CS 70 Discrete Mathematics for CS
Fall 2004 Rao

Lecture 10

An Introduction to Graphs

Formulating a simple, precise specification of a computational problem is often a prerequisite to writing a computer program for solving the problem. Many computational problems are best stated in terms of graphs: A directed graph $G(V,E)$ consists of a finite set of vertices V and a set of (directed) edges or arcs E. An arc is an ordered pair of vertices (v,w) and is usually indicated by drawing a line between v and w, with an arrow pointing towards w. Stated in mathematical terms, a directed graph $G(V,E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V. Undirected graphs may be regarded as special kinds of directed graphs, such that $(u,v) \in E \leftrightarrow (v,u) \in E$. Thus, since the directions of the edges are unimportant, an undirected graph $G(V,E)$ consists of a finite set of vertices V, and a set of edges E, each of which is an unordered pair of vertices $\{u,v\}$.

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 travelled? This is an example of an NP-complete problem, and one we will study towards the end of 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.

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 superfast algorithm for this problem, that we will see shortly.

Representing graphs on the computer

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 has 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 linear time algorithms for graphs with many fewer than n^2 edges.

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 m edges is $O(n+m)$. We will use this representation for all our graph algorithms that take linear or near linear time. 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.