CS61B Summer 2006
Instructor: Erin Korber
Lecture 25, 8 Aug.

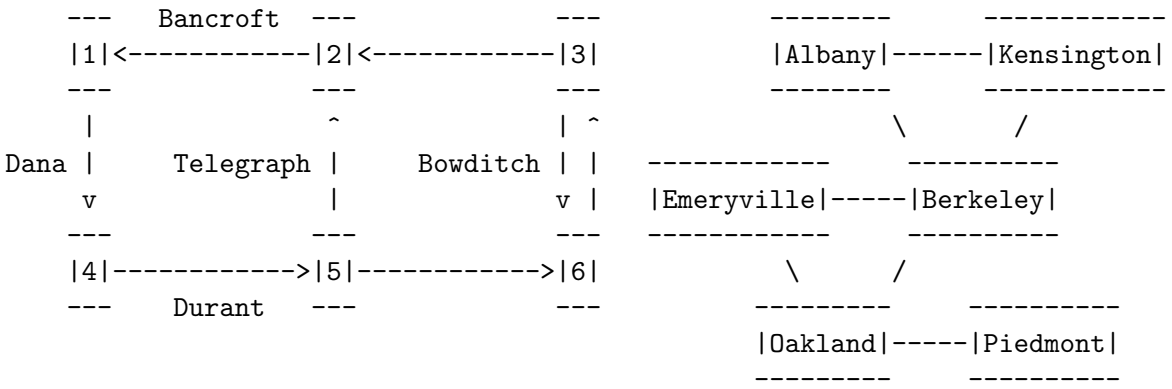
Graphs

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. Mathematicians often use the notation G = (V, E) - here, "(V,E')" is an ordered pair of sets.

Graphs come in two main 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)" (an ordered pair), and draw an arrowhead on the edge so that it points 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 simply draw a line 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.



Both directed and undirected graphs can have self-edges of the form (v, v), which connect a vertex to itself. (Many applications, like the example illustrated above, will not use these. Multiple copies of an edge are forbidden, but a directed graph may contain both (v, w) and (w, v).

1 Definitions

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, $\langle 4, 5, 6, 3 \rangle$ is a path of length 3. It is perfectly respectable to talk about a path of length zero, such as $\langle 2 \rangle$. The *distance* from one vertex to another is the length of the shortest path from one to the other. A *simple path* is one on which no vertex is repeated (but the starting and ending vertex might be the same). A *cycle* is a path with length greater than 0 from a vertex v to itself.

An undirected graph is *connected*, and a directed graph is *strongly connected*, if there is a path from any vertex to any other. A directed graph is *weakly connected* if for all u,v in V, there is a path either from u to v or from v to u. Both of the graphs shown above are strongly connected. A *complete* graph is one that contains every possible edge. Neither of the graphs shown above are complete.

The *degree* of a vertex is the number of edges incident on that vertex. Hence, in the above example, 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.

You may have noticed that graphs sometimes look like trees, and that is because trees are just a kind of graph. A *tree* is a connected graph with no cycles, and a *rooted tree* additionally has a node designated as the root. (Trees that are not rooted trees are more often referred to as DAGs - *directed acyclic graphs*, and the word "tree" is reserved for rooted trees.) The condition for being a tree (connected and acyclic) can be equivalently stated as: for any two vertices u,v in V, there is exactly one path between them.

2 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 a number of edges in $O(|V|)$. A graph is called *sparse* if it has far fewer edges than the maximum possible number, and *dense* if it has almost the maximum number of edges. ("Sparse" and "dense" do not have precise definitions, but "dense" usually implies that the number of edges is in $O(|V|^2.)$, and "sparse" that the number of edges is in $O(|V| \log |V|).$) For example, the densest graph possible for any set V of vertices is the complete graph on V.

For a sparse graph, an adjacency matrix representation is very wasteful. A more memory-efficient data structure for them is the *adjacency list*. An adjacency list is actually a collection of linked lists. Each vertex v maintains a linked list (usually singly-linked) of the edges directed out from v.

```
  ---   ----
1 |.+-> |4*|
  ---   ====
2 |.+-> |1*|
  ---   ====   ----
3 |.+-> |2.+-> |6*|
  ---   ====   ----
4 |.+-> |5*|
  ---   ====   ----
5 |.+-> |2.+-> |6*|
  ---   ====   ----
6 |.+-> |3*|
```

```
                ---   ------   ------
      Albany    |.+-> |Ken.+-> |Ber*|
                ===   ======   ======
   Kensington   |.+-> |Alb.+-> |Ber*|
                ===   ======   ======
   Emeryville   |.+-> |Ber.+-> |Oak*|
                ===   ======   ======   ------   ------
     Berkeley   |.+-> |Alb.+-> |Ken.+-> |Eme.+-> |Oak*|
                ===   ======   ======   ======   ------
      Oakland   |.+-> |Eme.+-> |Ber.+-> |Pie*|
                ===   ======   ------   ------
     Piedmont   |.+-> |Oak*|
```

