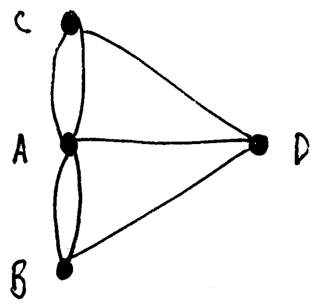
Introduction to Graphs

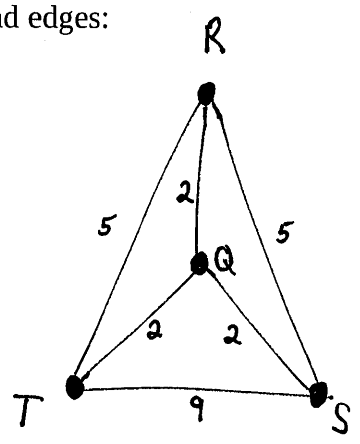
I. Basic Definition

A *graph* is a collection of vertices and edges:



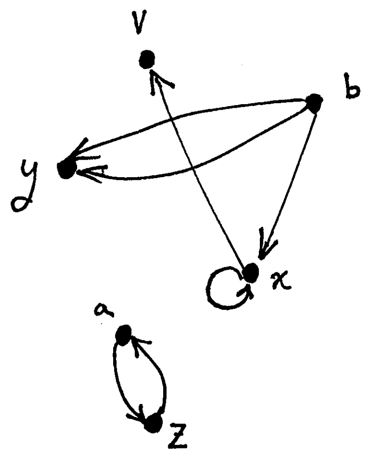
Graph 1

(Undirected and Unweighted)



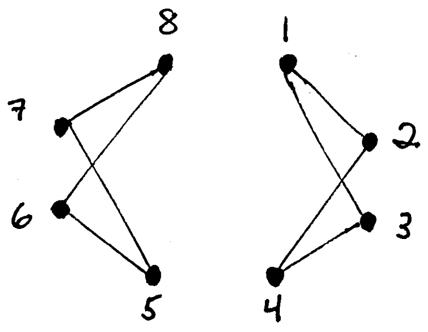
Graph 2

(Weighted)

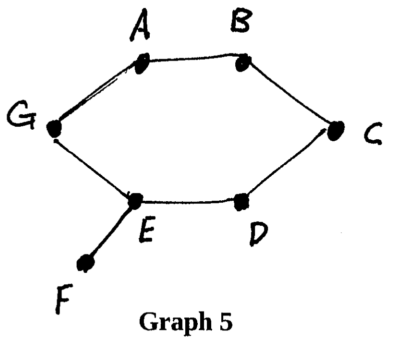


Graph 3

(Directed Graph)



Graph 4



Graph 5

The simplest type of graph is unweighted and undirected (Graph 1). The solid circles are the *vertices* (A, B, C, and D), and the lines are *edges*. Sometimes we write G = (V, E).

When we denote a graph as G = (V, E), V is our set of vertices (such as V = {A, B, C, D}), and E is our set of edges (such as E = {AB, BC, CD, DA}).

We can assign values to the edges (a cost related to connecting the two vertices together) and form a *weighted graph* (Graph 2).

There's also a *directed graph* (sometimes called a *digraph*), that makes the edges one-way (Graph 3). Since edges are only one-way in Graph 3, we say that graph has an edge from *b* to *x*, but it does not have an edge from *x* to *b*.

We can have weighted directed graphs, as well. We can even have graphs that have multiple edges between vertices (called *multigraphs*).

II. Some Terminology

We say two vertices are ***adjacent*** if there is an edge between them in a graph.

A ***path*** is a list of vertices in which successive vertices are connected by edges in the graph. For example, A → B → D → A → C is a path in Graph 1. In contrast, A → B → C is *not* a path in Graph 1, since there is no edge from B to C.

A ***simple path*** is a path in which no vertex is repeated. Hence, A → B → D → A → C is not a simple path, but A → B → D → C is.

A ***cycle*** is a path in which the first and last vertex are the same, but no edges are repeated, and there is at least one edge. (This is also called a ***closed path***.) For example, E → F → E is not a cycle in Graph 5, whereas A → B → C → D → E → G → A is. (Note that if we eliminate the requirement that a cycle must have no repeated edges, then the path E → F → E would be a cycle in Graph 5.)

