CS512 LECTURE NOTES - LECTURE 10

0.1 Connected Components

Let us consider the following relation:
u and v are related iff there is a path from u to v.

We can easily show that this relations is an equivalence relation.

- Reflexive: $\forall u \in V$ there is a path from u to u. This is true by definition of path (see above).
- Symmetric: If there is a path from u to v then there is a path from v to u. This is true, since the graph is undirected we can reverse the direction of the path from u to v to obtain a path from v to u.
- Transitive: If there is a path from u to v and there is a path from v to z, we can concatenate those two paths to get a path from u to z.

Since the relation is an equivalence relation, then it partitions the set of vertices V into equivalence classes, that we will call *Connected Components of G*.

Connected Components Algorithm

We can easily modify the *Explore* algorithm and add a driver to it in such a way that it is able to find all connected components of a graph G.

ConnectedComponents(G)

```
comp=0
for each v ∈ V
   if not marked(v)
      comp++ (found a new component)
      Explore(v)
```

Where

```
Explore(x)
    Mark(x)
    x.component=comp
    for each (x, v) ∈ E
       If not marked(v)
          Explore(v)
```

1 Application of DFS

1.1 pre-visit and post-visit numbers

We will use the following two methods to assign pre and post visit numbers to vertices.

previsit(v)
    count++
    v.previsit=count

postvisit(v)
    count++
    v.postvisit=count

Now we will modify **explore** so that the pre-visit number is assigned the first time that **explore** "visits" a vertex, i.e. when going into the recursion, and the post-visit number is assigned to a vertex v when backtracking.

Explore(x)
    Mark(x)
    previsit(x)
    for each $(x, v) \in E$
        If not marked(v)
            Explore(v)
    postvisit(x)

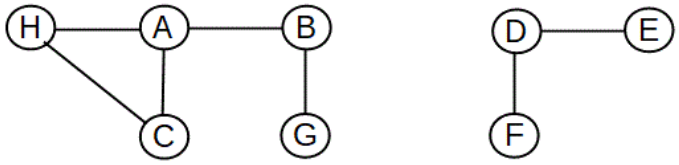
We will now provide a *driver* method **DFS** that calls **explore** on every unmarked vertex of G.

DFS($G = (V, E)$)
    count=0
    for all $v \in V$
        if not marked(v)
            explore(v)

Important: In the case of our examples we will always assume that **DFS** and **explore** will always process vertices in alphabetical (lexicographical) order.

Example

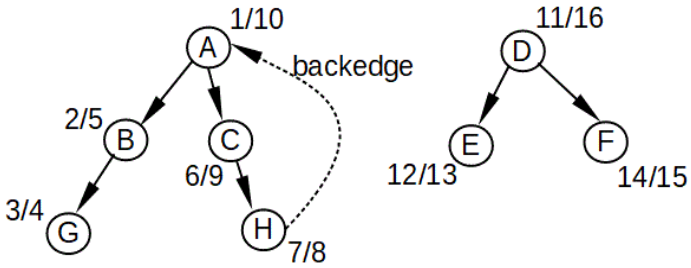
Find pre and post visit numbers for the following undirected graph:



Notice that when following **DFS** we are building a tree based on the order of the recursion. The figure illustrates the DFS tree obtained as well as the pre/post visit numbers.

While exploring a vertex, we can identify two types of edges (u, v)

- **Tree edges:** Those where v has not been marked, that will cause **explore** to recursively explore v.
- **Back edges:** Those where v is marked, therefore v must have been visited before u.



The dotted edge represents a back edge.

1.2 Directed graphs

The algorithm that assigns pre and post visit numbers also works in the case of directed graphs, but the types of edges that we can get is not the same as in the case of undirected graphs where we can only have back and tree edges.
Example

2 Acyclic graphs

A cycle is a path of length > 1 that starts and ends with the same vertex:

$$\text{cycle}=x_0x_1x_2\ldots x_nx_0$$

According to this definition a path of length 0 that goes from x_0 back to itself is not a cycle.

The following theorem can help us design an algorithm to determine whether a graph has a cycle or not.
Theorem
Let $G = (V, E)$ be an undirected graph. G has a cycle $\iff$ **DFS** finds a back edge.
Notice that Once we backtrack out of the recursion of v the entire DFS tree rooted at v will have all of its pre/post visit numbers already computed. This can be proven very easily.
Proof:

- ($\Longleftarrow$)

  Assume that DFS finds a cycle

  $$v_0v_1\ldots v_kv_0$$

  we ordered the vertices in the cycle in such a way that v_0 is the first vertex in the cycle visited by DFS.

  Notice that all the vertices in the cycle are reachable from v_0, including v_k.

  While visiting v_k explore will find the edge (v_k, v_0), but v_0 will have been visited, therefore it will be seen as a back-edge. $\square$

- ($\Longrightarrow$)

  If DFS finds a back-edge (u, v) then there must be a path $v\ x_1\ x_2 \ldots\ u$ from v to u using tree edges (or else (u, v) would not be a back-edge), so we can create a cycle appending edge (u, v) to the path $v\ x_1\ x_2 \ldots\ u$, obtaining the cycle

  $$v\ x_1\ x_2 \ldots\ u\ v$$

So we have an algorithm to detect if a given graph is acyclic. There are two possible cases:

- Directed Acyclic Graphs (DAGS)
- Undirected graphs with no cycles can be:
  - Tree (if connected)
  - Forest (if not connected)

2.1 Topological sorting

Suppose that we are given a set of tasks T_i, $i = 1, \ldots, n$ and a set of precedence constraints, i.e. a subset of tasks $T_{k_1}, T_{k_2}, \ldots, T_{k_r}$ that must be completed before starting task T_j.

This problem can be easily represented by using a DAG where each task is represented by a vertex, and $(i, j) \in E$ if and only if task T_i must precede task T_j.

A linearization is an ordering of the tasks where precedence is satisfied (for simplicity assume that $T_1, , T_2, \ldots, T_n$ is already this ordering):

It can never happen that $i < j$ in the ordering and that T_i does not precede T_j.

In other words (taking the negation of the previous statement) if $(i, j) \in E$ then $i < j$

Algorithm:

Remember that in the case of a DAG there are no back edges, hence we know that if $(u, v) \in E$ then `u.post>v.post`

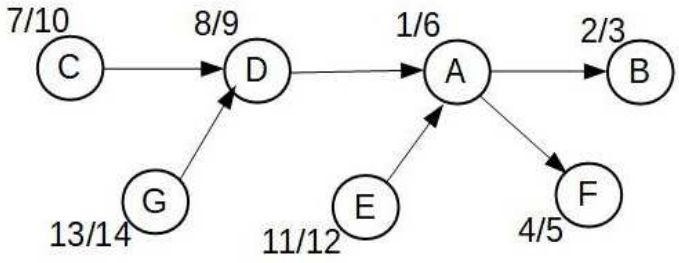
Therefore, if we order the vertices of the DAG (tasks) in decreasing order of their postvisit number, this ordering will satisfy the condition:

$(u, v) \in E \Rightarrow$ `u.post>v.post`

Therefore, task V must appear before task u in the ordering.

Examle

The following graph represents a set of tasks with their corresponding precedence. The pre/post visit numbers are computed using DFS in lexicographical order with respect to the labels of the vertices.



The linearization based on the postvisit numbers shows that all the edges go only forward, therefore it satisfies the required precedence constraints.