Deciding Directed Acyclic Graphs

How to decide if a given directed graph is DAG or not? The following variant of DFS (with timing) gives an algorithm. Specifically, when the algorithm examines an edge (v_i, v_j): if v_j has been explored *and* its post-number hasn't been set yet, then the algorithm reports that G is not a DAG.

```
function DFS ($G = (V,E)$)
    $clock = 1$;
    $visited[i] = 0, pre[i] = -1, post[i] = -1$, for $1 \leq i \leq |V|$;
    for $i = 1 \to |V|$
        if ($visited[i] = 0$)
            explore ($G, v_j$);
        end if;
    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 ($G, v_j$);
        else if ($post[j] = -1$): report "$G$ is not a DAG";
    end for;
    $post[i] = clock$;
    $clock++$;
end algorithm;
```

Now let's prove this algorithm is correct. We first prove that if G is not a DAG then the algorithm will always give that report at some time. Assume that G contains a cycle C as it is not a DAG. 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, within exploring v_j there will be a time that v_i will be explored. Consider the time of examining edge (v_i, v_j) within exploring v_i: at this time $visited[j]$ has been set as 1, but its post-number hasn't been set, as now the algorithm is still within exploring v_j. Therefore, the algorithm will report that G dis not a DAG.

We then prove that if the algorithm gives that report then G indeed is not a DAG. 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 $post[j]$ hasn't been set implies that the algorithm is within exploring v_j. So we have that v_j can reach v_i, as now we are exploring 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.)

Finding a Linearization of a DAG

DFS-with-timing can also be used to find a linearization of a DAG: we simply run DFS-with-timing on the given DAG G and get the postlist (the list of vertices in decreasing order of post value); the postlist will be a linearization of G.

Why? First, observe 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 1?) 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 the conclusions for general directed graphs we obtained in Lecture A11 in the context of DAGs. Consider Claim 1 in Lecture A11: *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* $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. As in a DAG, components C_i and C_j will degenerate into two vertices, say v_i and v_j, and its meta-graph G_M is the same as G, we can translate this claim for DAG as: if in a DAG we have $(v_i, v_j) \in E$ then we must have $post[i] > post[j]$. This immediately gives the desired conclusion following the definition of linearization and the definition of postlist: the postlist is a linearization of G.

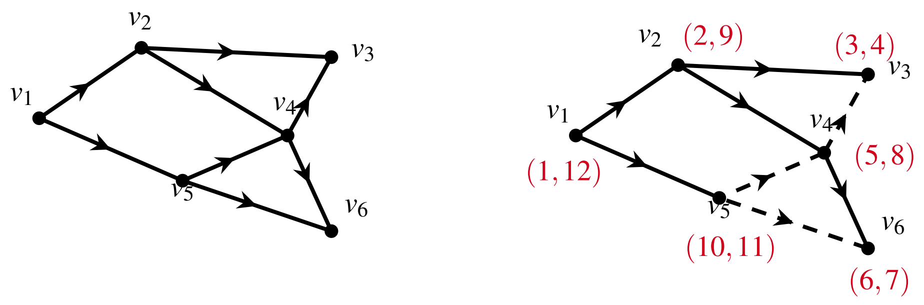

Figure 1: Example of running DFS (with timing) on a DAG G. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $(v_1, v_5, v_2, v_4, v_6, v_3)$, which is a linearization of G.

Queue

A *queue* data structure supports the following four operations.

1. empty (Q): decides if queue Q is empty;
2. insert (Q, x): add element x to Q;
3. find-earliest (Q): return the earliest added element in Q;
4. delete-earliest (Q): remove the earliest added element in Q.

To implement above operations, we can use a (dynamic) array S to stores all elements, and use two pointers, *head* and *tail*, where *head* pointer always points to the first available space in S, and *tail* pointer always points to the earliest added element in S. When we add an element to S we can directly add it to the place *head* points to, and when we delete the earliest added element, we can directly remove the one *tail* points to.

