removing C_2, C_3 and C_4 from the meta-graph. So, the next vertex the DFS should explore must be in $\{v_3, v_8, v_{10}\}$. Let's say we pick v_8. Clearly, exploring v_8 will exactly determine the last connected component C_1.

To abtract above observation, the determined connected components with above DFS forms a reverse-

linearization of the meta-graph. Therefore, the special ordering of vertices should satisfy this condition: *the ordering of connected components sorted by their first appearance in the special ordering of vertices should form a reverse-linearization of the meta-graph*. And if the special ordering of vertices satisfies this condition, the DFS algorithm will work—it will identify all connected components.

Again see Figure 2, the ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$ satisfies above condition. To see that, the corresponding list of connected components is $(C_3, C_4, C_2, C_2, C_1, C_4, C_1, C_2, C_1, C_2)$, and hence the ordering of their first appearance is (C_3, C_4, C_2, C_1) which is indeed a reviers-linearization of the meta-graph.

```
function DFS (G = (V, E))
    num-cc = 0;
    for v_i in a specific order
        if (visited[i] = 0)
            num-cc = num-cc + 1;
            explore (G, v_j);
        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;
```

To summarize, the algorithm to identify connected components of directed graphs is essentially the same with that for undirected graphs (introduced in Lecture A9, copied above), except a single line of difference (marked blue): for directed graphs, we need to explore vertices in a specific order that satisfy above condition, while for undirected graphs, we can explore all vertices in any arbitrary order.

How to find an ordering of vertices that satisfy above condition? We need a combination of two techniques: DFS-with-timing and reverse-graph.

DFS with Timing

The DFS-with-timing is a variant of DFS, which uses the following data structures (we assume $n = |V|$):

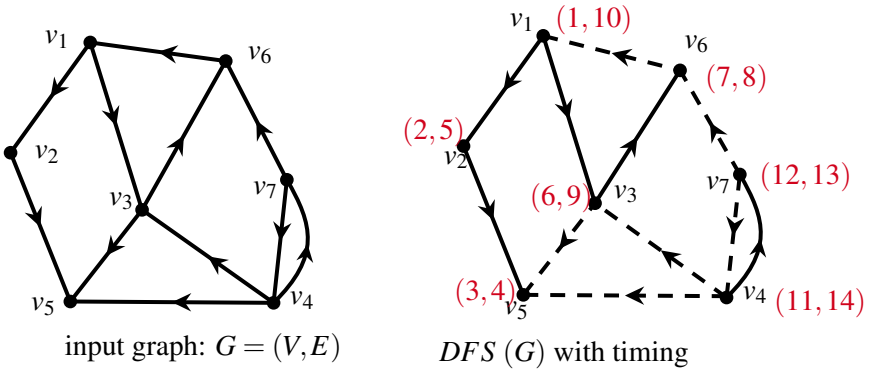
1. variable clock servers as a timer that stores the current time;
2. binary array $visited[1..n]$, where $visited[i]$ indicates if $v[i]$ has been explored, $1 \le i \le n$;
3. array $pre[1..n]$, where $pre[i]$ records the time of starting exploring v_i, $1 \le i \le n$;
4. array $post[1..n]$, where $post[i]$ records the time of finishing exploring v_i, $1 \le i \le n$;
5. array $postlist$, stores the vertices in decreasing order of $post[\cdot]$.

The pseudo-code of DFS with timing is given below.

function DFS-with-timing $(G = (V, E))$
    $clock = 1$;
    for $i = 1 \to |V|$
        if $(visited[i] = 0)$: explore (G, v_i);
    end for;
end algorithm;

function explore $(G = (V, E), v_i \in V)$
    $visited[i] = 1$;
    $pre[i] = clock$;
    $clock++$;
    for any edge $(v_i, v_j) \in E$
        if $(visited[j] = 0)$: explore (G, v_j);
    end for;
    $post[i] = clock$;
    $clock++$;
    add v_i to the front of $postlist$;
end algorithm;

An example of running DFS with timing is given below. Notice that the DFS searching partitions all edges into two categories: solid edges (u, v) implies that v is visited for the first time (and therefore explore v will start right now), while dashed edges (u, v) implies that at that time v has been visited already (and therefore v will be skipped and the next adjacent vertex of u will be examined in the for-loop).



input graph: $G = (V, E)$ $DFS\ (G)$ with timing

Figure 3: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex. The $postlist$ for this run is $postlist = (v_4, v_7, v_1, v_3, v_6, v_2, v_5)$.