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 linked list, there are references from one to the “next” (and/or “previous”) node. In a (rooted) tree, there are references were to children nodes (or siblings or parent nodes). In a general graph, there is no unique notion of “next” or “prev” as in a list, and there is no unique notion of child or parent. Every node can potentially reference every other node. We will discuss data structures for graphs below.

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.

Below is an example of a graph. Some of the arrows are in a single direction and some are in both directions. The latter are a convenient notation that there are two edges, namely one in each direction.

![Graph Example](image)

Examples of graphs include transportation networks. For example, $V$ might be a set of airports and $E$ might be direct flights between airports. Or $V$ might be a set of html documents and $E$ might be defined by URLs (links) between documents.

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 is the adjacency lists for the example above.

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

```java
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

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

ASIDE: The example in the slides is sort of a compromise between these. For V, I use Character. But the sketches suggest that the characters are inside the node, i.e. a field in the GNode class.

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.

```java
class Graph<V>{
    LinkedList<GNode<V>> gNodeList;
    // methods
}
```

One would build the graph by constructing each GNode and adding it to the list of GNode's.

A more sophisticated graph class might use a hash table:

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

The keys might be the vertices V themselves. Or the keys might a field within the vertices V. Another example: in a graph of airports and flight connections, the key might be the three letter name of an airport e.g. YUL or LAX.

**Adjacency Matrix**

A different data structure for representing 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 $(v_1, v_2)$ in the array indicates that $(v_1, v_2)$ is an edge, that is, $(v_1, v_2) \in E$, and the value 0 indicates that $(v_1, v_2) \notin E$. 


The adjacency matrix for the graph from earlier is shown below.

\[
\begin{array}{ccccccccc}
& a & b & c & d & e & f & g & h \\
\hline
a & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
b & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
d & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
e & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
f & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
g & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Note that in this example, the diagonals are all 0’s, meaning that there are no edges of the form \((v, v)\). But graphs can have such edges. An example is given in the slides.

**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.worldwidewebsize.com](http://www.worldwidewebsize.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 is 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.

- **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 \((v_1, \ldots, v_m)\) where \((v_i, v_{i+1}) \in E\) for each \(i\).

  Note that the definition of path is similar to the definition of path that we saw for trees.

• **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.

At the end of the lecture, I mentioned a few definitions and important graph problems that you will see in subsequent courses.