44 Graphs	*(4 activities)*

44.1 Work with the abstract concept of a graph.(13 steps)

44.1.1 (Display page) Overview of graphs

Overview of graphs

This week's activities involve studying graphs. Graphs are a representation used to show relationships between elements of a collection. First, a few definitions. An element in a graph is called a *vertex*. Vertices typically represent the objects in our graph, namely the things that have the relationships such as people, places, or things. A connection between two vertices is called an *edge*. Basically, an edge represents some kind of relationship between two vertices in a given graph. Quite a few examples of graphs exist in the everyday world:

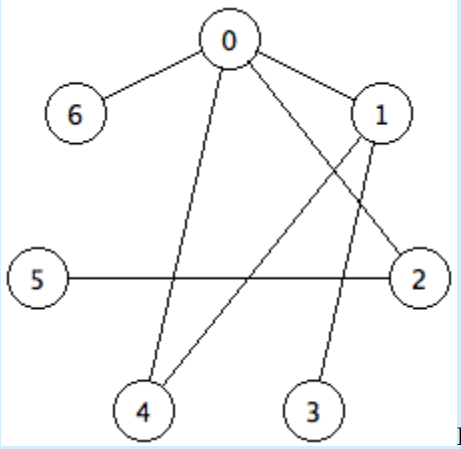
- **Road maps** are a great example of graphs. Each city is a vertex, and the edges that connect these cities are the roads and freeways. An abstract example of what such a graph would look like can be found here. For a more local example, each building on the Berkeley campus can be thought of as a vertex, and the paths that connect those buildings would be the edges.
- **Online social networks**, such as My Space, orkut, and Friendster are a recent phenomenon that depend on graphs. Each person that participates is a vertex, and an edge is established when two people claim to be friends or associates of each other.
- For a more technical example, a **computer network** is also a graph. In this case, computers and other network machinery (like routers and switches) are the vertices, and the edges are the network cables. Or for a wireless network, an edge is an established connection between a single computer and a single wireless router.

In more formal mathematical notation, a vertex is written as a variable, such as v_0, v_1, v_2, etc. An edge is written as a pair of vertices, such as (v_0,v_1), or (v_2,v_0).

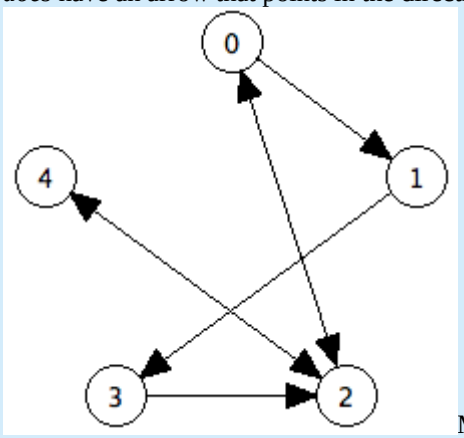
44.1.2 (Display page) Directed vs. undirected graphs

Directed vs. undirected graphs

Once again, an element in a graph is called a *vertex*, and a connection between two vertices is called an *edge*. If all edges in a graph are showing a relationship between two vertices that works in either direction, then it is called an *undirected* graph. A picture of an undirected graph looks like this:



But not all edges in graphs are the same. Sometimes the relationships between two vertices sometimes only go in one direction. Such a relationship is called a *directed* graph. An example of this could be a city map, where the edges are sometimes one-way streets between locations. A two-way street would actually have to be represented as *two* edges, one of them going from location A to location B, and the other from location B to location A. In terms of a visual representation of a graph, an undirected graph does not have arrows on its edges (because the edge connects the vertices in both

directions), whereas each edge in a directed graph *does* have an arrow that points in the direction the edge is going. An example directed graph appears below. 

More formally, an edge from a vertex v_0 to a vertex v_1 is printed as the pair (v_0,v_1). In a directed graph, the pair is ordered; thus (v_0,v_1) might be an edge, but (v_1,v_0) might not. In an undirected graph, the pair isn't ordered, so the edge (v_0,v_1) is the same as the edge (v_1,v_0).

