Eric Roberts and Kevin Miller
CS 106B

Handout #40
February 23, 2015

Graphs

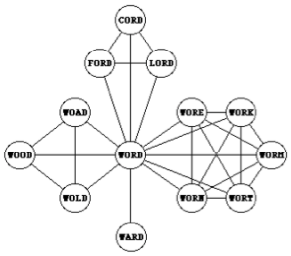
Graphs

Kevin Miller
(standing in for Eric Roberts)
CS 106B
February 23, 2015

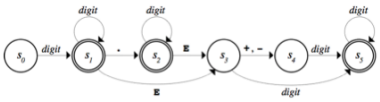
Outline

1. Examples of graphs
2. Defining a graph in terms of sets
3. Graph terminology
4. Designing the `graph.h` interface

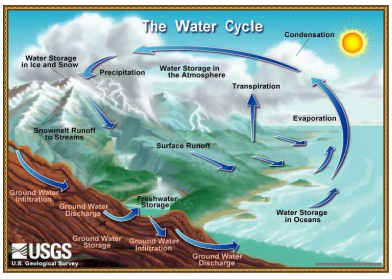
Examples of Graphs



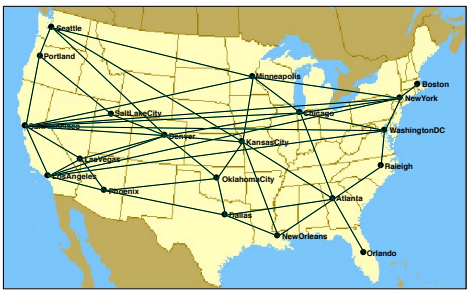
Examples of Graphs



Examples of Graphs



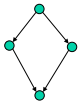
Transportation Graphs



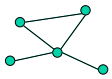
The Definition of a Graph

- 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 and the arcs correspond to the links that connect two nodes.
- A graph can either be ***directed*** *or* ***undirected***.

Directed Graph



Undirected Graph



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++.
- Both nodes and arcs have some properties that are closely tied to the graph in which they belong. Each node must keep track of the arcs that lead to its neighbors. Each arc needs to know what nodes it connects.
- Arcs and nodes may also need extra 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.*** We can make `Node` and `Arc` structs to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.

The low-level `graphtypes.h` Interface

```
/*
 * File: graphtypes.h
 * ------------------
 * This file defines low-level data structures that represent graphs.
 */

#ifndef _graphtypes_h
#define _graphtypes_h

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

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

/*
 * Type: SimpleGraph
 * -----------------
 * This type represents a graph and consists of a set of nodes, a set of
 * arcs, and a map that creates an association between names and nodes.
 */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

The low-level `graphtypes.h` Interface

```
/*
 * Type: Node
 * ----------
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct Node {
   std::string name;
   Set<Arc *> arcs;
};

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

struct Arc {
   Node *start;
   Node *finish;
   double cost;
};

#endif
```

Three Strategies

1. ***Use low-level structures***. We can make **`Node`** and **`Arc`** structs 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. ***Adopt 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 for nodes and arcs, respectively.  These types can contain any
 * fields or methods required by the client, but must contain the following
 * public fields required by the Graph package itself:
 *
 * The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 */

template <typename NodeType,typename ArcType>
class Graph {

public:
```

The class-based `graph.h` Interface

```
/*
 * 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();
```

The class-based `graph.h` Interface

```
/*
 * 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();
```

The class-based `graph.h` Interface

```
/*
 * Method: addNode
 * Usage: NodeType *node = g.addNode(name);
 *        NodeType *node = 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 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);
```

The class-based `graph.h` Interface

```
/*
 * Method: addArc
 * Usage: g.addArc(s1, s2);
 *        g.addArc(n1, n2);
 *        g.addArc(arc);
 * ----------------------
 * 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.  Alternatively, the client can create the arc
 * structure explicitly and pass that pointer to the addArc method.  All
 * three of these versions 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

```
/*
 * 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);

/*
 * Method: getNeighbors
 * Usage: for (NodeType *node : g.getNeighbors(node)) . . .
 *        for (NodeType *node : 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);
```

The class-based `graph.h` Interface

```
/*
 * Method: getNodeSet
 * Usage: for (NodeType *node : g.getNodeSet()) . . .
 * --------------------------------------------------
 * Returns the set of all nodes in the graph.
 */

   Set<NodeType *> & getNodeSet();

/*
 * Method: getArcSet
 * Usage: for (ArcType *arc : g.getArcSet()) . . .
 *        for (ArcType *arc : g.getArcSet(node)) . . .
 *        for (ArcType *arc : 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 private and implementation sections go here.

The Private Section of `graph.h`

```
/* Private section */

private:

/*
 * Implementation notes: Data structure
 * ------------------------------------
 * 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
 * chosen by the client, which is 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     */
```

The Implementation Section

```
/* Implementation notes: Graph class
 * ---------------------------------
 * As 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 hand all the hard work off to the underlying
 * class.
 */
```

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 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 in case the client needs to
 * capture this value.
 */

   NodeType *addNode(string name);
   NodeType *addNode(NodeType *node);
```

Closing Thoughts

1. Graphs are EVERYWHERE
2. Representing a graph effectively is super important
3. Next time: how to traverse graphs and do cool things