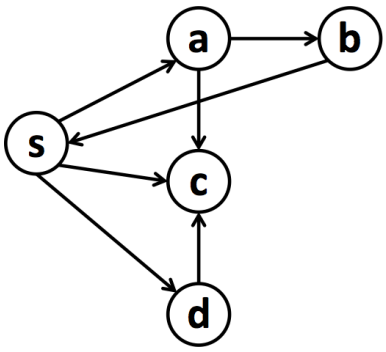
COMPSCI 330: Design and Analysis of Algorithms	September 9, 2014
Lecture # 5	
Lecturer: Debmalya Panigrahi	*Scribe: Roger Zou*

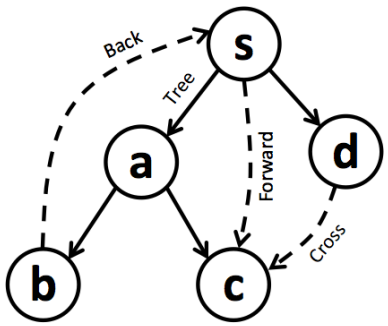
1 Overview

Previously we introduced basic definitions and algorithms in graph theory, with a focus on Depth-First Search (DFS). This lecture, after a brief review of DFS, covers applications of DFS in identifying graph connectivity structure. Content covered include applying DFS for detecting cycles, definitions for Directed Acyclic Graphs (DAG) and (Strongly) Connected Components (SCC), and algorithms for efficiently computing Topological Sort and SCC.

2 DFS Review

We will use the following running example. Let $G = (V, E)$ be a directed graph shown bottom left. Suppose DFS at each iteration alphabetically picks the next vertex to visit. Then T is the path (tree) traversed by DFS shown bottom right.





Note that the ordering of vertices visited by this DFS at each step of the algorithm is: $s\,a\,b\,b\,a\,c\,c\,a\,s\,d\,d\,s$.

2.1 Pre/Post Order and Edge Types

Recall a *pre order* of a vertex is the first time a vertex is discovered in the ordered list of DFS visits. A *post order* of a vertex is the last time a vertex appears in the list of DFS visits. In the example above, $pre(a) = 2$, and $post(a) = 8$.

We are able to classify edges traversed by DFS from looking at their $pre/post$ orders. Please be aware that the tree types and $pre/post$ order values are dependent on the exact DFS traversal order. In other words, there are other valid DFS traversals with different traversal orders and edge types. Below is a chart of edge types with their corresponding $pre/post$ orders.

Table 1: Edge Type and *Pre / Post* Order

Edge Type (u,v)	***pre/post* order**
Tree/forward	$pre(u) < pre(v) < post(v) < post(u)$
Back	$pre(v) < pre(u) < post(u) < post(v)$
Cross	$pre(v) < post(v) < pre(u) < post(u)$

3 Acyclicity in Directed Graphs

Using the properties of edge types and $pre/post$ ordering, we can come up with an algorithm to detect acyclicity in directed graphs. Our approach uses the properties of back edges. The claim is that a graph with no back edges is equivalent to an acyclic graph. Lets first prove a lemma that will help prove the claim.

Lemma 1. *For directed graph $G = (V,E)$ if there exists a cycle $u_1, u_2, \ldots, u_k$, then at least one of these edges is a back edge in ANY DFS.*

Proof. Suppose you run DFS and have a table of $pre/post$ values. Now sort the vertices in decreasing order of *post* values.

Observe that both tree/forward edges and cross edges point forward in this ordering (you can verify this by looking at the table of edge types and $pre/post$ ordering above). However, back edges point backwards to a previous vertex in this ordering. Since by assumption these vertices form a cycle, there must be at least one back edge. Otherwise, there is no way to go from a vertex later in the ordering to a vertex earlier, which contradicts the assumption of the existence of a cycle. □

Theorem 2. *For directed graph $G = (V,E)$, G has no back edges $\iff$ G is acyclic.*

Proof. To prove an if and only if statement, we must show two things: (1) if G has no back edges, then G is acyclic and (2) if G is acyclic, then G has no back edges.

(As a pedagogical statement on general proof strategy, observe that any statement $p \implies q$ is true if and only if its contrapositive $\neg q \implies \neg p$ is also true.)

Thus, lets prove the contrapositive of (1), that if G has a cycle, then G has a back edge. But we just proved this in Lemma 1.

