Harvard University
CS 124

Lecture 3

Lecture Notes by Jeremy Nixon

February 3, 2015

1 Graphs

n is always the number of verticies $|V|$
m is always the number of edges $|E|$

1. Adjacency matrix - an NxN matrix, where the element n(i,j) is 1 or 0 depending on whether or not a connectin exists.

2. Adjacency List - An array A of length n. $A[i]$ is a ointer to a linked list containing all of element i's neighbors.

Tradeoffs

	Adj. Matrix	Adj. List
space	n^2	$\Theta(n+m)$
$(i,j) \in E$?	$\Theta(1)$	$O(n)$ or $\Theta(\deg(i))$

Here the degree is how many things do vertex i point to.

Problem Set 1: Knowing that ou can represent graphs as an adjacency matrix together with ideas from lecture 2 is enough to solve the programming problem. Also, use the hint in the problem set.

Today: Graph Exploration

1. Depth first search (DFS)

2. Breadth first search (BFS)

1.1 Depth First Search (DFS)

DFS clock = 1
visited = boolean[n]
preorder = int[n]
postorder = int[n]

search(v):
    visited[v] = True
    preorder[v] = clock
    clock++
    for u s.t. $(v, u) \in E$
        if visited[u] = False
        then search[u].
    postorder[v] = clock
    clock++

DFS:
dfs($G(V, E)$):
    visited = [False, False,. . . ,False]
    for $u \in V$:
        if visited[u] = False
            search(u)

Run this search code offline to check your understanding.

Runtime of DFS:
If you're given the graph as an adjacency list, the runtime is θ(n+m). There are two types of work that we have to take into account. We have to run through the loop n times. For each time we do a search of v, we have to do some operations. We enter v n times, once per vertex. We do these 5 operations n times. The for loop's running time is the degree of v. How many times do we run this? We have to run it m times, baecause we encounter each edge exactly once in the adjacency list. $\theta(n + m)$

In the Adjacency matrix, we run through the search n times (once for each matrix) and have to run through the edges for each vertex. so we have a runtime $\theta(n^2)$.

So the Adjacency matrix is never better that the Adjacency list in runtime for DFS.

You should imagine that there's a clock that is running in the background. Every time I enter a recursive call, the clock advances.

Preorder and post order numbers - anytime you enter or leave a recursion, the clock advances. The preorder represents entering a recursion and the postorder represents leaving

a recursion.

Applications of DFS:

1. Articualtion points - a vertex without wich the graph would not be connected. By manipulating DFS with these pre and postorder variables, we can find these articulation points.

2. Finding the biconneced components - Just because there is a route from one vertex to another vertex, it doesn't mean that there isn't a single cut that will cut verticies off from one another. There is a modification of DFS that will measure connectivity.

3. topological sort/strongly connected components

4. planarity testing - given a graph, and asked if the graph is planar. Planar - Can you draw the graph on a board with no two edges crossing? This is the problem that we faced in making arduinos (minimizing crossing). Every planar graph will contain a version of either the completely interconnected 5 vertex graph or a 3x3 bipartite matching graph. Crossing number - for every version of drawing this graph, which one has the fewest number of crosses? Solving this problem can lead to faster/more efficient algorithms.

5. isomorphism - given two graphs, is there a way to relabel the verticies to make the identical? There is no fast algorithm for this, but there's a linear time algorithm for graphs that are planar.

Don't always expect road maps to be planar, because there are overpasses.
What does fast mean? In this context (graph isomorphism) he meant polynomial time.
Is graph isomorphism NP complete? If someone could prove that graph isomorphism was np complete, then every algorithm like it would have a polynomial time solution.

More Notation: We can categorize edges into various types based on behavior of DFS.

1. Forest/Tree edge: Edges that are taken in recursive searches - Edge of the graph that appears in the DFS forest.

2. forward edge: Edge from ancestor to descendant in the DFS tree (but not a tree edge)

3. Back Edge: From descendant to ancestor.

4. Cross Edge: Non of the above.

These edge definitions depend on where you started your search. If you start your search in a different place, you'll have a different classification.

Claim: Suppose$(u, v) \in E$, then $post[u] \leq post[v] \iff (u, v)$ is a back edge.
Proof: Back edge $\implies post[u \leq post[v]$ Since u finishes its function call to search before v does.

$((u,v) \in E)(post[u] \leq post[v]) \implies (u,v) is a back edge$. Reason that it's nested: By above claim, intervals are either nested or disjoint, and we know it is not disjoint. It is not disjoint because if we did a single search from some vertes, at some point we're going to enter a search of u. And u is going to try to recursively call a search of v. Either v had been visited already, or it will recursively work.

Problem: Detect if G has a cycle.
Claim: G has a cycle if and only if G has at least one back edge.

Proof: Suppose (u, v) is a back edge. Then if you draw the DFS forest, v is the ancestor of u, so we know that there is some path from v to u. But then there is a back edge (u, v), so there is also an edge from u to v. This is a cycle.

Now let's prove the other direction. Suppose G has a cycle. Consider a cycle $\{v_1, v_2, \ldots, v_l, v_1\}$. Let the vertex v_i have the **smallest** post order in the cycle. Then v_i, v_{i+1} is a back edge since v_{i+1} has post-order less than or equal to that of v_i.

□

1.2 Breadth first search (BFS)