A ***simple cycle*** is a cycle in which no vertices or edges are repeated (except, of course, for the first/last vertex in the cycle, which is visited exactly twice).

The ***path length*** of some path is the number of edges in that path. For example, the path E → F → E → G → A has a path length of 4.

An undirected graph that has a path from every vertex to every other vertex in the graph is said to be ***connected***. An undirected graph that does not satisfy this property is said to be ***disconnected***.

A ***connected component*** is a maximal connected subgraph. Graph 4 has two connected components.

III. Motivation

Graphs are very useful for representing a large array of problems and for solving everyday "real world" problems. Some examples include:

- Map intersections to vertices and roads to edges, and now you have a representation to help you route fire trucks to fires.

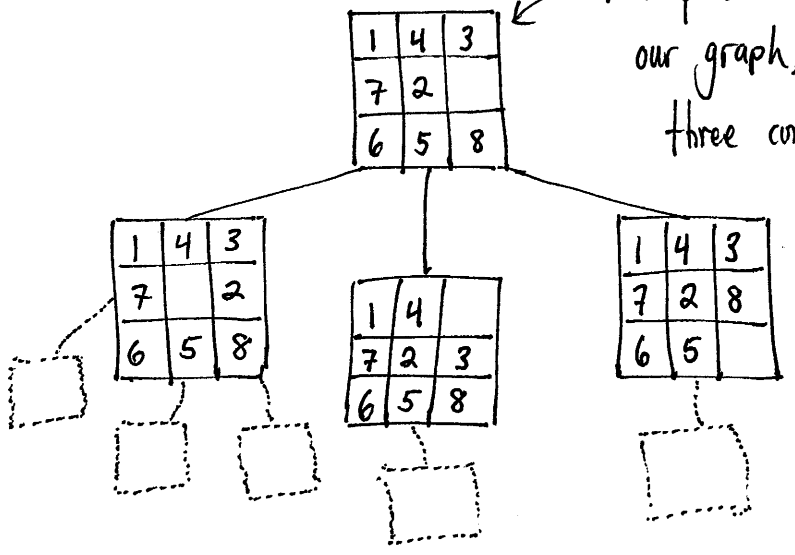
- Map intersections to vertices and roads to edges, and now you have a representation to help you route hungry students to delicious food. (Pom Pom's, Krungthep, Dandelion Cafe, Hawkers, Mamak, Tako Cheena, etc.)

- Map the buildings on UCF's campus to vertices, the sidewalks between buildings to edges, and then you'll be able to find the shortest walking distance from each building to any other building.

- Map people to vertices and the number of interactions or other connects they have on some social network to weighted edges, and you can quantify the value of human friendships.

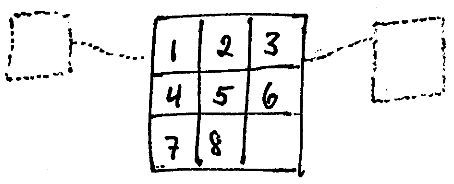
(Further Motivation)

- States on a game board can be mapped to vertices, with edges connecting some state (*x*) to another state, *y*, if there is a legal move that can get you from state *x* to state *y*. Consider, for example, the 8-puzzle problem:



This puzzle configuration is a vertex in our graph, and it's adjacent to the three configurations we can reach via a single move.

...elsewhere in the graph:

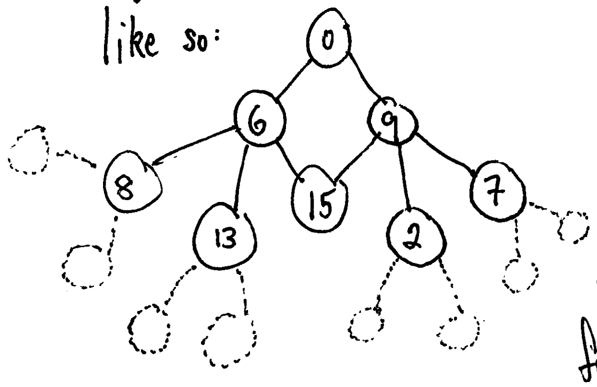


- Another problem (Knight's Tour): What is the smallest number of jumps for a knight to get from one position on a chess board to another? How can we represent this using a graph? How can we find the solution?

