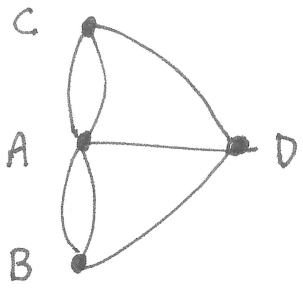
Introduction to Graphs

(Prepared by Dr. Sean Szumlanski)

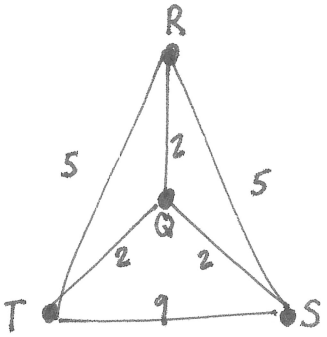
I. Basic Definition

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



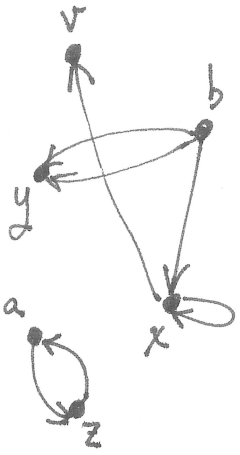
Graph 1

(Undirected and Unweighted)



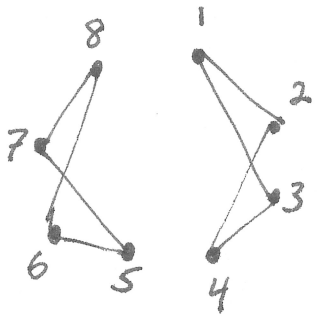
Graph 2

(Weighted)



Graph 3

(Directed Graph)



Graph 4

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).

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, ***b*** has an edge to ***x*** in Graph 3, but ***x*** does not have an edge to ***b***.

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

II. Some Terminology

A graph is *connected* if there is a path from every vertex to every other vertex in the graph. If a graph is not connected, we call it *disconnected.* (Above, Graph 1 is connected, but Graph 4 is disconnected.)

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.

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 that is simple, except that the first and last vertices are the same. In Graph 4, the path 7 → 5 → 6 → 8 → 7 is a cycle.

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

III. 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 *N*x*N* matrix (where *N* is the number of vertices in the graph). The (*i*, *j*) entry of the matrix is the number of edges from vertex *i* to vertex *j* (or the cost of an edge from *i* to *j* if one exists, or a boolean value indicating whether there's an edge at all). 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

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

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:

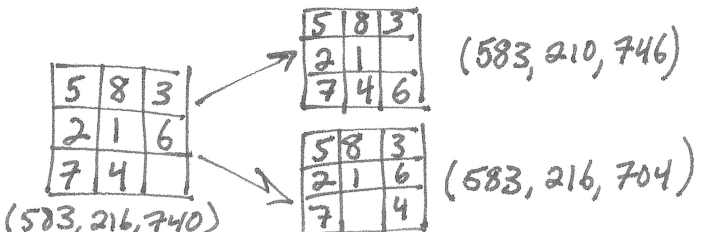
A: B → C → D
B: A → D
C: A → D
D: A → B → C

This representation is useful for sparse graphs (those with lots of vertices but few edges). You can implement this using an array of array lists (Java), or an array of lists (C++).

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 (like Kruskal's).

Adjacency Function – For generating adjacent vertices when the representation is implicit. (Example: the 8-puzzle problem from last week's contest.)

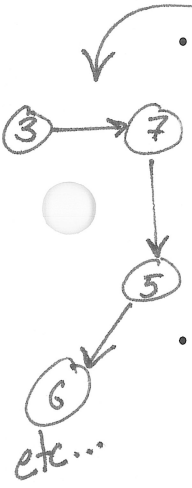


We can represent configurations using integers. (How many states? 9! = 362,880)
—— probably don't want to build the whole graph!

