Scientific Computation

Spring, 2019

Lecture 7

Notes

- Today's office hour will be in the MLC
- Clarifications on HW1 were added online Friday and Saturday

Today

- Graph search in undirected networks

Networks: basics

Network	Nodes	Links	Directed / Undirected	N	L	‹K›
Internet	Routers	Internet connections	Undirected	192,244	609,066	6.34
WWW	Webpages	Links	Directed	325,729	1,497,134	4.60
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594	2.67
Mobile-Phone Calls	Subscribers	Calls	Directed	36,595	91,826	2.51
Email	Email addresses	Emails	Directed	57,194	103,731	1.81
Science Collaboration	Scientists	Co-authorships	Undirected	23,133	93,437	8.08
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908	83.71
Citation Network	Papers	Citations	Directed	449,673	4,689,479	10.43
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802	5.58
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930	2.90

Table 2.1

Canonical Network Maps

The basic characteristics of ten networks used throughout this book to illustrate the tools of network science. The table lists the nature of their nodes and links, indicating if links are directed or undirected, the number of nodes (N) and links (L), and the average degree for each network. For directed networks the average degree shown is the average in- or out-degrees $\langle k \rangle = \langle k_{in} \rangle = \langle k_{out} \rangle$ (see Equation (2.5)).

Generally interested in large *complex* networks

Analysis can be complicated and expensive (classical example: computing shortest path between nodes)

Networkx package provides a suite of tools for working with complex networks

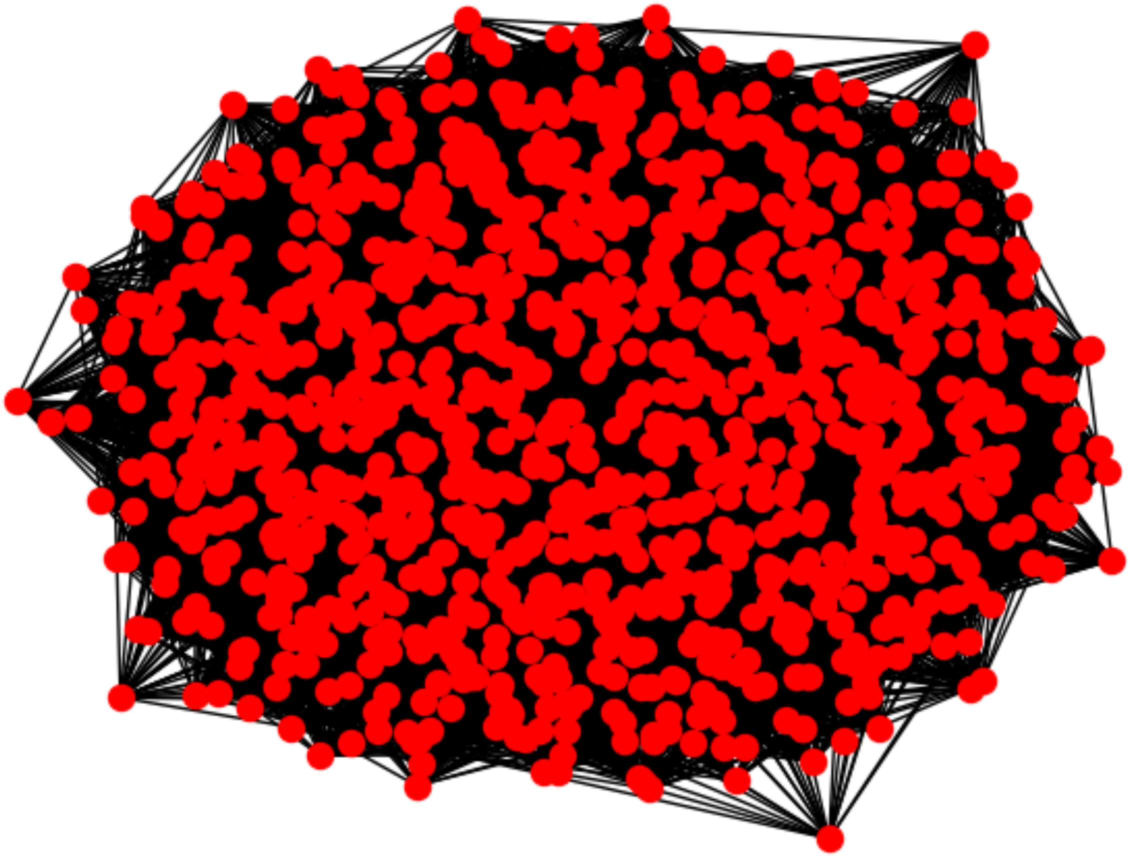
More generally: avoid writing own code whenever possible! Many powerful highly-efficient libraries are available

