Lecture 38

Graph Traversals

We've talked about graphs, and what sorts of real world data they could be used to represent. We also defined a lot of graph terms, and talked about how to represent a graph in our programs. We also talked a little about what sorts of questions we could ask of a graph. That is, what kind of analysis or information can we get out of a graph representation.

One fairly straightforward thing we can do is traverse a graph, either to list all the vertices in the graph or to search for a particular vertex in the graph. Another straightforward question we can ask is does the graph have any cycles?

Breadth First Search

Breadth first search, or BFS, is a simple algorithm to traverse a graph. It starts with some node, and visits that node. It then adds all of its adjacent nodes to the list of nodes to be visited, and visits them in order. You can think of it as searching all the nodes that are one step away, then all the nodes that are two steps away, and so on.

Here is the basic algorithm:

1. Enqueue the start node
2. while the queue is not empty
   (a) Dequeue a node
   (b) if the the node has not been visited previously, visit it and enqueue all of the nodes children

For an acyclic undirected graph (that is, a tree), this amounts to a *level-order traversal*. If we were actually searching for a value in the graph, the `visit` function would stop the traversal if the item was found.

Why do we need to keep track of whether or not the node has been visited before? If we had a tree (or a graph without cycles), then we wouldn't. We need it to make sure we don't start to loop.

Notice that we can save a little bit of work if we check the adjacent nodes before adding them to the queue.

There is still one small problem with this algorithm. What if the graph is not connected? We probably still want to search the whole thing. So, instead of enqueing just the start node, we need to iterate over the all the vertices.

For each node:

1. Enqueue the node
2. while the queue is not empty

(a) Dequeue a node

(b) if the the node has not been visited previously, visit it and enqueue all of the unvisited adjacent nodes

Graph Representation

Refer to `bfs_dfs_demo.cpp` on the code page for this lecture.

We use an adjacency list to represent our graph. For this example, is not the most efficient, but it is more intuitive and easier to follow, plus it is very different from your upcoming assignment. We can imagine that we have a list of the vertices in the graph. Each vertex has an associated list of adjacency nodes in it.

So, our graph then is really just a vector of pairs. Where each pair is a string (the name of the vertex), and another vector, which is a list of the adjacent vertices by name.

Thus, the type for our graph is

```
vector< pair<string, vector<string> > >
```

This is fairly cumbersome to write, and we're going to be passing the graph around the code as a parameter. Luckily, C++ gives us a useful keyword for just such an occasion – `typedef`. It allows us to define a new type, in terms of existing types.

```
typedef vector< pair<string, vector<string> > > Graph
```

Now, every time we say `Graph` we are really using our complex type.

BFS Code

The code begins by testing to see if the graph is empty. If so, we just won't do anything.

Next, we need to set up a queue to hold the vertices to visit. We also set up the `visited` list. Here, we use a C++ `set`. This data type is convenient for two reasons: 1. it ignores duplicates so we can just add the vertices without worrying about increasing size unnecessarily, and 2. it provides us with a simple `count` function, which returns 1 if an item is in the set, and 0 if it is not. This will make checking to see if a node has been visited easy.

We use a `queue` type for the queue for obvious reasons. We are adding to the back of the queue, and removing from the front. Remember, that C++ uses a slightly different setup than Java, and we must examine the first item, then pop it when we want it removed.

The `visit` function in this example simply prints out the name of the vertex, but it could be setup to do anything. It is separated out from the algorithm to emphasize this fact.

Notice that the `visit` function is called exactly once for each node in the graph. This is true even though we iterate over all vertices and add them to the queue because we maintain the `visited` list across `for` loop iterations.

Depth First Search

Depth first search, or DFS, can be thought of as sort of the opposite of breadth first search. Whereas BFS searches all of the closest nodes first, dfs goes in some direction as far as it can. When it hits a dead end, it backtracks to the last branch and goes as far as it can in that direction.

It is basically BFS with a `stack` instead of a `queue`. That is, for an acyclic undirected graph, it is equivalent to pre-order traversal. We use our pre-order recursive algorithm as a basis for DFS:

1. visit start node
2. for each adjacent node *adj*
   (a) if *adj* has not been visited recursively call dfs

Again, if we are searching, visit can indicate when we find what we are searching for.

We will have the same issue with unconnected graphs, but solve it in the same as with BFS.

As with BFS, we use the visited list to prevent looping through nodes we've already seen.

In fact, there are two conditions that indicate that we have encountered a cycle, as defined in lecture last time.

1. We try to visit a node that we have visited before.
2. The visited node is not the parent of the node we are currently visiting.

We need the second condition because in an undirected graph, a node is always adjacent to the node that caused it to be added to the queue. Thus, we track the parent (the node we just came from), the current node, and the nodes we want to add to the queue because they are adjacent.

We can verify a cycle at the time that we are adding the adjacent nodes to the queue. If we are adding a node to the queue that has already been visited, and it is not the node we just came from, then there must be a path from the adjacent node to the parent. We know there is a path from the parent to the current node by definition. We know there is a path to the adjacent from the current node, because it is in the current nodes adjacency list. Thus, there must be a cycle of at least 3 nodes.

1. visit start node
2. for each adjacent node *adj*
   (a) if *adj* has not been visited recursively call dfs with node as parent and *adj* as node.
   (b) else if *adj* is visited, and is not the parent, then cycle detected.