Graphs

Eric Roberts  
CS 106B  
February 25, 2013

Outline

1. Examples of graphs  
2. Defining a graph in terms of sets  
3. Graph terminology  
4. Designing the graph.h interface  
5. Traversal strategies: depth-first and breadth-first search

Examples of Graphs

The Definition of a Graph

- A graph consists of a set of nodes together with a set of arcs. The nodes correspond to the dots or circles in a graph diagram (which might be cities, words, neurons, or who knows what) and the arcs correspond to the links that connect two nodes.
- In some graphs, arcs are shown with an arrow that indicates the direction in which two nodes are linked. Such graphs are called directed graphs.
- Other graphs, arcs are simple connections indicating that one can move in either direction between the two nodes. These graphs are undirected graphs.
- Question: Which of these two structures is more general?
Graph Terminology

- Mathematicians often use the words **vertex** and **edge** for the structures that computer scientists call **node** and **arc**. We'll stick with the computer science names, but you might see their mathematical counterparts in algorithmic descriptions.
- The nodes to which a particular node is connected are called its **neighbors**.
- In an undirected graph, the number of connections from a node is called its **degree**. In a directed graph, the number of arcs leading outward is called the **out-degree**, and the number of arcs leading inward is called the **in-degree**.
- A series of arcs that connect two nodes is called a **path**. A path that returns to its starting point is called a **cycle**.
- A graph in which there is a path between every pair of nodes is said to be **connected**.

Designing a Graph Interface

- To take maximum advantage of the intuition that people have with graphs, it makes sense to design a graph package in C++ so that it adheres as closely as possible to the way graphs in mathematics. In particular, the idea that a graph is a set of nodes together with a set of arcs should be clear from the definition.
- Both nodes and arcs have some properties that are closely tied to the graph in which they belong. Each node, for example, must keep track of the arcs that lead to its neighbors. In much the same way, each arc needs to know what nodes it connects.
- At the same time, both nodes and arcs are likely to contain other information that depends on the application, such as the name of a node or the cost of traversing an arc.

Three Strategies

1. **Use low-level structures.** This design uses the structure types **Node** and **Arc** to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.

2. **Define classes for each of the component types.** This design uses the **Node** and **Arc** classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application.

3. **Adapt a hybrid strategy.** This design defines a **Graph** class but parameterizes that class so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.
The class-based graph.h Interface

/*
* File: graph.h
* ------------
* This interface exports a parameterized graph class used to represent
* graphs, which consist of a set of nodes and a set of arcs.
*/

#ifndef _graph_h
#define _graph_h

#include <string>
#include "error.h"
#include "map.h"
#include "set.h"

/*
* Functions: nodeCompare, arcCompare
* ----------------------------------
* Standard comparison functions for nodes and arcs.
*/
template <typename NodeType>
int nodeCompare(NodeType *n1, NodeType *n2);
template <typename NodeType,typename ArcType>
int arcCompare(ArcType *a1, ArcType *a2);

The class-based graph.h Interface

/*
* Class: Graph<NodeType,ArcType>
* ------------------------------
* This class represents a graph with the specified node and arc types.
* The NodeType and ArcType parameters indicate the structure type or class
* used to represent the nodes and arcs, respectively.  These types must
* contain the following:
* - A string field called name
* - A Set<ArcType *> field called arcs
* - A NodeType * field called start
* - A NodeType * field called finish
*/
template <typename NodeType,typename ArcType>
class Graph {
public:

    // Constructor: Graph
    // Usage: Graph<NodeType,ArcType> g;
    // Creates an empty Graph object.
    Graph();

    // Destructor: ~Graph
    // Usage: (usually implicit)
    // Frees the internal storage allocated to represent the graph.
    ~Graph();

    // Method: size
    // Usage: int size = g.size();
    // Returns the number of nodes in the graph.
    int size();

    // Method: isEmpty
    // Usage: if (g.isEmpty()) . . .
    // Returns true if the graph is empty.
    bool isEmpty();

    // Method: clear
    // Usage: g.clear();
    // Reinitializes the graph to be empty, freeing any heap storage.
    void clear();

    // Method: addNode
    // Usage: NodeType *node = g.addNode(name);
    // Adds a node to the graph.  The first version of this method creates a
    // new node of the appropriate type and initializes its fields; the second
    // assumes that the client has already created the node and simply adds it
    // to the graph.  Both versions of this method return a pointer to the node.
    NodeType *addNode(std::string name);
    NodeType *addNode(NodeType *node);

    // Method: getNode
    // Usage: NodeType *node = g.getNode(name);
    // Looks up a node in the name table attached to the graph and returns a
    // pointer to that node.  If no node with the specified name exists,
    // getNode signals an error.
    NodeType *getNode(std::string name);

    // Method: addArc
    // Usage: g.addArc(s1, s2);
    // Adds an arc to the graph.  The endpoints of the arc can be specified
    // either as strings indicating the names of the nodes or as pointers to
    // the node structures.  Alternately, the client can create the arc
    // structure explicitly and pass that pointer to the addArc method.  All
    // three versions of this method return a pointer to the arc in case the
    // client needs to capture this value.
    ArcType *addArc(std::string s1, std::string s2);
    ArcType *addArc(NodeType *n1, NodeType *n2);
    ArcType *addArc(ArcType *arc);

