Graphs

A graph, in the most basic sense, is a collection of vertices and edges. Graphs can be thought of as an ADT if you think of them as the collection of operations like:

- `addEdge(Vertex source, Vertex dest)`
- `isVertex(Vertex v)`
- `getNeighbors(Vertex v)`
- `findPath(Vertex source, Vertex dest)`
- `findShortestCostPath(Vertex source, Vertex dest)`
- `(and many, many others)`

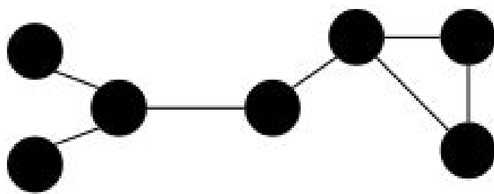
However, it is usually more useful to think of graphs in a more general sense, since there are so many variations of different types of graphs. In class we talked about all of the following terminology.

Vertices and Edges

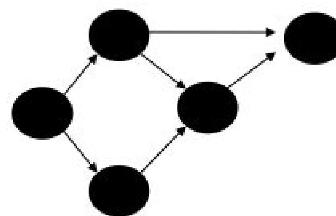
Vertices (or nodes) are the entities in your data that may have relationships with each other. The edges are how those entities are related, usually representated as the ordered pair (vertexSource, vertexDestination). For example, friends in a social network: the entities are the people in the network, the edges represent the friendship. Those edges could be directed, undirected, weighted, unweighted. The graph could have cycles, no cycles, be connected, fully connected, strongly/weakly connected, be dense or sparse, have self edges, etc. A **self edge (or self loop)** is when a vertex ‘A’ has an edge to itself ‘(A, A)’. The degree of a vertex (or in-degree and out-degree for directed graphs) is how many edges are connected to that vertex.

Directed and Undirected Graphs

This has to do with whether the edges have a direction to them. If they have a direction, then the graph is directed. Examples:



Undirected Graph

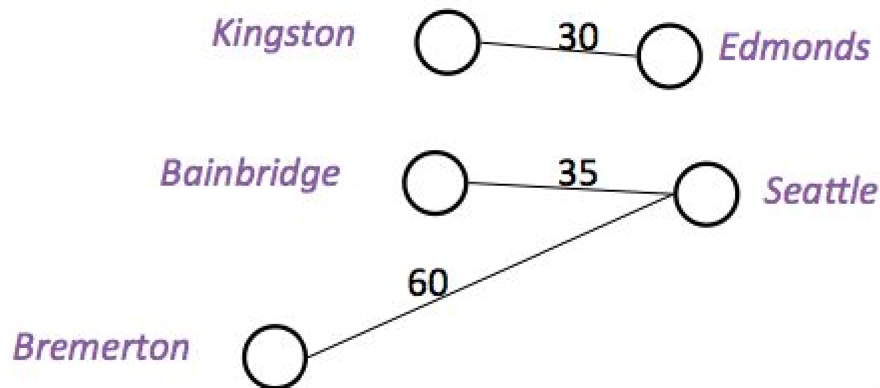


Directed Graph

If you are traversing an undirected graph, you can go along an edge in either direction. If you are traversing a directed graph, each edge has a source and a destination and you can only traverse along the edge in that direction. In an undirected graph, the edge (a,b) implies that edge (b,a) is also in the graph, so you can traverse in either direction.

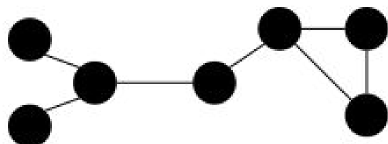
Weighted and Unweighted Graphs

This has to do with whether the edges have weights (or costs) to them. You can use anything as a weight, though typically we think of weights as numeric costs. We use weights when looking at paths. For example, a graph representing locations might use distances as edge weights. Then taking a path from Seattle -> Dallas -> Chicago will cost the sum of the edges along that path. This is useful when comparing paths. Example of a weighted graph:

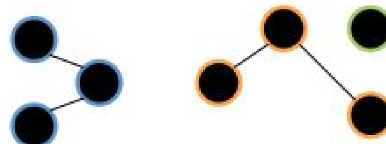


Connected, Fully Connected, Strongly/Weakly Connected

The definition of connectedness is slightly different for directed vs undirected graphs. For undirected graphs, it is considered connected if you can get from any vertex a to any other vertex b. (i.e. the graph is all in one cluster, there are not disjoint sets of nodes in your graph).

