DFS with Timing

The Algorithm

The DFS-with-timing is a variant of DFS that records the time of starting and finishing the explore of each vertex. It uses the following data structures (we assume $n = |V|$). These data structures are global variables, so that the explore function can get access to and edit them.

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/visited, $1 \leq i \leq n$;
3. array $pre[1..n]$, where $pre[i]$ records the time of starting exploring v_i, $1 \leq i \leq n$;
4. array $post[1..n]$, where $post[i]$ records the time of finishing exploring v_i, $1 \leq i \leq n$;
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 ($G = (V, E)$)
- $clock = 1$;
- $postlist = \emptyset$;
- $pre[i] = post[i] = -1$, for $1 \leq i \leq n$;
- for $i = 1 \to |V|$
  - if ($visited[i] = 0$): explore (G, v_i);
- end for;

end algorithm;

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

end algorithm;

An example of running DFS with timing is given below. Notice that this algorithm 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, and after exploring v the program will return to explore u), while dashed edges (u, v) implies that at that time v has been visited already (and therefore v will be skipped and the next out-edge of u will be examined in the for-loop).

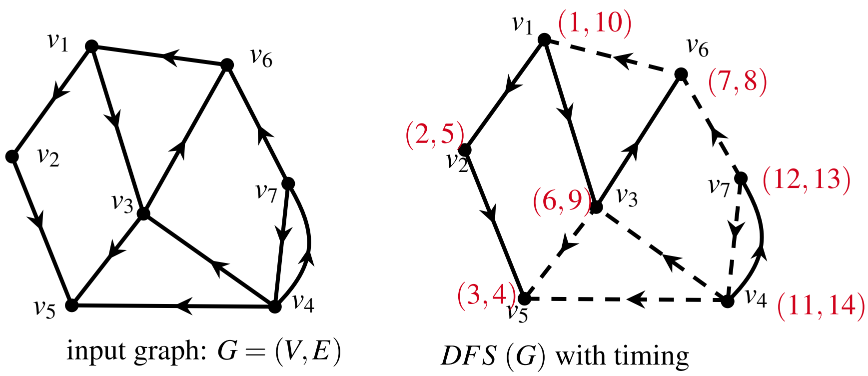

Figure 1: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex. The $postlist$ for this run is $postlist = (v_4, v_7, v_1, v_3, v_6, v_2, v_5)$.

In explore v_i, the adjacent vertices $\{v_j \mid (v_i, v_j) \in E\}$ can be examined in any arbitrary order, i.e., all conclusions/properties we show hold regardless the order that $\{v_j\}$ gets examined. In practice though, we might follow a specific order; in Figure 1, we examine $\{v_j\}$ in increasing order of their indexes.

The above DFS-with-timing algorithm runs in $\Theta(|V| + |E|)$ time, since each vertex and each edge will be examined a constant number of times (once for directed graph, twice for undirected graph).

The above DFS-with-timing algorithm gives an interval $[pre, post]$ for each vertex. For two vertices $v_i, v_j \in V$, their corresponding intervals can either be *disjoint*, i.e., the two intervals do not overlap, or *nested*, i.e., one interval is within the other. See Figure 2. But two intervals cannot be *partially overlapping*. Why? This is because the explore funtion is recursive. There are only two possiblities that $pre[i] < pre[j]$. The first one is that explore v_j is *within* explore v_i; in this case the recursive behaviour of explore leads to that $post[j] < post[i]$, as explore v_j must terminate first and return to explore v_i. This case corresponds to that the two intervals are nested. The second one is that explore v_j starts after explore v_i finishes; this case corresponds to that the two intervals are disjoint.

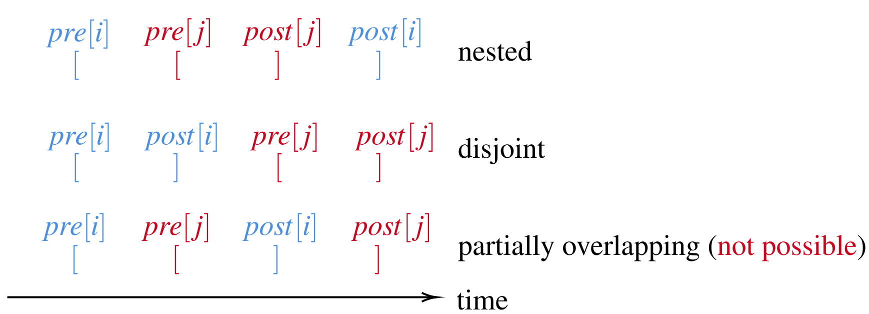

Figure 2: Relations between two $[pre, post]$ intervals.

Claim 1. If the $[pre[j], post[j]]$ is nested within $[pre[i], post[i]]$, then v_j is reachable from v_i.

