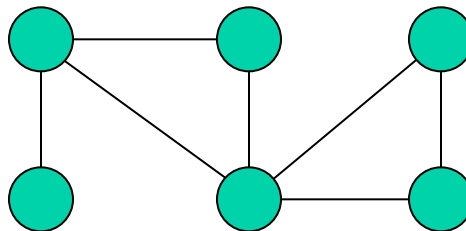


Graphs

A graph G consists of a set of *vertices* V together with a set E of vertex pairs or *edges*.

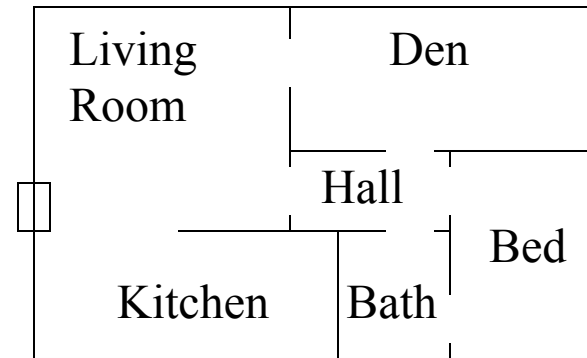
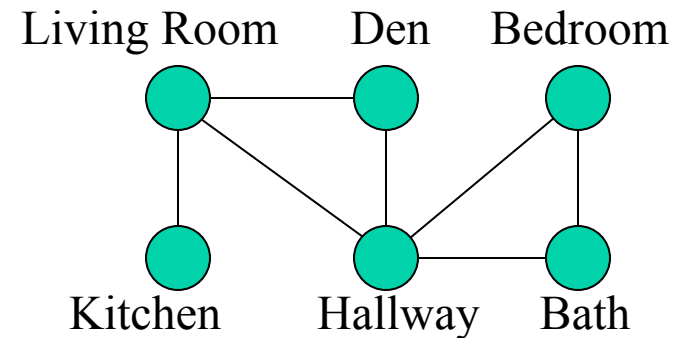
Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially *any* relationship.



What could this mean?

The vertices could represent rooms in a house, and the edges could indicate which of those rooms are connected to each other.

Sometimes a using a graph will be an easy simplification for a problem.

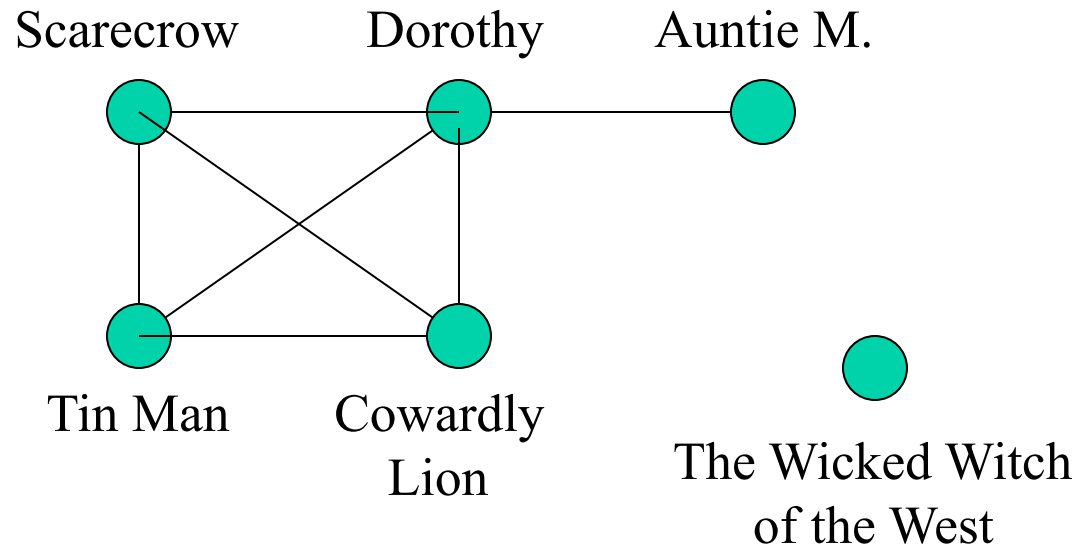


Apartment Blueprint

What else could a Graph mean?

- Vertices are cities and edges are the roads connecting them.
- Edges are the components in a circuit and vertices are junctions where they connect.
- Vertices are software packages and edges indicate those that can interact.
- Edges are phone conversations and vertices are the households being connected.

Friendship Graphs



Each vertex represents a person, and each edge indicates that the two people are friends.

Questions...

If I'm your friend, are you my friend?

A graph is said to be *undirected* if edge (x, y) always implies (y, x) . Otherwise it is said to be *directed*.

Am I my own friend?

An edge of the form (x, x) is said to be a *loop*. If x was y 's friend several times over, we can model this relationship using *multiedges*. A graph is said to be *simple* if it contains no loops or multiedges.

How close a friend are you?

A graph is said to be *weighted* if each edge has an associated numerical attribute. In an *unweighted* graph, all edges are assumed to be of equal weight.

More questions...

Am I linked by some chain of friends to someone famous?

A *path* is a any sequence of edges that connect two vertices. A *simple path* never goes through any vertex more than once. The *shortest path* is the minimum number edges needed to connect two vertices.

Is there a path connecting every two people in the world?

The “six degrees of separation” theory argues that there is always a short path between any two people in the world. A graph is *connected* if there is there is a path between any two vertices. A directed graph is *strongly connected* if there is always a directed path between vertices. Any subgraph that is connected can be referred to as a *connected component*.

Still More Questions...

Who has the most and who has the fewest friends?

The *degree* of a vertex is the number of edges connected to it. The most popular person will have a vertex of the highest degree. Remote hermits may have degree-zero vertices. In *dense* graphs, most vertices have high degree. In *sparse* graphs, most vertices have low degree. In a *regular graph*, all vertices have exactly the same degree.

What is the largest clique?

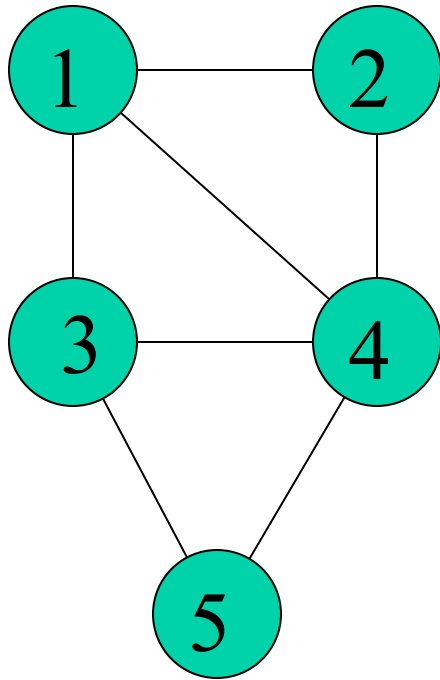
A graph is called *complete* if every pair of vertices is connected by an edge. A *clique* is a sub-graph that is complete.

Yet Another Question...

How long will it take for my gossip to get back to me?