Networkx: basics

Erods-Renyi network N nodes, a link is placed between a pair of nodes with probability p:

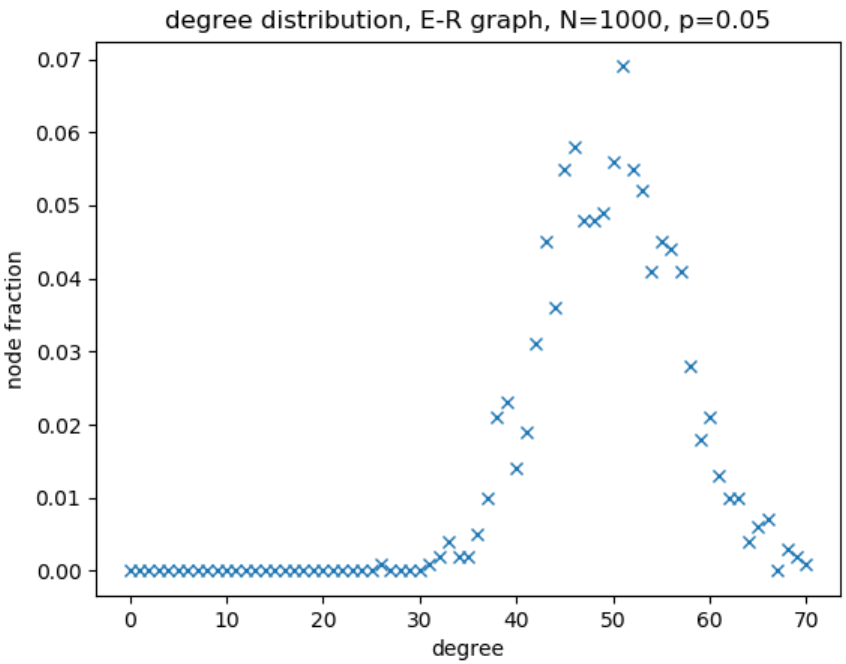
```
In [119]: Grandom = nx.gnp_random_graph(1000,0.05)

In [120]: nx.draw(Grandom,node_shape='.')
```



Getting started with NetworkX

- **Degree distribution follows a binomial distribution:**



- Should compute degree distributions for several graphs (with fixed N,P) and average

- Generally, when there is randomness in the problem, statistics are the quantities of interest (mean, variance, etc…)

- For large degree, distribution decays away exponentially – most real complex networks have large-degree hubs

Getting started with NetworkX

- Two other important quantities are the *clustering coefficient* and *shortest path*
- Clustering coefficient for node i with degree q_i:

$$C_i = \# \text{ of links between neighbors}/(q_i/2*(q_i-1))$$

```
In [16]: nx.clustering(G,500)
Out[16]: 0.044096728307254626

In [17]: nx.clustering(G,100)
Out[17]: 0.06464646464646465

In [18]: nx.clustering(G,0)
Out[18]: 0.04645760743321719
```

For $G_{N,P}$ graph, expect $C_i = P$

Getting started with NetworkX

- **Two other important quantities are the *clustering coefficient* and *shortest path***
- **Shortest path: find route between two nodes traversing fewest number of links**

```
In [20]: nx.shortest_path(G,source=0,target=500)
Out[20]: [0, 233, 15, 500]
```

