Eric Roberts
CS106B

Handout #51
May 27, 2009

Graph Algorithms

This handout contains additional material that will someday be in a new chapter on graphs. It also include a new example of the important graph algorithms you need for the Pathfinder assignment.

Changes to the assignment

After implementing the Pathfinder several different ways over the long weekend, I have become convinced that doing it the "right" way—by taking advantage of inheritance to create a general graph abstraction and then allow clients to extend it through subclassing—cannot work as an assignment in CS106B without a lot more work. I have therefore reverted to the more traditional approach of building the Pathfinder application using structures instead of objects. Much of the handout doesn't change beyond the discussion of the `graph.h` interface on page 2. Rather than reprint the handout, here is the relevant section:

> The starter project contains a file `graph.h` that defines the following types:
>
> 1. The structure `graphT`, which represents the graph as a whole. Mathematically, a graph consists of two sets: a set of nodes and a set of arcs. The two `Set`-valued fields in the `graphT` structure follow immediately from this definition. In addition, each graph contains a `Map` for converting names to nodes and the name of the image file.
> 2. The structure `nodeT`, which defines the data structure for a node in the graph. In Pathfinder, each `nodeT` has a name and a set of arcs leading to other nodes, along with coordinate information and a `visited` flag that clients can set to indicate that the node has been visited.
> 3. The structure `arcT`, which represents a connection between two nodes. Each `arcT` represents a connection in one direction, which makes it possible to use the `graph.h` interface to model directed graphs. If you want to use it to represent an undirected graph, you need to add an arc in each direction. In Pathfinder, each `arcT` has the addresses of the "from" and "to" nodes, along with the cost of traversing the arc.

The contents of the revised `graph.h` interface appear in Figure 1 on the next page.

The other important change is that you no longer have to implement this interface in the traditional sense. Step 1 has vanished from the list of tasks, along with some of Step 2. You've made considerable progress toward the goal just by waiting for us to make things easier.

The `foreach` macro

A little more than a decade ago, I introduced a simplification into the CS106B curriculum that reduced considerably the overall level of confusion and ended up, I'm convinced, saving an entire lecture day of explanation. Through all of my planning and redesign work, I had concluded that the same simplification could not be made to work in C++, and that we'd have to suffer through the considerably less convenient approach to using iterators. An new idea this weekend ended up giving new life to that simplification, and I've added it to the `Set` class as you have it in the starter folders.

Suppose that you want to step through all of the nodes in a graph to which you have a pointer in the variable `gp`. If you look at Figure 1, you'll see that the `graphT` class has a field called `nodes`, which is a `Set<nodeT *>`. If you use iterators the way we have been all quarter, you first have to declare an iterator, initialize it to contain an iterator over the nodes in the graph, and then use the standard iterator idiom to cycle through the nodes.

Figure 1 The `graph.h` interface

```
/*
 * File: graph.h
 * -------------
 * This file defines structure types for the graphT, nodeT, and arcT
 * types used in the Pathfinder assignment and section problems.
 */

#ifndef _graph_h
#define _graph_h

#include "coord.h"
#include "map.h"
#include "set.h"

struct nodeT;      /* Forward references to these two types so  */
struct arcT;       /* that the C++ compiler can recognize them. */

/*
 * Type: graphT
 * ------------
 * This type represents a graph and consists primarily of two
 * sets -- a set of nodes and a set of arcs -- along with a map
 * of names to nodes and display information.
 */

struct graphT {
    Set<nodeT *> nodes;
    Set<arcT *> arcs;
    Map<nodeT *> nodeMap;
    string imageName;
};

/*
 * Type: nodeT
 * -----------
 * This type represents an individual node and consists of the set
 * of arcs leaving this node, along with a name, a flag indicating
 * whether the node has been visited, and the node coordinates.
 */

struct nodeT {
    Set<arcT *> arcs;
    string name;
    bool visited;
    coordT loc;
};

/*
 * Type: arcT
 * ----------
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct arcT {
    nodeT *from;
    nodeT *to;
    double cost;
};

#endif
```

The resulting code looks like this:

```
Set<nodeT *>::Iterator iter = gp->nodes.iterator();
while (iter.hasNext()) {
    nodeT *np = iter.next();
    . . . do something with that node . . .
}
```

While that code isn't all that bad, most of you will end up thinking a lot about exactly what goes on each line and how to get the various bits of syntax right. In the process, your mind will be occupied with those details long enough to lose track of the big picture. Wouldn't it be better if you could say something closer to what you meant in the first place—possibly like this:

```
foreach (nodeT *np in gp->nodes) {
    . . . do something with that node . . .
}
```

It's now almost English. The good news is that you can now write precisely that. Indeed, for any **Set<***type***>**, you can write

```
foreach (type var in set) {
    . . . do something with the element stored in var . . .
}
```

Depth-first search

The depth-first strategy for traversing a graph is quite similar to the preorder traversal of trees and has the same recursive structure. The only additional complication is that graphs—unlike trees—can contain cycles. If you don't check to make sure that nodes are not processed many times during the traversal, the recursive process can go on forever as the algorithm proceeds.

The algorithm to perform depth-first search on a graph, which is written using the **foreach** idiom, looks like this:

```
void DepthFirstSearch(nodeT *start) {
    if (start->visited) return;
    Visit(start);
    foreach (arcT *ap in start->arcs) {
        DepthFirstSearch(ap->to);
    }
}
```

The depth-first strategy is most easily understood by tracing its operation in the context of a simple example, such as the following directed graph: