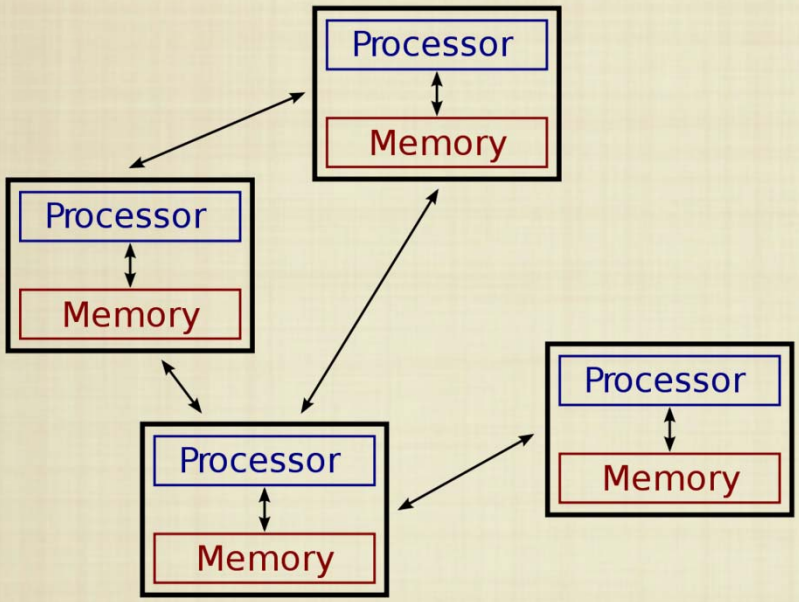
Introduction To Graphs and Networks

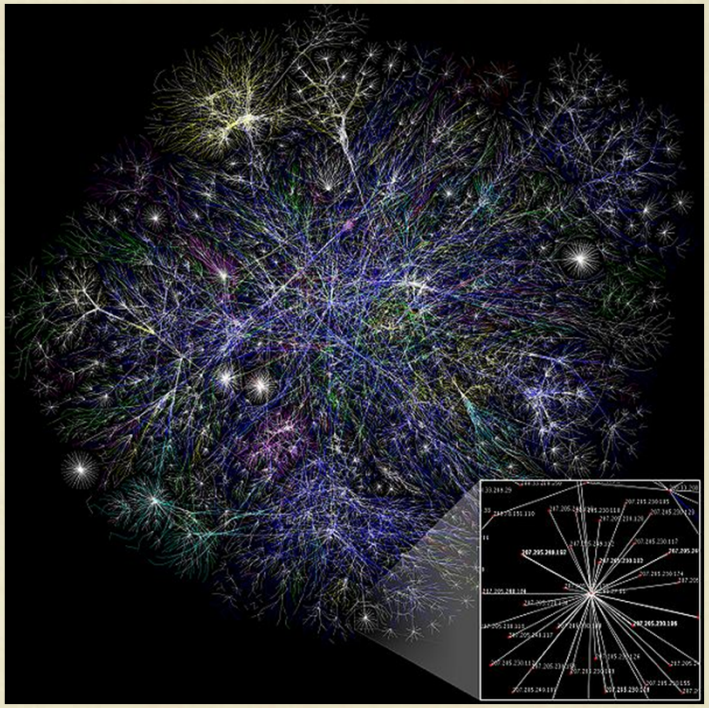
Fall 2013

Carola Wenk

What is a Network?

- We have thought of a computer as a single entity, but they can also be connected to one another.





Internet

What are the advantages of connecting computers together?
Communication and Computation.

Communication and Computation

- "A network of such [computers], connected to one another by wide-band communication lines [which provided] the functions of present-day libraries together with anticipated advances in information storage and retrieval and [other] symbiotic functions."
  —J.C.R. Licklider, 1960, *Man-Computer Symbiosis*

- In 2010, there were 107 trillion emails sent; there are 1.88 billion email users.

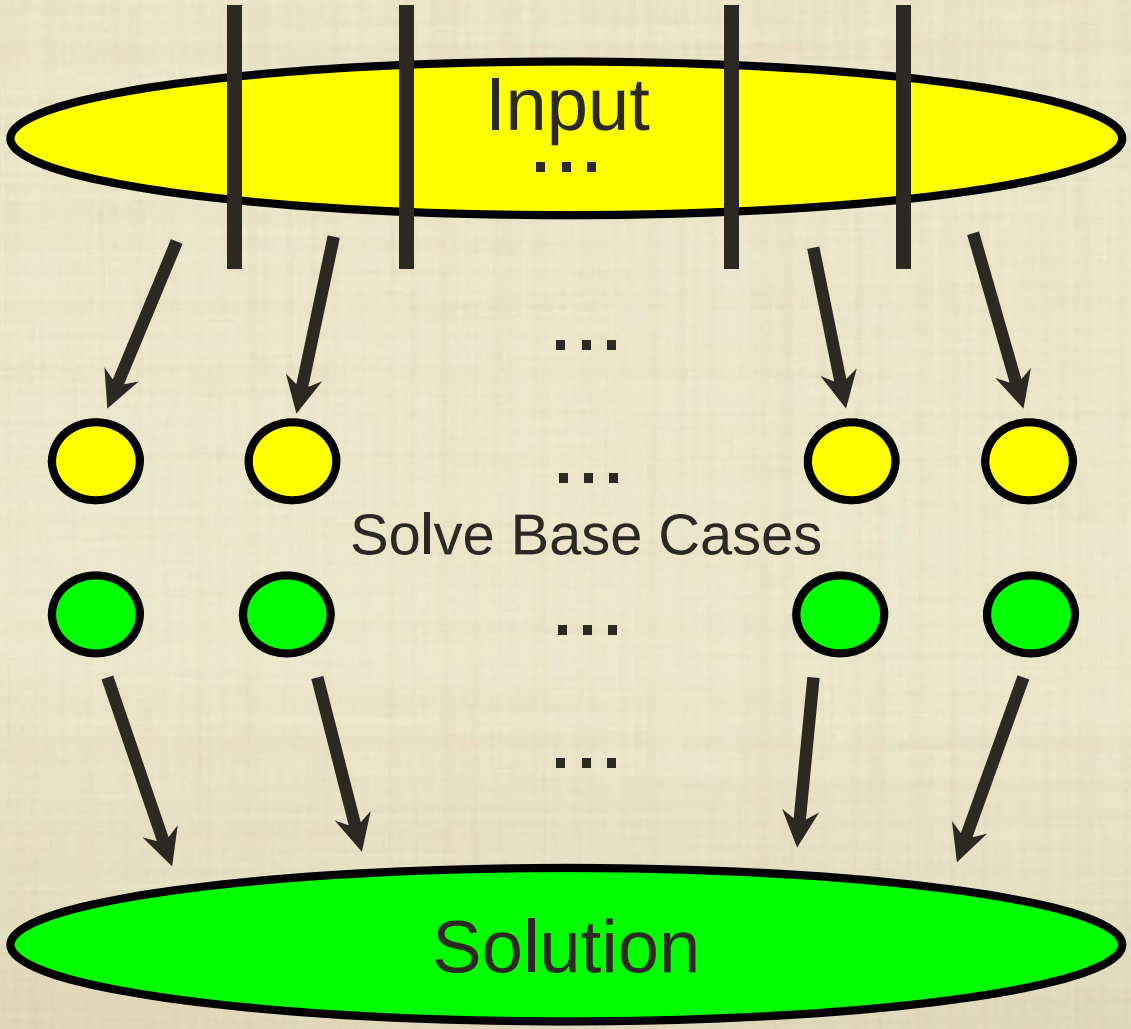
- Communication enables long-distance computation; in fact this is one of the reasons networking was initially studied.

“Dumb” Terminal

Mainframe

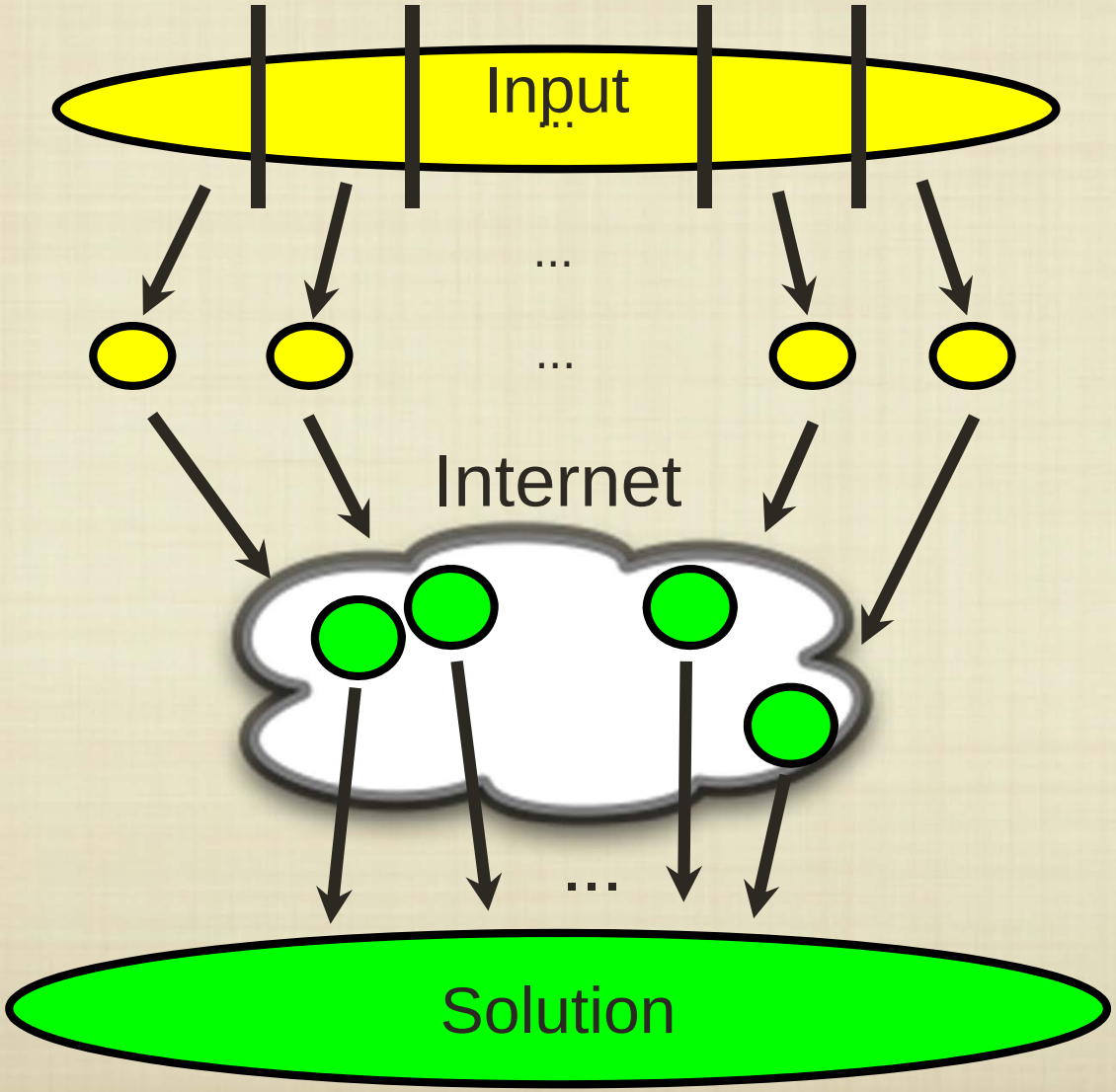
Communication and Computation

- Communication also enables coordinated computation; if a task can be split up into smaller parts, we can solve each part simultaneously.