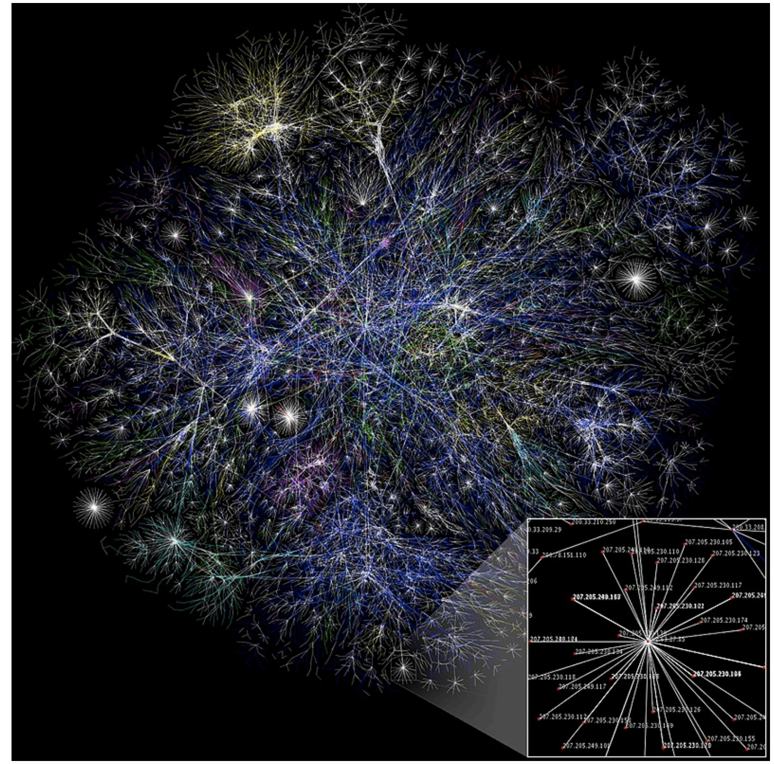
→ Very important in study *algorithms (lectures 7+8)*

Notes: GNP graph is not a good model for large complex networks

- **Degree distribution should include large-degree nodes, power-law decay for large q**
- **Clustering coefficient should be large and the average degree should be small**
- **Cf. Barabasi-Albert model from Lab 3**

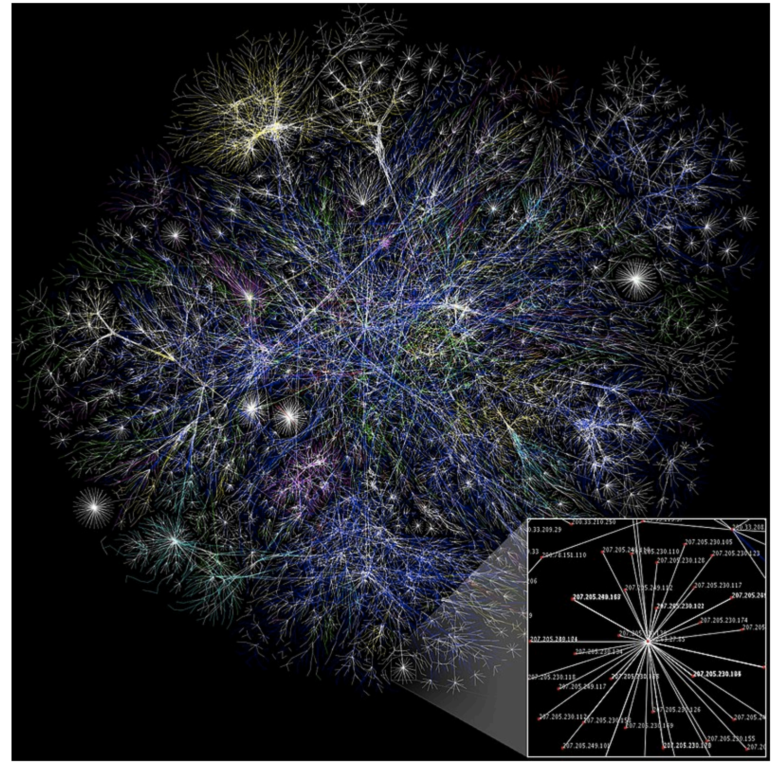
Networks

- We’ve looked at search for lists/arrays in some detail
- Can we construct efficient routines for searching through networks?



https://en.wikipedia.org/wiki/Complex_network

Networks



https://en.wikipedia.org/wiki/Complex_network

- We've looked at search for lists/arrays in some detail
- Can we construct efficient routines for searching through networks?
- Are graph searches useful?
  - Provide basic information about network structure, e.g. which nodes are (un)reachable from a given node?
  - Find shortest paths, what is the "fastest" route between two nodes?
- Many networks of interest have 1e5+ edges, essential that implementation is efficient!