The second step is proving the contrapositive of (2), that if G has a back edge, then G has a cycle. Suppose $e(u,v) \in E$ is a back edge. Then by definition of back edges there exists a path P from v to u in the DFS tree. Simply set $P \leftarrow P + \{e(u,v)\}$, and so P is a cycle in G. □

Remark 1. *Theorem 2 provides good intuition for an efficient algorithm to detect cycles in directed graphs: Simply do DFS. If at any point you encounter a back edge (which can be determined from $pre/post$ values), declare the existence of a cycle.*

4 Topological Sort and Directed Acyclic Graphs

Definition 1. *A Directed Acyclic Graph (DAG) is a directed graph with no cycles.*

Definition 2. *A Topological Sort of a DAG $G = (V,E)$ is a linear ordering of all $v \in V$.*

4.1 Algorithm for Topological sorting of DAGs

Recall the first step of the proof for Lemma 1 orders the vertices of a cycle by decreasing order of *post* values in directed graph $G = (V, E)$ However, since DAGs have no cycles, and thus no back edges (as proven in Theorem 2), we can order the vertices by decreasing *post* value. This follows from the property that in DAGs for every $e(u, v) \in E$, $post(u) > post(v)$, since only tree/forward and cross edges exist. This provides intuition for the following algorithm for computing a topological ordering in a DAG:

1. Given DAG $G = (V, E)$, find its source v_s (vertex with largest *post* value). v_s is next in the ordering.
2. $V \leftarrow V \setminus \{v_s\}$
3. Goto Step 1 until $G = \{\emptyset\}$

5 (Strong) Connectivity

Definition 3. *In a directed graph $G = (V, E)$, vertex u is connected to vertex v if there exists a path from u to v.*

Definition 4. *In a directed graph $G = (V, E)$, vertex u is **strongly** connected to v if there exists a path from u to v, and path from v to u.*

Strong connectivity of vertices in a directed graph $G = (V, E)$ can be thought of as an equivalence relation. This is seen by demonstrating the reflexivity, transitivity, and symmetry properties of strongly connected vertices. Let $u, v, w \in V$ be any vertices in a strongly connected component. Then:

- Reflexivity: trivial path from u to itself.
- Transitivity: if there exists path from u to v, and from v to w, then there exist path from u to w.
- Symmetry: if there exists path from u to v, there exists path from v to u.

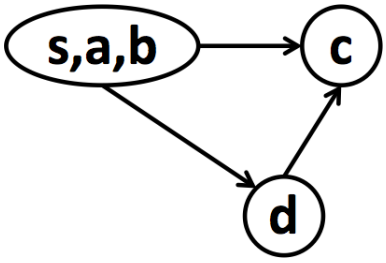
Which of these properties does **not** hold for connectivity in general (not strongly connected)?

5.1 Strong Connected Components (SCC)

Definition 5. *In a directed graph $G = (V, E)$, a Strongly Connected Component (SCC) is a **maximal** subset $C \subseteq V$ s.t. any two vertices $u, v \in C$ are strongly connected.*

Informally, we define **maximal** set in general to mean the largest subset one can construct to satisfy a certain property.

We can partition groups of vertices that are strongly connected into SCCs. Thus ANY directed graph can be decomposed into a DAG on SCCs! From the example directed graph shown above, its SCC is shown below.



For directed graph $G = (V, E)$, and its decomposition into a DAG of SCCs, a *source* SCC C_{src} has no edges from other SCCs going into C_{src}. A *sink* SCC C_{sink} has no edges from C_{sink} going out to other SCCs in G.

5.2 An Efficient Algorithm for SCCs

To come up with an efficient algorithm to find all the SCCs of a directed graph, note that DFS from any vertex v in a *sink* SCC C_{sink} only visits vertices in C_{sink}. Thus, if we have some way of finding such a v, we can do DFS starting from v, delete all nodes DFS visits, and repeat. However, the problem is finding such a v.

The solution arises when one observes that the vertex of highest post value in the DFS of any graph is a source vertex, thus residing in a *source* SCC. By switching the direction of all edges in G to construct new graph G^R, the source vertex in G^R (found using DFS) corresponds to a vertex in the *sink* SCC in G. Then algorithm is as follows:

Algorithm to compute SCCs in directed graph $G = (V, E)$

1. Reverse all edges in G to construct graph G^R. (Note that this process depends on graph implementation, and can be nontrivial to implement efficiently).
2. DFS on G^R, keeping a table of *post* values for all vertices. Name the vertex with largest *post* value v. This is a vertex in *sink* SCC in G.
3. DFS from v in G.
4. Let C_i be the set of all vertices visited by DFS starting from v. Set $V \leftarrow V \setminus C_i$
5. Check the *post* value table (of the vertices remaining in G) and identify the vertex v corresponding to the largest *post* value. Goto Step 3 until $V = \{\emptyset\}$

Running Time: Constructing G^R from G (Step 1) can be performed in $O(m)$. Finding a vertex in the *sink* SCC can be done in $O(m)$. These two steps can be done before the main loop (Steps 3-5). Each iteration i of Steps 3-5 take $O(m_i)$, where m_i is the size of C_i. Because once a vertex (and associated edges) are deleted they can never be visited again, this means that $O(\sum_i m_i) = O(m)$. Thus the worst case running time is $O(m)$.

6 Summary

In this lecture we first reviewed edge types and *pre*/*post* values obtained from the Depth-First Search (DFS) algorithm. Through theoretical results, we proved strong relationships between back edges, acyclicity, and

topological ordering in directed graphs. Then we demonstrated various applications of DFS in revealing the connectivity structure of a directed graph. Specifically, we used DFS to *efficiently* detect cycles in directed graphs, compute Topological ordering on a Directed Acyclic Graph (DAG), identify Strongly Connected Components (SCCs), and decompose a general directed graph to a DAG of SCCs.