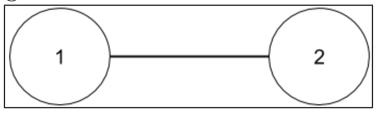
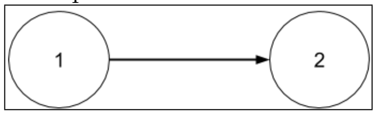
Basics and Representations of Graphs

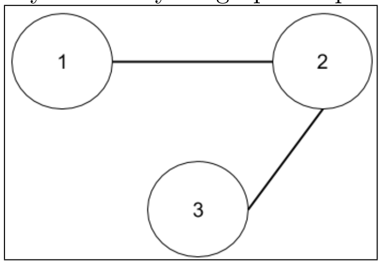
1 Basics of Graphs

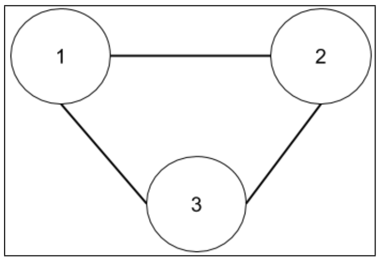
Graphs, in the most simple way, are a set of nodes with edges between them. There are a few types of graphs, one way of categorizing is **directed** and **undirected**. An undirected graph is one where the edges is "2 way" which means if there is an edge between vertex A and B you can go from A to B and B to A. A directed graph is one where the edge only goes "1 way" so an edge between A and B will either allow you to go from A to B or B to A. Below is are examples of undirected and directed graphs respectively.





Another two categories of graphs are ***cyclic*** and ***acyclic***. A graph is cyclic if there is a cycle. A cycle exists if at any point there is a path where the start and end vertex are the same. Below are examples of acyclic and cyclic graphs respectively.

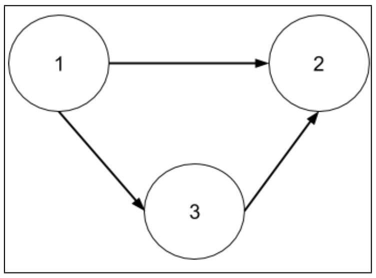


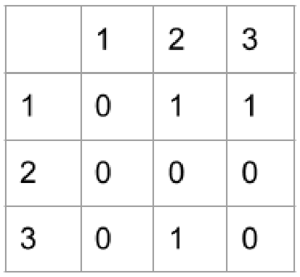


Two nodes or vertices are ***adjacent*** to each other if they have an edge between them. Sometimes, edges can have some ***weight*** on them which represent some sort of cost that it takes to cross the path. Vertices are ***connected*** if there is an edge between them; graphs are **connected** if all vertices are connected.

2 Representations of Graphs

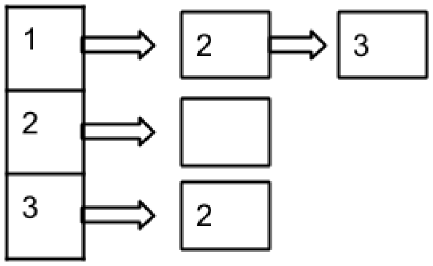
There are a 2 main ways to represent graphs: ***Adjacency Matrices*** and ***Adjacency Lists***.
An Adjacency Matrix is represented with a 2d array that is N x N. The x and y axes of this 2d array will be the vertices and the value at each "block" at row r and column c will be 0 or 1 (or some other true or false distinction value) if the vertices r and c are connected. Below is an example for how an Adjacency Matrix would look for a graph.





	1	2	3
1	0	1	1
2	0	0	0
3	0	1	0

The other representation of graphs, which tends to be used more frequently, is the Adjacency List. An Adjacency List is similar to a HashMap in that they both are represented with an array of "buckets". Below is a an example of how an Adjacency List would look for the same graph as above.



To compare the above two representations, we will take a look at the Graph API and see what functions what we need to be able to account for.

```
void addEdge(int v, int w) //adds an edge between the vertices v and w
Iterable<Integer> adj(int v) // returns an iterable that has all the certices adjacent to
int V(): //number of vertices in the graph
int E(): // number of edges in the graph
```

To add an edge in the Adjacency Matrix, we would simply just go to the proper index in the 2-d array and replace the false value with a true one. For the Adjacency List, you would go to the proper index and add an element to the end of the bucket. As a result, this takes $\Theta(1)$ time.
Finding the vertices adjacent to a certain vertex in an adjacency matrix would require you to go to the

proper row and go through each one of its columns, adding the vertex if there is a true value, as a result, this takes $\Theta(V)$ time since you need to go through V buckets. In an Adjacency List, you simply go return the corresponding bucket since the bucket contains all the adjacent vertices, as a result, this takes $\Theta(1)$ time.
Finally, we will look at the space used. For the Adjacency Matrix, regardless of how big it is, there will

always take $\Theta(V^2)$ space because the 2d array is V x V. The Adjacency List will take up $\Theta(E + V)$ space (whichever term is bigger).
So which representation should be used? Well it really depends on your graph. If the graph is very sparse,

it does not make sense to use $\Theta(V^2)$ space, so an Adjacency Matrix would be the superior choice. However, to see if two vertices are connected, it can be faster to use and Adjacency Matrix. A specific case that this may occur is when the graph is fully connected, that is the edges amount is equal to V^2. This means that a bucket would be V length long and you may need to iterate over all of it in an Adjacency List representation; however, and Adjacency Matrix would allow you to plug in indices and see connectivity in constant time.