15.1 Introduction

Today we're going to continue our discussion of basic graph algorithms. Last class we talked about BFS and DFS, ending with a discussion of DFS. Today we're going to use DFS to design some interesting and extremely fast algorithms.

15.2 Topological Sort

DFS has a number of nice applications, some of which are discussed in the book. One of its most famous applications is doing a "topological sort" on a directed acyclic graph (DAG). A DAG is a directed graph in which there are no directed cycles, although if we reinterpret edges as undirected there might be cycles. A topological sort of a DAG is an ordering of the vertices $v_1, \ldots, v_n$ so that all edges are of the form (v_i, v_j) where $i < j$. In other words, we can line up all of the nodes so that there are no edges going backwards.

It turns out that we can use DFS to find topological sorts of DAGs. First, let's use DFS to characterize DAGs

Theorem 15.2.1 *A directed graph G is a DAG if and only if DFS(G) has no back edges.*

Proof: One direction is obvious: if there is a back edge then we clearly have a directed cycle. For the other direction, suppose that there is a directed cycle C in G. Then consider the node $u \in C$ with minimum start value. Since all nodes in C are reachable from u, they will be descendants of u and so any $v \in C$ with $v \neq u$ has finish$(v) <$ finish(u). But then if v is the predecessor of u on the cycle, the edge (v, u) will be a back edge.

So if G has a back edge then it has a directed cycle and if it has a directed cycle then it has a back edge. This proves the theorem. ■

So now we know that if we call DFS on a DAG, we will never find any back edges. But this automatically gives us a topological sort! When a node finished (i.e. we return from DFS(v)), we just put v at the head of the list. Since there are no back edges, every edge (v, u) is either a forward edge or a cross edge, and thus u has already finished and been put in the list. So no edges go backwards in the list, and we have a topological sort in $O(m + n)$ time.

15.3 Strongly Connected Components

Let's do another application of DFS. This was originally invented by Rao Kosaraju, who you may know or have heard of – he's a professor here, and was one of the founders of the JHU CS department.

Let $G = (V, E)$ be a directed graph. We say that two vertices v and u are *equivalent*, denoted

$u \equiv v$, if v is reachable from u and u is reachable from v. It is not hard to see that this is formally an *equivalence relation*, i.e. it satisfies the following three properties.

1. Reflexivity: $v \equiv v$ (obviously).
2. Symmetry: If $v \equiv w$ then $w \equiv v$. This is immediate from the definition.
3. Transitivity: If $v \equiv w$ and $w \equiv u$ then $v \equiv u$. This is also reasonably straightforward: if there is a path from v to w and a path from w to u, then there is a path from v to u. And the same reasoning implies that there is a path from u to v.

Since $\equiv$ satisfies all three properties, it naturally gives us a partition of V into components in which each pair of vertices are equivalent. These are called the *strongly connected components* (or SCCs) of the graph. So C is a SCC if it is a maximal set such that if $u, v \in C$ then $u \equiv v$. We want to design an algorithm to compute the SCCs of G.

A trivial algorithm would be to run DFS or BFS from each node to see what you can reach, and then if two nodes can reach each other we put them in the same SCC. But this requires n different full DFS runs, each of which takes time $O(n+m)$, so the total running time is $O(n^2 + mn)$. This is not so good.

Let's be a little more careful. Before we define the algorithm, let's first set up some notation and get some intuition. Let $\hat{G}$ be the graph of SCCs: there is a vertex $v(C)$ in $\hat{G}$ for each SCC of G, and there is an edge from $v(C)$ to $v(C')$ if there is an edge from some $u \in C$ to some v in C'.

Lemma 15.3.1 *$\hat{G}$ is a DAG.*

Proof: By definition, if two nodes u and v are in the same SCC then they are each reachable from the other. Thus if there is a path in $\hat{G}$ from $v(C)$ to $v(C')$, every node in C' is reachable from every node in C. Thus if there is a directed cycle $v(C_1), v(C_2), \ldots, v(C_k)$ in $\hat{G}$, then $C_1 \cup C_2 \cup \cdots \cup C_k$ would be a SCC in G. This contradicts our definition of $\hat{G}$, and thus $\hat{G}$ cannot have a directed cycle and so is a DAG. ■

Since $\hat{G}$ is a DAG, there is a topological sort of $\hat{G}$. Suppose I knew this topological sort, and suppose that $v(C)$ is a sink $\hat{G}$ (and so has no edges leaving it). Then if we were to run a DFS from a node in C, we would mark *exactly* the nodes in C. In other words, we would find C! We could then remove these nodes from G, and run a DFS from the new final node in the topological sort of $\hat{G}$ (which was previously the next-to-last node), to find the next SCC. We can just keep repeating this until we find all of the SCCs.

Of course, we don't know $\hat{G}$ or else we would already know the SCCs. What Rao figured out is that it's actually easy to find a node in a sink SCC, even though we don't know $\hat{G}$. First, we need to extend our notion of finishing time to SCCs. Let $\text{finish}(C) = \max_{v \in C} \text{finish}(v)$.

Lemma 15.3.2 *Suppose we run a DFS of G. Let C_1 and C_2 be distinct SCCs, and suppose there is an edge $(u, v) \in E$ where $u \in C_1$ and $v \in C_2$. Then finish$(C_1) >$ finish(C_2).*

Proof: Let $x \in C_1 \cup C_2$ be the first node encountered in $C_1 \cup C_2$ by the DFS. We break into two cases. First, suppose that $x \in C_2$. Then the DFS will visit *all* nodes in C_2 before it visits *any* nodes in C_1, so clearly $\text{finish}(C_1) > \text{finish}(C_2)$. On the other hand, suppose that $x \in C_1$. Then all

other nodes in $C_1 \cup C_2$ will be descendants of x and hence the finish time of x will be larger than the finish time of any other node in $C_1 \cup C_2$. Thus finish$(C_1) >$ finish(C_2) ■

Note that this lemma implies that the node with largest finishing time is in a source SCC (an SCC with no incoming edges in $\hat{G}$). Of course, what we wanted was a node in a *sink* SCC, not a node in a *source* SCC. But this is easily fixed: let G^T be the graph we get by reversing every edge of G. Then the SCCs do not change, but clearly the edges of $\hat{G}$ are all reversed. So a node which is in a source SCC of G^T is in a sink SCC of G.

So we can find a node in a sink SCC. Of course, to get overall fast running time we can't do a DFS each time we remove a SCC. But it turns out that we don't need to (see below and the book)! The finishing times of a DFS of G^T are enough. This gives the following overall algorithm.

```
DFS(G^T) to get finishing times
Repeat until G is empty:
  Let v be the vertex in G with largest finishing time
  Runs DFS(v) in G to define an SCC C of all nodes found in this DFS
  Delete C from G
```

Let's first analyze the running time. Flipping the graph is $O(n+m)$. The first DFS call is $O(n+m)$. Then all of the remaining calls combined are only $O(n+m)$, since we only consider each edge once. Thus the total running time is $O(n+m)$. (Note: there are some missing details here. For example, how do we repeatedly find the vertex with largest finishing time without paying a log factor to maintain a heap? Good exercise to do at home: fill in the blanks! Hint: make a list of vertices ordered by finishing time in $O(n+m)$ time (without paying an extra log factor to sort).

For the correctness proof, see the book (I only sketched it in class, and will do only a sketch here).

Theorem 15.3.3 *Kosaraju's SCC algorithm correctly identifies all SCCs.*

Proof: Let $C_1, C_2, \ldots, C_k$ be the sets identified by the algorithm, in order (so the first set deleted is C_1, then C_2, etc). We claim that for all $i \in \{1, \ldots, k\}$, the set C_i is a sink SCC of $G \setminus \cup_{j=1}^{i-1} C_j$. We prove this by induction on i. For the base case, when $i = 1$, we know from Lemma 15.3.2 and the definition of G^T that the vertex v with largest finishing time is in a sink SCC of G. Hence C_1, the set of nodes reachable from v, is exactly the SCC containing v and so is a sink SCC of G as desired.

For the inductive step, suppose that the statement is true for all $j \leq i-1$, and now we prove it for i. Lemma 15.3.2 implies that the vertex v remaining with largest finishing time must be in an SCC which does not have an edge to any still remaining SCC (or else a node from that SCC would have larger finishing time by Lemma 15.3.2). Hence C_i, the set of nodes reachable from v in $G \setminus \cup_{j=1}^{i-1} C_j$, is indeed a sink SCC of $G \setminus \cup_{j=1}^{i-1} C_j$ as claimed. ■