44.1.3 (Brainstorm) More situations that can be represented by graphs

Give another situation that can be modeled with graphs. Describe what the vertices are, and define the conditions under which two vertices are connected by an edge. Can your example be represented as an undirected graph, or does it have to be a directed graph?

44.1.4 (Brainstorm) Graphs vs. trees

An Amoeba family tree may be interpreted as a graph. What are the graph's vertices, and what defines when two vertices v and w are adjacent?

44.1.5 (Display page) A few more definitions

A few more definitions

Now that we have the basics of what a graph is, here are a few more definitions that apply to graphs that will come in handy while discussing graphs. When an edge (v_0,v_1) exists between vertices v_0 and v_1, then v_1 is said to be *adjacent to* or *connected to* v_0, and a *neighbor* of v_0. Basically, "adjacent" is the official term for saying that a vertex is pointed to by another vertex, but we also refer to the "set of neighbors" or the "neighborhood" of a vertex. In addition, in the above example, the vertices v_0 and v_1 are said to be *incident* with edge (v_0,v_1). So when an edge is touching a vertex, that vertex is incident with that edge. A *path* between two vertices is a sequence of edges that can be followed from one vertex to another. A special kind of path is called a *cycle*, which is a path that ends at the same vertex where it originally started. The existence of cycles are one reason why graphs are more complicated than trees. Finally, a graph is *connected* when for every pair of vertices u and v, there is a path from u to v.

44.1.6 (Self Test) Edge count vs. vertex count

Suppose that G is an *directed* graph with N vertices. What's the maximum number of edges that G can have? Assume that a vertex cannot have an edge pointing to itself, and that for each vertex u and vertex v, there is at most one edge (u,v).

Now suppose the same graph G in the above question is an *undirected* graph. Again assume that no vertex is adjacent to itself, and at most one edge connects any pair of vertices. What's the maximum number of edges that G can have?

What's the *minimum* number of edges that a connected undirected graph with N vertices can have?

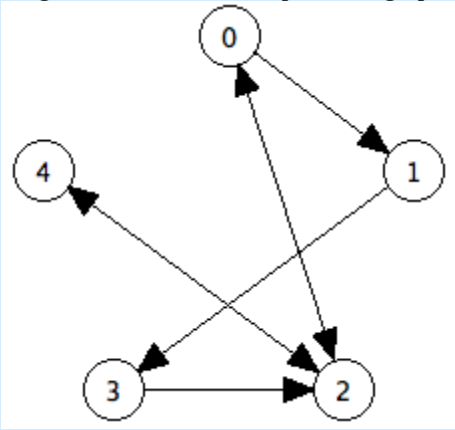
44.1.7 (Display page) Data structures for graphs

Data structures for graphs

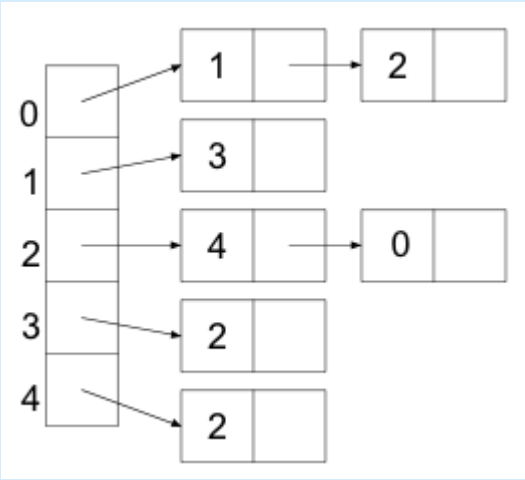
Now that we know how to draw a graph on paper and understand the basic concepts and definitions, we can consider how a graph should be represented inside of a computer. We need quick answers for the following questions about a graph:

- Are vertices v and w adjacent?
- Is vertex v incident to edge e?
- What vertices are adjacent to v?
- What edges are incident to v?