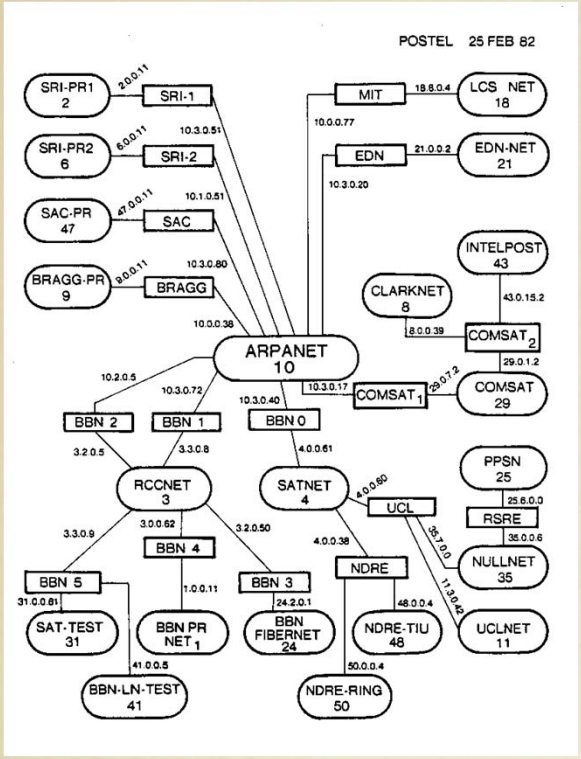


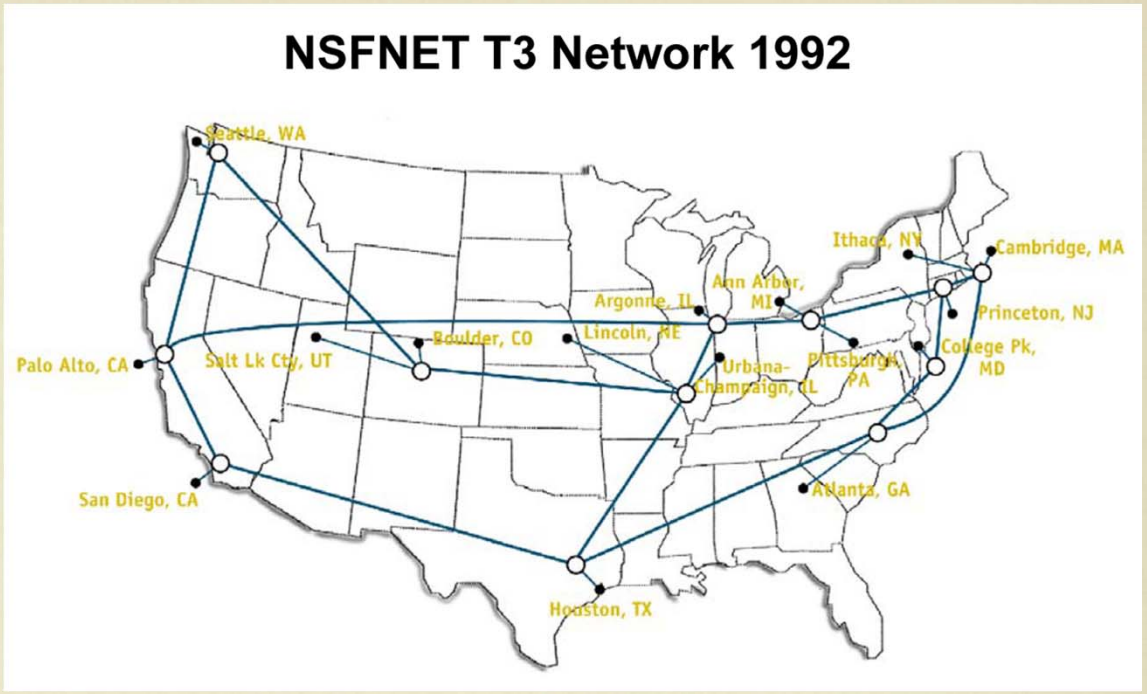
Communication and Computation

- Communication also enables coordinated computation; if a task can be split up into smaller parts, we can solve each part simultaneously.



History of the Internet

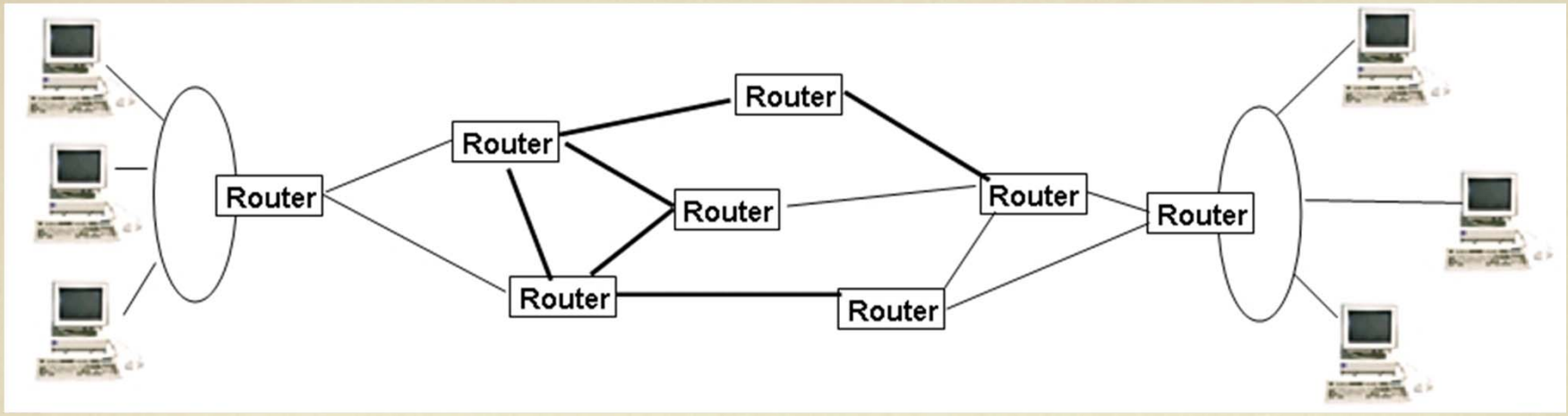




The Internet was initially developed as a military research project in the 1970s. The goal was to connect computers across a wide geographic area with very high “availability”.

Communication on the Internet works by a protocol in which “packets” of information are transmitted. Each packet is routed from source to destination.

Internet Routing



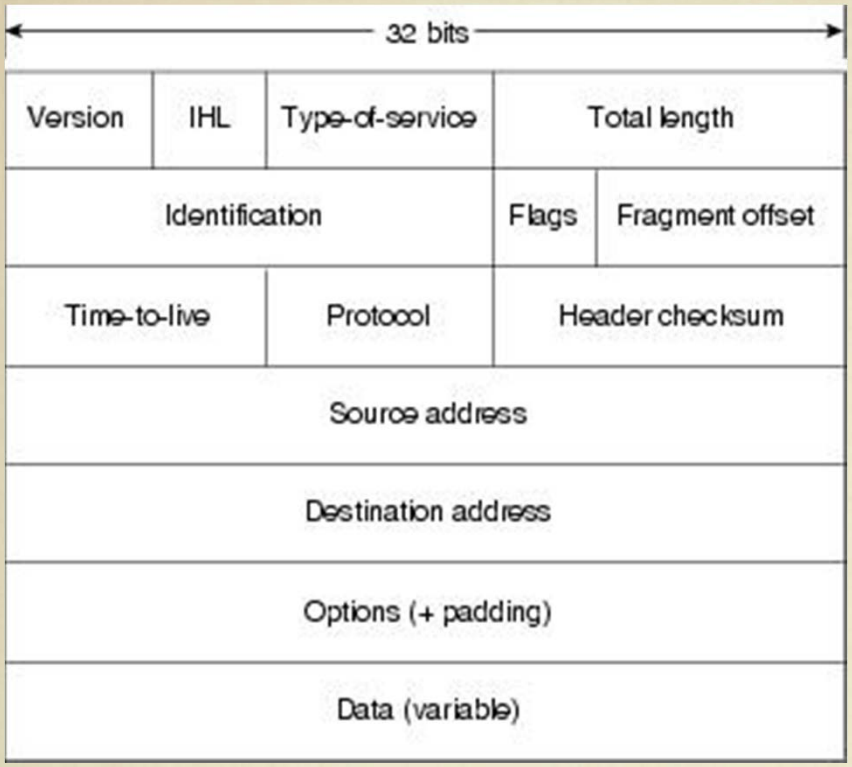
IP Address

IP Address

Each device connected to the Internet has an “IP” address, which is just a 32-bit number. The route of each transmitted packet is determined as it moves along its path.

Due to the ubiquity of connected devices the current protocol for naming devices will run out of addresses soon - IPv6 uses 128-bit addresses.

Internet Routing



TCP/IP = “Transmission Control Protocol over Internet Protocol”

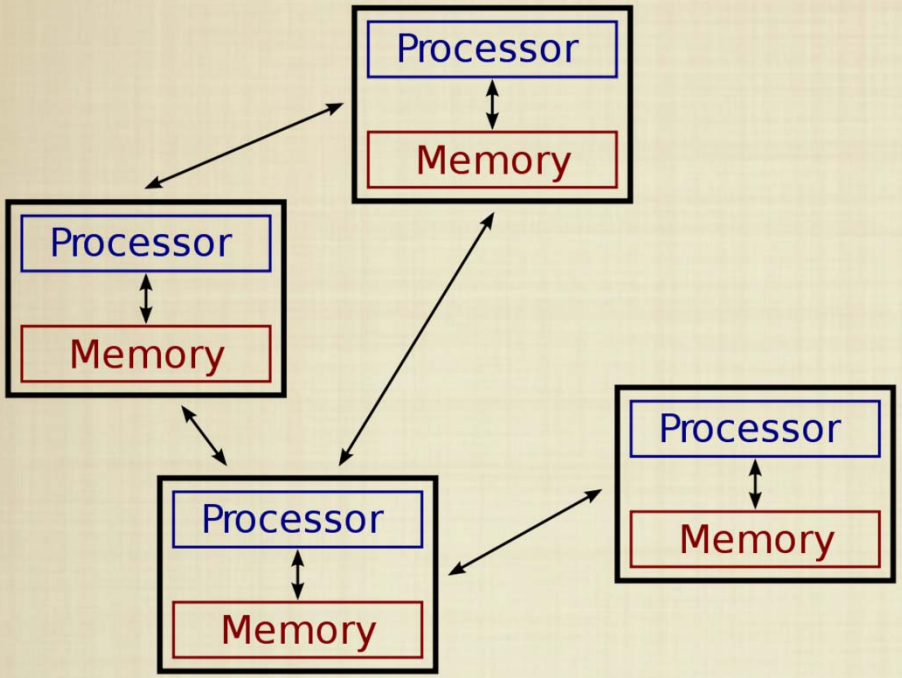
TCP/IP was designed in the early 1970s to be used in a “packet switching” network.

It was initially tested on a 4-node network and now is used on millions of computers.

TCP/IP works at a low level to break up communication tasks into packets, and manages how they propagate from source to destination.

Internet routers do most of the work in guiding TCP/IP packets; because of this distribution of work, there are often many paths that can be taken from a source to a destination.