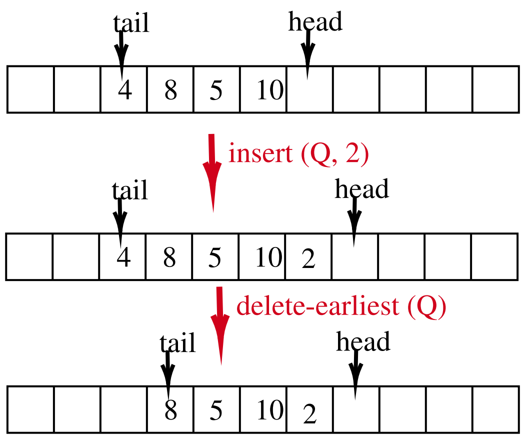

Figure 2: An example of queue.

```
function empty($Q$)
    if $head = tail$: return true;
    else: return false;
end function;

function insert($Q$, $x$)
    $S[head] = x$;
    head = head + 1;
end function;

function find-earliest($Q$)
    return $S[tail]$;
end function;

function delete-earliest($Q$)
    tail = tail + 1;
end function;
```

Note, a queue data structure exhibits a first-in-first-out property (while a stack is first-in-last-out).

Priority Queue

In a priority queue, each element is associated with a *priority* (also called key). In other words, each element in a priority queue is a pair (*key, value*), where *key* indicates its priority, while *value* stores the actual data. A *priority queue* data structure supports the following operations.

1. empty (PQ): decides if priority queue PQ is empty;
2. insert (PQ, x): add element x to PQ;
3. find-min (PQ): return the element in PQ with smallest key (i.e., highest priority);
4. delete-min (PQ): remove the element in PQ with smallest key (i.e., highest priority).

5. decrease-key (*PQ*, pointer-to-an-element, new-key): set the key of the specified element as the given new-key.

Note that a queue can be regarded as a special case of priority, for which the priority is the time an element is added to the queue.

There are numerous different implementations for priority queue (check wikipedia). Here we introduce one of them, *binary heap*. To do it, let's first formally introduce *heap*.

A *heap* is a (rooted) tree data structure satisfies the *heap property*. A heap is either a *min-heap* if it satisfies the *min-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u is smaller than that of v, or a *max-heap* if it satisfies the *max-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u is larger than that of v. Here we always consider a heap as a min-heap, unless otherwise specified.

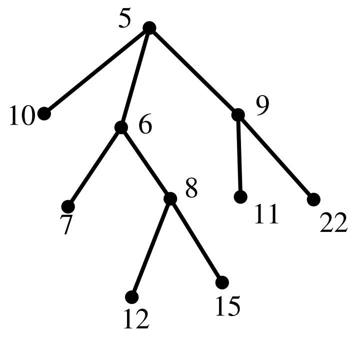

Figure 3: An example of heap. The key of an element is next to vertices

A *binary heap* is a heap with the tree being the *complete binary tree*. A *complete binary tree* is a binary tree (i.e., each vertex has at most 2 children) and that in every layer of the tree, except possibly the last, is completely filled, and all vertices in the last layer are placed from left to right.

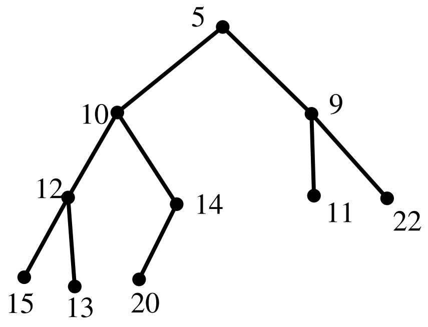

Figure 4: An example of binary heap.