Imperial College London

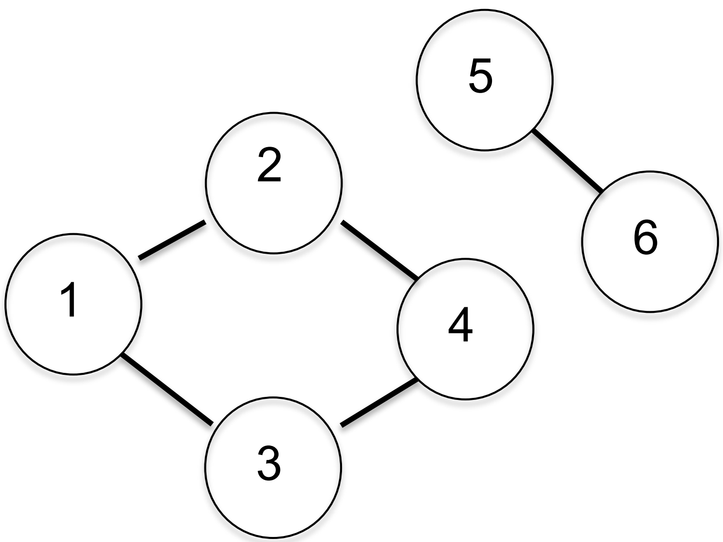
Graph search

- **Basic idea: Given a graph, G, and a source node, s, find all other nodes that can be reached from s**

- **Basic approach:**
  - **Label s as "explored" and all other nodes as "unexplored"**

  While there is at least one edge between an explored and unexplored node:

  Select one such edge and re-label the unexplored node as explored



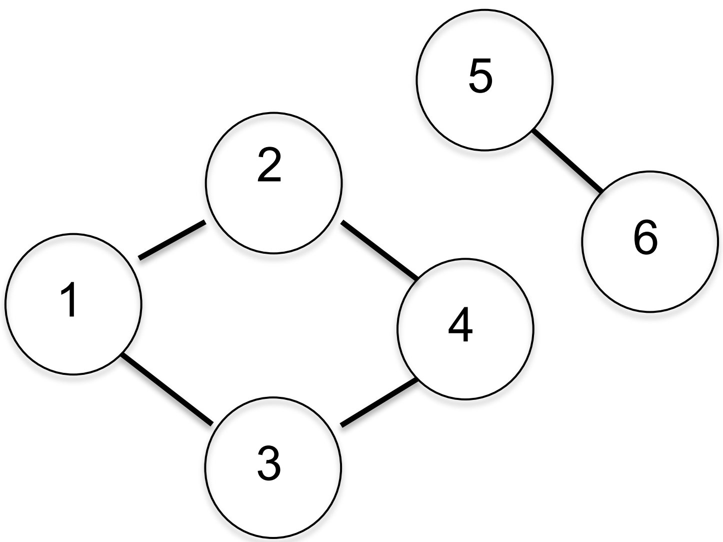
Graph search

- **Basic idea: Given a graph, G, and a source node, s, find all other nodes that can be reached from s**

- **Basic approach:**
  - **Label s as "explored" and all other nodes as "unexplored"**

  While there is at least one edge between an explored and unexplored node:

