Graphs

We saw most of the following definitions and notation in CISC-203. This information is included here for completeness.

Graphs can be viewed as generalizations of trees (actually, trees are a subclass of graphs). A graph $G$ consists of two sets: a set $V$ of vertices, and a set $E$ of edges, where each edge consists of a pair of vertices in $V$.

Terminology:

We use $n$ to denote $|V|$

We use $m$ to denote $|E|$. Note that if we disallow duplicate edges, $m \in O(n^2)$

Two vertices $x$ and $y$ are adjacent if $(x, y)$ is in $E$

If $x$ and $y$ are adjacent, we call them neighbours.

A loop is an edge where both vertices are the same, such as $(x, x)$. Loops are useful in a few situations but for the most part we will focus on graphs that have no loops.

Sometimes it is reasonable to have multiple edges between a pair of vertices. These are called multiple edges (surprise!) - usually we prohibit this for the sake of simplicity.

The degree of a vertex is the number of times the vertex appears in $E$. If loops are prohibited (as is usual), the degree of a vertex is the number of edges that touch the vertex. We use $d(v)$ to denote the degree of $v$.

A path is a sequence of vertices and edges such that each edge joins the previous vertex to the next vertex.

A cycle is a path where the first and last vertices are the same.

A simple path is a path in which all the vertices and edges are distinct.

A simple cycle is a cycle in which all the vertices and edges are distinct (with the exception of the first vertex, which is also the last vertex).
Unless otherwise noted, we will always limit our discussions to graphs with no multiple edges and no loops
- simple paths
- simple cycles

Because we are (almost always) going to talk about simple paths and simple cycles, we will just call them "paths" and "cycles"

A **directed edge** is an edge with one vertex identified as the head and the other as the tail. The edge is directed from the tail to the head. This is usually signified in drawings of the graph by placing an arrow-head on the edge, pointing into the head vertex. **Edges are undirected unless otherwise specified.**

Edges may be weighted. The weight of an edge $e$ is usually denoted by $w(e)$. If the edge is identified by its end vertices, such as $e = (x,y)$, we will sometimes indicate its weight by $w(x,y)$. Edge weights are sometimes referred to as costs.

In general, graphs are used to represent bilateral (ie. undirected) or unilateral (ie. directed) pairwise relationships between discrete entities. In a communication network, the edges could represent the existence of a wired link between two objects in the network. In a social network, the edges could represent "friend" links.

A graph $G$ is **connected** if for each pair of vertices $x$ and $y$, there is a path in $G$ that starts at $x$ and ends at $y$. Connectivity is a fundamentally important property of graphs and we will look at methods of determining if a graph is connected.

Before addressing these algorithms we must first discuss appropriate data structures for representing graphs. There are two well-known approaches: **adjacency matrices** and **adjacency lists**.
Adjacency Matrix Representation

An adjacency matrix for a graph G is an n*n matrix A where
\[ A[i,j] = 1 \text{ if vertex } i \text{ is adjacent to vertex } j \]
\[ A[i,j] = 0 \text{ if vertex } i \text{ is not adjacent to vertex } j \]

If the edges have weights, then
\[ A[i,j] = w(i,j) \text{ if vertex } i \text{ is adjacent to vertex } j \]
\[ A[i,j] = 0 \text{ if vertex } i \text{ is not adjacent to vertex } j \]

Note that the size of A is \( n^2 \), regardless of the number of edges in E

If the graph is undirected, \( A[i,j] = A[j,i] \). The lower left diagonal half of the matrix is the mirror image of the upper right diagonal half. In practice we often don’t bother filling in the lower left diagonal half because all of its information is already in the upper right diagonal half.