Since a binary heap T is so regular, we can use an array S to store its elements (rather than using adjacency list). The root (i.e., layer 0) of T is placed in $S[1]$ (we assume that the index of S starts from 1), the first element of the layer 1 is placed in $S[2]$, and so on. Generally, the j-th element in the i-th layer of T will be placed in $S[2^i + j - 1]$. We can also easily access the parent and children of an element:

1. the parent element of $S[k]$ is $S[k/2]$ (floor of $k/2$);
2. the left child of $S[k]$ is $S[2k]$; the right child of $S[k]$ is $S[2k+1]$.

We now introduce two common procedures used in implementing a binary heap. These procedures apply when one vertex violates the heap property, and they can adjust the heap to make it satisfy the heap property.

The *bubble-up* function applies when a vertex has a smaller key than its parent.

```
function bubble-up ($S$, $k$)
    $p = k/2$;
    if ($S[k].key < S[p].key$);
        swap $S[p]$ and $S[k]$;
        bubble-up ($S$, $p$);
    end if;
end function;
```

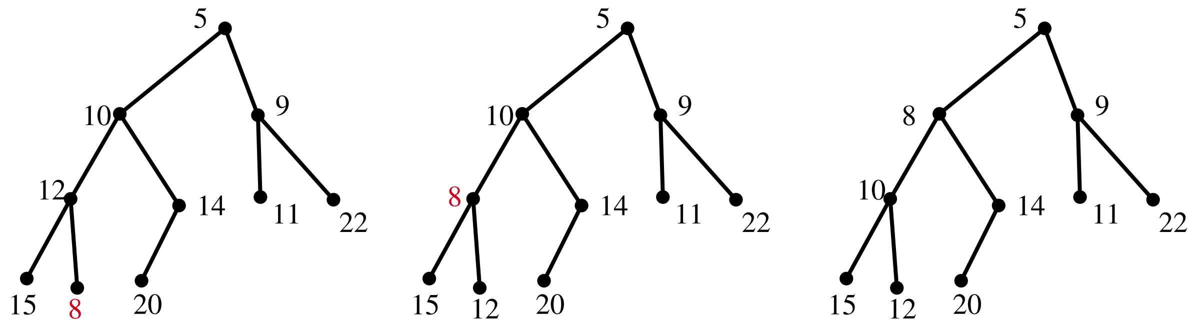

Figure 5: Illustrating bubble-up procedure.

The *sift-down* function applies when a vertex has a larger key than its children.

```
function sift-down ($S$, $k$)
    $c = \arg\min_{t \in \{2k, 2k+1\}} S[t].key$ be the index of the child of $S[k]$ with smaller key;
    if ($S[k].key > S[c].key$);
        swap $S[c]$ and $S[k]$;
        sift-down ($S$, $c$);
    end if;
end function;
```

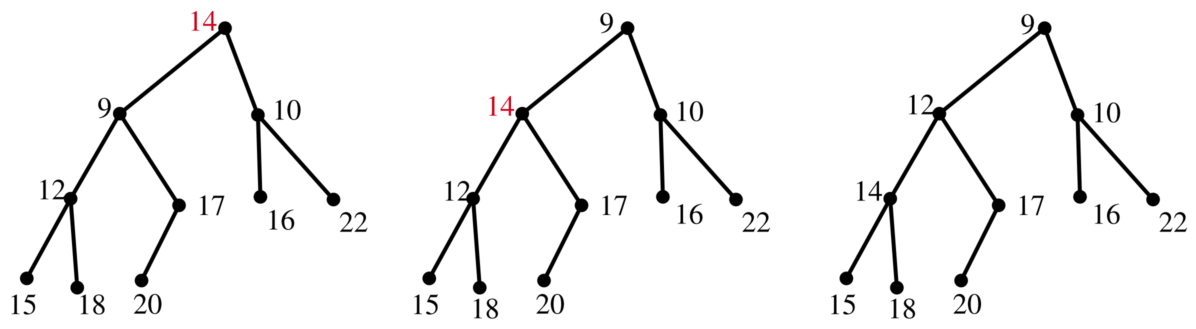

Figure 6: Illustrating sift-down procedure.