CS512 LECTURE NOTES - LECTURE 13

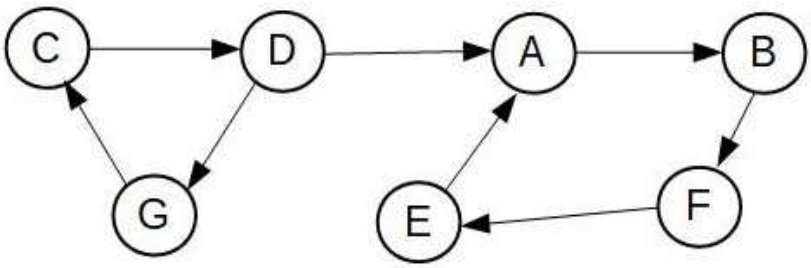
1 Strongly Connected Components

We defined before the concept of *Connectivity* in the case of undirected graphs. There is a similar definition that is useful in the case of directed graphs:

Definition A graph $G = (V, E)$ is said to be strongly connected if and only if $\forall\ u,\ v \in V$ there exists a path from u to v **and** a path from v to u.

We can use the definition above to describe a relation R in such a way that u and v are related if and only if there exists a path from u to v **and** a path from v to u.

It is not difficult to show that the above relation is an equivalence relation, and as such it partitions the se of vertices into equivalence classes called *Strongly Connected Components*.

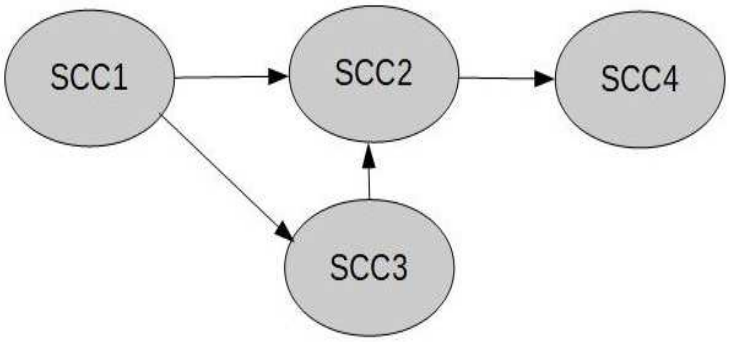


In the graph shown we can see that there is a path from C to D and also a path from D to C, which implies that C and D belong to the same *strongly connected component*, we can also see that even though there is a path from C to F, there is no path from F to C, therefore C and F are not in the same strongly connected component.

From the definitions given above, it is clear that the graph has two strongly connected components: $\{C, D, G\}$ and $\{A, B, E, F\}$.

Notice also that any graph can be decomposed into a DAG of strongly connected components.

The question now is to design an efficient algorithm that is able to find all the strongly connected components of a graph.



Suppose that we have a DAG of strongly connected components and we were able to find a vertex in a sink strongly connected component of the graph. In the figure, SCC4 would be a sink strongly connected component, and SCC1 would be a source strongly connected component.

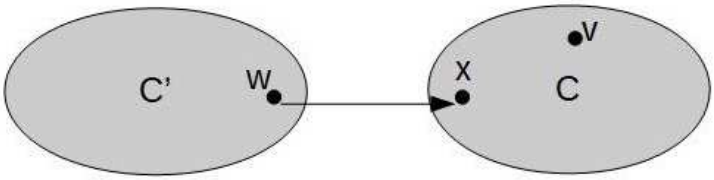
If we use *explore* starting from any vertex of a sink strongly connected component, we would mark only the vertices of that particular component and no other. Once we were able to do this, we can delete the vertices from that component and find another sink SCC and repeat the process:

```
Algorithm SCC(V,E)

comp=0
repeat until V is empty
   find a vertex v in a sink SCC of (V,E)
   comp++
   explore(v)
   delete from V all marked vertices
```

The only problem is to find a sink SCC of a graph G.

Remember that from our discussion about linearization we can see that the vertex with largest postvisit number is in a source strongly connected component. This fact is easy to prove using the order in which DFS processes the vertices. Assume that v is the vertex with maximum postvisit number, but that the component where it is located C is not a sink strongly connected component. That means that there has to be another strongly connected component C' that has an edge (w, x) to C. When DFS is being executed there are only two choices:



1. v is visited before any vertex from C'. But in this case, since there is no path from v to any vertex in C', vertex v has to be assigned a postvisit number before any vertex of C', therefore v cannot have the maximum postvisit number.

2. a vertex in C' is visited first. But in this case, since there is a path from the vertices in C' to v, there is a point during DFS where w is visited before going into x, which is the first vertex in C' visited by DFS. Clearly since there is a path from x to v, then v must be visited before w backtracks from the recursion, which implies that the postvisit number of w must be larger than the postvisit number of v, therefore v cannot be have the maximum postvisit number.

Therefore if v is the vertex with maximum postvisit number, v must be located in a source strongly connected component of G.

The final observation to be able to complete our algorithm is to notice that if a vertex is in a source SCC of the reverse of a graph (G^R) then it is in a sink SCC of the original graph. The reverse of a graph is very easy to compute, just reverse every edge of the original graph G, so we have our algorithm:

```
Algorithm StronglyConnectedComponents(V,E)

Compute the GR=the reverse of G

comp=0
DFS(GR)
order the vertices of V in descending order of postvisit number
for each v in this order
   if not v.marked
      comp++
      explore(v)
```