600.363 Introduction to Algorithms / 600.463 Algorithms I **Lecturer:** Michael Dinitz
Topic: Basic Graph Algorithms **Date:** 10/9/14

12.1 Introduction

Graphs are an incredibly important abstraction in computer science, so there has been a huge amount of work on algorithms for graphs. We'll spend much of the rest of the semester studying graph algorithms. Today we'll begin by talking about a few of the simplest, oldest, and (possibly) most useful algorithms.

Everyone should know what a graph is already, but in case you don't: a graph is a pair $G = (V, E)$ where V is a set of elements (usually called *vertices* or *nodes*) and $E \subseteq \binom{V}{2}$ is a set of edges. The notation $\binom{V}{2}$ means that the edges are unordered or *undirected*: the edge $\{u, v\}$ is the same as the edge $\{v, u\}$. We will sometimes talk about directed graphs, in which $E \subseteq V \times V$ and the edge (u, v) is different from the edge (v, u).

It is reasonably standard to let $n = |V|$ and $m = |E|$, so I'm going to use that notation throughout.

12.2 Graph Representations

The first obvious question is: how do we represent a graph? What's the actual data structure we're going to use? There are basically two options: adjacency list and adjacency matrix. Unless otherwise specified, we're going to assume that we use adjacency lists.

Adjacency Matrix: We have an $n \times n$ matrix A. We set $A_{ij} = 1$ if $\{i, j\} \in E$ and set $A_{ij} = 0$ otherwise.

Major benefits: can check whether an edge exists in constant time. It also turns out this matrix has really, really nice properties as a matrix. Obviously it is symmetric, but studying its eigenvalues and eigenvectors can actually tell us a huge amount about the graph. There's a whole area of graph theory called *spectral graph theory* which studies this.

Major drawbacks: takes $\Theta(n^2)$ space even if m is small. Iterating over the edges incident on a vertex v takes time $\Omega(n)$ even if there are only a few such edges (need to scan a whole row or column of the matrix).

Adjacency List: We have an array A of length n, and $A[v]$ is a pointer to a linked list which contains the edges incident on v. In other words, $A[v]$ is a linked list of the vertices that are adjacent to v.

Major benefits: only takes $O(n + m)$ space, can iterate over edges very efficiently.

Major drawbacks: takes a long time to check if $\{u, v\}$ is an edge (time $O(d(u))$ or $O(d(v))$, where $d(u)$ denotes the degree of u).

12.3 Breadth-First Search

BFS is a way of traversing or searching a graph. It is one of the simplest and most obvious ways of doing this, and has a few nice properties.

Pseudocode is in CLRS – I'd suggest looking at it. But basically we start out from some node v. We mark it as complete, and put all of its neighbors in a queue (first-in first-out). We then pull from the queue, mark that node as complete, put its neighbors in the queue, etc. The key point is that we only put a node in the queue if it is not complete. This means that we essentially search by levels from v. We first see all of the neighbors of v, then we see all vertices at distance 2 from v, then all vertices at distance 3 from v, etc.

The running time is $O(n + m)$. We need to spend $\Theta(n)$ time on initialization, e.g. setting the "completed" field of each node to 0. Once we've started the algorithm, note that the adjacency list of a vertex is scanned only when that vertex is pulled from the queue. Each node is pulled from the queue only once, so each adjacency list is scanned only once, so the time is only $O(m)$. Thus the total running time is $O(n + m)$.

A very nice feature of BFS is that it gives *shortest paths*. This is proved formally in the book, but let's give an informal argument here. Suppose we start at some node v. We claim that for any node u, the path in the BFS tree from v to u is the shortest possible path.