  Select one such edge and re-label the unexplored node as explored

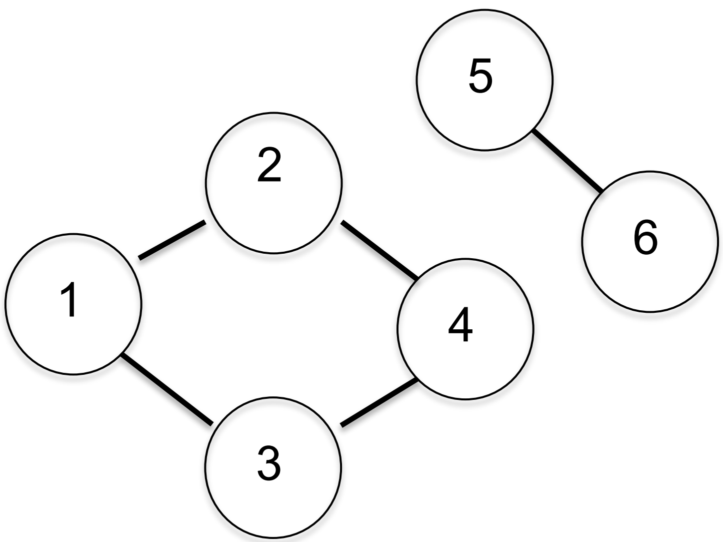


Claim: Upon completion, a node is labeled explored if and only if a path exists between it and the source node

Graph search

- **Basic approach:**
  - **Label s as "explored" and all other nodes as "unexplored"**

While there is at least one edge between an explored and unexplored node:
Select one such edge and re-label the unexplored node as explored



Implementation: Depends on how edges are selected each iteration

- **Depth-first search – aggressively move into the graph (1-2, 2–4)**
- **Breadth-first search – consider one "layer" at a time (1-2, 1-3)**
- **Today: BFS**

Breadth-first search

- **Python implementation**

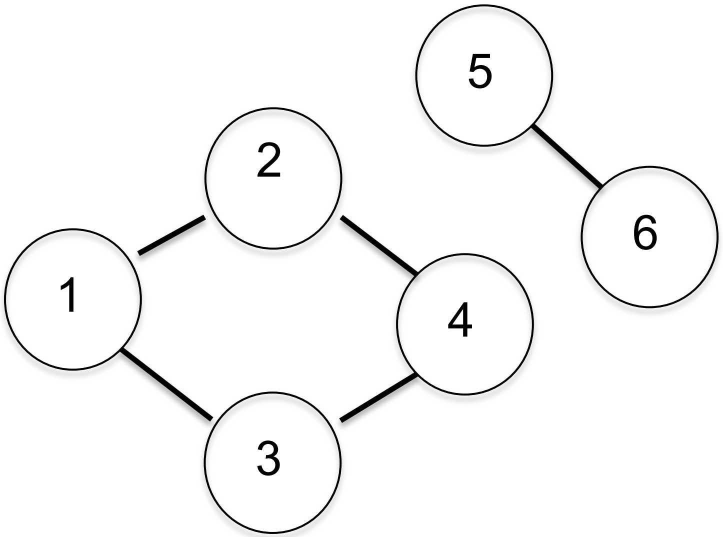
    - **Specify: Graph and source node**
    - **Maintain: 1) list of nodes, 2) list of labels for nodes and 3) *queue* of nodes to-be explored**
        - **Initialize the queue with the source node and mark it as explored**
        - **Remove nodes from the queue in the order they were added (first in, first out)**
        - **Search through edges of removed node and add unexplored nodes to queue**
            - **Label added nodes as explored**
        - **Terminate search when queue is empty**

Breadth-first search

- **Python implementation**

- **Specify: Graph and source node**

```
G = nx.Graph()
edges = [[1,2],[1,3],[1,2],[2,4],[3,4],[5,6]]
G.add_edges_from(edges)
s = 1
Q = [s] #Nodes to be explored
```