Network Abstraction

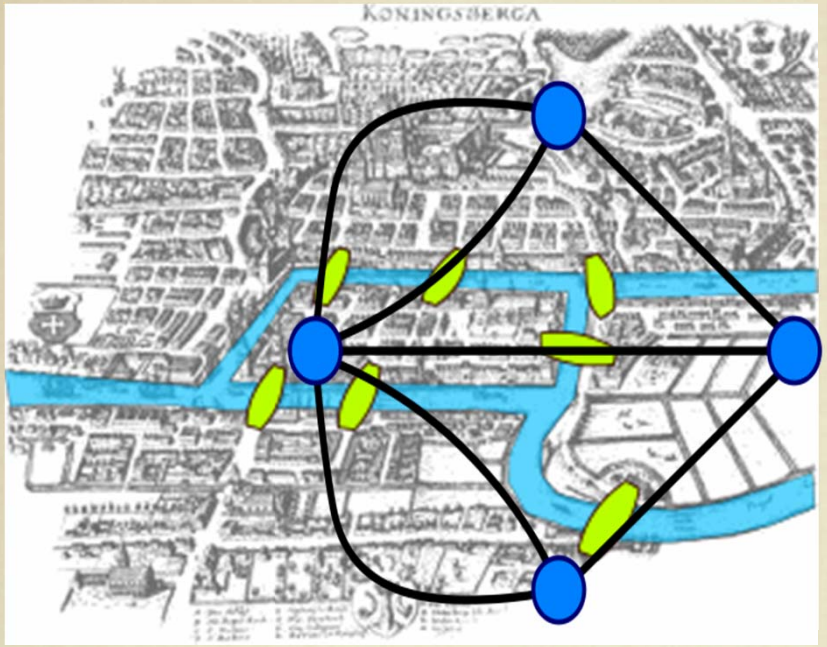


The Internet is modeled as a set of “nodes” that are connected by “links”.

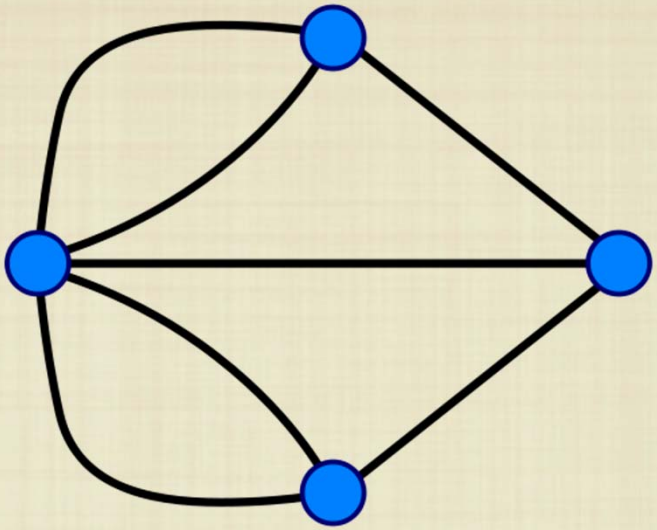
For the purposes of communication, we only care about how we are able to get from one node to another.

How can we determine the best path from point A to point B in a large network? Can this be done efficiently?

Bridges of Königsberg



Map of Konigsberg

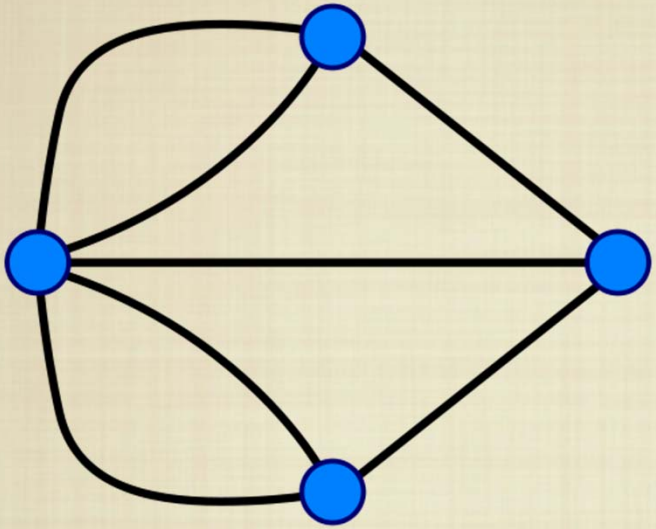


The Abstraction

In 1735, Leonhard Euler posed the question of whether there was a path that crossed every bridge exactly once. Is there such a path?

Abstracting away from the specific city of Königsberg allows us to answer this question for any city. Do you see a rule?

Graphs: Abstract Networks



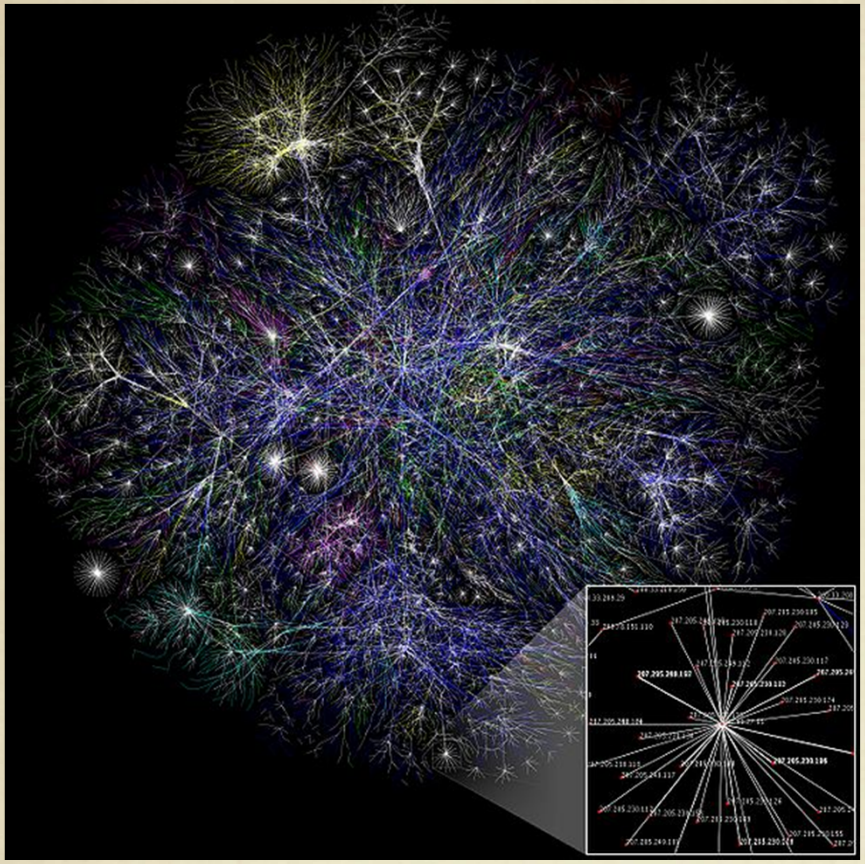
A graph G consists of a set of vertices that are connected by edges. We write G=(V,E), where V is the set of vertices and E is the set of edges.

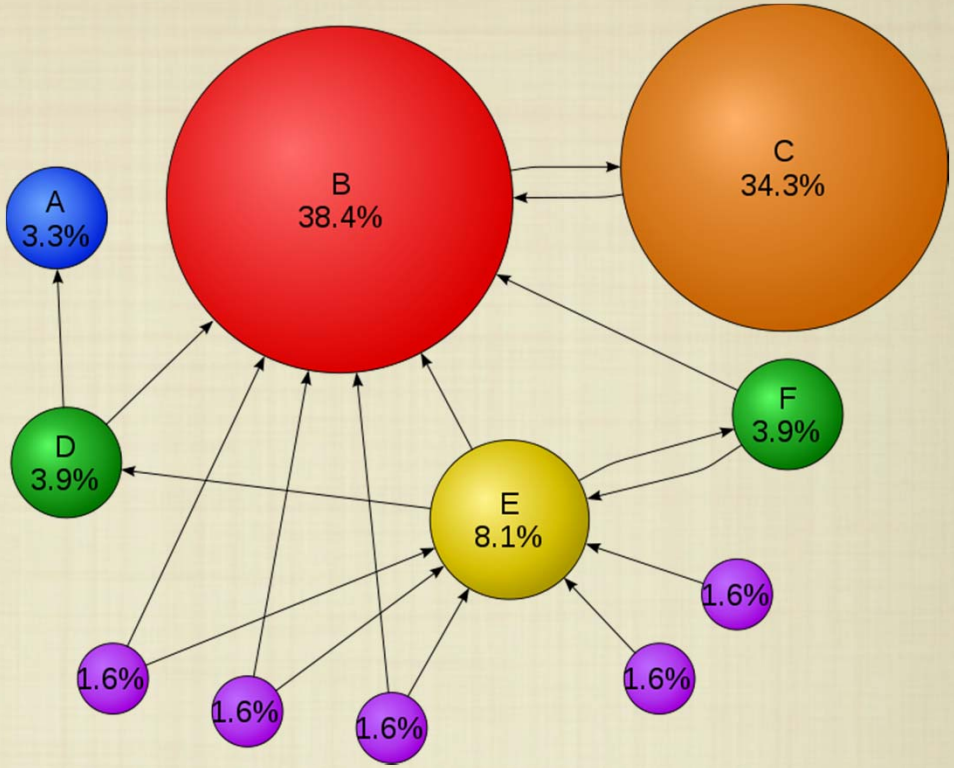
What is the maximum number of edges a graph can have (i.e., every pair of vertices has one edge)?

Generalizing the Königsberg problem, a graph has an Eulerian path if and only if all but two vertices are involved in an even number of edges.

Graphs can represent any abstract relationship between pairs of objects, and are particularly useful in analyzing computer networks.

The Internet Graph



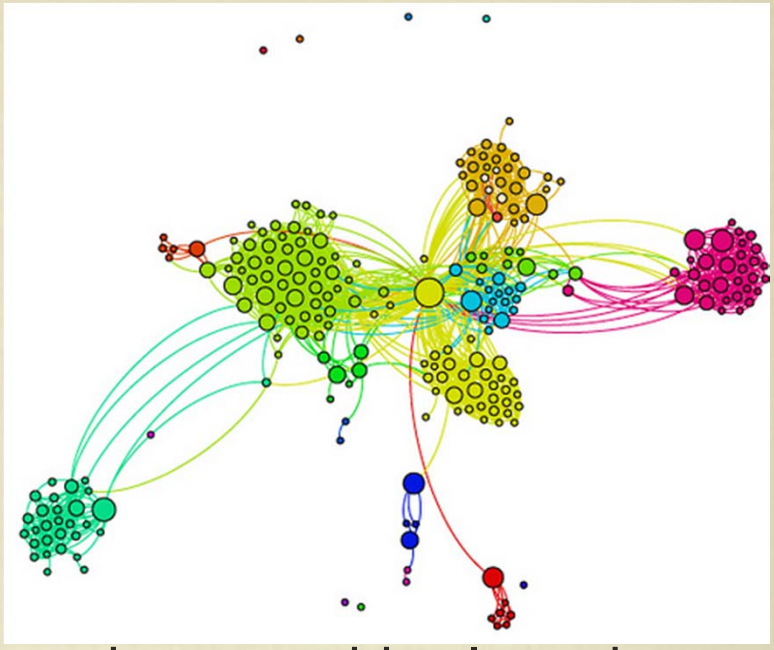


Nearly every profitable and/or useful application on the Internet must use algorithms on graphs.

Example: Google PageRank analyzes the web-graph to determine which pages are most relevant to search queries.

Virtual Networks

worldwide facebook graph



demographic clustering

- Social networks aren’t formed from physical hardware, but by publicly-stated relationships.
- By analyzing ‘friendships’, social networking companies can gather information for targeted advertising.

Analyzing Graphs

- Two fundamental questions on graphs:
  - Can we get from any vertex to any other vertex?
  - What is the shortest path from between a given pair of vertices?
- In a computer network, these questions give us a basic idea of where communication is possible, and how quickly it can be accomplished.
- In a social network, these questions tell us how “social” a particular population is, and how well two individuals know one another.