Proof. Consider when an explore will be called within another explore: only if there is an edge $(v_i, v_j) \in E$ (and $visited[j] = 0$), explore v_j will be called within explore v_i. Consequently, the time interval for v_j will be within the interval for v_i. Note that explore v_j might call explore other vertices, such as explore v_k. When this happens, the time interval for v_k will be within the interval for v_j, and therefore within the interval for v_i. But again this happens only if there exists edge (v_j, v_k), and hence a path $v_i \to v_j \to v_k$. This argument can be extended to longer paths, proving the conclusion above. $\square$

Determine of Existence of Cycles

Let's see the first application of DFS-with-timing—to decide if a given (directed) graph contains cycles or not. We can simply modify the explore function, given below, and use the same DFS-with-timining function. Specifically, when the algorithm examines an edge (v_i, v_j): if v_j has been explored *and* its post-number has not been set yet, then the algorithm reports that G contains cycle.

```
function explore (G = (V,E), v_i ∈ V)
    visited[i] = 1;
    pre[i] = clock;
    clock = clock + 1;
    for any edge (v_i, v_j) ∈ E
        if (visited[j] = 0): explore (G, v_j);
        else if (post[j] = −1): report "G contains cycle";
    end for;
    post[i] = clock;
    clock = clock + 1;
    add v_i to the front of postlist;
end algorithm;
```

Now let's prove this algorithm is correct. We first prove that if G contains cycle then the algorithm will always report at some time. Let C be the cycle. Let $v_j \in C$ be the first vertex that is explored in C. Let $(v_i, v_j) \in E$ be an edge in C. As v_j can reach v_i (reason: both in cycle C), within exploring v_j there will be a time that v_i gets explored. In explore v_i, consider the time of examining edge (v_i, v_j): at this time $visited[j]$ has been set as 1, but its post-number has not been set, as now the algorithm is still within exploring v_j. Therefore, the algorithm will report that G contains cycle.

We then prove that if the algorithm reports, then G must contain cycles. Consider that the algorithm is exploring v_i, examining edge (v_i, v_j) and finds $visited[j] = 1$ *and* $post[j] = -1$. The fact that $post[j]$ has not been set implies that the algorithm is within exploring v_j. Therefore the interval for v_i must be nested within the interval for v_j. Following Claim 1, we know that v_j can reach v_i. In addition, there exists edge (v_i, v_j). Combined, G contains cycle.

Note that this algorithm is essentially determining if there exists edge $(v_i, v_j) \in E$ such that the interval $[pre[i], post[i]]$ is within interval $[pre[j], post[j]]$. (Such edges are called *back edges* in textbook [DPV], page 95.)

Key Property

Before seeing more applications, we prove an important property that relates the post values and meta-graph.

Claim 2. Let C_i and C_j be two connected components of directed graph $G = (V, E)$, i.e., C_i and C_j are two vertices in its coresponding meta-graph $G_M = (V_M, E_M)$. If we have $(C_i, C_j) \in E_M$ then we must have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Intuitively, following an edge in the meta-graph, the largest post value decreases. Before seeing a formal proof, please see an example in Figure 3: the largest post values for C_1, C_2, C_3, and C_4 are 9, 6, 10, and 14, and you may verify that following any edge in the meta-graph, the largest post value always decreases.

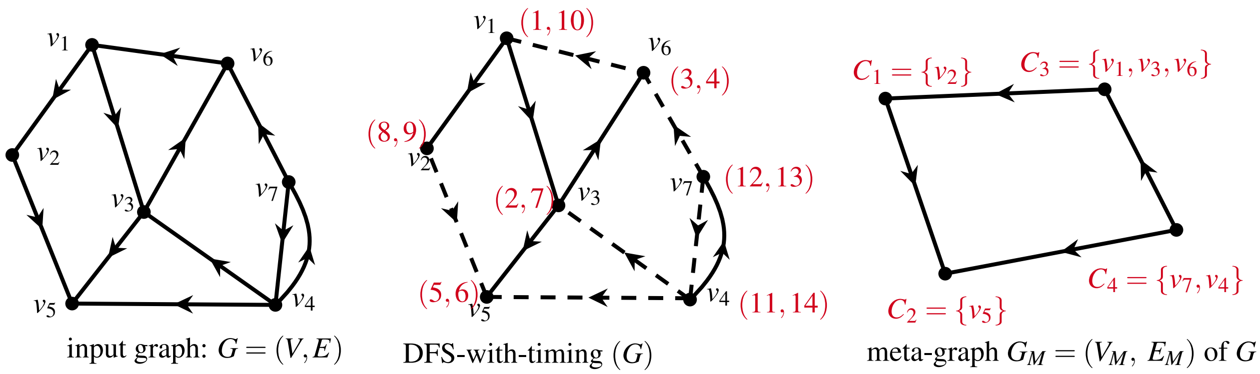

Figure 3: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex.