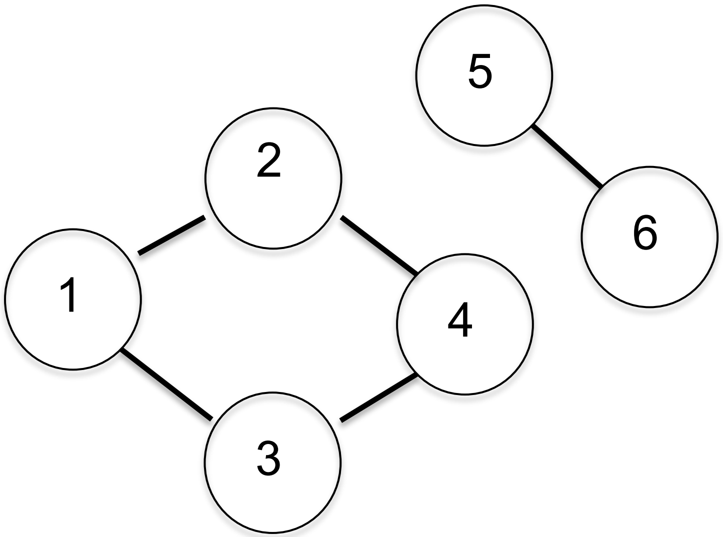
Breadth-first search

- **Specify: Graph and source node**

```
G = nx.Graph()
edges = [[1,2],[1,3],[1,2],[2,4],[3,4],[5,6]]
G.add_edges_from(edges)
s = 1
Q = [s] #Nodes to be explored
```

- **Create list of nodes and labels**

```
nodes = G.nodes()
z = [0 for i in nodes] #labels
L = [nodes,z]
L[1][s-1]=1 #mark source node as explored
```

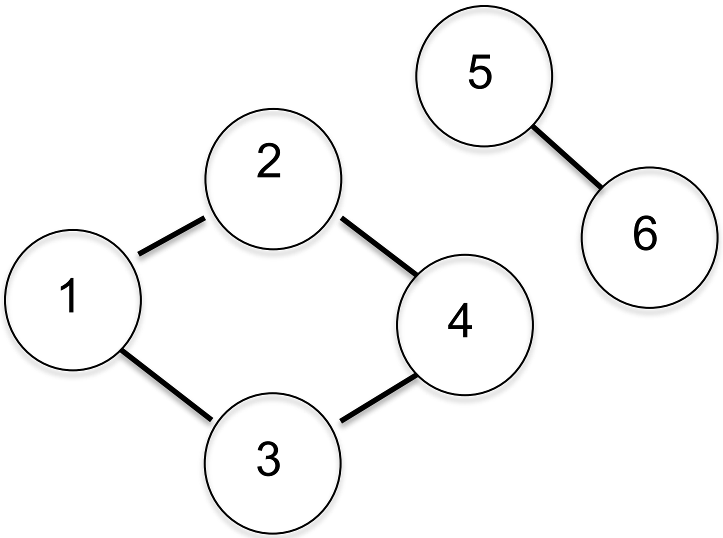


Breadth-first search

- **Specify: Graph and source node**

```
G = nx.Graph()
edges = [[1,2],[1,3],[1,2],[2,4],[3,4],[5,6]]
G.add_edges_from(edges)
s = 1
Q = [s] #Nodes to be explored
```

- **Create list of nodes and labels**




```
nodes = G.nodes()
z = [0 for i in nodes] #labels
L = [nodes,z]
L[1][s-1]=1 #mark source node as explored
```

- **Iterate through nodes in queue (updating Q as appropriate)**

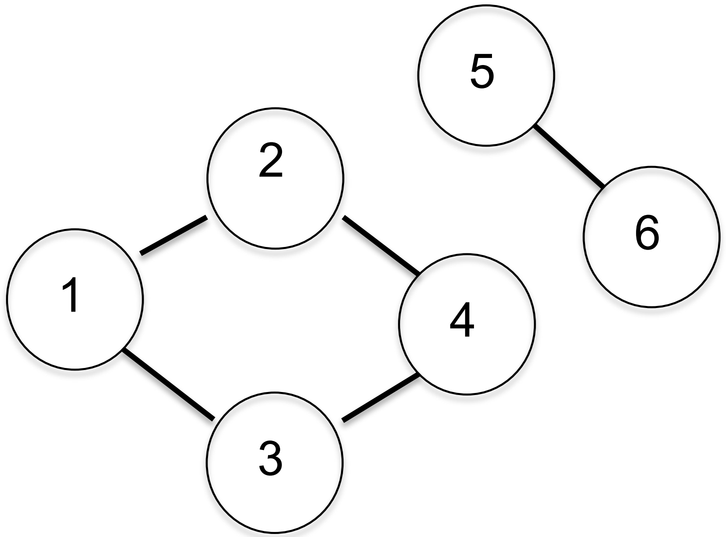
```
while len(Q)>0:
    n = Q.pop(0)
    for v in G.adj[x].keys(): #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            Q.append(v)
```