Road networks

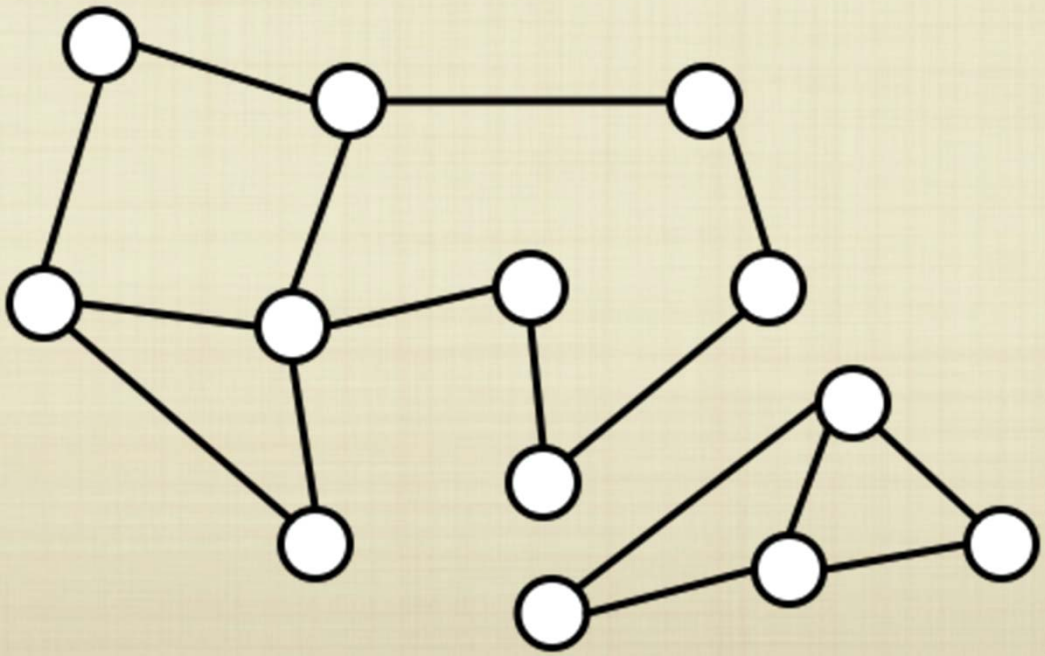
- Graph: Vertices are intersections, edges are road segments.
- What is the shortest path from A to B?

Navigation systems answer shortest path queries based on travel times on road segments.

Two Networking Questions

- Connectivity: Given a graph, is any vertex reachable from any other vertex?

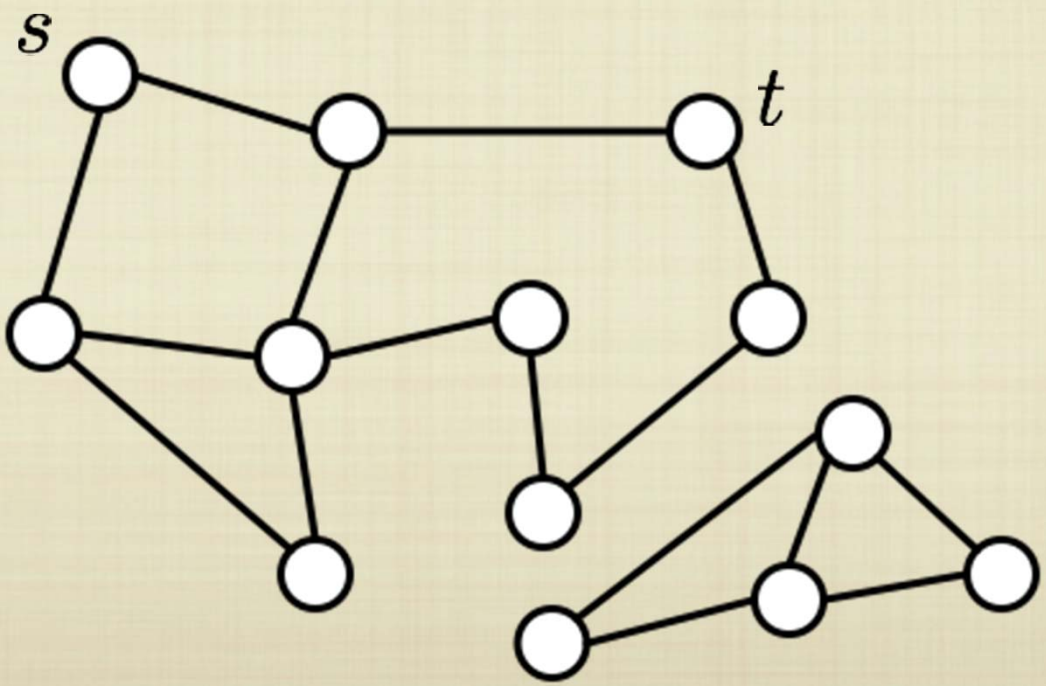
- Shortest Paths: Given a graph and two vertices, what is the shortest path between the two?



Two Networking Questions

- Connectivity: Given a graph, is any vertex reachable from any other vertex?

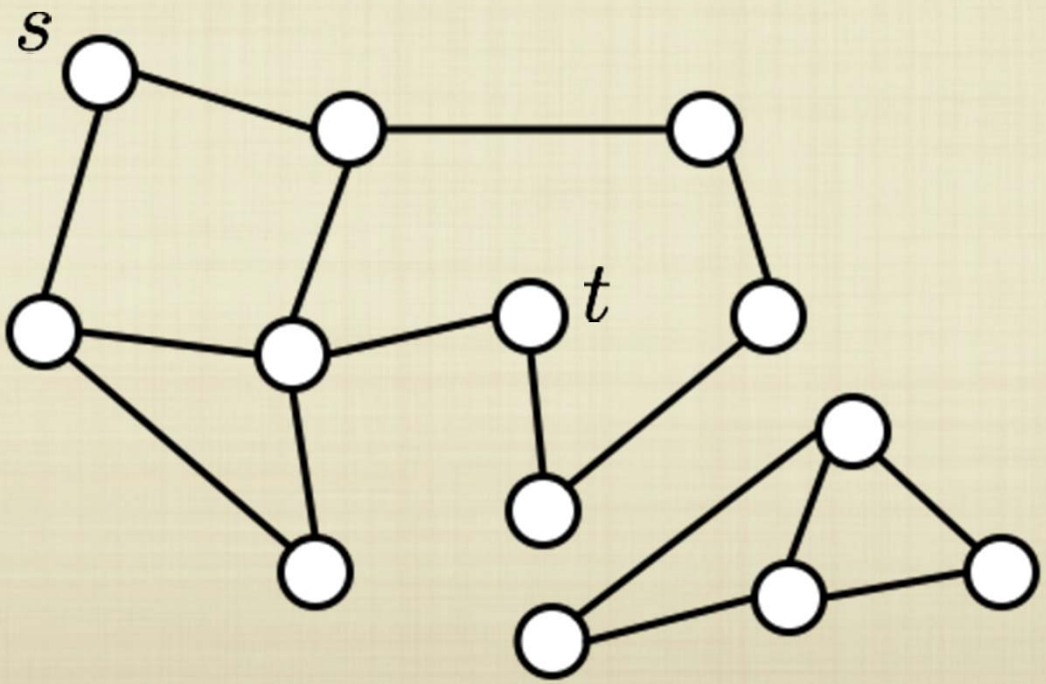
- Shortest Paths: Given a graph and two vertices, what is the shortest path between the two?



Two Networking Questions

- Connectivity: Given a graph, is any vertex reachable from any other vertex?

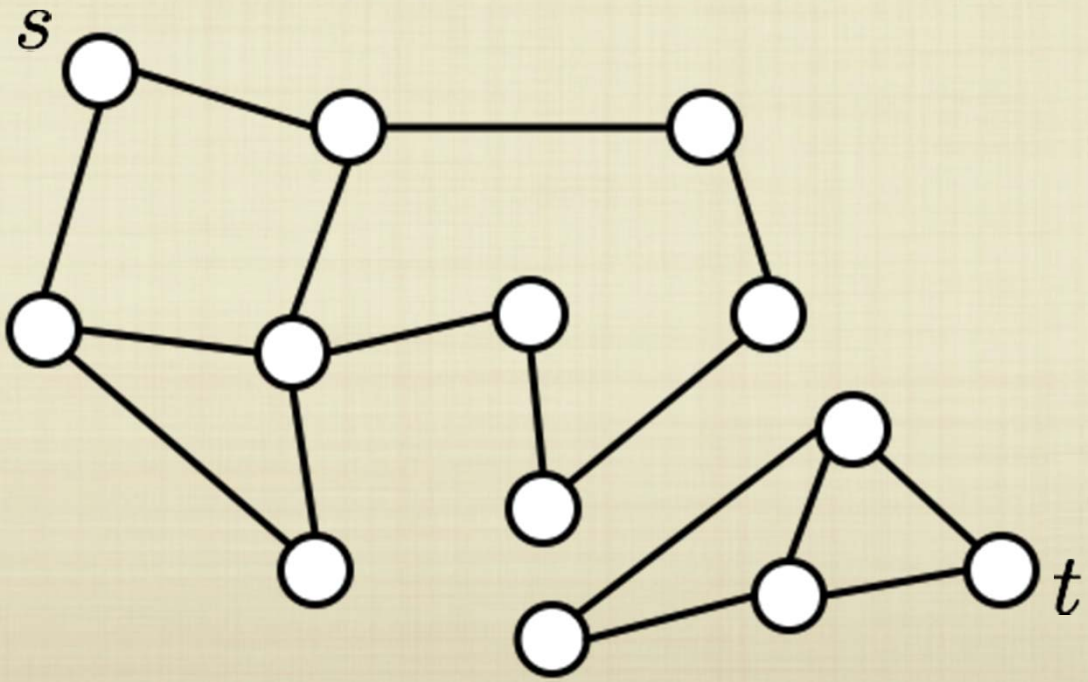
- Shortest Paths: Given a graph and two vertices, what is the shortest path between the two?



Two Networking Questions

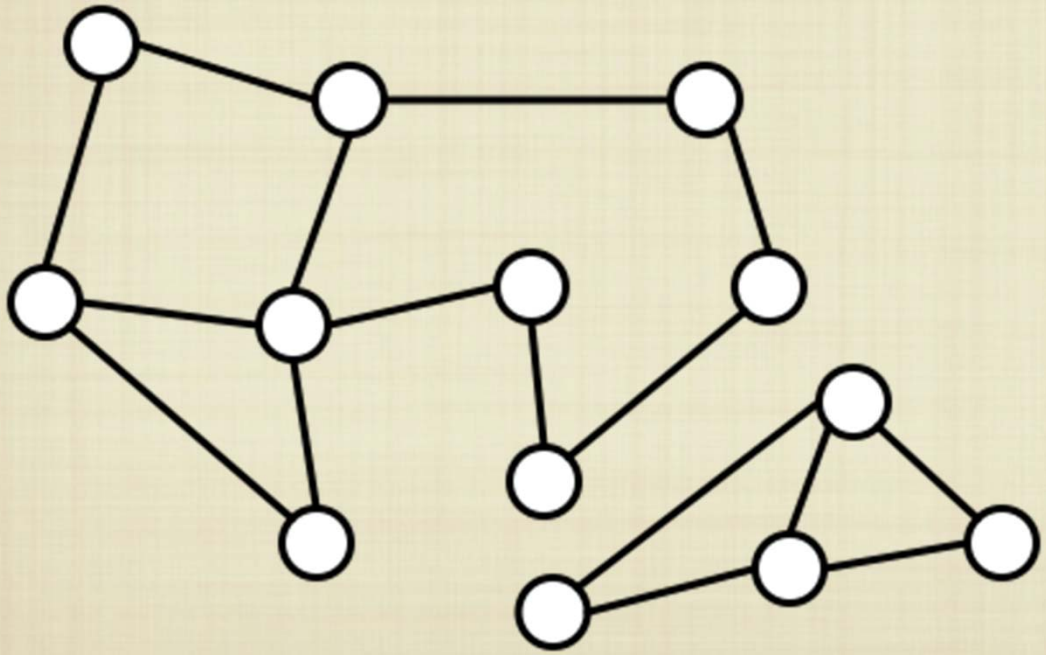
- Connectivity: Given a graph, is any vertex reachable from any other vertex?