Proof. Let $u^* := \arg\min_{u \in C_i \cup C_j} pre[u]$, i.e., u^* is the first explored vertex in $C_i \cup C_j$. Consider the two cases.

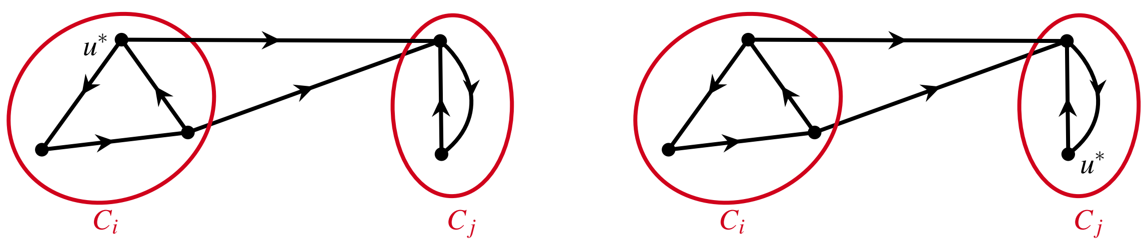

Figure 4: Two cases in proving above claim.

First, assume that $u^* \in C_i$. Then u^* can reach all vertices in $C_i \cup C_j \setminus \{u^*\}$. Hence, all vertices in $C_i \cup C_j \setminus \{u^*\}$ will be explored *within* exploring u^*. In other words, for any vertex $v \in C_i \cup C_j \setminus \{u^*\}$, the interval $[pre[v], post[v]]$ is a subset of $[pre[u^*], post[u^*]]$. This results in two facts: $\max_{u \in C_i} post[u] = post[u^*]$ and $\max_{v \in C_j} post[v] < post[u^*]$. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Second, assume that $u^* \in C_j$. Then u^* can *not* reach any vertex in C_i; otherwise $C_i \cup C_j$ form a single connected component, conflicting to the fact that any connected component must be maximal. Hence, all vertices in C_i will remain unexplored after exploring u^*. In other words, for any vertex $v \in C_i$, the interval $[pre[v], post[v]]$ locates after (and disjoint with) $[pre[u^*], post[u^*]]$. This gives that $\max_{u \in C_i} post[u] > post[u^*]$. Besides, we also have $\max_{v \in C_j} post[v] = post[u^*]$ as all vertices in $C_j \setminus \{u^*\}$ will be examined within exploring u^*. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. $\square$

Finding a Linearization of a DAG

The above key property holds for all directed graphs. We now consider DAGs. Note that each connected component of a DAG G contains exactly one vertex, i.e., each vertex in a DAG G forms the connected component of its own. (Can you spot this using Figure 5?) This is because, if a connected component contains at least two vertices u and v then u can reach v and v can reach u so a cycle must exist. Consequently, the meta-graph G_M of any DAG G is also itself, i.e., $G = G_M$.

Now let's interpret above key conclusion in the context of DAGs. For a DAG $G = (V, E)$, components

C_i and C_j will degenerate into two vertices, say v_i and v_j, edge $(C_i, C_j) \in E_M$ becomes $(v_i, v_j) \in E$, and $\max_{u \in C_i} post[u] = post[i]$, and $\max_{v \in C_j} post[v] = post[j]$. We have

Corollary 1. Let $G = (V, E)$ be a DAG. If $(v_i, v_j) \in E$, then $post[i] > post[j]$.

Now recall the definition of linearization: X is a linearization if and only if for every edge $(v_i, v_j) \in E$, v_i is before v_j in X. Since it is guaranteed above that, for every edge $(v_i, v_j) \in E$, $post[i] > post[j]$, we can immediately conclude that vertices sorted in decreasing order of post-values is a linearization! This order is nothing else but the postlist.

Corollary 2. Let $G = (V, E)$ be a DAG. The postlist generated in the DFS-with-timing algorithm is a linearization of G.

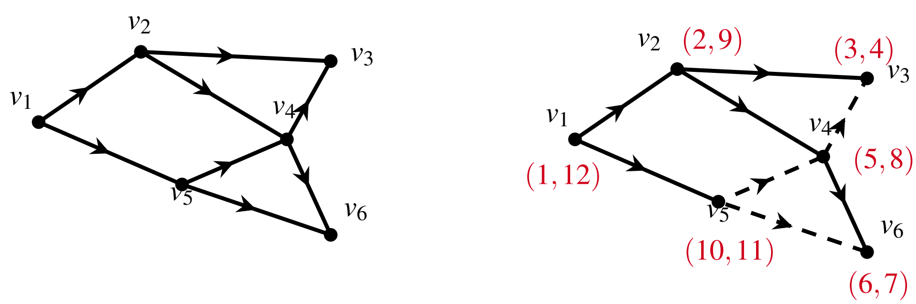

Figure 5: Example of running DFS (with timing) on a DAG G. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $(v_1, v_5, v_2, v_4, v_6, v_3)$, which is a linearization of G.