Imagine that we want to represent a graph that looks like this:



Probably the easiest-to-understand representation is called an *adjacency list*. In such a data structure, an array is created that has the same size as the number of vertices in the graph. Then, each location in the array points to a list that contains indexes to other vertices. It looks like this:



Another way to represent the edges in a graph is to use an *adjacency matrix*. In this data structure, it works the same way as an adjacency list, but each linked list is replaced with another array of the same length as the number of vertices in the graph. This matrix just contains boolean values, true when there is an edge between the two given vertices, false when no edge exists. Thus, each vertex has a row and a column in the matrix, and the value in that table says true or false whether or not that edge exists. Such a representation looks like this:

```
      0     1     2     3     4
0  false  true  true false false
1  false false false  true false
2   true false false false  true
3  false false  true false false
```

```
    4  false false  true false false
```

44.1.8 (Display page) Visualize graph data structures

Here is a useful applet we've created to help visualize a graph, given an adjacency matrix or adjacency lists. For the matrix, a checkbox that is turned off represents a false value in the matrix, and a checkbox turned on represents a true value. In the lists, type the numbers of the vertices, separated by spaces. Experiment with it to get a better understanding of what a corresponding graph looks like with a given adjacency matrix or a given set of adjacency lists. If you want, you can open up the applet in a separate window by clicking here.

44.1.9 (Self Test) Time estimates for accessing graph information

Using an adjacency matrix, how long in the worst case does it take to determine if vertex v is adjacent to vertex w? (Assume vertices are represented by integers.)

Using an array of adjacency lists, how long in the worst case does it take to determine if vertex v is adjacent to vertex w? (Assume vertices are represented by integers.)

44.1.10 (Brainstorm) Memory use

Suppose we are representing a graph with N vertices and E edges. The memory required to store an adjacency matrix is N^2 times the memory required to store a boolean value. How much memory is required to represent the graph as an array of adjacency lists? Assume that references and integers each use 1 unit of memory. Briefly explain your answer.

44.1.11 (Brainstorm) A time estimate for finding a common neighbor

A common neighbor c between two vertices v and w is a vertex that has edges (v,c) and (w,c). In other words, both v and w both have an edge that points to c. In a graph with N vertices and E edges that is implemented using an adjacency matrix, how many comparisons are necessary in the worst case to determine if vertices v and w have a common neighbor? (Assume that v and w are represented as integers.) Briefly explain your answer.

44.1.12 (Brainstorm) Another estimate for neighbors in common

A CS 61B student claims that, in a graph with N vertices and E edges that's implemented with an array of adjacency lists, the worst-case time to see if vertices v and w have a common neighbor is proportional to N^2. (Vertices are not in any particular order in an adjacency list.) This estimate is insufficiently specific. Explain why, and give a more specific estimate.

44.1.13 (Self Test) Adjacency lists vs. adjacency matrix

Imagine you have a graph with lots of edges. Another way to put that is that each vertex almost always is adjacent to most of the other vertices in the graph. Which data structure is best for representing such a graph?

Imagine you have a graph with very few edges. Some vertices may not even be connected to other vertices, and most vertices are adjacent to only one or two other vertices. (Another term for this kind of graph is a *sparse* graph.) Which data structure is best for representing such a graph?

44.2 Work with programs that process graphs.*(4 steps)*

44.2.1 (Display page) Graph traversal

Graph traversal

Earlier in the course, we used the general traversal algorithm to process all elements of a tree:

```
    Stack fringe = new Stack ();
    fringe.push (myRoot);
    while (!fringe.isEmpty()) {
        // select a tree node from fringe
        TreeNode node = fringe.pop();

        // process the node
```

```
        [do whatever processing operation here...]

        // add node's children to fringe
        fringe.push(node.myRight);
        fringe.push(node.myLeft);
          // Note: If this was a tree with more than
          //       two children, we'd push ALL of the
          //       children onto the stack.
    }
```

The code just given returns tree values in depth-first order. If we wanted a breadth-first traversal of a tree, we'd replace the Stack with a queue (the LinkedList class in java.util would work well here). Analogous code to process every vertex in a connected graph is

```
    Stack stack = new Stack();
    fringe.push (some vertex);
    while (!fringe.isEmpty()) {
        // select a vertex from fringe
        // process the vertex
        // add the vertex's neighbors to fringe
    }
```