- Shortest Paths: Given a graph and two vertices, what is the shortest path between the two?



Graph Connectivity

- Given a graph as input (vertices and edges), we want to produce as output the number of “connected components” of the input graph.



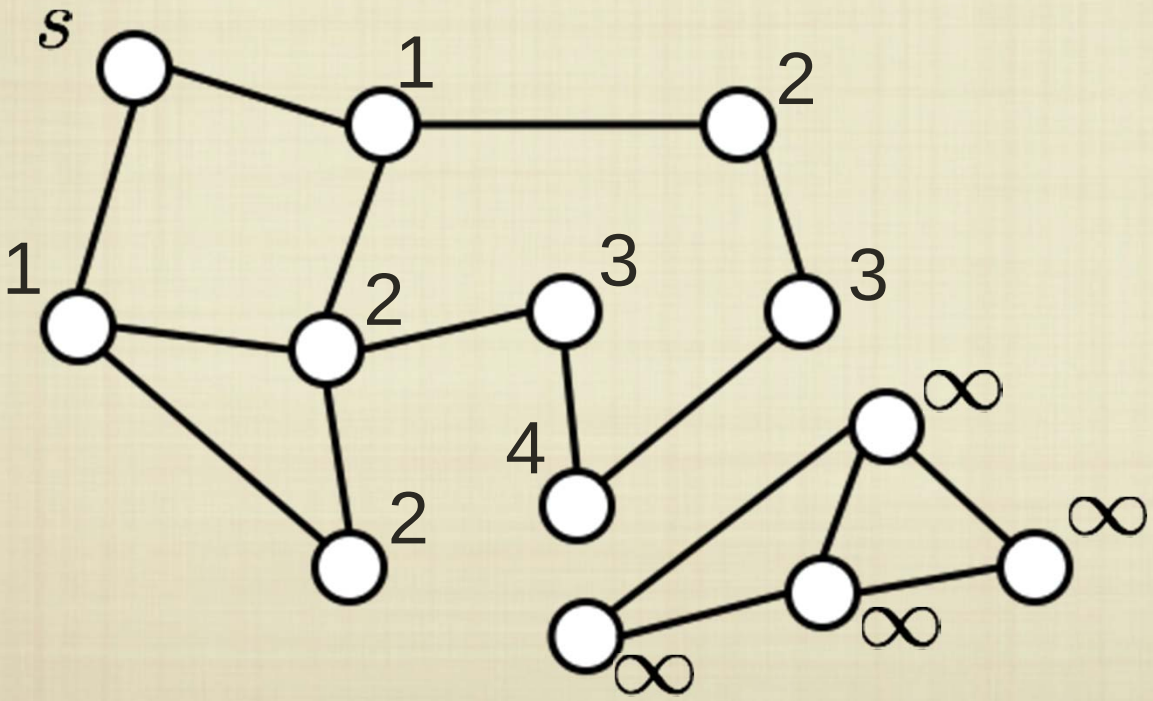
$G = (V, E)$

2 Connected Components

“Single-Source” Shortest Paths

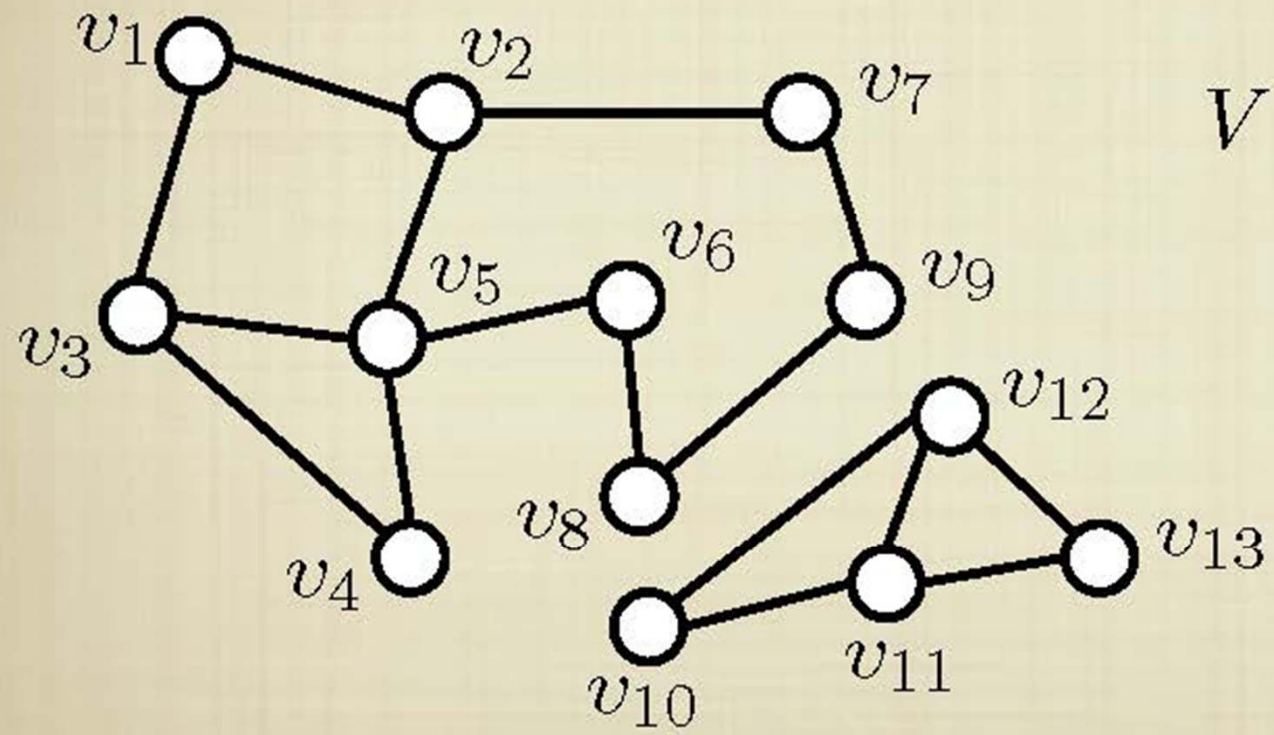
- Given a graph as input (vertices and edges) and a vertex s, we want to produce as output the shortest paths from s to all other vertices.

$$G = (V, E)$$



Graph Representations

Both of these problems require a graph as input. How do we represent the vertices and edges? How can we check if an edge is in the graph?



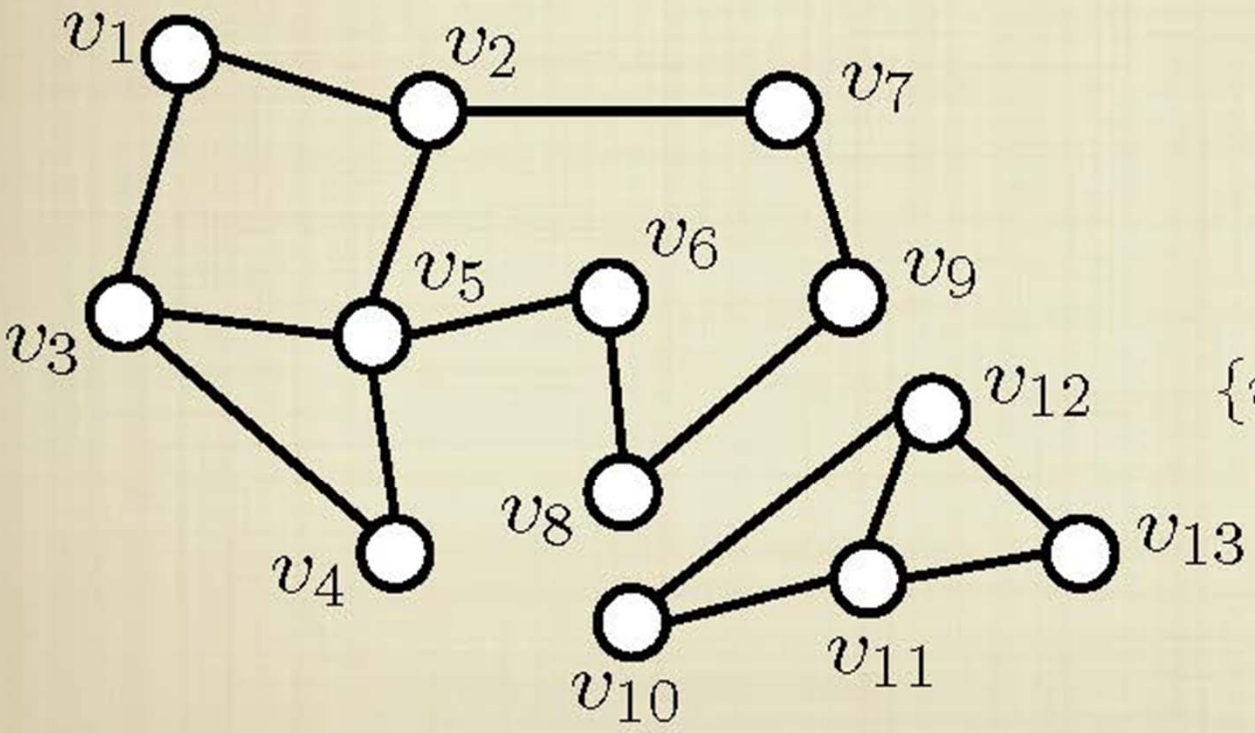
$$G = (V, E)$$

$$V = \{v_1, v_2, \ldots, v_{13}\}$$

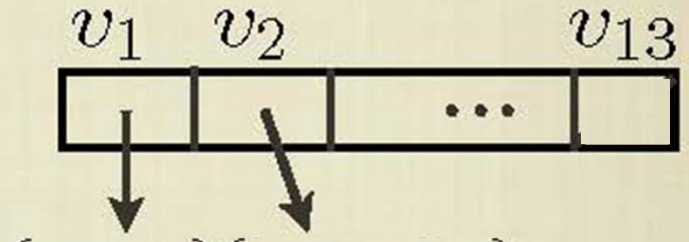
$$E = \{\{v_1, v_2\},$$
$$\{v_1, v_3\},$$
$$\{v_2, v_5\},$$
$$\ldots,$$
$$\{v_{12}, v_{13}\}$$

Graph Representations

Both of these problems require a graph as input. How do we represent the vertices and edges? How can we check if an edge is in the graph?



$G = (V, E)$



"Adjacency List"

With an adjacency list, we can store all edges and examine any edge at any vertex.

Lists and Dictionaries

Both of these problems require a graph as input. How do we represent the vertices and edges? How can we check if an edge is in the graph?

```
A = [5, 20, 99]
print A[0]
print A[0]+A[1]
A.append(66)

B = {'a':99, 'b':2, '3':1, 5:'x'}
print B.values()
print B.keys()
B['x'] = 5
B['dog'] = 'cat'
```

Python lists and dictionaries allow “random access”, and dictionaries can have arbitrary “keys”.