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

If your vertices have consecutive integer names, you can declare an array of linked 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 linked lists. Each entry in the hash table uses a vertex name as a key, and a List as the associated value. You can still find the linked list for any label in O(1) time.

An adjacency list is more space- and time-efficient than an adjacency matrix for a sparse graph, but less efficient for a dense graph.

3 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 a tree, DFS performs a preorder or postorder tree traversal.

Breadth-first search (BFS) starts at an arbitrary vertex and 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 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 "visited" to true.

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

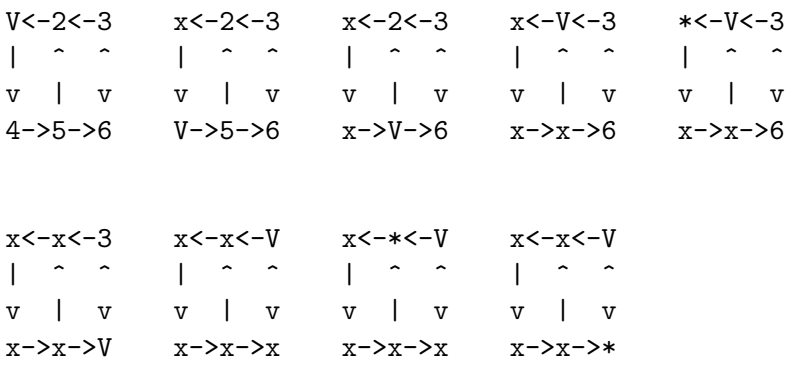
3.1 Depth-first search

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 void dfs(Vertex u) {
  u.visit();                          // Do some unspecified action 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);
    }
  }
```

```
}
```

The sequence of figures below shows the behavior of DFS on our street map, starting at vertex 1. Note that the order in which nodes are visited depends partly on their order in the adjacency lists. A "V" is currently being visited; an "x" is marked visited; a "*" is a vertex which we try to visit but discover has already been visited.



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.

3.2 Breadth-first search

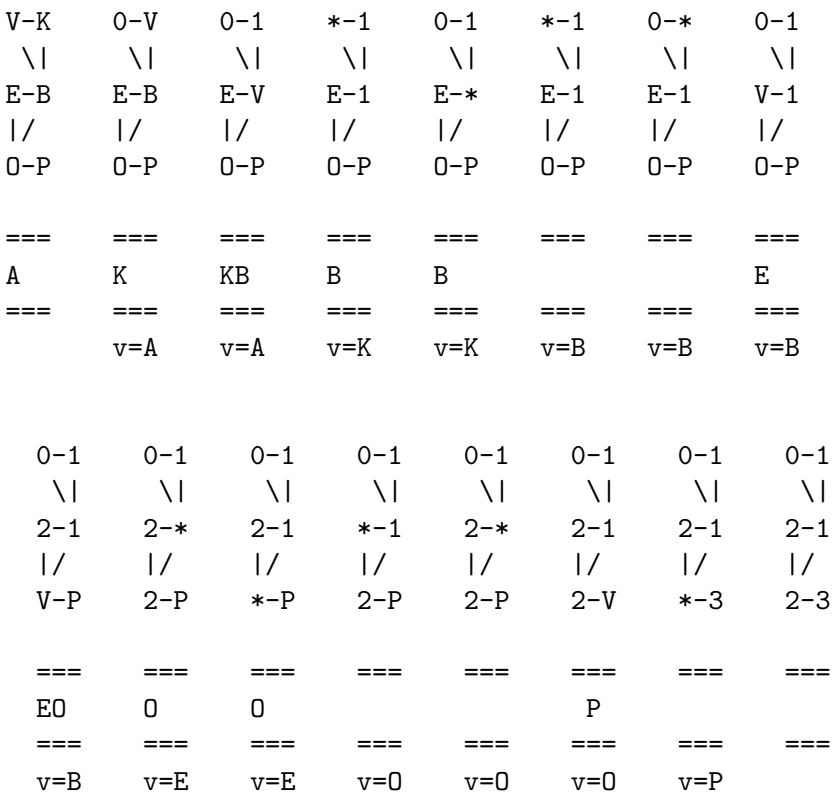
Breadth-first search (BFS) is a little more complicated than depth-first search, because it's not naturally recursive. We use a queue so that vertices are visited in order according to their distance from the starting vertex. (Hopefully you recall this from Exam 2!)