IV. Motivation

Graphs are very useful for representing a large array of problems and for solving everyday "real world" problems, on which the contest loves to focus. Some examples include:

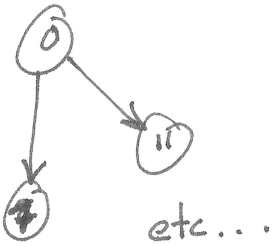
- Map intersections to vertices and roads to edges, and now you have a representation to help you route fire trucks to fires.
- Map a network to computers and routers to vertices, physical connections to edges, and you have a way to find the minimum number of hops between two given computers.
- Map a set of rooms that need water to vertices, all combinations of distances between pairs of rooms to the edges, and then you can determine the smallest amount of pipe needed to connect all the rooms together.
- Map the buildings on the UCF campus to vertices, the sidewalks between the buildings to edges, and then you'll be able to find the shortest walking distance from HEC to any other building.



- A problem: Given N matrices just by size ($N \leq 50$), is there an ordering that allows us to perform matrix multiplication? (E.g.: 3x7, 7x5, 5x6, 6x12, 12x3, 3x2) The brute force approach (permutations) is too slow; we don't want to try all $N!$ possibilities. If we create a graph where each dimension is represented as a vertex, and edges indicate two dimensions are used in one matrix (e.g., if we have a 3x7 matrix, we want an edge from vertex 3 to vertex 7), then we have an Euler Path problem. (Is there a path that visits every edge exactly once? We can re-visit nodes, but not edges.)
- 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?

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

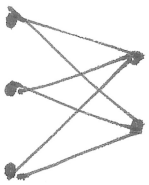
Knight can move from 0 to 7 or 11, so:



V. Graphs for Testing

Empty Graph
(n nodes, zero edges)

$K_{3,2}$

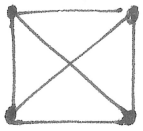


Bipartite Graph ($K_{m,n}$)

Ask them: How many edges in $K_{m,n}$?

$(m \cdot n)$

K_4

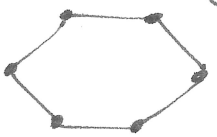


Complete Graph (K_n)

(How many edges?)

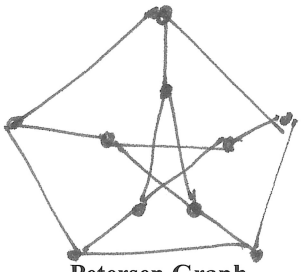
$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$

C_6



Cycle (C_n)

(How many edges? —— n)



Petersen Graph

VI. Algorithms

Depth-First Search

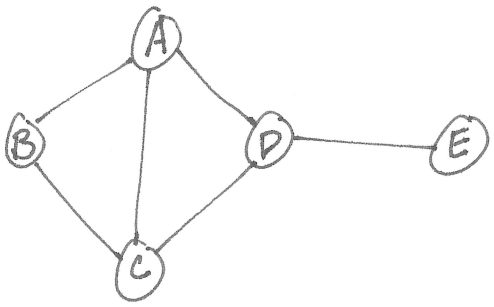
The name "Depth-First Search" comes from the fact that you search as deep as you can in the graph first.

```
adj : array[1..N, 1..N] of Boolean
visited : array[1..N] of Boolean      (← initialized to all false!)

DFS(node)
{
   visited[node] = true;
   if (node is target node)
      process it accordingly
   for i = 1 to N
      if (adj[node][i]) and !visited[i])
         DFS(i);
}
```

Motivation: Most basic search algorithm. Useful for mazes, path finding, cycle detection...

Show example. Mention backpaths.



A B C D E isn't the only valid DFS...

A D E C B is another one, albeit unlikely if using adj. matrix.