Lists and Dictionaries

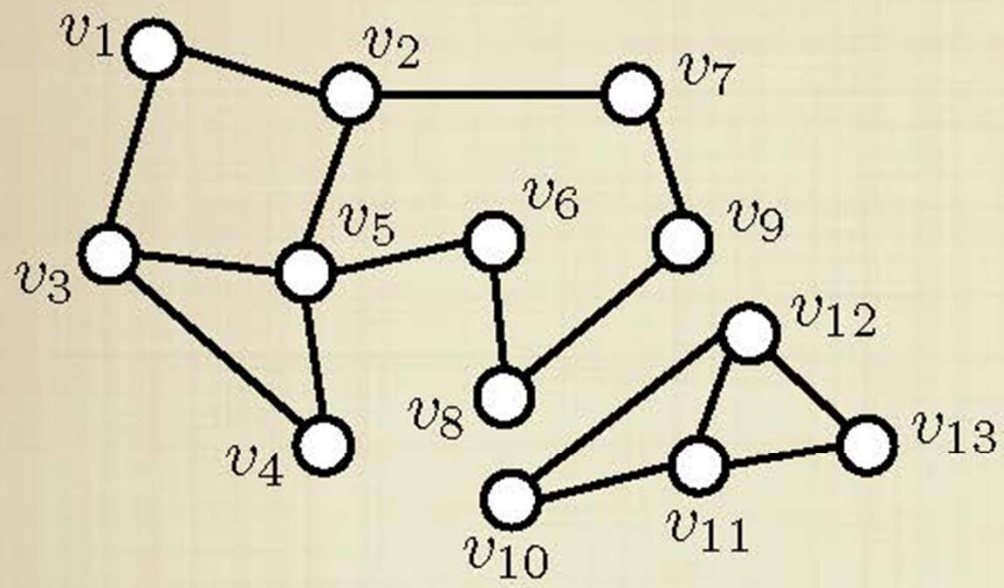
Both of these problems require a graph as input. How do we represent the vertices and edges? How can we check if an edge is in the graph?

```
A = [5, 20, 99]
print A[0]
print A[0]+A[1]
A.append(66)

B = {'a':99, 'b':2, '3':1, 5:'x'}
print B.values()
print B.keys()
B['x'] = 5
B['dog'] = 'cat'
```

Does one of these look like a graph?

Graphs as Dictionaries

Both of these problems require a graph as input. How do we represent the vertices and edges? How can we check if an edge is in the graph?



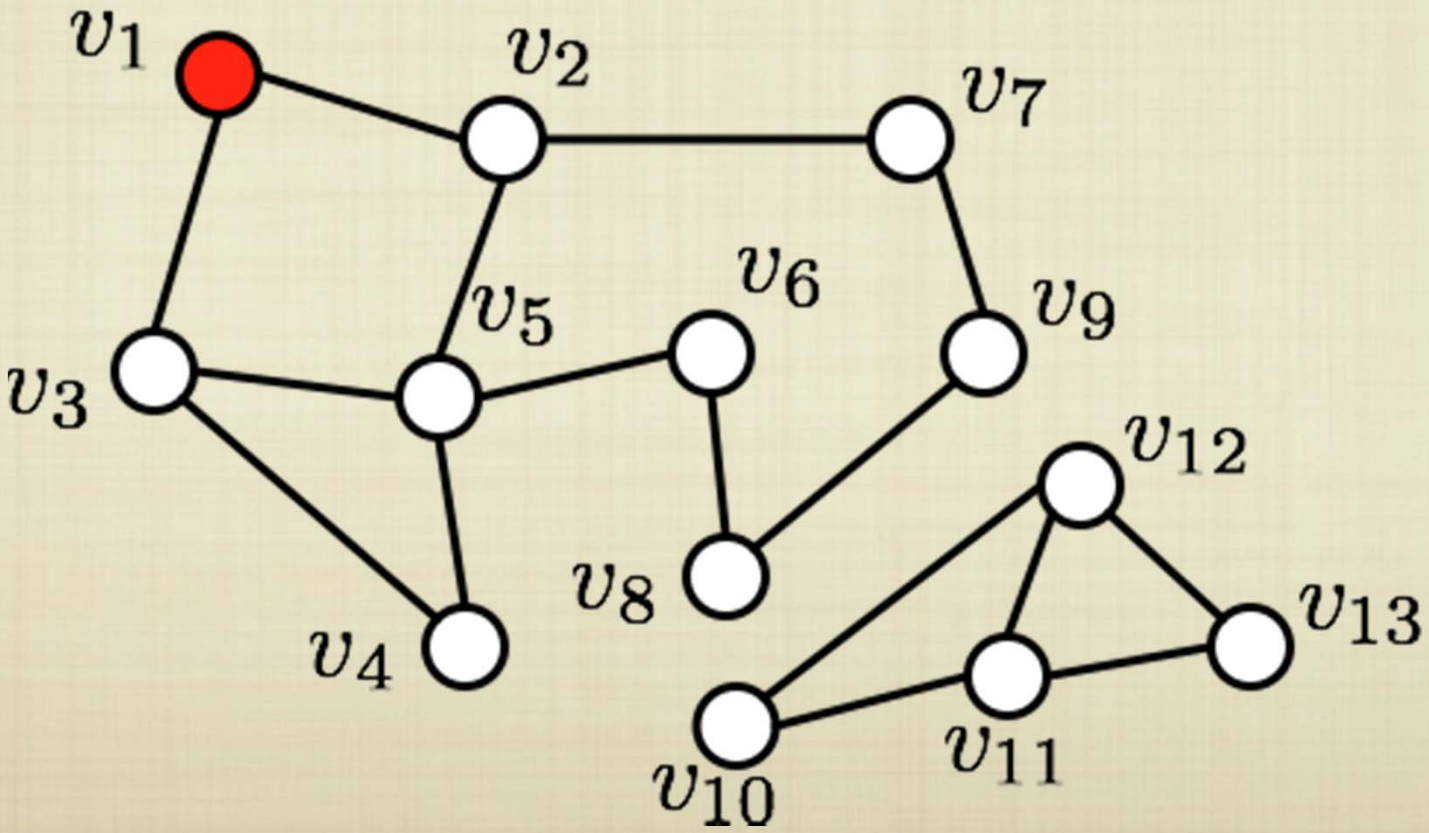

```
G = {1:[2,3], 2:[1,5,7],
3:[1,4,5], 4:[3,5], 5:
[2,3,4,6], 6:[5,8], 7:
[2,9], 8:[6,9], 9:[7,8],
10:[11,12], 11:[10,12,13],
12:[10,11,13], 13:[11,12]}

print G.keys()
print G.values()
```

The dictionary in Python can implement an adjacency list.

Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

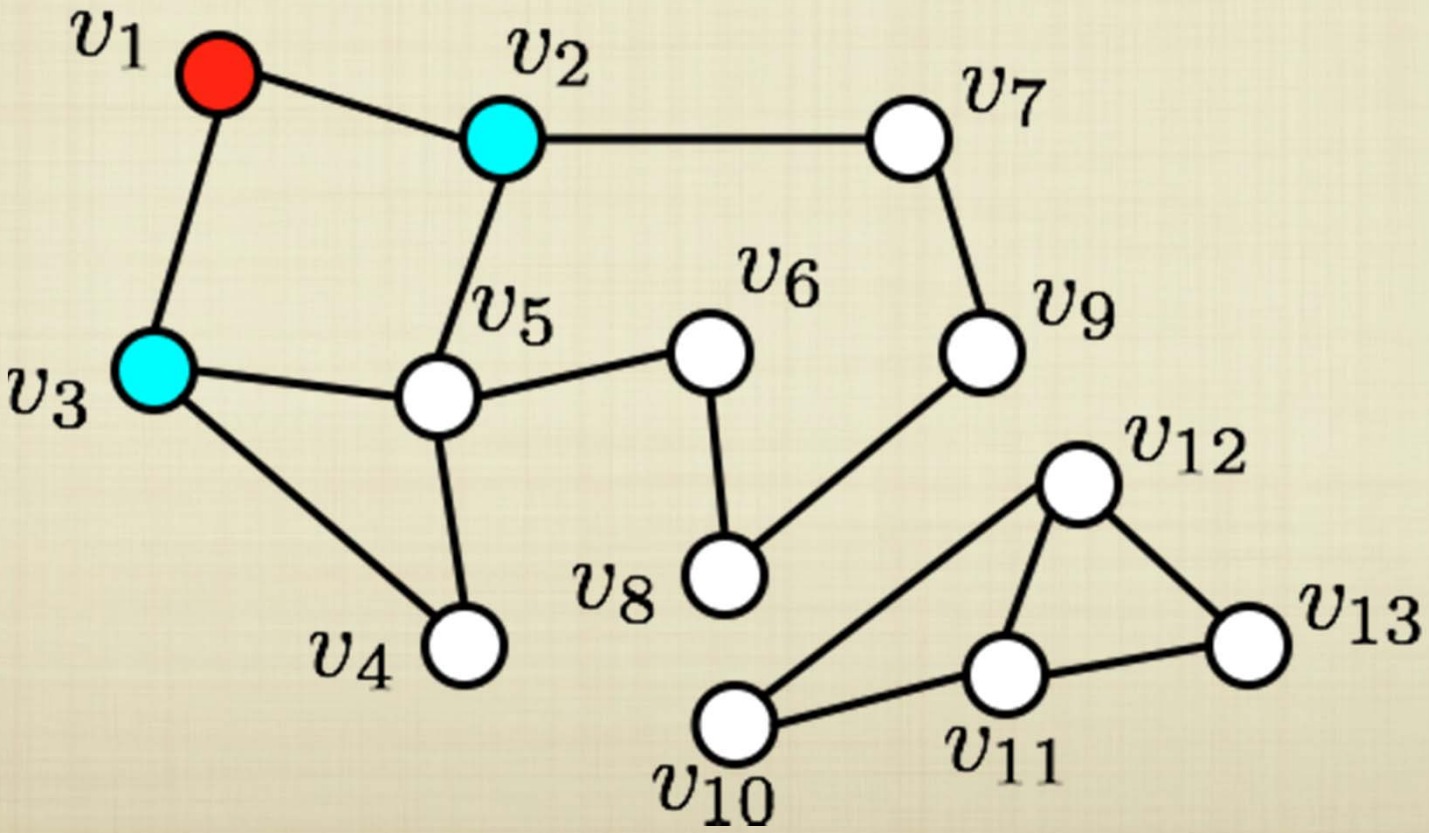
- How can we find out if a graph is “connected”?



Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

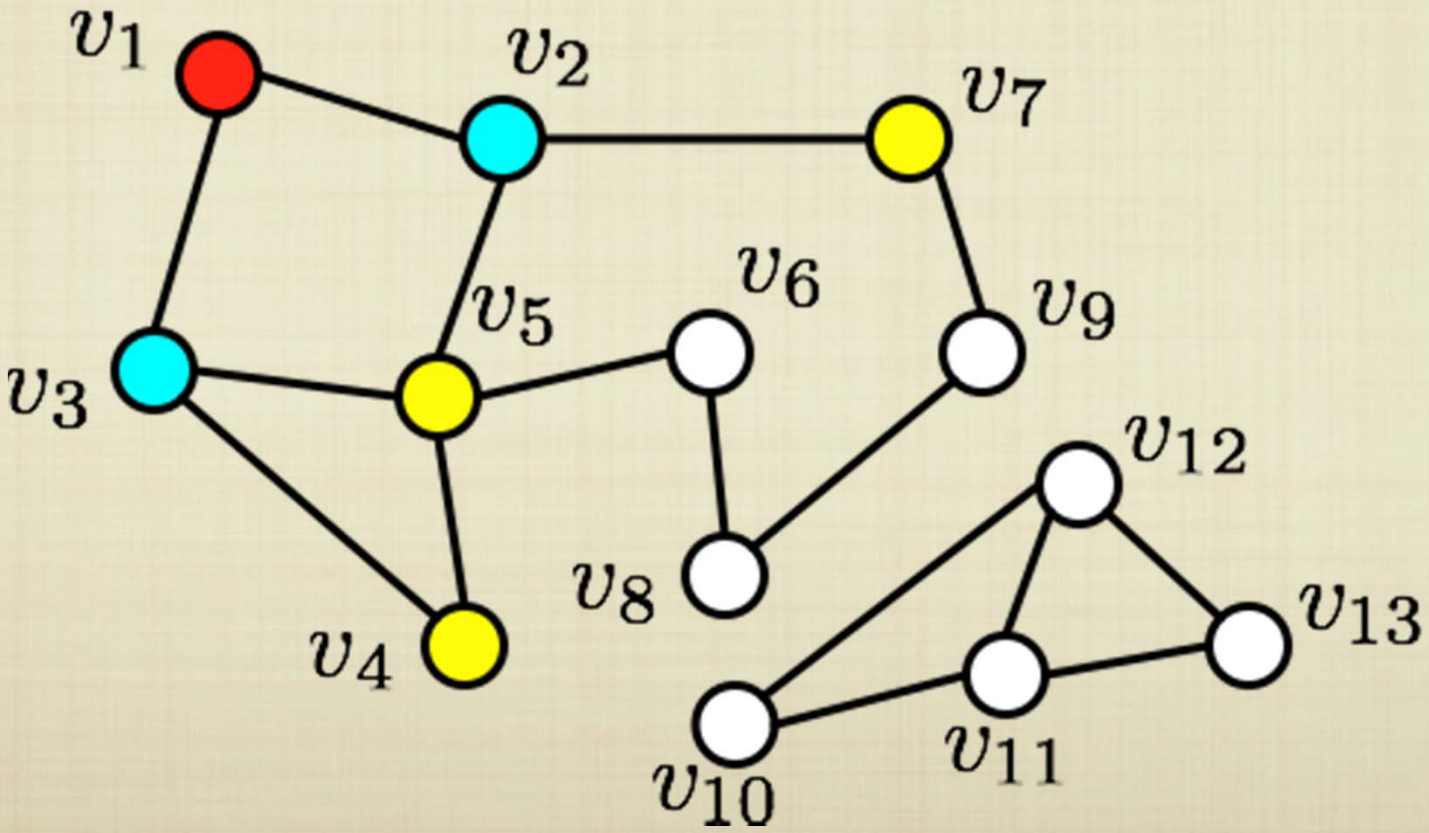
- How can we find out if a graph is “connected”?



Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

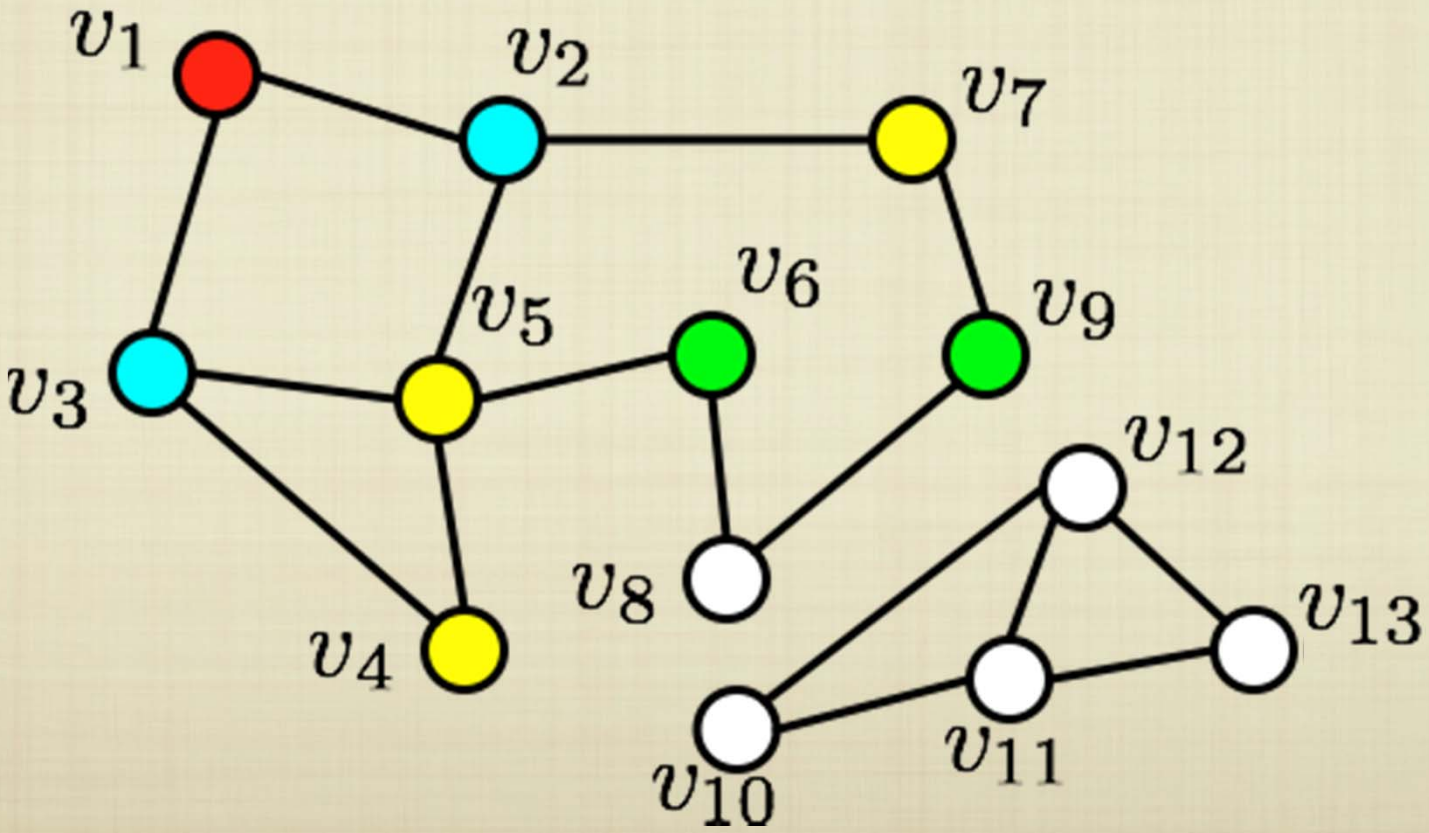
- How can we find out if a graph is “connected”?



Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

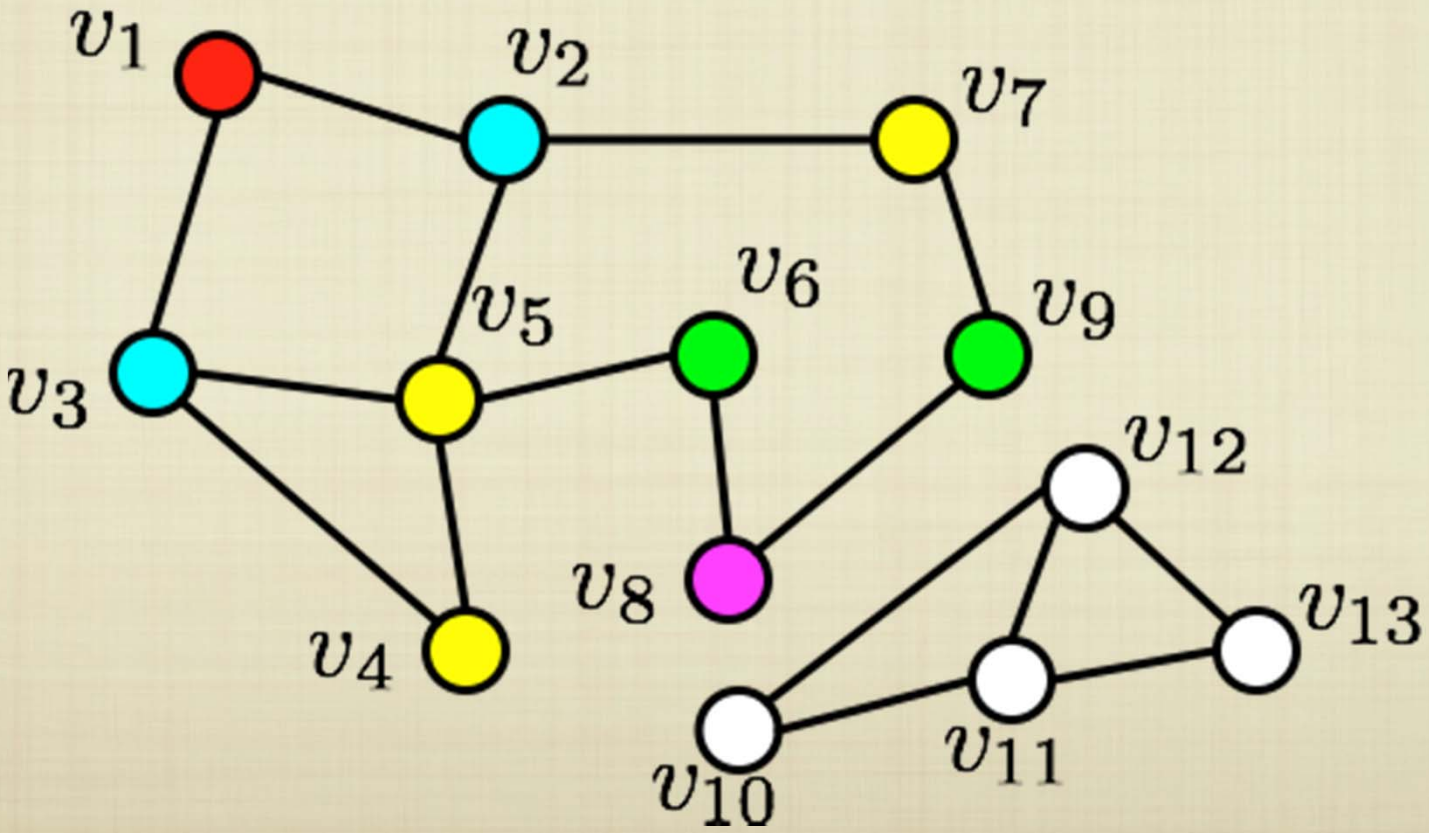
- How can we find out if a graph is “connected”?



Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

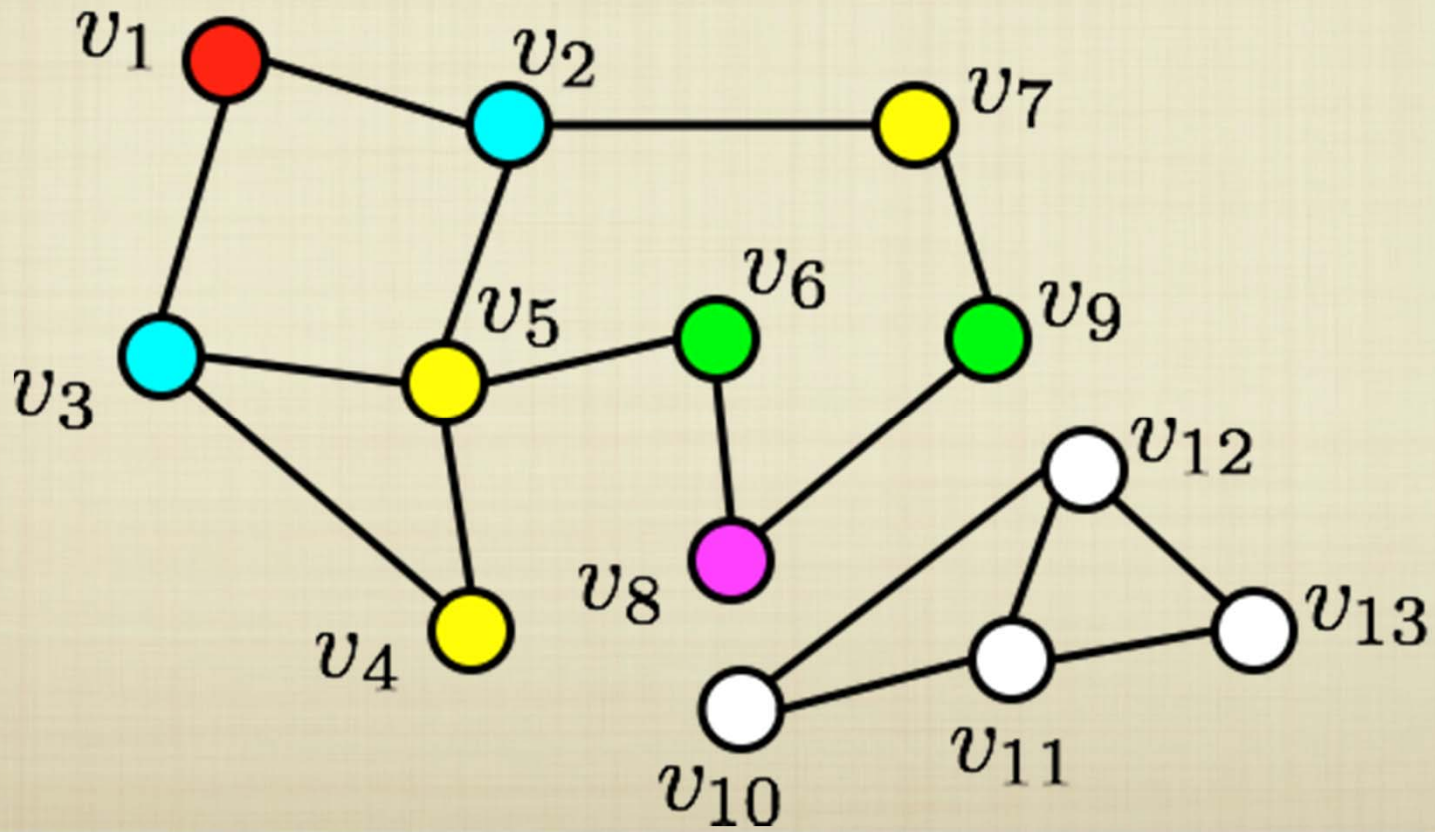
- How can we find out if a graph is “connected”?