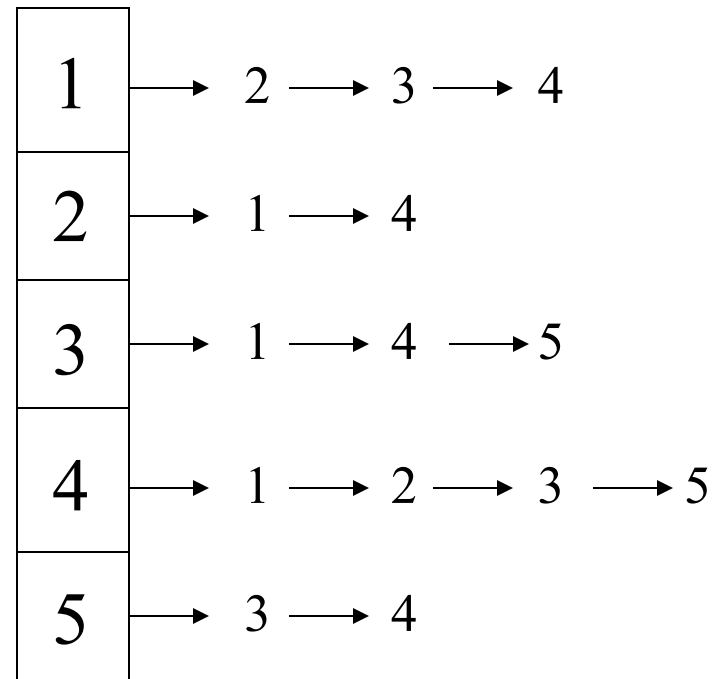
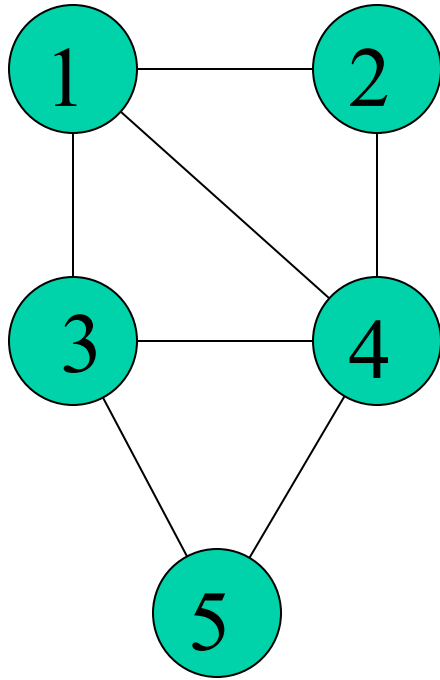
A *cycle* is a path where the last vertex is adjacent to the first. A cycle in which no vertex is repeated is said to be a *simple cycle*. The shortest cycle in a graph determines the graph's *girth*. A simple cycle that passes through every vertex is said to be a *Hamiltonian cycle*. An undirected graph with no cycles is a *tree* if it is connected, or a *forest* if it is not. A directed graph with no directed cycles is said to be a *directed acyclic graph* (or a DAG)

Adjacency Matrices



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Adjacency Lists



Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comparison

Faster to test if (x, y) exists?

matrices: $\Theta(1)$ vs. $\Theta(m)$

Faster to find vertex degree?

lists: $\Theta(1)$ vs. $\Theta(n)$

Less memory on sparse graphs?

lists: $\Theta(m+n)$ vs. $\Theta(n^2)$

Less memory on dense graphs?

matrices: (small win)

Edge insertion or deletion?

matrices: $\Theta(1)$ vs. $\Theta(m)$

Faster to traverse the graph?

lists: $\Theta(m+n)$ vs. $\Theta(n^2)$

Better for most problems?

lists

Problem:

The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, such that $(x, y) \in E^2$ iff, for some z , both (x, z) and $(z, y) \in E$; i.e., there is a path of exactly two edges.

Give efficient algorithms to square a graph on both adjacency lists and matrices.

G^2 with Adjacency Matrices

Given an adjacency matrix, we can check in constant time whether a given edge exists. To discover whether there is an edge (x, y) in E^2 , for each possible intermediate vertex z we can check whether (x, z) and (z, y) exist in $O(1)$.

Since there are $O(n)$ intermediate vertices to check, and $O(n^2)$ pairs of vertices to ask about, this takes $O(n^3)$ time.

G^2 with Adjacency Lists

For a given edge (x, z) , we can run through all the edges from z in $O(n)$ time, and fill the results into an adjacency matrix of G^2 , which is initially empty.

It takes $O(mn)$ to construct the edges, and $O(n^2)$ to initialize and read the adjacency matrix, for a total of $O((n+m)n)$. Since $m+1 \geq n$ (unless the graph is disconnected), this is usually simplified to $O(mn)$, and is faster than the previous algorithm on sparse graphs.

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

For *efficiency*, we must make sure we visit each edge at most twice.

Marking Vertices

The idea in graph traversal is that we mark each vertex when we first visit it, and keep track of what is not yet completely explored.

For each vertex, we maintain two flags:

- ***discovered*** - have we encountered this vertex before?
- ***explored*** - have we finished exploring this vertex?

We must maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is set to be discovered.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

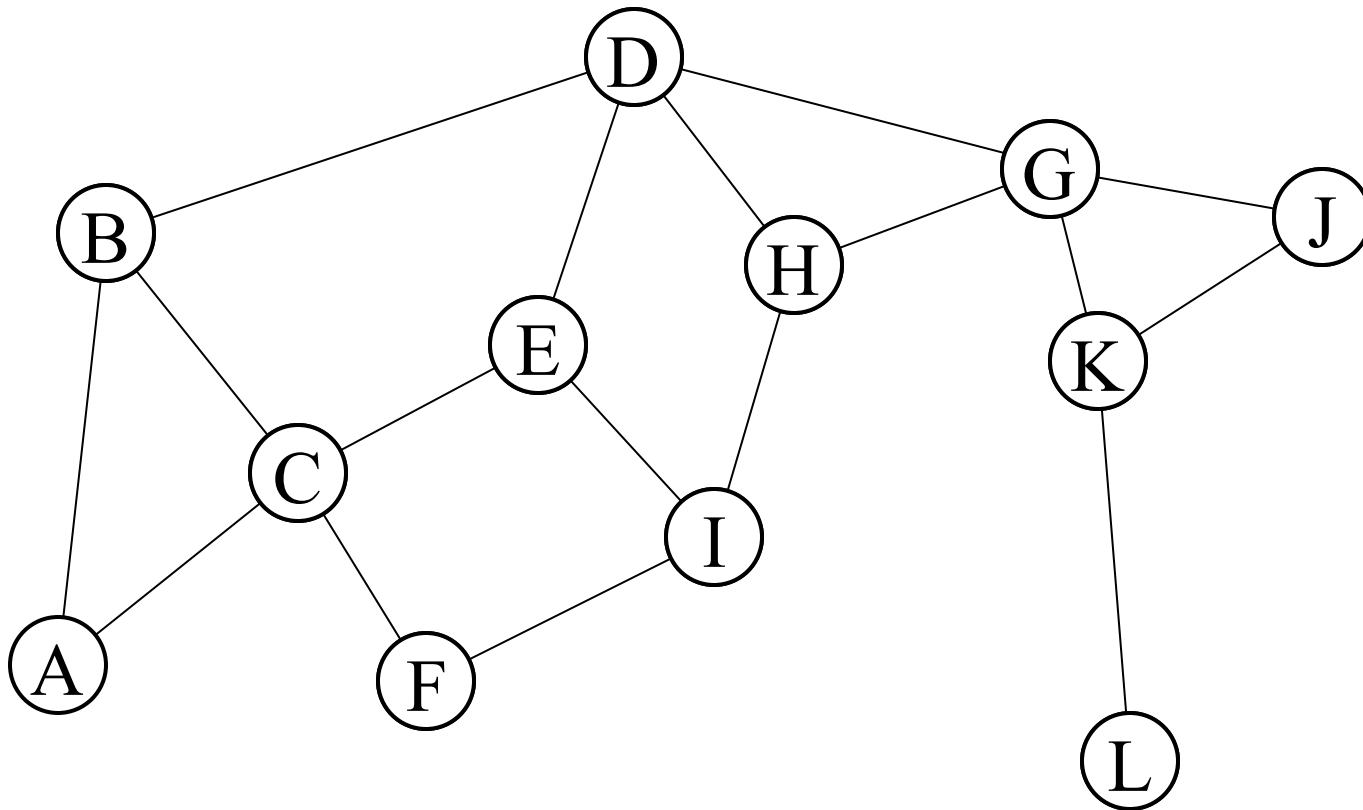
Suppose not, i.e. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it....