Let's do an informal proof by contradiction. Suppose that u is the closest node to v in which the BFS does not return the shortest path. Let w be the node just before u on the shortest path, and let w' be the node just before u on the BFS path. If $w = w'$ then we are done – w is closer to v than u is, so by assumption the BFS tree has the shortest path to w, and so it also has the shortest path to u. If $w \neq w'$, then we have two cases. If $d(v, w) = d(v, w')$ then we are done, since this means the BFS has a shortest path to u. Otherwise $d(v, w) < d(v, w')$. But this means that the BFS would have found u from w before it found u from w', giving a contradiction.

12.4 Depth-First Search

DFS is the other obvious way to explore a graph. Instead of going by breadth, we go by depth. Again, pseudocode is in the book. This is super easy to implement recursively: when we call DFS on a node v, we iterate through its neighbors and for any unmarked neighbor u we mark it and then call DFS on u. It can also be implemented iteratively using a stack rather than a queue.

DFS also takes time $O(n+m)$. We again have to spend $\Theta(m)$ time initializing the graph, and then since we only visit each vertex once we only scan each adjacency list once, so we spend time $O(m)$ in the main algorithm.

12.5 Topological Sort

DFS has a number of nice properties, 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 is not immediately obvious, but is not too hard to see, that every DAG has a topological sort (and indeed a directed graph has a topological sort if and only if it is a DAG). Lets see how to use DFS in a DAG to construct one.

Let G be a DAG. First note that we can assume that G is connected when interpreted as an undirected graph, since if it's not we can just run the algorithm separately in each connected component. We start by finding a node v which has no incoming edges. Such a node must exist, or else there would be a directed cycle. Now from this node we recursively call DFS on each of its neighbors. When all of these recursive calls return, we add v to the head of the list. At this point there might still be some vertices that we didn't find, so we simply start from another vertex with no incoming edges.

The key point is that we only insert a node into the list when all of its recursive calls have completed, and then we add it to the head of the list. To prove that this work, consider an edge (u, v). We need to prove that u comes before v in our final list. Since G is a DAG, there is no path from v to u. So once v is reached by the DFS, u cannot be reached until v is completed and added into the list. Thus v must finish before u, so will come after u in the list.

12.6 Strongly Connected Components

Let's do another application of DFS. This was originally invented by S. Rao Kosaraju, who you may know or have heard of.

A *strongly connected component* of a directed graph G is a subset $C \subseteq V$ such that for every pair $u, v \in C$, there is a directed path from u to v and from v to u. Our goal is to decompose a graph into the smallest number of connected components. It's not too hard to see that the decomposition is essentially unique.

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 component. 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. Let's start by first running DFS on the graph G. This assigns a finishing time to each node. We now flip all of the edges (in time $O(n + m)$) to get a new graph G^T. We now run DFS in G^T, but we do it *in decreasing order of finishing times.* That is, the first node we start from is the last node to complete from the original call. This hits some nodes, but others are not hit. So we choose the remaining node with largest (original) finishing time, and do a DFS on that (but stopping when we hit any node marked by the previous DFS). We just repeat this. Then in the end we return each tree from these final DFS calls.

Let's first analyze the running time. The first DFS call is $O(n+m)$. Flipping the graph 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 $O(n + m)$.

Now let's prove correctness. We first want to argue that if two nodes u and v can each reach each other then they will be in the same component. This is trivial – once one of them is hit by a final DFS, the other one will also be hit by the same DFS. So they will be in the same final tree and thus the same component.

Now we want to argue that if u and v are not strongly connected (say v cannot reach u) then they will not be in the same component. This is not quite as trivial. Let C and C' be two arbitrary strongly connected components. Then there can only be edge from C to C' or vice versa – if there are edges in both directions then $C \cup C'$ would be a strongly connected component. So the graph of strongly connected components is a DAG. If in a topological sort of this DAG the SCC containing v is further down than the SCC containing u, this must mean that the finishing time of v is smaller than the finishing time of u (since u cannot finish until v has finished). So when we do our reverse DFSes, we will see u before v. But since there is no directed path from u to v in the flipped graph, the DFS tree containing u will not contain v. Thus they will be in separate outputted components.