Breadth-First Search

- Now that we have a way to actually implement graphs, how do we solve our two basic problems?

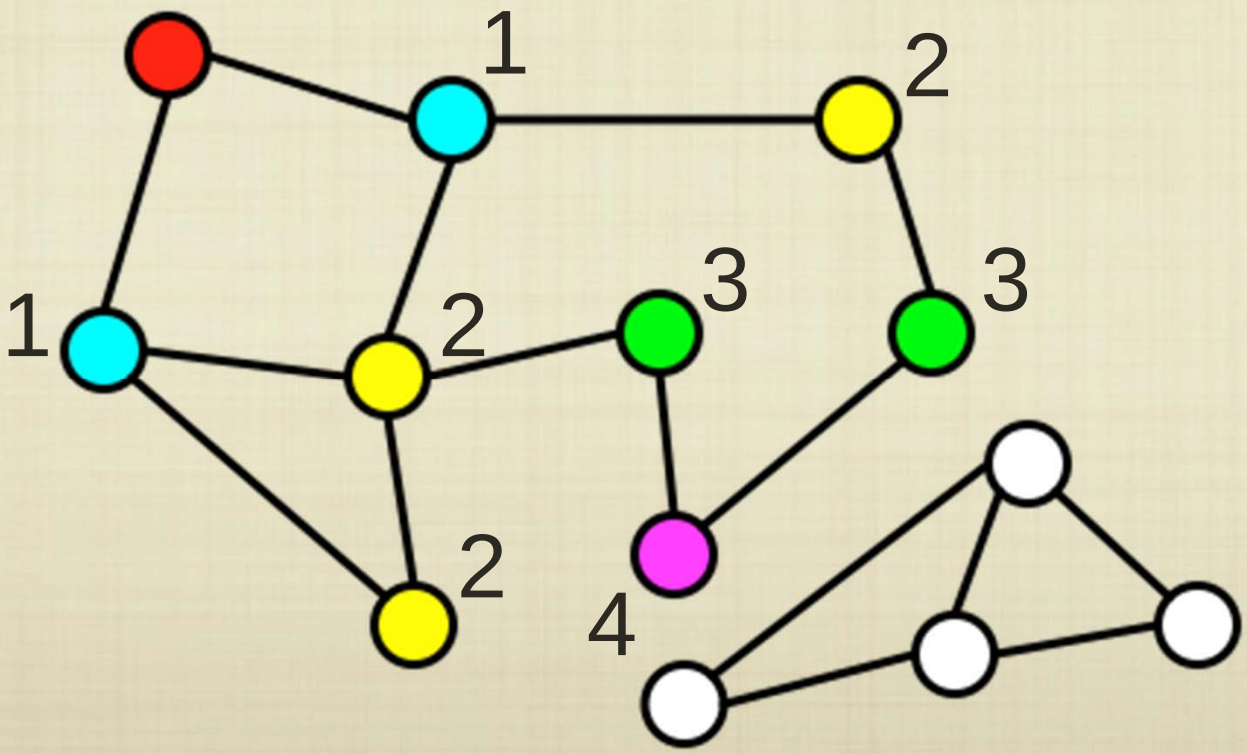
- The graph is not connected, because we cannot visit any new vertices, but have not visited all of them.



Breadth-First Search

- This intuitive idea to “visit” vertices can be made more concrete, but how do we organize our search? How do we keep track of vertices that have been visited?

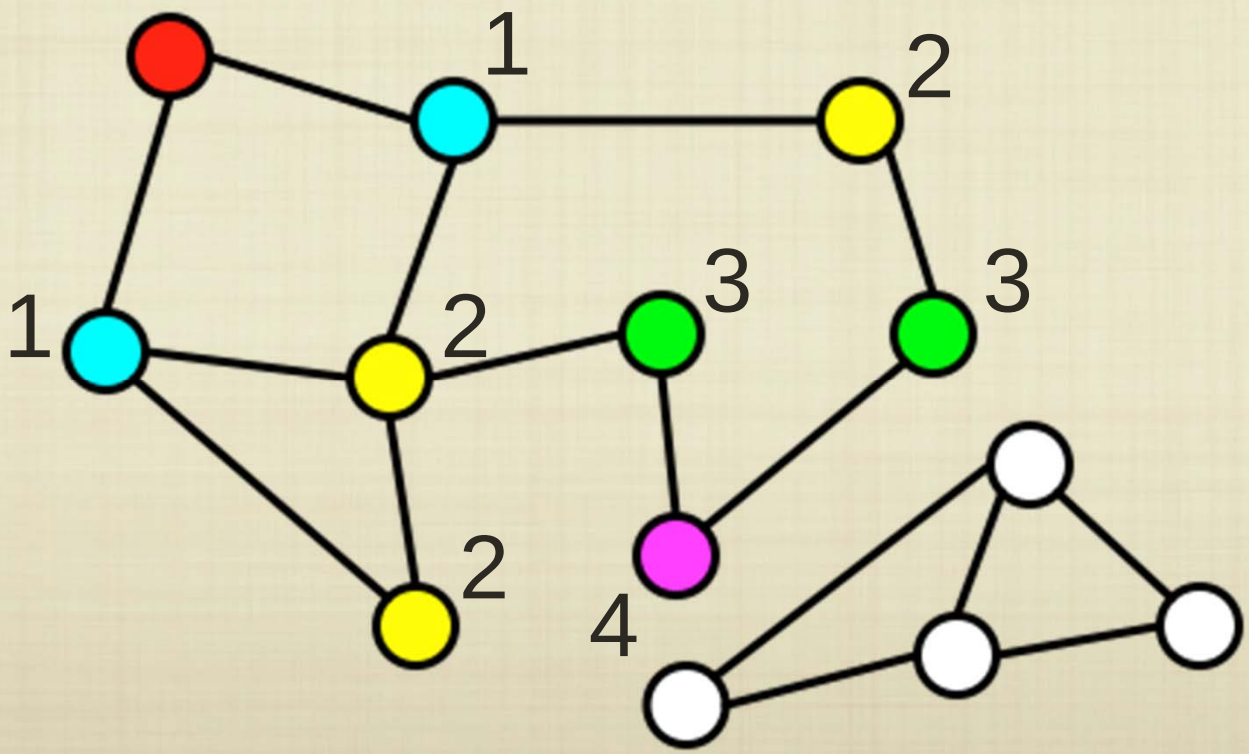
- Note that the order of visitation gives us shortest path distances!



Breadth-First Search

- To keep track of the order of visitation, we can use a queue.

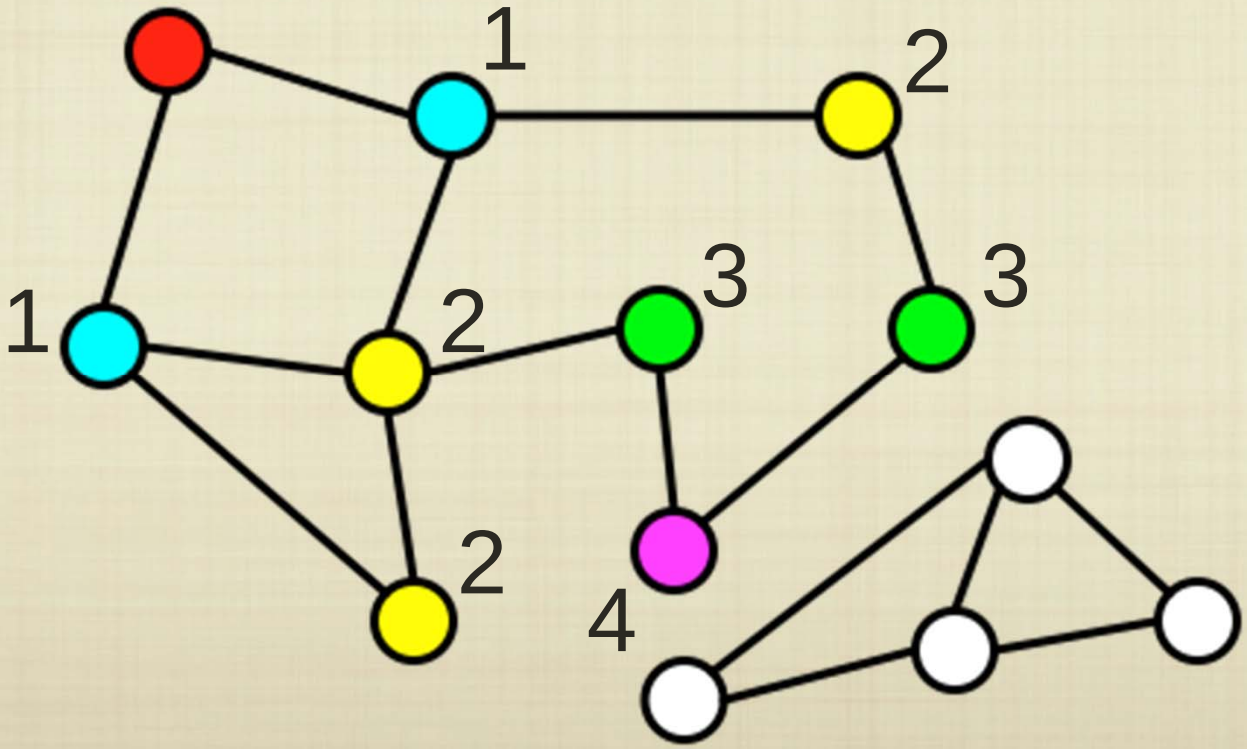
- A queue is just a list, but we only remove items from the front, and add items at the end.



Breadth-First Search

Algorithm:

1. Initialize a queue with the starting vertex
2. While the queue is not empty
    a. Dequeue the next vertex from the queue
    b. Enqueue all its neighbors that have not been visited/enqueued before



Breadth-First Search

Runtime:

- Each vertex is enqueued at most once, and therefore dequeued at most once
- We need to examine each edge once to update distances

$\Rightarrow$ Runtime $O(n+m)$ where $n=|V|$ is the number of vertices in G, and $m=|E|$ is the number of edges in G

$\Rightarrow$ The runtime is linear in the size of the graph