Traversal Orders

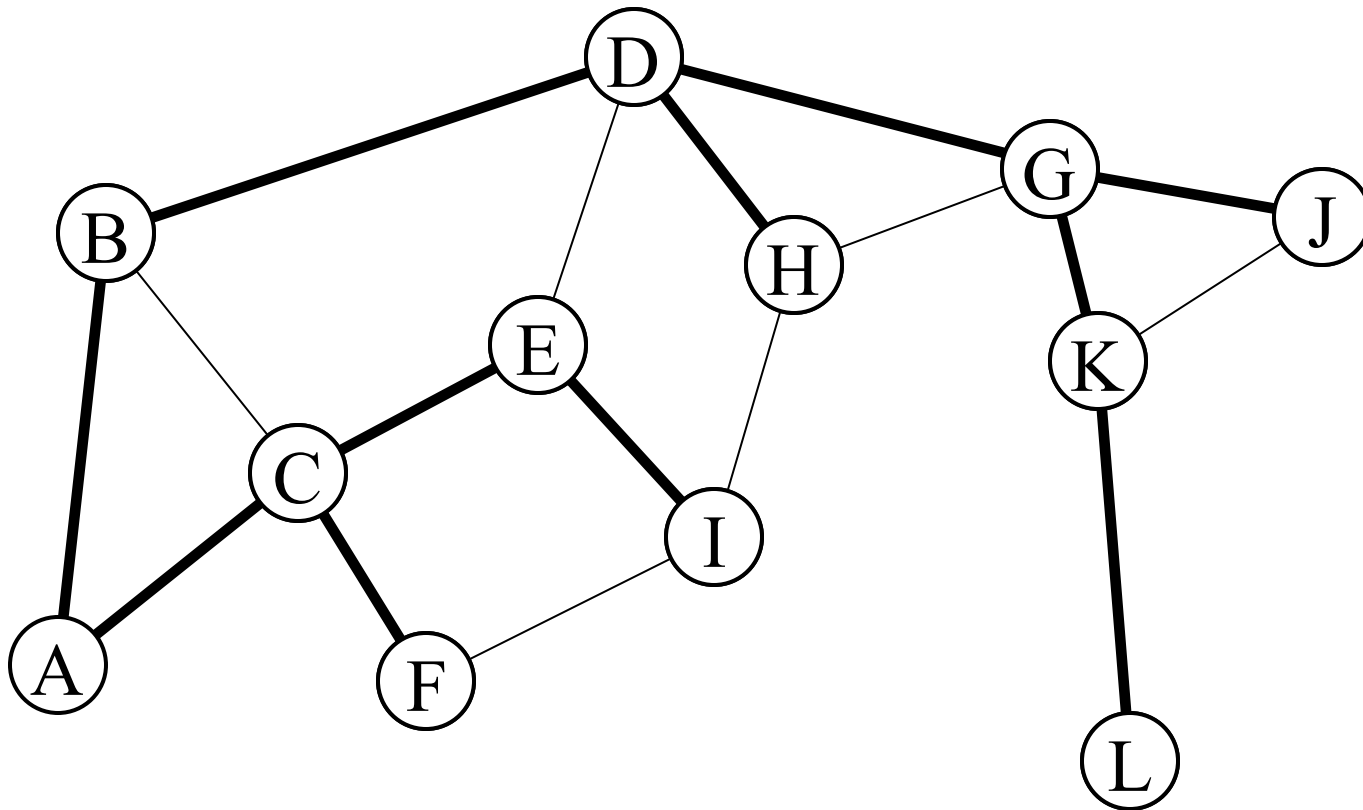
The order we explore the vertices depends upon the data structure used to hold the discovered vertices yet to be fully explored:

- ***Queue*** - by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus we radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.
- ***Stack*** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only when surrounded by previously discovered vertices. Thus our explorations speed away from our starting point, defining a so-called *depth-first search*.

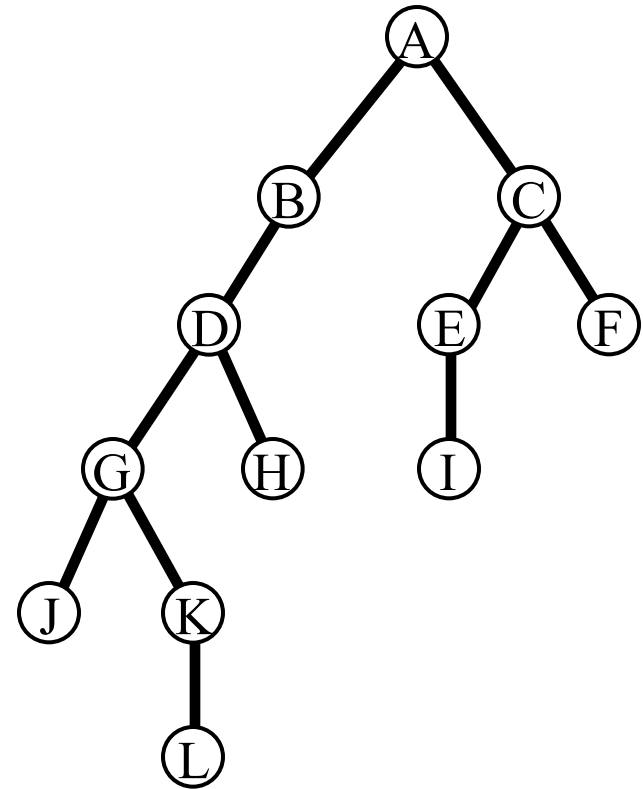
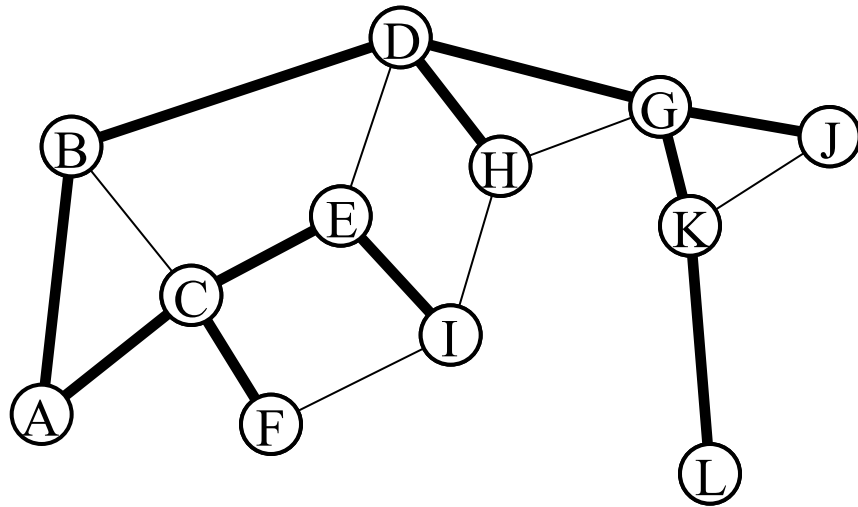
Example Graph



Breadth-First Search Tree



Breadth-First Search Tree

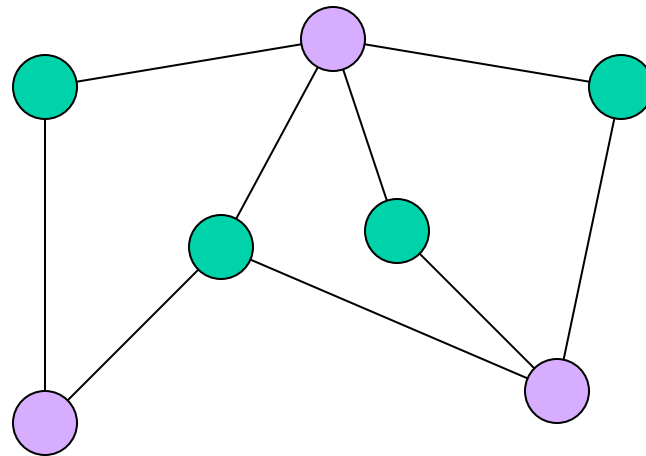


Depth-First Search Tree



Question

Give an efficient algorithm to determine if a graph is bipartite. (Bipartite means that the graph can be colored with 2 colors such that all edges connect vertices of different colors.)



Question

Give an $O(n)$ algorithm to determine if an undirected graph contains a cycle.