Breadth-First Search

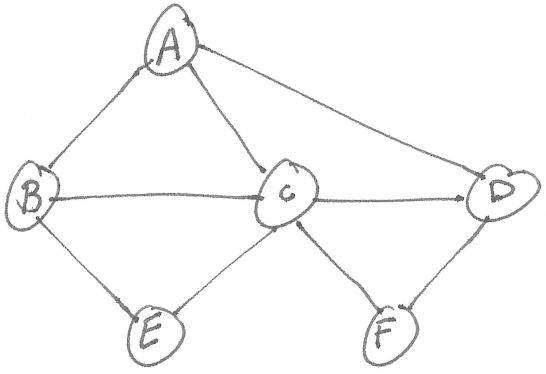
The name "Breadth-First Search" comes from the idea that you search as wide as possible at each step.

```
adj : array[1..N, 1..N] of Boolean
visited : array[1..N] of Boolean      (← initialized to all false!)

BFS(start)
{
   enqueue(start)
   visited[start] = true
   while (queue is not empty)
      v = dequeue();
      for i = 1 to N
         if (adj[v][i]) and !visited[i])
            visited[i] = true
            enqueue(i)
}
```

Motivation: Finding all connected components in a graph, finding the shortest path between vertices *u* and *v* (in an unweighted graph), counting the number of connected components in a graph, testing a graph for bipartiteness (which also tests for 2-colorability)...

Show example. Mention this is the solution to the 8-puzzle problem mentioned earlier. Always finds shortest path to goal (in unweighted graph).



For counting connected components:

```
int cc(){
  int cnt = 0
  for v in graph
    if v unvisited
      BFS(v)
      cnt ++
  return cnt
}
```

(Uses adj[][] and visited[] from above.)

Floyd's All Shortest Paths (Floyd-Warshall)

The basic idea of this algorithm is to construct a matrix that gives the length of the shortest path between each pair of nodes. The algorithm is based on dynamic programming.

We'll initialize our matrix to the adjacency matrix and then perform N iterations. After each iteration k, the matrix will give the length of the shortest paths between each pair of nodes that use only nodes numbered 1 to k as intermediary nodes.

At each iteration, the algorithm must check whether, for each pair of nodes i and j, there exists a path from i to j going through the node k, that is better than the current shortest path (which only uses nodes 1 to k-1). After n iterations, the matrix will give the length of the shortest paths using any of the nodes, which is the result we want.

```
adj : array[1..N, 1..N] of ~~Boolean~~ Int
A : array[1..N, 1..N] of ~~Boolean~~ Int      (A is our shortest path matrix)

Floyd()
{
   A = adj
   for k = 1 to N
      for i = 1 to N
         for j = 1 to N
            A[i][j] = MIN(A[i][j], A[i][k] + A[k][j])
}
```

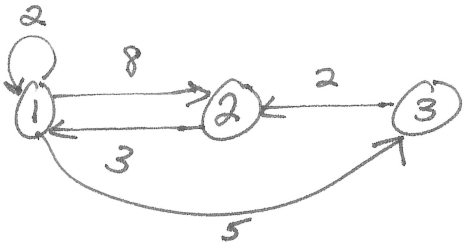
$O(N^3)$ — avoid for $|V| > 200$ or 300 or so.

Be sure to zero out the diagonal of the matrix before starting (A[i][i] = 0 for all i), and where there are no edges, initialize to "∞".

For ∞, choose something that won't cause an integer overflow but will still be greater than any real value we'll see in the matrix. (Common value: 1 billion; note that 2 billion is still a valid *int*.)

See also: the Bellman-Ford algorithm for computing single-source shortest paths in a weighted digraphs.

Show example:



$$\begin{bmatrix} 2 & 8 & 5 \\ 3 & -1 & -1 \\ \infty & 2 & -1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \xrightarrow{k=1} \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix} \xrightarrow{k=2} \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \xrightarrow{k=3} \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

eg: $2 \to 3$ vs. $2 \to 1, 1 \to 3$

∞ vs. $3 + 5 = 8$