CS 61B: Lecture 28
Wednesday, April 2, 2014

GRAPHS

A graph G is a set V of vertices (sometimes called nodes), and a set E of edges (sometimes called arcs) that each connect two vertices together. To confuse you, mathematicians often use the notation G = (V, E). Here, "(V, E)" is an _ordered_pair_ of sets. This isn't as deep and meaningful as it sounds; some people just love formalism. The notation is convenient when you want to discuss several graphs with the same vertices; e.g. G = (V, E) and T = (V, F).

Graphs come in two types: _directed_ and _undirected_. In a directed graph (or _digraph_ for short), every edge e is directed from some vertex v to some vertex w. We write "e = (v, w)" (also an ordered pair), and draw an arrow pointing from v to w. The vertex v is called the _origin_ of e, and w is the _destination_ of e.

In an undirected graph, edges have no favored direction, so we draw a curve connecting v and w. We still write e = (v, w), but now it's an unordered pair, which means that (v, w) = (w, v).

One application of a graph is to model a street map. For each intersection, define a vertex that represents it. If two intersections are connected by a length of street with no intervening intersection, define an edge connecting them. We might use an undirected graph, but if there are one-way streets, a directed graph is more appropriate. We can model a two-way street with two edges, one pointing in each direction. On the other hand, if we want a graph that tells us which cities adjoin each other, an undirected graph makes sense.

```
     ---   Bancroft  ---             ---            --------      ------------
     |1|<------------|2|<------------|3|            |Albany|------|Kensington|
     ---             ---             ---            --------      ------------
      |               ^              | ^                    \       /
 Dana |     Telegraph |     Bowditch | |      ------------     ----------
      v               |              v |      |Emeryville|-----|Berkeley|
     ---             ---             ---      ------------     ----------
     |4|------------>|5|------------>|6|            \      /
     ---    Durant   ---             ---          ---------     ----------
                                                  |Oakland|-----|Piedmont|
                                                  ---------     ----------
```

Multiple copies of an edge are forbidden, but a directed graph may contain both (v, w) and (w, v). Both types of graph can have _self-edges_ of the form (v, v), which connect a vertex to itself. (Many applications, like the two illustrated above, don't use these.)

A _path_ is a sequence of vertices such that each adjacent pair of vertices is connected by an edge. If the graph is directed, the edges that form the path must all be aligned with the direction of the path. The _length_ of a path is the number of edges it traverses. Above, <4, 5, 6, 3> is a path of length 3. It is perfectly respectable to talk about a path of length zero, such as <2>. The _distance_ from one vertex to another is the length of the shortest path from one to the other.

A graph is _strongly_connected_ if there is a path from every vertex to every other vertex. (This is just called _connected_ in undirected graphs.) Both graphs above are strongly connected.

The _degree_ of a vertex is the number of edges incident on that vertex. (Self-edges count just once in 61B.) Hence, Berkeley has degree 4, and Piedmont has degree 1. A vertex in a directed graph has an _indegree_ (the number of edges directed toward it) and an _outdegree_ (the number of edges directed away). Intersection 6 above has indegree 2 and outdegree 1.

Graph Representations

There are two popular ways to represent a graph. The first is an _adjacency_ _matrix_, a |V|-by-|V| array of boolean values (where |V| is the number of vertices in the graph). Each row and column represents a vertex of the graph. Set the value at row i, column j to true if (i, j) is an edge of the graph. If the graph is undirected (below right), the adjacency matrix is _symmetric_: row i, column j has the same value as row j, column i.

	1	2	3	4	5	6
1	-	-	-	T	-	-
2	T	-	-	-	-	-
3	-	T	-	-	-	T
4	-	-	-	-	T	-
5	-	T	-	-	-	T
6	-	-	T	-	-	-

	Alb	Ken	Eme	Ber	Oak	Pie
Albany	-	T	-	T	-	-
Kensington	T	-	-	T	-	-
Emeryville	-	-	-	T	T	-
Berkeley	T	T	T	-	T	-
Oakland	-	-	T	T	-	T
Piedmont	-	-	-	-	T	-

It should be clear that the maximum possible number of edges is |V|^2 for a directed graph, and slightly more than half that for an undirected graph. In many applications, however, the number of edges is much less than Theta(|V|^2). For instance, our maps above are _planar_graphs_ (graphs that can be drawn without edges crossing), and all planar graphs have O(|V|) edges. A graph is called _sparse_ if it has far fewer edges than the maximum possible number. ("Sparse" has no precise definition, but it usually implies that the number of edges is asymptotically smaller than |V|^2.)