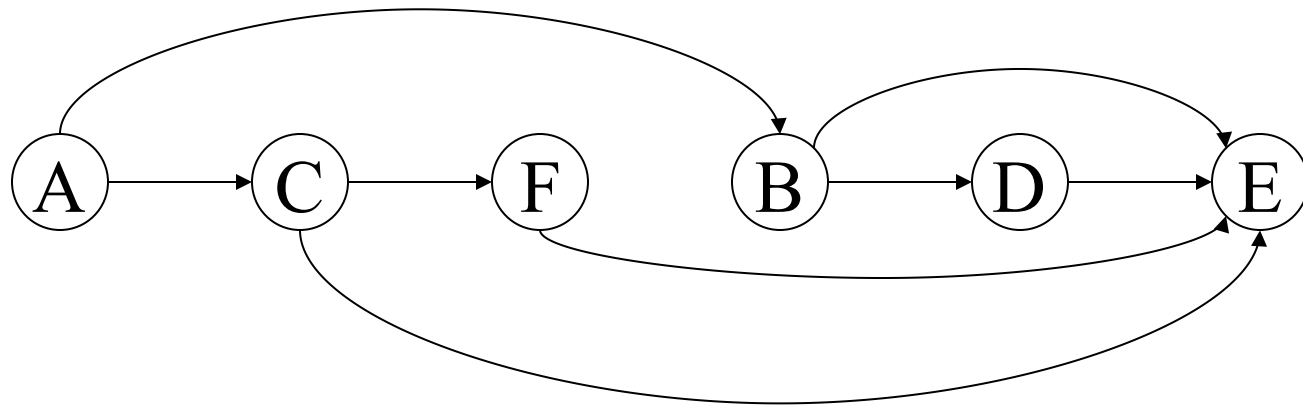
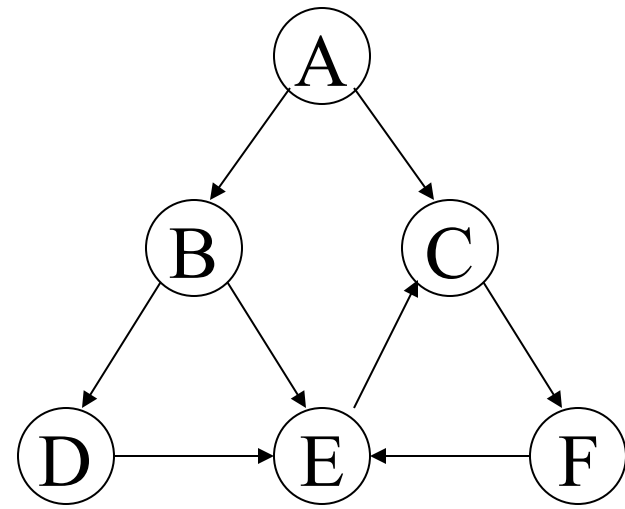
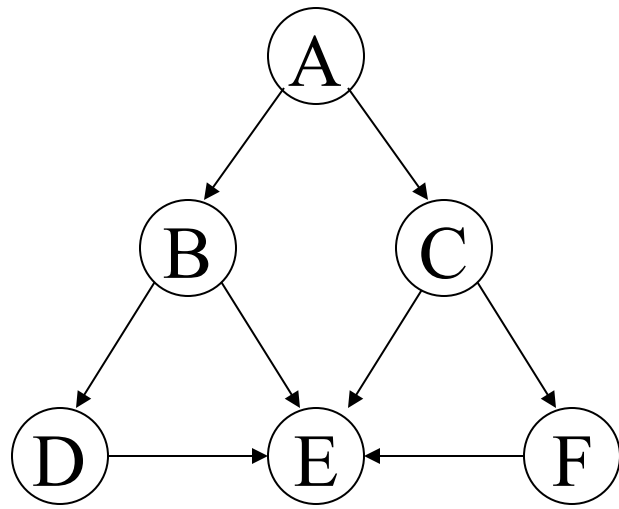
Note: Traversal by BFS and DFS both take $O(n+m)$ time.... What can we do?

Topological Sorting

A directed acyclic graph (DAG) is a directed graph with no directed cycles.

A topological sort is an ordering of vertices such that all edges go from left to right. Only an acyclic graph can have a topological sort because eventually all cycles must return to their starting point.

How can we find a topological sort?



Topological Sorting

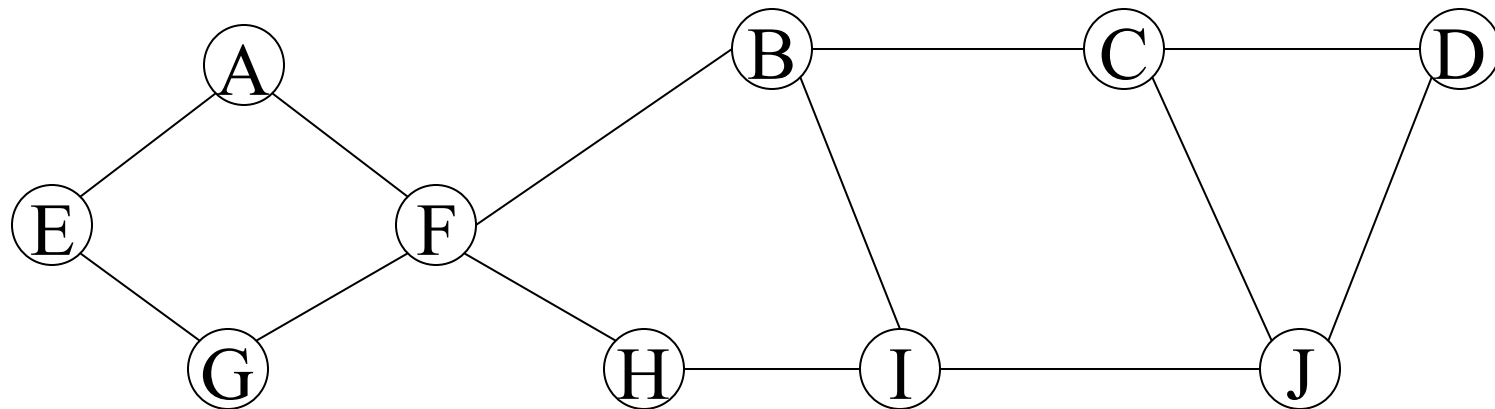
Consider doing a depth-first-search on the graph - what properties does the resulting tree have? Can we use this to find a topological sorting?

Since a DAG may have multiple sources in it (vertices with outgoing edges but none incoming), it is not always possible to construct a depth-first-search tree.

What if we do a DFS from all of the sources? How can we combine the results together?

Articulation Vertices

Suppose you are a hacker seeking to disrupt a company's intranet. Which router in the diagram below should you choose to blow up in order to cause the maximum amount of damage?



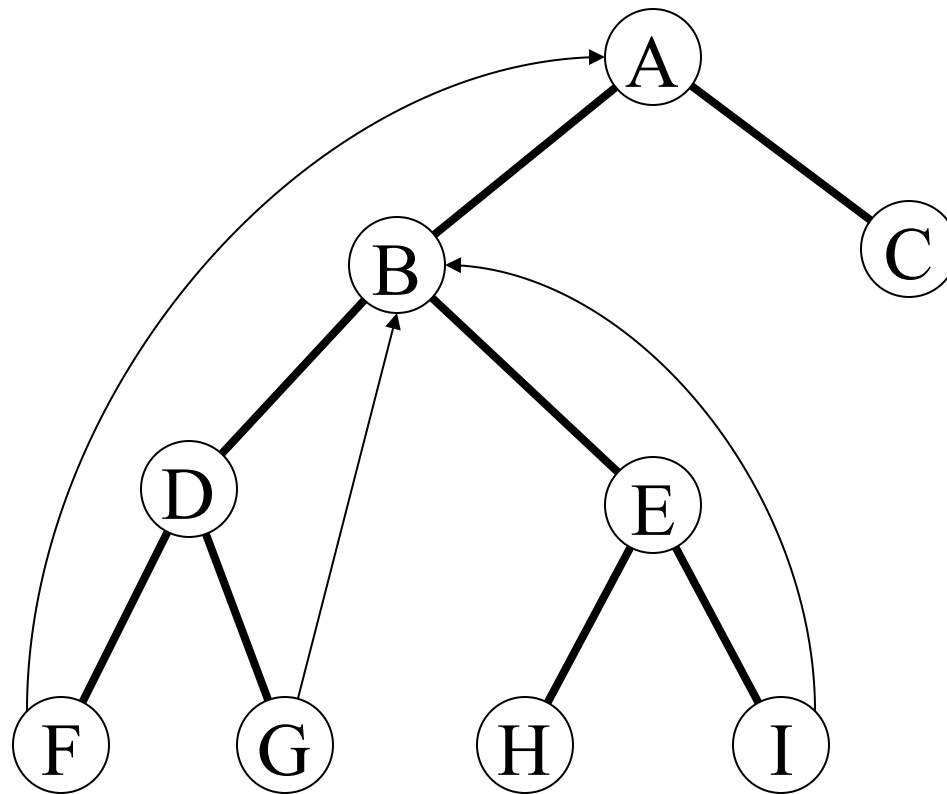
Connectivity

The connectivity of a graph is the smallest number of vertices whose deletion will disconnect a graph. For graphs with an articulation vertex, the connectivity is one.

