

Graphs

You are familiar with data structures such as arrays and lists (linear), and trees (non-linear). We next consider a more general non-linear data structure, known as a graph. Like the previous data structures, a graph has a set nodes. Each node has a reference to other nodes. In the context of graphs, a reference from one node to another is called an edge. In a (rooted) tree, we saw that the references were either to a child or parent node. In a general graph, there is no notion of child and parent. Every node can potentially reference every other node.

Graphs have been studied and used for many years, and some of the main basic results go back a few hundred years. Mathematically, a graph consists of a set V called “vertices” and a set of edges $E \subseteq V \times V$, that is, the edges E in a graph is some set of ordered pairs of vertices.

For certain graphs, we do not distinguish edges (v, w) and (w, v) . These graphs are called *undirected*. If we do care about the order of vertices in an edge, then we say that the graph is *directed*.

Below is an example of two graphs. On the left, there are double arrows on each edges. You can think of the graph having two edges at each node, one in each direction. On the right, some of the edges have only a single arrow. In that case, there is only one edge there. Sometimes we draw the undirected graph on the left without the arrowheads. (See slides.)



Examples of graphs include transportation networks. For example, V might be a set of airports and E might be direct flights between airports. In computer system, V might be a set of computers and E might be a direct communication link between them. Or V might be a set of html documents on a web site and E might be defined by URLs (links) between documents. You can think of each of the above as directed graphs, since it makes sense to distinguish the order. (An example of an undirected graph would be a set of tunnels or roads that allow two-way traffic.)

Data structures

There are two very common data structures for graphs: adjacency lists and adjacency matrices.

Adjacency List

For each vertex v , consider a list of vertices w such that $(v, w) \in E$. For example, here are the adjacency lists for the examples above.

- | | |
|---------|-------|
| a - c,d | a - c |
| b - e,f | b - f |

```

c - a,d,f      c - f
d - a,c        d - a,c
e - b,f        e - b,f
f - b,c,e      f - b,e
g - h          g - h
h - g          h -

```

In the slides, I discussed how one might represent adjacency lists as part of a graph class in Java. Take the simple case in which the vertices are characters. Then one could define a node:

```

class GNode{
    char          vertex;
    LinkedList<GNode> adjList;
}

```

See the example in the slides.

A more general scheme allows the vertices to be objects of some generic class *V*. One could then define

```

class GNode<V>{
    V          vertex;
    LinkedList<GNode> adjList;
}

```

The next question is how to organize these *GNode* objects into a graph. A simple way to do this is just to have a list of *GNodes*.

```

class Graph<V>{
    LinkedList<GNode<V>> vertices;
    :
}

```

One builds the graph by building each *GNode* and adding it to the list of *GNodes*.

A more sophisticated graph class might use a hash table:

```

class Graph<K,V>{
    HashMap<K, GNode<V>> graph;
}

```

where the keys might be the vertices *V* themselves. Or the keys might a field within the vertices *V*. For example, in a graph of airports and flight connections, the key might be the name of an airport.

See the lecture slides for a simplified example in which the vertices (and keys) are characters, and the values are *GNodes*.

Adjacency Matrix

A totally different way to represent the edges in a graph is to use an *adjacency matrix* which is a $|V| \times |V|$ array of booleans, where $|V|$ is the number of elements in set V i.e. the number of vertices. The value 1 at entry $(v1, v2)$ in the array indicates that $(v1, v2)$ is an edge, that is, $(v1, v2) \in E$, and the value 0 indicates that $(v1, v2) \notin E$.

Adjacency matrices for the two graphs are shown below. Note that the adjacency matrix on the left is symmetric about the diagonal.

	abcdefgh		abcdefgh
a	00110000	a	00100000
b	00001100	b	00000100
c	10010100	c	00000100
d	10100000	d	10100000
e	01000100	e	01000100
f	01101000	f	01001000
g	00000001	g	00000001
h	00000010	h	00000000

Adjacency list versus adjacency matrix

Space considerations

If the number of entries in an adjacency matrix is n^2 , and if n is large, then it is not practical to build such a matrix. For example, if the graph represents web pages on the world wide web, then $n \approx 20,000,000,000$. See <http://www.worldwidewebsite.com>. Most of the web pages point to a small number of other web pages, and so it would be much more space efficient to use an adjacency list.

Another consideration is that the adjacency matrix requires (strictly speaking) only one bit per edge, whereas an adjacency list requires much more, namely it requires a node in a linked list, plus other linked list overhead requirements.

Time considerations

Suppose you wish to know which edges a given vertex v is connected to, namely you want to know for which w 's there is an edge $(v, w) \in E$. Using an adjacency matrix, it takes time $|V|$ time steps to determine this, namely you need to examine all elements in a row of the adjacency matrix and see which ones have the value 1. If you use an adjacency list, then the time it takes is proportional to the size of the adjacency list of v .

When would be advantageous to use an adjacency matrix? If you want to know if a graph contains a particular edge (v, w) , then an adjacency matrix allows you to determine this in one step (an array lookup), whereas an adjacency list requires you to search through the list and this on average takes time proportional to the number of edges in the list.

Terminology

Finally, here is some basic graph terminology that you should become familiar with, and that is heavily used in discussing properties of graphs. The ideas are intuitive enough.

- if we have an edge $e = (v_1, v_2)$ then v_1 and v_2 are *adjacent vertices*, the edge e is *incident* on these vertices.
- *outgoing edges from v* - the set of edges of the form $(v, w) \in E$ where $w \in V$
- *incoming edges to v* - the set of edges of the form $(w, v) \in E$ where $w \in V$
- *in-degree of v* - the number incoming edges to v
- *out-degree of v* - the number outgoing edges from v
- *path* - a sequence of vertices (v_1, \dots, v_m) where $(v_i, v_{i+1}) \in E$ for each i .

Note that the definition of path is similar too the definition of path that we saw for trees. See lecture 17 page 2. That definition of a path in a tree only applied to trees for which the edges are from parent to child, though. To generalize that definition to (say) trees that have (child,parent) edges also, one would use the definition that I just gave for graphs.

- *cycle* - a path such that the first vertex is the same as the last vertex

If there were an edge (v, v) , then this would be considered as a cycle. Such edges are certainly allowed in graphs and indeed are quite common.