However if the edges are directed, we set \( A[i,j] = 1 \) (or \( w(i,j) \)) if there is a directed edge from vertex i to vertex j, and \( A[i,j] = 0 \) otherwise. In the adjacency matrix for a directed graph, it is not necessarily true that \( A[i,j] = A[j,i] \)

Here is a small graph:
and here is its adjacency matrix. I have left out all the 0's.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>3</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the rows and columns could be re-ordered – as long as we know which row and column correspond to each vertex, we can still build the matrix.
Adjacency Lists Representation

The idea behind the adjacency list representation of a graph is to store a list of all the neighbours of each vertex. We use a 1-dimensional array for the vertices. To the array element for vertex k, we attach a linked list of the neighbours of that vertex - the list of neighbours is in no particular order. If the graph is undirected, each edge will create an entry in two of the adjacency lists. If the edges are directed, an edge from vertex i to vertex j would be represented by including j in the adjacency list for vertex i.

If the edges are weighted, the weights of the edges can be stored as auxiliary data in the adjacency lists. Here is an adjacency list representation of the graph shown above:
Now that we have two different ways of implementing the graph ADT, we can compare them. Consider a few basic operations that we might want to perform on a graph:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a new vertex</td>
<td>Add a new row and column to the matrix</td>
<td>Add a new element to the 1-dimensional vertex vector</td>
</tr>
<tr>
<td>Remove a vertex</td>
<td>Remove a row and column from the matrix … or just flag it as “deleted”</td>
<td>Remove one element from the vertex vector … or just flag it as “deleted”</td>
</tr>
<tr>
<td>Add an edge between two vertices</td>
<td>Change 2 elements of the matrix</td>
<td>Append the new edge info to two of the adjacency lists</td>
</tr>
<tr>
<td>Remove an edge between two vertices</td>
<td>Change 2 elements of the matrix</td>
<td>Search the adjacency lists of the two vertices and remove the edge info from both of them</td>
</tr>
<tr>
<td>Query if two vertices are adjacent</td>
<td>Examine 1 element of the matrix</td>
<td>Search the adjacency list of one of the two vertices</td>
</tr>
<tr>
<td>List all neighbours of a particular vertex</td>
<td>Examine each element of the matrix row for that vertex</td>
<td>Traverse the adjacency list of that vertex</td>
</tr>
</tbody>
</table>

Looking at this list we can see that some operations are faster for Adjacency Matrices and some are faster for Adjacency lists. Our choice of which representation to use for a particular algorithm will depend on which operations are used in the algorithm.

The last operation in the table (list all neighbours of a particular vertex) is an important one. In practical applications, it is often the case that there is a fixed upper bound on the number of neighbours each vertex can have. If that is the case then traversing an adjacency list takes constant time, whereas examining each element of a row in the Adjacency Matrix will take \( O(n) \) time. We will see that even when there is no fixed upper limit on the number of neighbours a vertex can have, the Adjacency Lists representation is almost always preferable to the Adjacency Matrix representation if we need to perform this “look at all neighbours of a vertex” operation.
One important consideration when choosing between these two structures is the number of edges in the graph, so we should consider upper and lower bounds on $m$.

If every vertex is adjacent to all other vertices, the number of edges is $n*(n-1)/2$ (remember we do not allow loops or duplicate edges). Thus we know $m$ is in $O(n^2)$

Can we put a lower bound on $m$? Well, clearly $m$ can equal 0 ... but it’s more interesting if we require that the graph is connected ... so the question becomes "What is the smallest number of edges in a connected graph on $n$ vertices?"

**Theorem:** Let $G$ be a connected graph on $n$ vertices. Then $m \geq n-1$

**Proof:** We proved this theorem in CISC-203

Thus if $G$ is a connected graph (without loops or multiple edges), $n-1 \leq m \leq n(n-1)/2$

As mentioned above, in many practical applications of graphs the degree of each vertex is bounded by a constant. (For example, we may have a restriction that each device in a network has $\leq 4$ direct connections to other devices.) If this is true, then $m$ is in $O(n)$. We call such a graph **sparse**.

If a graph is sparse, many of the entries in its adjacency matrix will be 0 ... for such graphs, adjacency lists are often used since the space requirement is proportional to the number of edges.