The class-based graph.h Interface
The class-based `graph.h` Interface

```cpp
/* Method: isConnected */
* Usage: if (g.isConnected(n1, n2)) . . .
*        if (g.isConnected(s1, s2)) . . .
* ---------------------------------------
* Returns true if the graph contains an arc from n1 to n2. As in the
* addArc method, nodes can be specified either as node pointers or by
* name.
*/
bool isConnected(NodeType *n1, NodeType *n2);
bool isConnected(std::string s1, std::string s2);
```

---

The private section of `graph.h`

```cpp
/* Private section */
private:

  /* Implementation notes: Graph class */
  /* The graph class is defined -- as it traditionally is in
   * mathematics -- as a set of nodes and a set of arcs. These
   * structures, in turn, are implemented using the Set class.
   * The element type for each set is a pointer to a structure
   * specified as one of the
   * parameters to the class template.
   */
  /* Instance variables */
  Set<NodeType *> nodes;  // The set of nodes in the graph
  Set<ArcType *> arcs;    // The set of arcs in the graph
  Map<std::string, NodeType *> nodeMap; // A map from names to nodes

  /* Implementation notes: data structure */
  /* The class-based graph.h Interface */
  /* Method: getNodeSet */
  * Usage: foreach (NodeType *node in g.getNodeSet()) . . .
  * -------------------------------------------------------
  * Returns the set of all nodes in the graph.
  */
  Set<NodeType *> & getNodeSet();

  /* Method: getArcSet */
  * Usage: foreach (ArcType *arc in g.getArcSet()) . . .
  *        foreach (ArcType *arc in g.getArcSet(node)) . . .
  *        foreach (ArcType *arc in g.getArcSet(name)) . . .
  * --------------------------------------------------------
  * Returns the set of all arcs in the graph or, in the second and third
  * forms, the arcs that start at the specified node, which can be indicated
  * either as a pointer or by name.
  */
  Set<ArcType *> & getArcSet();
  Set<ArcType *> & getArcSet(NodeType *node);
  Set<ArcType *> & getArcSet(std::string name);
```

---

The class-based `graph.h` Interface

```cpp
/* Method: getNeighbors */
* Usage: foreach (NodeType *node in g.getNeighbors(node)) . . .
*        foreach (NodeType *node in g.getNeighbors(name)) . . .
* -------------------------------------------------------------
* Returns the set of nodes that are neighbors of the specified node, which
* can be indicated either as a pointer or by name.
*/
Set<NodeType *> getNeighbors(NodeType *node);
Set<NodeType *> getNeighbors(std::string node);
```

---

Depth-First Search

- The traversal strategy of **depth-first search** (or **DFS** for short) recursively processes the graph, following each branch, visiting nodes as it goes, until every node is visited.
- The depth-first search algorithm requires some structure to keep track of nodes that have already been visited. Common strategies are to include a `visited` flag in each node or to pass a set of visited nodes, as shown in the following code:

```cpp
void depthFirstSearch(Node* start) {  
    Set<Node*> visited;  
    visitUsingDFS(start, visited);  
}
```

```cpp
void visitUsingDFS(Node* start, Set<Node*> & visited) {  
    if (visited.contains(start)) return;  
    Node* node = start;  
    Node* next = node->arcs.first();  
    while (next) {  
        if (!visited.contains(next)) {  
            visited.add(next);  
            visitUsingDFS(next, visited);  
        }  
        next = node->arcs.next();  
    }  
}
```

---

The Implementation Section

```cpp
/* Implementation notes: Graph class */
/* An is typical for layered abstractions built on top of other classes,
 * the implementations of the methods in the graph class tend to be very
 * short, because they can handle most of the work off to the underlying
 * classes.
 */

/* Exercise: Implement the two versions of addNode */
/* Method: addNode */
* Usage: g.addNode(name);
*        g.addNode(node);
* -----------------------------------------------------
* Adds a node to the graph. The first version of this method
* creates a new node of the appropriate type and initializes it
* (if needed); the second assumes that the client has already
* created the node and simply adds it to the graph. Both versions of this
* method return a pointer to the node in case the client needs to
* capture this value.
*/
NodeType *addNode(string name);
NodeType *addNode(NodeType *node);
```

---

Breadth-First Search

- The traversal strategy of breadth-first search (which you used on Assignment #2) proceeds outward from the starting node, visiting the start node, then all nodes one hop away, and so on.
- For example, consider the graph:

```
  n1
 /|\ 
/ | 
/ 2\ 6
n2  n6
 /|\ 
/ 3| 
/ 
/ 5
n3  n5
 /|\ 
/ 4
```

- Breadth-first search begins at the start node (n1), then does the one-hops (n2 and n6), then the two hops (n3, n5, and n7) and finally the three hops (n4).