SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

Topic Notes: Graph Structure Design

We have seen that Java's object-oriented constructs: interfaces, abstract classes, inheritance with overriding, etc., can be used to design a collection of data structures, in particular, lists.

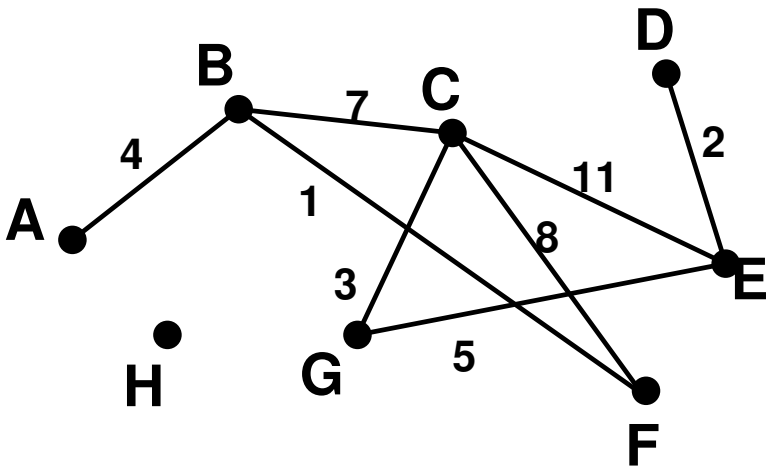
Next, we will consider the design of structures to represent graphs.

Graphs

A *graph* G is a collection of *nodes* or *vertices*, in a set V, joined by *edges* in a set E. Vertices have labels. Edges can also have labels (which often represent *weights*). Such a graph would be called a *weighted graph*.

The graph structure represents relationships (the edges) among the objects stored (the vertices).

For a tree, we might think of the tree nodes as vertices and edges labeled "parent" and "child" to represent nodes that have those relationships.



- Two vertices are *adjacent* if there exists an edge between them.
  e.g., A is adjacent to B, G is adjacent to E, but A is not adjacent to C.
- A *path* is a sequence of adjacent vertices.
  e.g., A-B-C-F-B is a path.
- A *simple path* has no vertices repeated (except that the first and last may be the same).
  e.g., A-B-C-E is a simple path.

- A simple path is a *cycle* if the first and last vertex in the path are same.

  e.g., B-C-F-B is a cycle.

- *Directed graphs* (or *digraphs*) differ from *undirected graphs* in that each edge is given a direction.

- The *degree* of a vertex is the number of edges incident on that vertex.

  e.g., the degree of C is 4, the degree of D is 1, the degree of H is 0.

  For a directed graph, we have more specific *out-degree* and *in-degree*.

- Two vertices u and v are *connected* if a simple path exists between them.

- A *subgraph* S is a *connected component* iff there exists a path between every pair of vertices in S.

  e.g., {A,B,C,D,E,F,G} and {H} are the connected components of our example.

- A graph is *acyclic* if it contains no cycles.

- A graph is *complete* if every pair of vertices is connected by an edge.

There are two principal ways that a graph is usually represented:

1. an *adjacency matrix*, or
2. *adjacency lists*.

As a running example, we will consider an undirected graph where the vertices represent the states in the northeastern U.S.: NY, VT, NH, ME, MA, CT, and RI. An edge exists between two states if they share a common border, and we assign edge weights to represent the length of their border.

We will represent this graph as both an adjacency matrix and an adjacency list.

In an adjacency matrix, we have a two-dimensional array, indexed by the graph vertices. Entries in this array give information about the existence or non-existence of edges.