```
public void bfs(Vertex u) {
  u.visit(null);                  // Do some action to u
  u.visited = true;               // Mark the vertex u visited
  q = new Queue();
  q.enqueue(u);
  while (q is not empty) {
    v = q.dequeue();
    for (each vertex w such that (v, w) is an edge in E) {
      if (!w.visited) {
        w.visit(v);
        w.visited = true;
        q.enqueue(w);
      }
    }
  }
}
```

Notice that when we visit a vertex, we pass the edge's origin vertex in as a parameter. This allows us to do a computation such as finding the distance of the vertex from the starting vertex, or finding the shortest path between them. The visit() method below accomplishes both these tasks. (assume that a vertex has parent and depth parameters):

```
public void visit(Vertex origin) {
   this.parent = origin;
   if (origin == null) {
      this.depth = 0;
   } else {
      this.depth = origin.depth + 1;
   }
}
```

When an edge (v, w) is traversed to visit a Vertex w, the depth of w is set to the depth of v plus one, and v is set to be the "parent" of w. In the sequence of figures below, a "V" is currently visited; a digit shows the depth of a vertex that is marked visited; a "*" is a vertex which we try to visit but discover has already been visited. Underneath each figure of the graph, the queue and the current value of the variable "v" in bfs() appear. This figure shows BFS on the "cities" graph from above.



After we finish, we can find the shortest path from any vertex to the starting vertex by following the parent pointers. These pointers form a tree rooted at the starting vertex. Note that they point in the direction *opposite* the search direction that got us there.

Why does this work? The starting vertex is enqueued first, then all the vertices at a distance of 1 from the start, then all the vertices at a distance of 2, and so on. Why? When the starting vertex is dequeued, all the vertices at a distance of 1 are enqueued, but no other vertex is. When the depth-1 vertices are dequeued and processed, all the vertices at a distance of 2 are enqueued, because every vertex at a distance of 2 must be reachable by a single edge from some vertex at a distance of 1. No other vertex is enqueued, because every vertex at a distance less than 2 has been marked, and every vertex at a distance greater than 2 is not reachable by a single edge from some vertex at a distance of 1.

BFS, like DFS, runs in $O(|V| + |E|)$ time if you use an adjacency list; $O(|V|^2)$ time if you use an adjacency matrix.

4 Weighted Graphs

A *weighted graph* is a graph in which each edge is labeled with a numerical weight. A weight might express the distance between two nodes, the cost of moving from one to the other, the resistance between two points in an electrical circuit, or many other things.

In an adjacency matrix, each weight is stored in the matrix. Whereas an unweighted graph uses an array of booleans, a weighted graph uses an array of ints, doubles, or some other numerical type.

In an adjacency list, recall that each edge is represented by a node in a list. Each node must be enlarged to include a weight, in addition to the reference to the destination vertex.

There are two particularly common problems involving weighted graphs. One is the "shortest path problem". Suppose a graph represents a street map, and each road is labeled with the amount of time it takes to drive from one intersection to the next. What's the fastest way to drive from one city to another? A shortest path algorithm will tell us. (Stated more generally, the shortest path problem is the problem of finding a path between two vertices such that the total weight of the edges on the path is minimized.) A closely related problem is the well-known "travelling salesman" problem, which is the problem of finding the shortest path that goes through each node exactly once and returns to the source. We'll discuss several shortest path algorithms over the next couple of days.

The second common problem is constructing a "minimum spanning tree." Suppose that you're wiring a house for electricity. Each node of the graph represents an outlet, or the source of electricity. Every outlet needs to be connected to the source, but not necessarily directly–possibly routed via another outlet. The edges of the graph are labeled with the length of wire you'll need to connect one node to another. How do you connect all the nodes together with the shortest length of wire? Stated more generally, this is the problem of finding the "minimum cost" subgraph that connects all the vertices, where "cost" is the total weight of the edges in the graph. We'll discuss some algorithms for solving this problem next week.