CS 61B: Lecture 29
Monday, April 7, 2014

## GRAPHS (continued)

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.

```
  public void bfs(Vertex u) {
    for (each vertex v in V) {                                 // O(|V|) time
      v.visited = false;
    }
    u.visit(null);                              // Do some unspecified thing to u
    u.visited = true;                                // Mark the vertex u visited
    q = new Queue();                                              // New queue...
    q.enqueue(u);                                    // ...initially containing u
    while (q is not empty) {                  // With adjacency list, O(|E|) time
      v = q.dequeue();
      for (each vertex w such that (v, w) is an edge in E) {
        if (!w.visited) {
          w.visit(v);                           // Do some unspecified thing to w
          w.visited = true;                          // Mark the vertex w visited
          q.enqueue(w);
        }
      }
    }
  }
```

```
public class Vertex {
  protected Vertex parent;
  protected int depth;
  protected boolean visited;

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

Notice that when we visit a vertex, we pass the edge’s origin vertex 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 at right accomplishes both these tasks.

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 become the _parent_ of w.

The sequence of figures below shows BFS running on the city adjacency graph (Albany, Kensington, Emeryville, Berkeley, Oakland, Piedmont) from last lecture, starting from Albany. 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, I depict the queue and the current value of the variable "v" in bfs().

```
V-K  0-V  0-1  *-1  0-1  *-1  0-*  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1
 \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|
E-B  E-B  E-V  E-1  E-*  E-1  E-1  V-1  2-1  2-*  2-1  *-1  2-*  2-1  2-1  2-1
|/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/
O-P  O-P  O-P  O-P  O-P  O-P  O-P  O-P  V-P  2-P  *-P  2-P  2-P  2-V  *-3  2-3

===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===
A    K    KB   B    B              E    EO   O    O              P
===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===
     v=A  v=A  v=K  v=K  v=B  v=B  v=B  v=B  v=E  v=E  v=O  v=O  v=O  v=P
```

After we finish, we can find the shortest path from any vertex to the starting vertex by following the parent pointers (right). 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.

```
0<--1
 ^
  \
   \
2-->1
  ^
 /
/
2<--3
```

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.

Recommendation: pull out a piece of paper, draw a graph and a program stack, and simulate BFS, with you acting as the computer and executing bfs() line by line. You will understand it much better after taking the time to do this.

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.

## 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. Edges missing from the graph can be represented by a special number like Integer.MIN_VALUE, at the cost of declaring that number invalid as an edge weight. (If you want to permit every int to be a valid edge weight, you might use an additional array of booleans as well.)

In an adjacency list, recall that each edge is represented by a listnode. Each listnode must be enlarged to include a weight, in addition to the reference to the destination vertex. (If you’re using an array implementation of lists, you’ll need two separate arrays: one for weights, and one for destinations.)

There are two particularly common problems involving weighted graphs. One is the _shortest_path_problem_. Suppose a graph represents a highway map, and each road is labeled with the amount of time it takes to drive from one interchange to the next. What’s the fastest way to drive from Berkeley to Los Angeles? A shortest path algorithm will tell us. You’ll learn several of these algorithms if you take CS 170.

The second 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?

Kruskal’s Algorithm for Finding Mimumum Spanning Trees
------------------------------------------------------
Let G = (V, E) be an undirected graph. A _spanning_tree_ T = (V, F) of G is a graph containing the same vertices as G, and |V| - 1 edges of G that form a tree. (Hence, there is exactly one path between any two vertices of T.)

If G is not connected, it has no spanning tree, but we can instead compute a _spanning_forest_, or collection of trees, having one tree for each connected component of G.

If G is weighted, then a _minimum_spanning_tree_ T of G is a spanning tree of G whose total weight (summed over all edges of T) is minimal. In other words, no other spanning tree of G has a smaller total weight.

Kruskal’s algorithm computes the mimimum spanning tree of G as follows.

```
[1]  Create a new graph T with the same vertices as G, but no edges (yet).
[2]  Make a list of all the edges in G.
[3]  Sort the edges by weight, from least to greatest.
[4]  Iterate through the edges in sorted order.
     For each edge (u, w):
[4a]   If u and w are not connected by a path in T, add (u, w) to T.
```

Because this algorithm never adds (u, w) if some path already connects u and w, T is guaranteed to be a tree (if G is connected) or a forest (if G is not).

Why is T a minimum spanning tree in the end? Suppose the algorithm is considering adding an edge (u, w) to T, and there is not yet a path connecting u to w. Let U be the set of vertices in T that are connected (so far) to u, and let W be a set containing all the other vertices, including w. Let the _bridge_edges_ be any edges in G that have one end vertex in U and one end vertex in W. Any spanning tree must contain at least one of these bridge edges. As long as we choose a bridge edge with the least weight, we are safe. (There may be several bridge edges with the same least weight, in which case it doesn’t matter which one we choose.)

Because we go through the edges of G in order by weight, (u, w) must have the least weight, because it’s the first edge we encountered connecting U to W. (See Goodrich and Tamassia page 649 for a proof that choosing the bridge edge with least weight is always the right thing to do.)

What is the running time of Kruskal’s algorithm? As we’ll discover in the next two lectures, sorting |E| edges takes O(|E| log |E|) time. The tricky part is, in [4a], determining whether u and w are already connected by a path. The simplest way to do this is by doing a depth-first search on T starting at u, and seeing if we visit w. But if we do that, Kruskal’s algorithm might take Theta(|E| |V|)) time.

We can do better. In Lecture 33, we’ll learn how to solve that problem quickly, so that all the iterations of [4a] together take less than O(|E| log |E|) time.

If we use an adjacency list, the running time is in O(|V| + |E| log |E|). But |E| < |V|^2, so log |E| < 2 log |V|. Therefore, Kruskal’s algorithm runs in O(|V| + |E| log |V|) time.

If we use an adjacency matrix, the running time is in O(|V|^2 + |E| log |E|), because it takes Theta(|V|^2) time simply to make a list of all the edges.