If we number the positions on, say, a 4×4 board:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

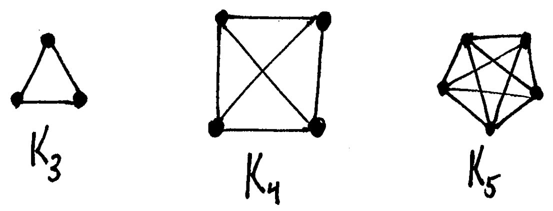
...then we can create a graph where each tile is a node, and there is an edge between two nodes if a knight could jump from one of those tiles to the other, like so:



Now the problem has been transformed into the question: What is the shortest path in the graph from some node, x, to some other node, y?

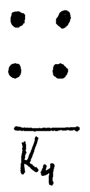
IV. Families of Graphs

Note: We saw the number of edges in K_n is $\sum_{i=0}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2-n}{2}$



Complete Graph (K_n)

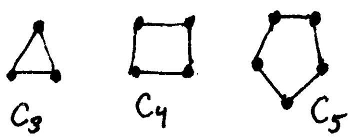
(*n* nodes, every possible edge)



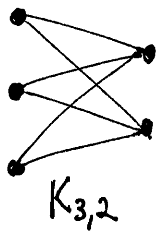
Empty Graph ($\overline{K_n}$)

(*n* nodes, zero edges)

Note: The bar denotes the complement of a graph.



Cycle (C_n)



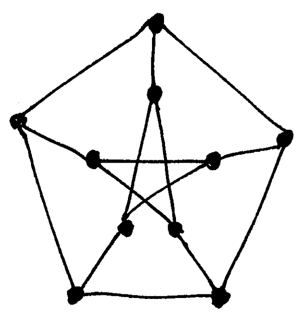
Complete Bipartite Graph ($K_{m,n}$)

There are two partitions here. All the nodes in one partition are adjacent to all nodes in the other, and no two nodes within one partition are adjacent to one another.

V. Graphs for Testing

When testing some implementation of a graph algorithm, you should try it out on graphs from all the families above. Also, consider things like even and odd vertex counts.

In addition to the graphs listed above, the following is a very good graph for testing:



Petersen Graph

VI. Representations of Graphs

The two most common ways of representing a graph in a program are the adjacency matrix and the adjacency list:

Adjacency Matrix

An adjacency matrix is an NxN matrix (or array), where N is the number of vertices in the graph. The (i, j) entry of the matrix is the weight of the edge from *i* to *j* (or a boolean value indicating whether there's an edge at all, or, in the case of multigraphs, the number of edges from *i* to *j*).

The adjacency matrix for Graph 1, above, would be:

	A	B	C	D
A	0	2	2	1
B	2	0	0	1
C	2	0	0	1
D	1	1	1	0

Note: For an undirected graph, matrix[i][j] = matrix[j][i], so half the matrix is unnecessary / superfluous / wasted space.

The adjacency matrix is the most common method for representing graphs that you'll use in this course.

Adjacency List

The adjacency list simply keeps a linked list for each vertex, indicating the adjacent vertices for that vertex (that is, a list of vertices such that there exists an edge from the source node to each of the other vertices). For Graph 1, the adjacency list would be:

A: $B \rightarrow C \rightarrow D$
B: $A \rightarrow D$
C: $A \rightarrow D$
D: $A \rightarrow B \rightarrow C$

← Runtime for finding out whether two particular vertices are adjacent is very slow if these lists are long!

Where would you prefer to use an adjacency list instead of an adjacency matrix?

When we have a <u>sparse</u> graph — one with lots of nodes, but few edges.

What kind of data structure might you use to implement this in Java?

Other Representations

Edge List – A list of pairs of adjacent vertices. Finding all the edges adjacent to a given vertex is inefficient, but there are algorithms where this representation might be useful.

Adjacency Function – For generating adjacent vertices where the representation is implicit. (Example: The 8-puzzle problem mentioned above.)