How can we test for 0 or 1 connectivity by brute force?

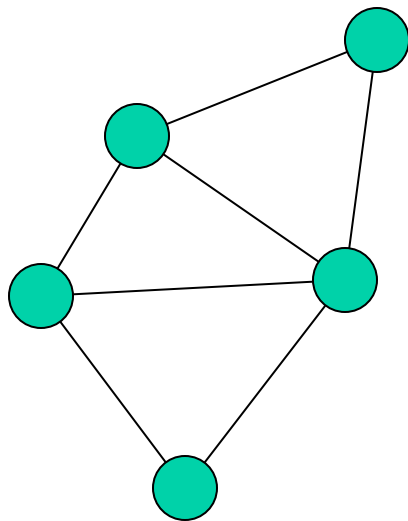
Is there a better algorithm we can use?

Finding Articulation Vertices with DFS



Minimum Height Spanning Trees

Both DFS and BFS produce spanning trees, but how can we guarantee that we find one of minimal height?



← What height will BFS give if we start from the top node?

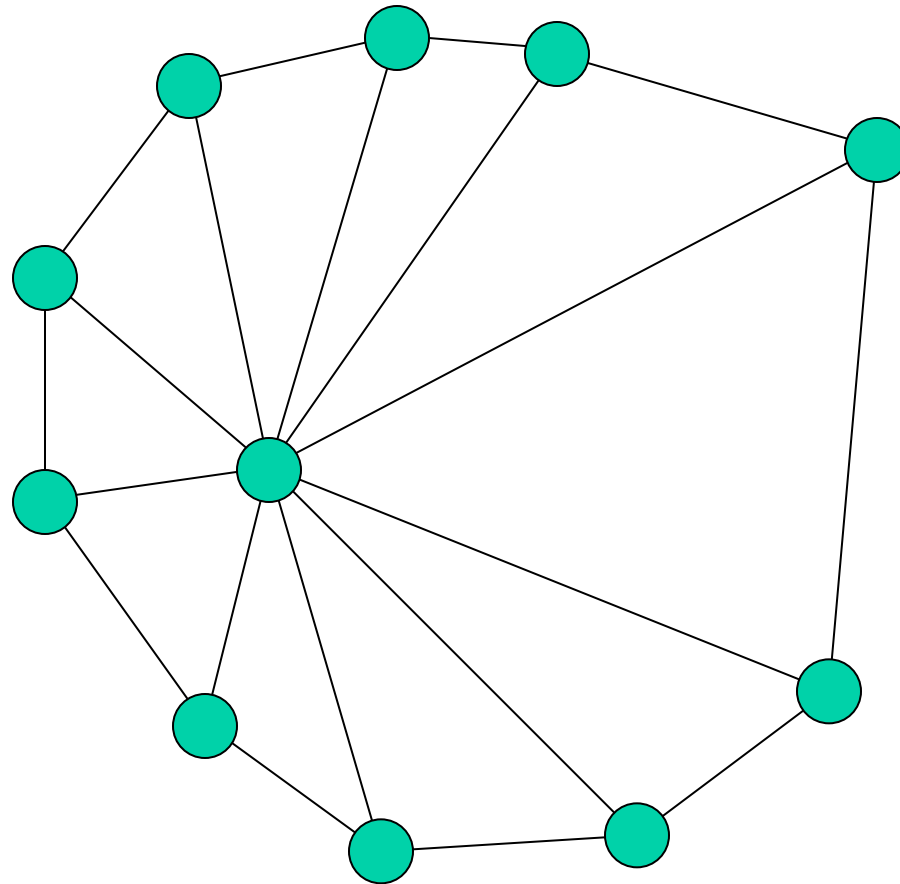
Can we do better than this?

Weighted Graphs

A weighted graph is one in which each edge has a “cost” of some kind associated with it. Many algorithms become more complex when in addition to finding *any* solution, we also want to minimize the cost of that solution.

Example: it is easy to find a spanning tree (using BFS or DFS) and even the minimum height spanning tree, but how hard is it to find the minimum *cost* spanning tree? (typically this is simply called the “minimum spanning tree”)

Example Graph



Prim's Algorithm

```
Prim-MinSpanningTree(G) {  
    Select an arbitrary vertex to start  
    the tree with;  
    While(non-tree vertices exist) {  
        Select the edge of minimum weight  
        between a tree and non-tree vertex;  
        Add the selected edge to the tree  
        and mark the new vertex as being in  
        the tree ;  
    }  
}
```

Kruskal's Algorithm

```
Kruskal-MinSpanningTree( $G$ ) {  
    Put the edges in a priority queue  
    ordered by weight;  
    count = 0;  
    while (count <  $n-1$ ) {  
        get next edge ( $v, w$ );  
        if (component ( $v$ ) != component ( $w$ ));  
            merge (component ( $v$ ), component ( $w$ ));  
            count++;  
        }  
    }  
}
```

Shortest Paths

The shortest path between two vertices s and t in an unweighted graph can be constructed using a breadth-first search from s . When we first encounter t in the search, we will have reached it from s using the minimum number of possible edges.

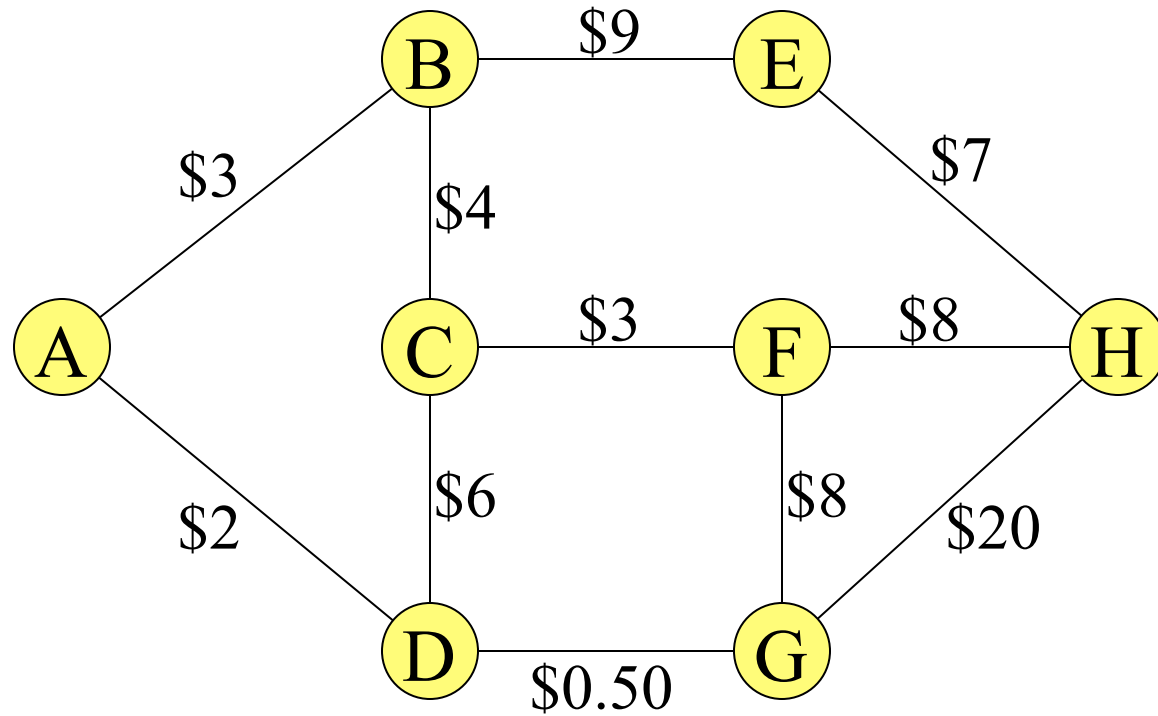
This is only true if all edges have the same weight; when they have differing weights we need to do something slightly more complex.

Dijkstra's Algorithm

Dijkstra's algorithm is another variation on Breadth-First-Search where, like minimum spanning trees, a priority queue determines which vertex is to be explored next. In this case, the priority of each vertex is the minimum total path found thus-far to that vertex.

The principle behind Dijkstra's algorithm is that given the shortest path between vertices x and y , any vertex v_i on that path must bisect it into the shortest path from x to v_i , and the shortest path from v_i to y .