For a sparse graph, an adjacency matrix representation is very wasteful. A more memory-efficient data structure for sparse graphs is the _adjacency_ _list_. An adjacency list is actually a collection of lists. Each vertex v maintains a list of the edges directed out from v. The standard list representations all work--arrays (below left), linked lists (below right), even search trees (because you can traverse one in linear time).

```
  ---   -----                 ---   ------   ------
1 |.+-> | 4 |          Albany |.+-> |Ken.+-> |Ber*|
  ---   =====                 ===   ======   ======
2 |.+-> | 1 |      Kensington |.+-> |Alb.+-> |Ber*|
  ---   =========             ===   ======   ======
3 |.+-> | 2 | 6 |  Emeryville |.+-> |Ber.+-> |Oak*|
  ---   =========             ===   ======   ======   ------   ------
4 |.+-> | 5 |        Berkeley |.+-> |Alb.+-> |Ken.+-> |Eme.+-> |Oak*|
  ---   =========             ===   ======   ======   ======   ------
5 |.+-> | 2 | 6 |     Oakland |.+-> |Eme.+-> |Ber.+-> |Pie*|
  ---   =========             ===   ======   ------   ------
6 |.+-> | 3 |        Piedmont |.+-> |Oak*|
  ---   -----                 ---   ------
```

The total memory used by all the lists is Theta(|V| + |E|).

If your vertices have consecutive integer names, you can declare an array of lists, and find any vertex's list in O(1) time. If your vertices have names like "Albany," you can use a hash table to map names to lists. Each entry in the hash table uses a vertex name as a key, and a List object as the associated value. You can find the list for any label in O(1) average time.

An adjacency list is more space- and time-efficient than an adjacency matrix for a sparse graph, but less efficient for a _complete_graph_. A complete graph is a graph having every possible edge; that is, for every vertex u and every vertex v, (u, v) is an edge of the graph.

Graph Traversals

We'll look at two types of graph traversals, which can be used on either directed or undirected graphs to visit each vertex once. Depth-first search (DFS) starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible. For example, if your graph is an undirected tree, DFS performs a preorder (or if you prefer, postorder) tree traversal.

Breadth-first search (BFS) starts by visiting an arbitrary vertex, then visits all vertices whose distance from the starting vertex is one, then all vertices whose distance from the starting vertex is two, and so on. If your graph is an undirected tree, BFS performs a level-order tree traversal.

In a graph, unlike a tree, there may be several ways to get from one vertex to another. Therefore, each vertex has a boolean field called "visited" that tells us if we have visited the vertex before, so we don't visit it twice. When we say we are "marking a vertex visited", we are setting its "visited" field to true.

Assume that we are traversing a strongly connected graph, thus there is a path from the starting vertex to every other vertex.

When DFS visits a vertex u, it checks every vertex v that can be reached by some edge (u, v). If v has not yet been visited, DFS visits it recursively.

```
public class Graph {
  // Before calling dfs(), set every "visited" flag to false; takes O(|V|) time
  public void dfs(Vertex u) {
    u.visit();                                // Do some unspecified thing to u
    u.visited = true;                                // Mark the vertex u visited
    for (each vertex v such that (u, v) is an edge in E) {
      if (!v.visited) {
        dfs(v);
      }
    }
  }
}
```

In this DFS pseudocode, a "visit()" method is defined that performs some action on a specified vertex. For instance, if we want to count the total population of the city graph above, "visit()" might add the population of the visited city to the grand total. The order in which cities are visited depends partly on their order in the adjacency lists.

The sequence of figures below shows the behavior of DFS on our street map, starting at vertex 1. A "V" is currently visited; an "x" is marked visited; a "*" is a vertex which we try to visit but discover has already been visited.

```
V<-2<-3  x<-2<-3  x<-2<-3  x<-V<-3  *<-V<-3  x<-x<-3  x<-x<-V  x<-*<-V  x<-x<-V
|  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^
v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v
4->5->6  V->5->6  x->V->6  x->x->6  x->x->6  x->x->V  x->x->x  x->x->x  x->x->*
```

DFS runs in O(|V| + |E|) time if you use an adjacency list; O(|V|^2) time if you use an adjacency matrix. Hence, an adjacency list is asymptotically faster if the graph is sparse.

What's an application of DFS? Suppose you want to determine whether there is a path from a vertex u to another vertex v. Just do DFS from u, and see if v gets visited. (If not, you can't there from here.)

I'll discuss BFS in the next lecture.

More on the Running Time of DFS

Why does DFS on an adjacency list run in O(|V| + |E|) time?

The O(|V|) component comes up solely because we have to initialize all the "visited" flags to false (or at least construct an array of flags) before we start.

The O(|E|) component is trickier. Take a look at the "for" loop of the dfs() pseudocode above. How many times does it iterate? If the vertex u has outdegree d(u), then the loop iterates d(u) times. Different vertices have different degrees. Let the total degree D be the sum of the outdegrees of all the vertices in V.

```
  D =  sum  d(v).
     v in V
```

A call to dfs(u) takes O(d(u)) time, NOT counting the time for the recursive calls it makes to dfs(). A depth-first search never calls dfs() more than once on the same vertex, so the total running time of all the calls to dfs() is in O(D). How large is D?

- If G is a directed graph, then D = |E|, the number of edges.
- If G is an undirected graph with no self-edges, then D = 2|E|, because each edge offers a path out of two vertices.
- If G is an undirected graph with one or more self-edges, then D < 2|E|.

In all three cases, the running time of depth-first search is in O(|E|), NOT counting the time required to initialize the "visited" flags.