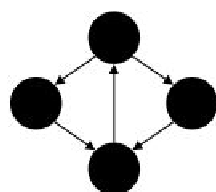


Connected graph

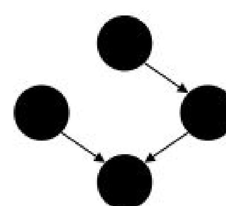


Disconnected graph

For directed graphs, we introduce the idea of strongly and weakly connected. The definition of strongly connected is the same as the undirected graph's definition of connection. If you can get from every vertex a to every other vertex b, the graph is strongly connected. The graph is weakly connected if, even if you can't get from every vertex a to every other vertex b, you could if you ignored the direction of the edges. (i.e. treat it like an undirected graph, if there are not disjoint sets of nodes, then it is weakly connected)



Strongly Connected

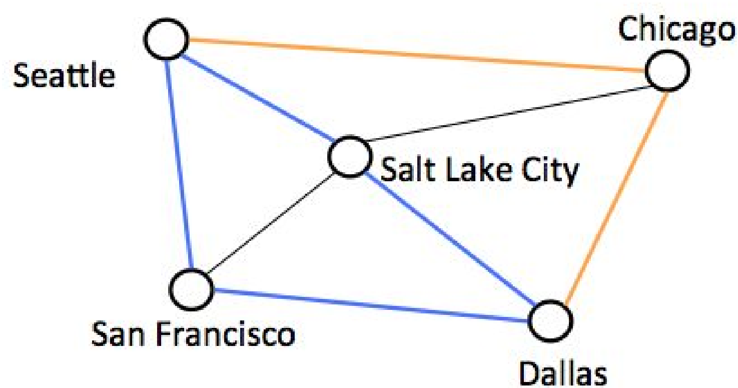


Weakly Connected

For both directed and undirected graphs, they are considered **fully connected** if there is an edge from every vertex to every other vertex (i.e. there is a path of length 1 from every vertex a to every other vertex b).

Paths and Cycles

A path is a set of edges from a vertex ‘A’ to another vertex ‘B’ in the graph. A cycle is a path that starts and ends at the same vertex. Optionally, a cycle can go through that vertex multiple times. If it doesn’t repeat vertices, then a cycle is a simple cycle. If it is a weighted graph, the cost of a path or a cycle is the sum of the costs for each of the edges. Whether it is weighted or not, the length of a path or cycle is the number of edges. Example:



Paths and cycles in directed graphs are exactly the same, except you can only use an edge in the direction it points (from source to destination).

DAG: Directed Acyclic Graphs

A DAG is a graph that is both directed and has no cycles. The edges can be weighted or unweighted. All trees are DAGs. DAGs are the only types of graphs you can do Topological Sorts on.

Density and Sparsity

Density and sparsity are terms used (in any kind of graph) to discuss how many edges are in a graph. A dense graph has lots of edges and a sparse graph has few edges.
As described in the lecture slides:
In an **undirected** graph, $\mathbf{0 \le |E| < |V|^2}$ In a **directed** graph: $\mathbf{0 \le |E| \le |V|^2}$
Therefore, for any graph: $\mathbf{O(|E|+|V|^2)}$ is still $\mathbf{O(|V|^2)}$

However, because **|E|** is often much smaller than $\mathbf{|V|^2}$, we do not always approximate **|E|** as $\mathbf{O(|V|^2)}$. It is useful to be able to describe an algorithm as **O(|E|)**, since that means something more specific than **O(|V|).**

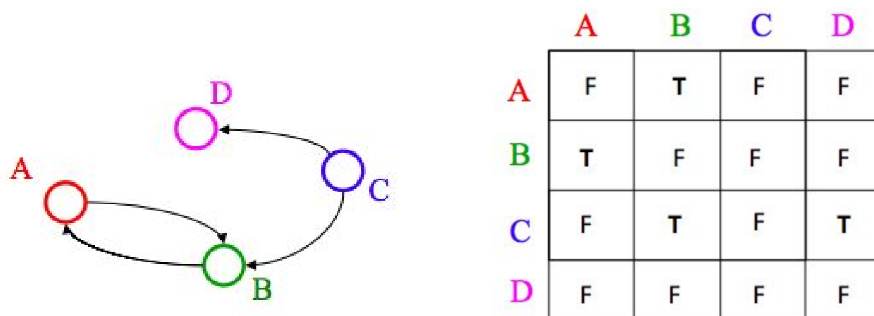
This definition is also useful, because if **|E|** is $\mathbf{O(|V|^2)}$ we say the graph is dense and if **|E|** is **O(|V|)** we say the graph is sparse.

Implementing a Graph

We still need to talk about how to actually implement a graph. So far, we've discussed graphs in a general sense, but we need to actually implement how to store a graph so we can implement functions like `isEdge(Vertex source, Vertex dest)`. The **efficiency analysis** of various operations on the various representations are in the lecture slides.

Adjacency Matrix

For each vertex in the graph, we can assign an index. Then we can use a matrix, or a 2D array to represent where edges exist in our graph. Here is an example where the graph is directed and the edges have no weights. We can use booleans, true or false, to represent that an edge between those two vertices exist. **This is better for dense graphs.**



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

We could use numbers instead of booleans if the edges have weights. Or we can minimize the amount of space in our 2D array if the graph is undirected, because we can use edge (a,b) to represent the implied edge (b,a).

Adjacency List

For each vertex in the graph, we can store the collection of edges from that vertex. There are many variations on how to actually implement this. Commonly, you will use a Map<Vertex, Collection<Edge>> where you are mapping a vertex to the edges it stores. That 'edge' could be a vertex by itself like below, or an edge object which stores the cost. The collection of edges in the map could be a Set, a List, an Array, or any other collection. If it's a directed graph, we can store both a collection of in-edges and a collection of out-edges. Adjacency Lists are a more common implementation because they are very flexible **and are better for sparse graphs,** which more closely models many real world problems**.**