Dijkstra Example



All-Pairs Shortest Path

What if we want to know the shortest path between all pairs of vertices in a weighted graph? How long will this take?

Example Problem

At an arm wrestling competition, one of the organizers is told to setup the order of contestants entering the awards ceremony such that no one who lost a competition comes into the room before the person who beat them.

Assuming that in every case the stronger person won, and no two people are the exact same strength, *but* not every pair of people arm-wrestled, how can he come up with a good ordering for them to enter in?

Example Problem

Marleen Thompson was taking the sixth grade class that she teaches to the zoo on a field trip. As part of the school policy, the students need to be broken up into small groups (of at least two) that must stay together at all times.

Each of the students gives her a list of other students that they are willing to pair with. How can she group the students into as many groups as possible, where each is with at least one other student on their list? Can we do this using graph theory?

Example Problem

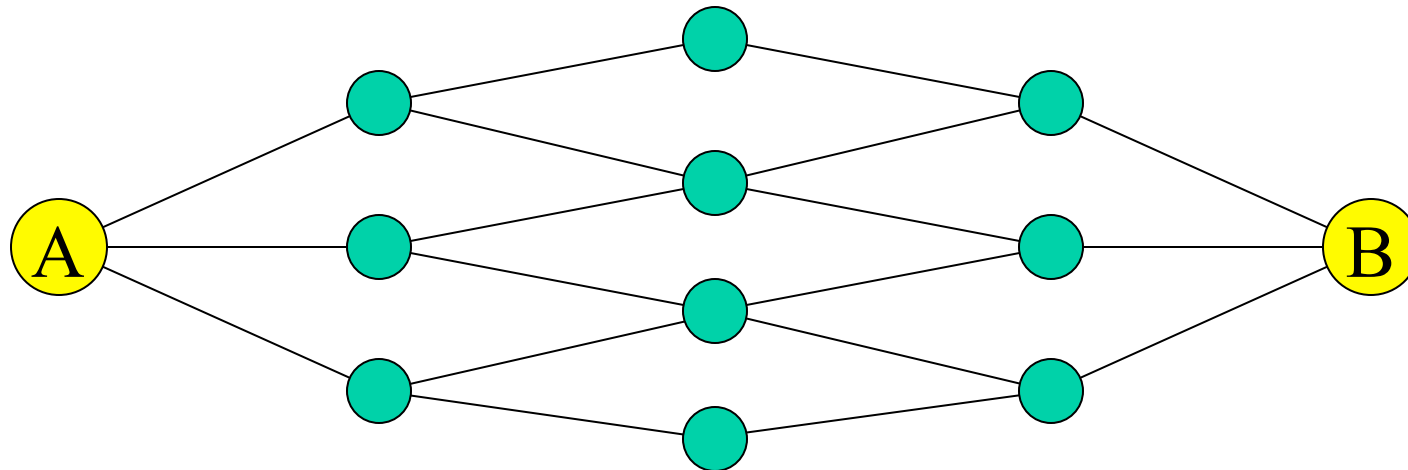
One way that professors found to detect students cheating on programming assignments is to study the distances between each pair of programs, and picking out those that had very similar code for further examination.

To get around this, groups of cheaters banded together and a couple of people wrote each part, and then the sections of the programs were mixed-and-matched, so that while two programs might look similar, they were not close enough to be sure of cheating.

What should the professors do now?

Example Problem

A company has all of its branches in a particular city all wired directly to each other to maximize the rate that data can be transferred around. If a critical operation needs to transfer information from A to B, what is the fastest it can be done given each connection has its own rate?

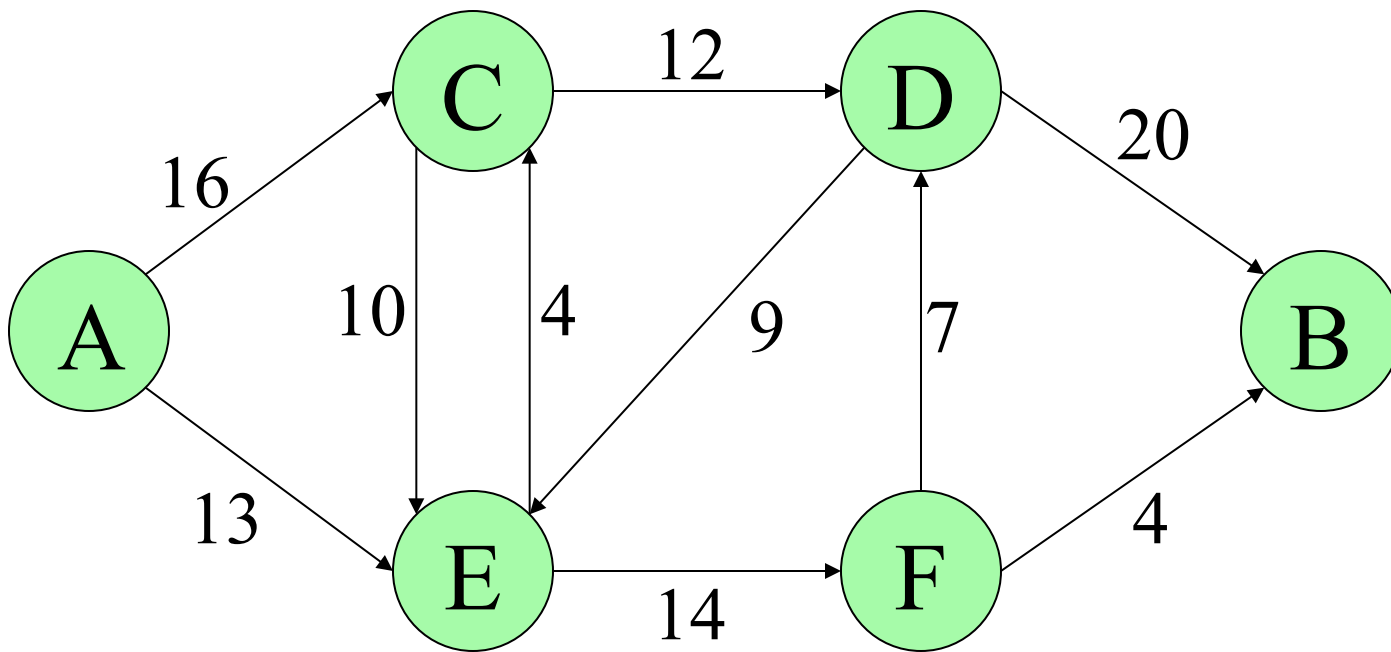


Network Flow

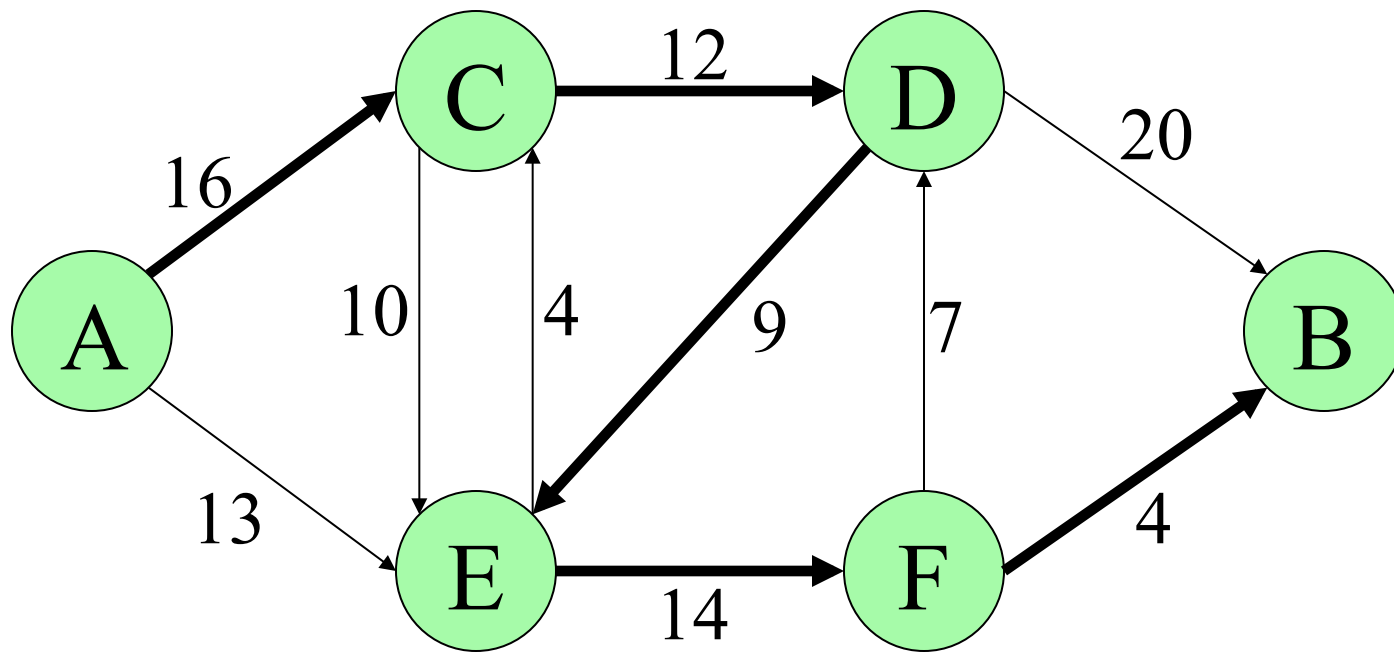
Inputs: A graph G , where each edge $e = (i, j)$ has a capacity c_e . A source node A , and a sink node B .