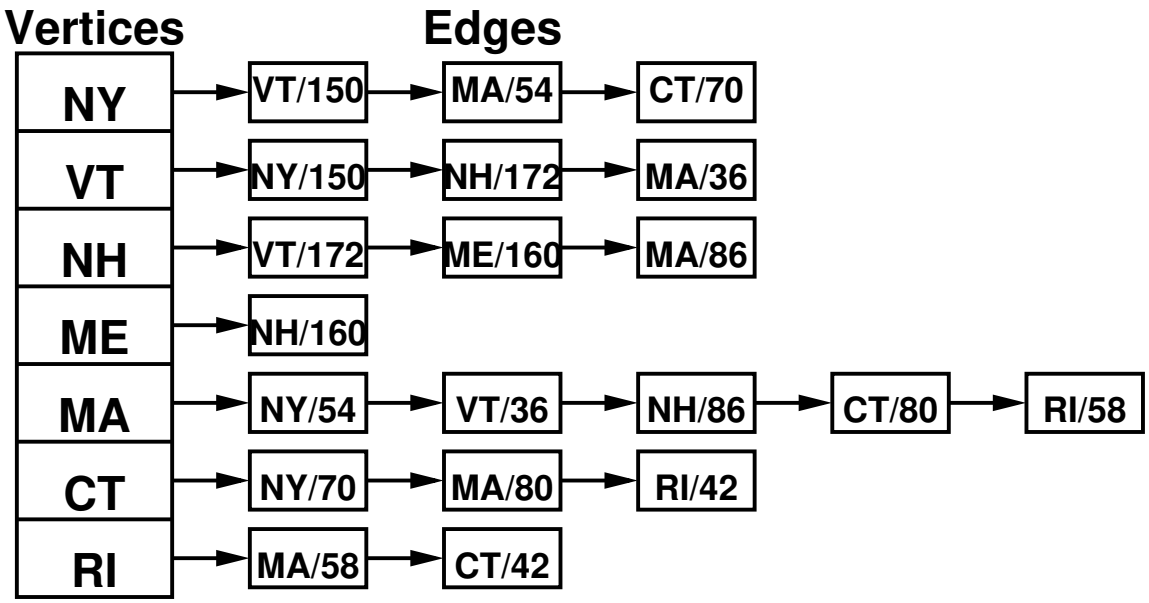
We represent a missing edge with `null` and the existence of an edge with a label (often a positive number) representing the edge label (often representing a weight).

Adjacency matrix representation of NE graph

	NY	VT	NH	ME	MA	CT	RI
NY	null	150	null	null	54	70	null
VT	150	null	172	null	36	null	null
NH	null	172	null	160	86	null	null
ME	null	null	160	null	null	null	null
MA	54	36	86	null	null	80	58
CT	70	null	null	null	80	null	42
RI	null	null	null	null	58	42	null

If the graph is undirected, then we could store only the lower (or upper) triangular part, since the matrix is symmetric.

An adjacency list is composed of a list of vertices. Associated with each each vertex is a linked list of the edges adjacent to that vertex.



In some cases, a matrix representation is more desirable. In other cases, it is the list representation. It depends on the density of the graph and which graph operations need to be most efficient for the task at hand.

Developing a Design for Graph Structures

In class, we will break down the objects and methods needed, then consider how to make use of interfaces, abstract classes, and concrete classes.

Some key points:

- edges and vertices have labels of arbitrary types, specified by type parameters
- vertex labels must be unique, edge labels need not be
- functionality is driven by the kinds of things needed by various graph algorithms:
  - adjacency information
  - visit all vertices/edges
  - counts/degrees
- add/remove vertices and edges in ways that do not invalidate the structure

- consider both directed and undirected graphs
- consider both adjacency matrix and adjacency list implementations

---

An Example: Bailey's Java Structures Graph Implementations

Some key points of Bailey's design and implementation:

- Interface `Graph` defines the basic operations
- `Graph` is treated as a `Structure`, so has common method names with some other structures (apply generally to vertices, not edges)
- `Vertex` and `Edge` are concrete classes, `Vertex` is extended by `GraphListVertex` and `GraphMatrixVertex` to support adjancency list and adjacency matrix implementations
- Abstract classes `GraphMatrix` and `GraphList` each implement the `Graph` interface
- Four concrete classes represent the combinations of directedness and underlying implementation: `GraphMatrixDirected`, `GraphMatrixUndirected`, `GraphListDirected`, `GraphListUndirected`
- Hash tables allow efficient mapping from vertex labels to `Vertex` objects