Notes for Lecture 3

1 Introduction

Sections 5.3 and 5.4 of CLR, which you should read, introduce a lot of standard notation for talking about graphs and trees. For example a graph $G = G(V, E)$ is defined by the finite set V of *vertices* and the finite set E of *edges* (u, v) (which connect pairs of vertices u and v). The edges may be *directed or undirected* (depending on whether the edge points from one vertex to the other or not). We will draw graphs with the vertices as dots and the edges as arcs connecting them (with arrows if the graph is directed). We will use terms like *adjacency*, *neighbors*, *paths* (directed or undirected), *cycles*, etc. to mean the obvious things. We will define other notation as we introduce it.

Graphs are useful for modeling a diverse number of situations. For example, the vertices of a graph might represent cities, and edges might represent highways that connect them. In this case, each edge might also have an associated length. Alternatively, an edge might represent a flight from one city to another, and each edge might have a weight which represents the cost of the flight. The typical problem in this context is computing shortest paths: given that you wish to travel from city X to city Y, what is the shortest path (or the cheapest flight schedule). There are very efficient algorithms for solving these problems. A different kind of problem—the traveling salesman problem—is very hard to solve. Suppose a traveling salesman wishes to visit each city exactly once and return to his starting point, in which order should he visit the cities to minimize the total distance traveled? This is an example of an NP-complete problem, and one we will study later in this course.

A different context in which graphs play a critical modeling role is in networks of pipes or communication links. These can, in general, be modeled by directed graphs with capacities on the edges. A directed edge from u to v with capacity c might represent a pipeline that can carry a flow of at most c units of oil per unit time from u to v. A typical problem in this context is the max-flow problem: given a network of pipes modeled by a directed graph with capacities on the edges, and two special vertices—a source s and a sink t—what is the maximum rate at which oil can be transported from s to t over this network of pipes? There are ingenious techniques for solving these types of flow problems, and we shall see some of them later in this course.

In all the cases mentioned above, the vertices and edges of the graph represented something quite concrete such as cities and highways. Often, graphs will be used to represent more abstract relationships. For example, the vertices of a graph might represent tasks, and the edges might represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. There is a very fast algorithm for solving this problem that we will see shortly.

We usually denote the number of nodes of a graph by $|V| = n$, and its number of edges by $|E| = e$. e is always less than or equal to n^2, since this is the number of all ordered pairs of n nodes—for undirected graphs, this upper bound is $\frac{n(n-1)}{2}$. On the other hand, it is reasonable to assume that e is not much smaller than n; for example, if the graph has no *isolated nodes*, that is, if each node has at least one edge into or out of it, then $e \geq \frac{n}{2}$. Hence, e ranges roughly between n and n^2. Graphs that have about n^2 edges—that is, nearly as many as they could possibly have—are called *dense*. Graphs with far fewer than n^2 edges are called *sparse*. Planar graphs (graphs that can be drawn on a sheet of paper without crossings of edges) are always sparse, having $O(n)$ edges.

2 Reasoning about Graphs

We will do a few problems to show how to reason about graphs.

Handshaking Lemma: Let $G = (V, E)$ be an undirected graph. Let degree(v) be the degree of vertex v, i.e. the number of edges incident on v, Then $\sum_{v \in V} \text{degree}(v) = 2e$.

Proof: Let E_v be the list of edges adjacent to vertex v, so degree$(v) = |E_v|$. Then $\sum_{v \in V} \text{degree}(v) = \sum_{v \in V} |E_v|$. Since every edge (u, v) appears in both E_u and E_v, each edge is counted exactly twice in $\sum_{v \in V} |E_v|$, which must then equal $2e$.

A *free tree* is a connected, acyclic undirected graph. A *rooted tree* is a free tree where one vertex is identified as the *root*.

Corollary: In a tree (free or rooted), $e = n - 1$.

Proof: If the tree is free, pick any vertex to be the root, so we can identify the *parent* and *children* of any vertex. Every vertex in the tree except the root is a child of another vertex, and is connected by an edge to its parent. Thus e is the number of children in the tree, or $n - 1$, since all vertices except the root are children.

In fact one can show that if $G = (V, E)$ is undirected, then it is a free tree if and only if it is connected (or acyclic) and $e = n - 1$ (see Theorem 5.2 in CLR).

3 Data Structures for Graphs

One common representation for a graph $G(V, E)$ is the *adjacency matrix*. Suppose $V = \{1, \cdots, n\}$. The adjacency matrix for $G(V, E)$ is an $n \times n$ matrix A, where $a_{i,j} = 1$ if $(i, j) \in E$ and $a_{i,j} = 0$ otherwise. The advantage of the adjacency matrix representation is that it takes constant time (just one memory access) to determine whether or not there is an edge between any two given vertices. In the case that each edge has an associated length or a weight, the adjacency matrix representation can be appropriately modified so entry $a_{i,j}$ contains that length or weight instead of just a 1. The most important disadvantage of the adjacency matrix representation is that it requires n^2 storage, even if the graph is very sparse, having as few as $O(n)$ edges. Moreover, just examining all the entries of the matrix would require n^2 steps, thus precluding the possibility of algorithms for sparse graphs that run in linear $(O(e))$ time.

The *adjacency list* representation avoids these disadvantages. The adjacency list for a vertex i is just a list of all the vertices adjacent to i (in any order). In the adjacency list representation, we have an array of size n to represent the vertices of the graph, and the i^{th} element of the array points to the adjacency list of the i^{th} vertex. The total storage used by an adjacency list representation of a graph with n vertices and e edges is $O(n + e)$. We will use this representation for all our graph algorithms that take linear or near linear time. Using the adjacency lists representation, it is easy to iterate through all edges going out of a particular vertex v—and this is a very useful kind of iteration in graph algorithms (see for example the depth-first search procedure in a later section); with an adjacency matrix, this would take n steps, even if there were only a few edges going out of v. One potential disadvantage of adjacency lists is that determining whether there is an edge from vertex i to vertex j may take as many as n steps, since there is no systematic shortcut to scanning the adjacency list of vertex i.

So far we have discussed how to represent directed graphs on the computer. To represent undirected graphs, just remember that an undirected graph is just a special case of directed graph: one that is symmetric and has no loops. The adjacency matrix of an undirected graph has the additional property that $a_{ij} = a_{ji}$ for all vertices i and j. This means the adjacency matrix is symmetric.

What data structure(s) would you use if you wanted the advantages of both adjacency lists and adjacency matrices? That is, you wanted to loop though all $k = \text{degree}(v)$ edges adjacent to v in $O(k)$ time, determine if edge (u, v) exists in $O(1)$ time on the average for any pair of vertices u and v, and use $O(n + e)$ total storage?