This doesn't quite work, however. Applied to a cycle, for example, the code loops infinitely. The fix is to keep track of vertices that we've visited already, in order not to process them twice. Here is pseudocode:

```
    Stack fringe = new Stack ();
    fringe.add (some vertex);
    while (!fringe.isEmpty()) {
        // select a vertex from fringe
        // process the vertex
        // add the vertex's unvisited neighbors to fringe
    }
```

As with tree traversal, we can visit vertices in depth-first or breadth-first order merely by choosing a stack or a queue to represent the fringe. Typically though, because graphs are usually interconnected, the ordering of vertices can be scrambled up quite a bit. Thus, we don't worry too much about using a depth-first or a breadth-first traversal. Instead, in the next lab session we will see applications that use a *priority queue* to implement *best-first* traversal.

44.2.2 (Display page) Using an iterator to find a path, part 1

Using an iterator to find a path, part 1

The file ~cs61b/labcode/lab22/Graph.java contains an implementation of a graph as an array of adjacency lists, and includes a method for depth-first iteration. Add a method pathExists (int v,int w) that returns whether or not any path exists that goes from vertex v to vertex w. Remember that a path is any set of edges that exist which you can follow such that you can travel from one vertex to another. Your method should call visitAll. For an example of an undirected graph this should work on, try testing it with the following two graphs:

```
    addUndirectedEdge(0,2);
    addUndirectedEdge(0,3);
    addUndirectedEdge(1,4);
    addUndirectedEdge(1,5);
    addUndirectedEdge(2,3);
    addUndirectedEdge(2,6);
    addUndirectedEdge(4,5);
    addEdge(0,1);
    addEdge(1,2);
    addEdge(2,0);
    addEdge(2,3);
    addEdge(4,2);
```

44.2.3 (Display page) Using an iterator to find a path, part 2

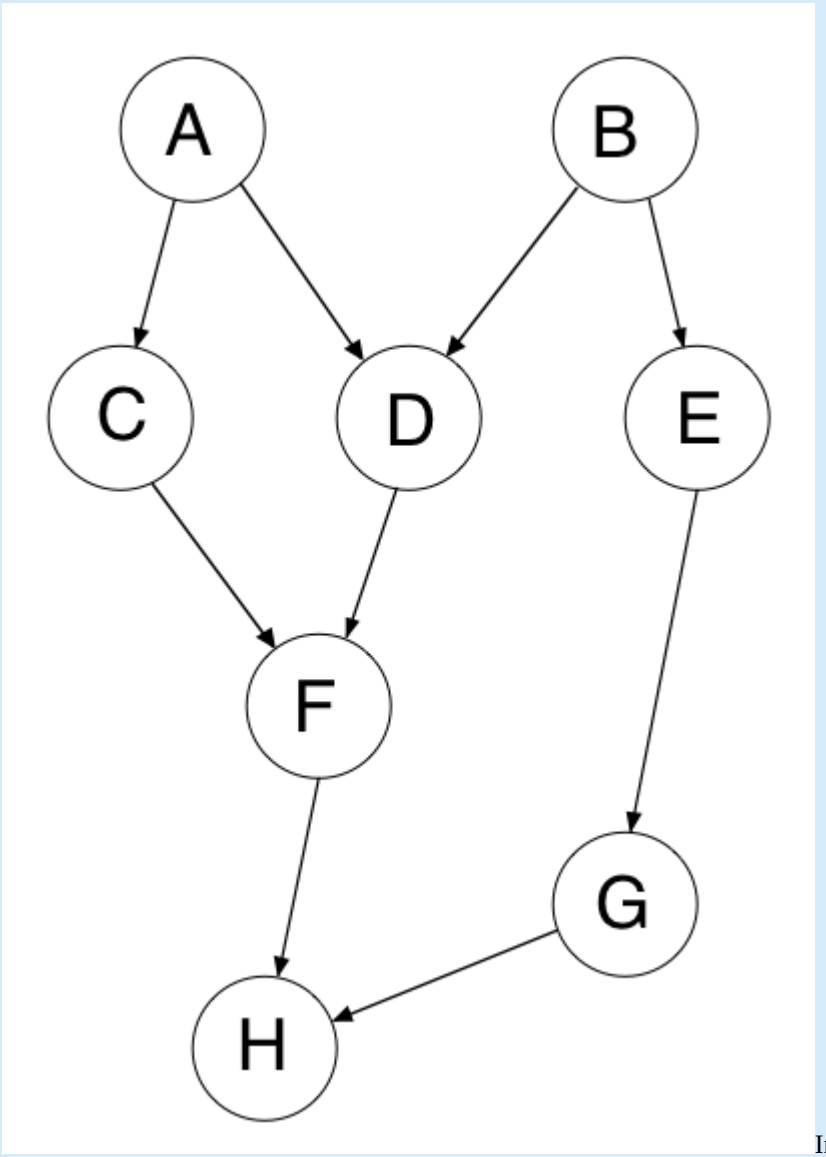
Using an iterator to find a path, part 2