Problem: What is the maximum flow you can route from A to B while respecting the capacity constraint of each edge?

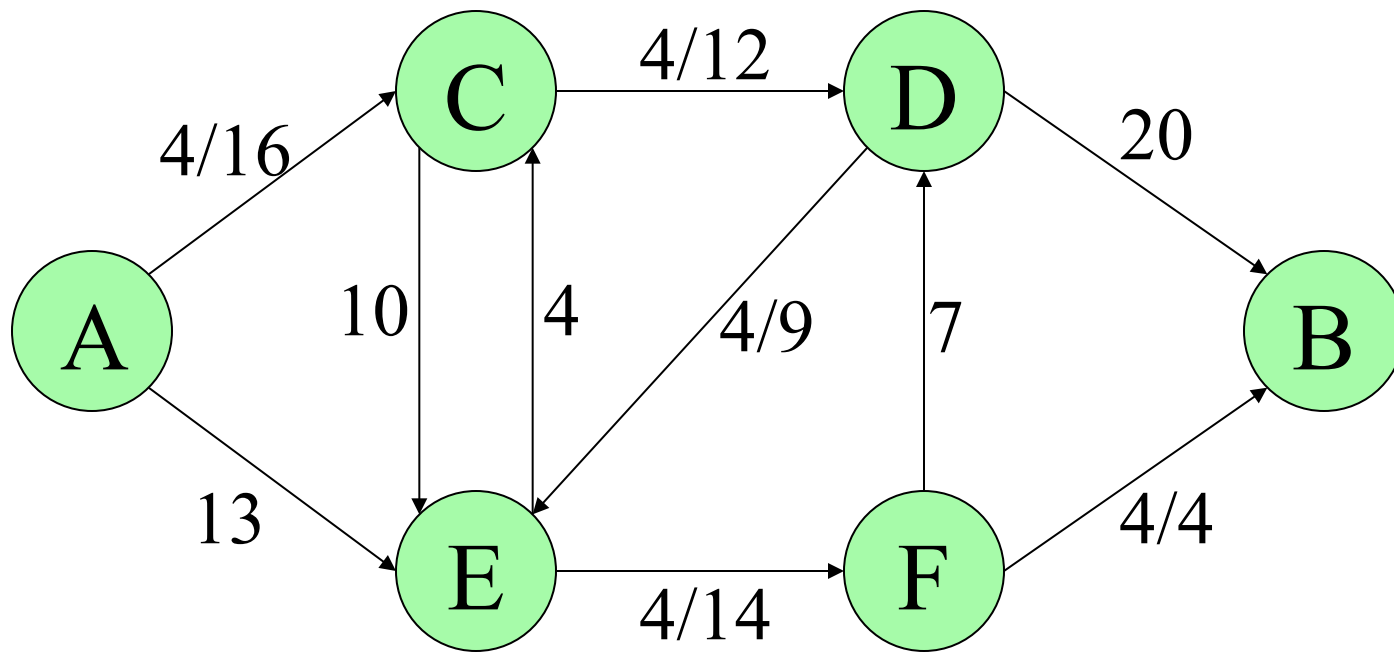
Example



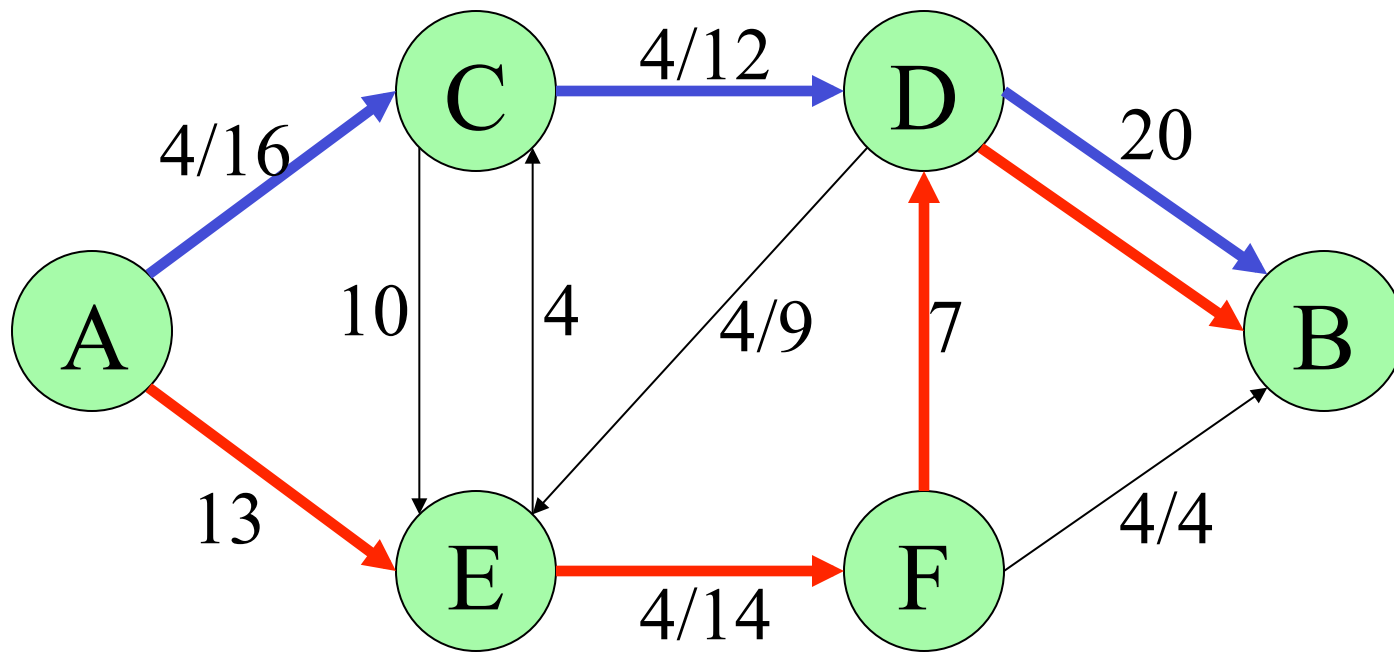
One Possible Path



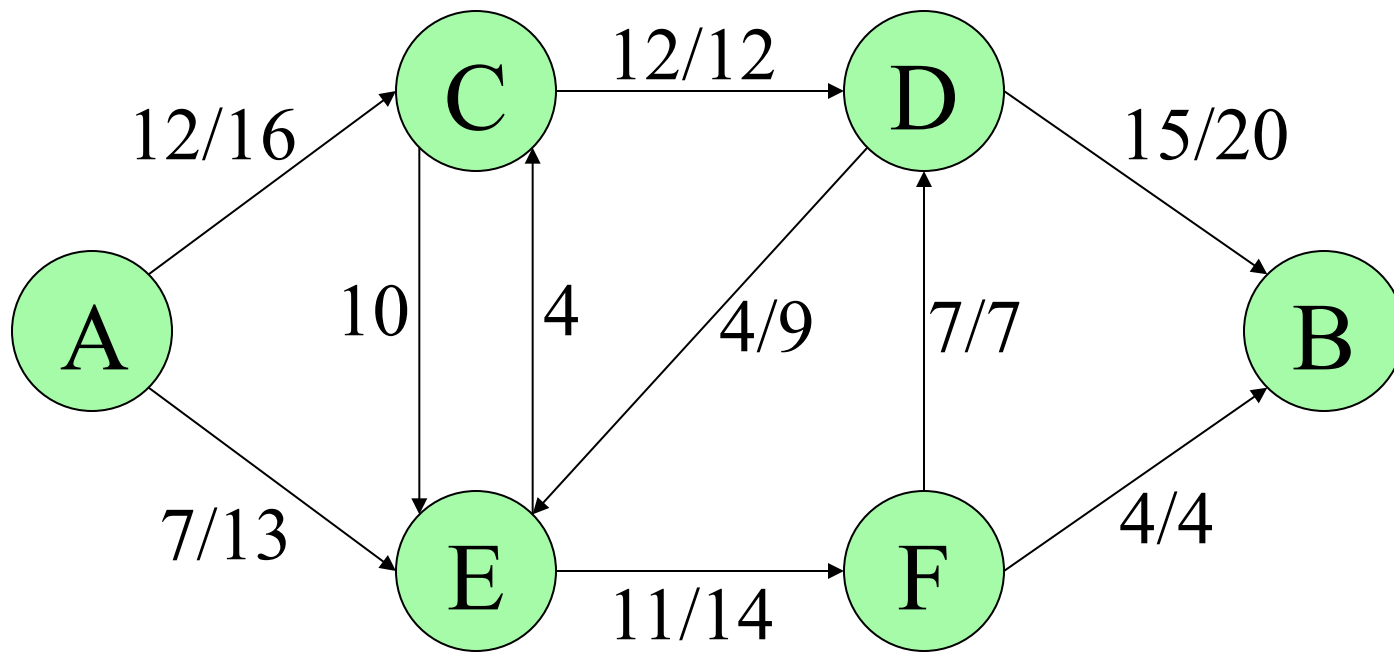
The capacity used so far...



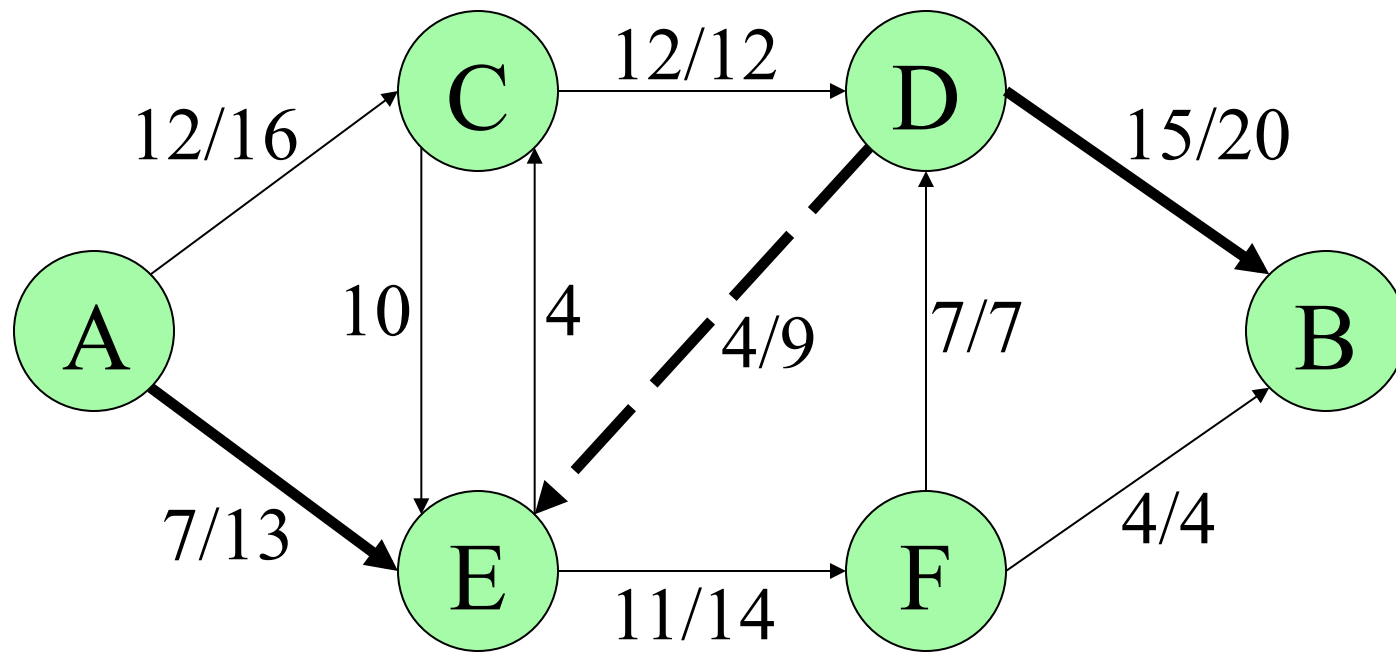
Two more paths...



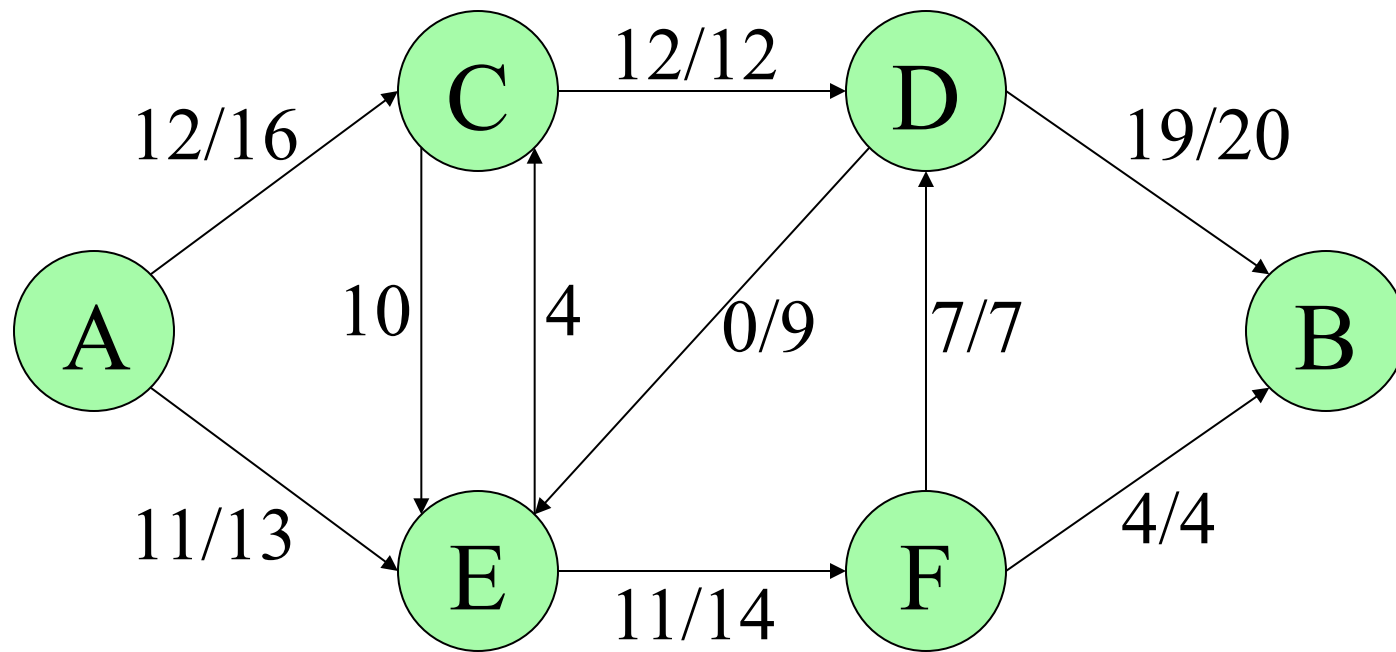
Final Capacities?



One final possible path...



And the resulting flow...



In any other words...

MegaCorp has expanded internationally, and needs to hire translators in a hurry. They found m people that can translate between pairs of languages, representing a total of n different languages. Of these, only k distinct key languages are needed by the company and they need to be able to translate between all combinations of them.

The bottom line is that they want to hire as few translators as possible that, through some chain of translations, can convert between any pair of the key languages. They don't care how long such a chain is, as long as it exists.