Now you will actually find a path from one vertex to another if it exists. Write a method named path that, given two ints that represent a start vertex and a finish vertex, returns an ArrayList of Integers that represent vertices that lie on the path from the start to the finish. If no such path exists, you should return an empty ArrayList. Pattern your method on visitAll, with the following differences. First, add code to stop calling next when you encounter the finish vertex. Then, trace back from the finish vertex to the start, by first finding a visited vertex u for which (u, finish) is an edge, then a vertex v visited earlier than u for which (v, u) is an edge, and so on, finally finding a vertex w for which (start, w) is an edge. Collecting all these vertices in the correct sequence produces the desired path.

44.2.4 (Display page) Topological sort

Topological sort

A *topological sort* of a directed graph is a list of the vertices in such an order that if there is a directed path from vertex v to vertex w, then v precedes w in the list. (The graph must be acyclic in order for this to work. Directed acyclic graphs are common enough to be referred to by their acronym: DAGs.) Here is an example of a DAG:



In the above DAG, a few of the

possible topological sorts could be:

```
A B C D E F G H
A C B E G D F H
B E G A D C F H
```

Notice that the topological sort for the above DAG has to start with either A or B, and must end with H
Another way to think about it is that the vertices in a DAG represent a bunch of tasks you need to complete on a to-do list. Some of those tasks cannot be completed until others are done. For example, when getting dressed in the morning, you may need to put on shoes and socks, but you can't just do them in any order. The socks must be put on before the shoes. The edges in the DAG represent dependencies between the tasks. In this example above, that would mean that task A must be completed before tasks C and D, task B must be completed before tasks D and E, E before G, C and D before F, and finally F and G before H. The topological sort algorithm uses an array named currentInDegree with one element per vertex. currentInDegree[v] is initialized with the in-degree of each vertex v; as each vertex is added to the result list, the currentInDegree values of its neighbors are reduced by 1. The fringe is initialized with all the vertices whose in-degree is 0. Once the currentInDegree of a vertex becomes 0, that means that all of the vertices that had edges pointing to it have already been processed. Thus, a vertex is added to the fringe when its currentInDegree becomes 0. Your task is to fill in the blanks in the TopologicalIterator class so that it successively returns vertices in topological order as described above. The TopologicalIterator class will resemble the VertexIterator class, except that it will use a currentInDegree array as described above, and instead of pushing unvisited vertices on the stack, it will push only vertices whose in-degree is 0.

45 Quiz 22 *(1 activity)*

45.1 (Quiz) Review heap construction and structure.*(1 step)*

45.1.1 (Student Assessment) Quiz questions

1. Suppose that the values 3, 1, 4, 5, 2, and 9 are inserted sequentially into an initially empty max heap. What does the underlying array contain after the insertions?

2. List the contents of another array that represents a max heap with the elements 3, 1, 4, 5, 2, and 9, in which the last element of the array—the "bottom" of the heap—is as large as possible.