Breadth-first search

- Adding `print("n=%d,Q=" %(n),Q)` to the while loop and running the code:

```
n=1,Q= [2, 3]
n=2,Q= [3, 4]
n=3,Q= [4]
n=4,Q= []
```

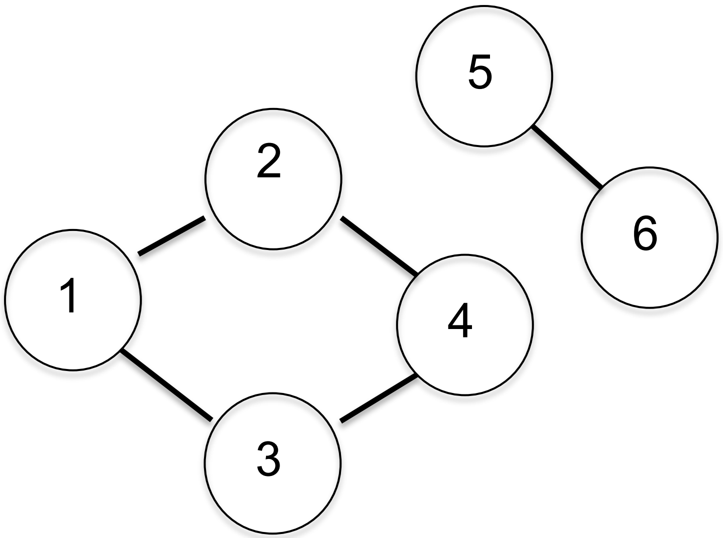
- n is the node removed from the queue and Q is the queue after edges from n have been added (if n is unexplored



Breadth-first search

- Adding `print("n=%d,Q=" %(n),Q)` to the while loop and running the code:

```
n=1,Q= [2, 3]
n=2,Q= [3, 4]
n=3,Q= [4]
n=4,Q= []
```



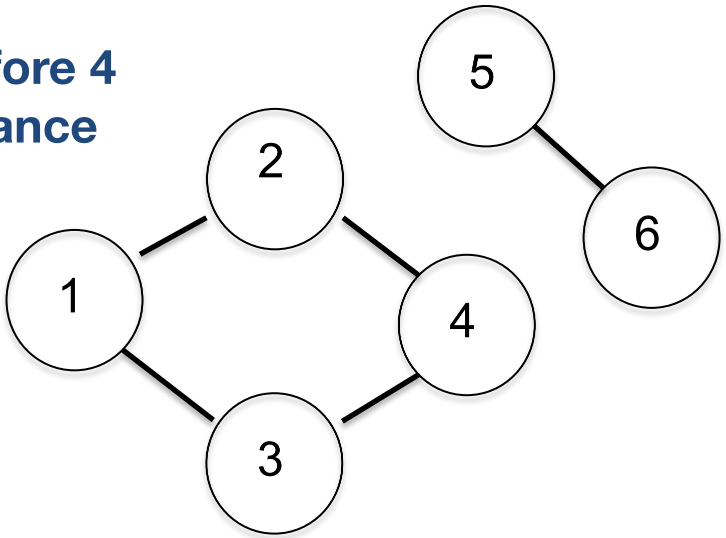
- n is the node removed from the queue and Q is the queue after edges from n have been added (if n is unexplored

- What is the cost of BFS?
  - Each reachable node is relabeled and each (reachable) edge is encountered twice
  - For a graph with N nodes and M edges, cost is O(M+N)
    - Linear time!

Breadth-first search

- How can we use BFS to compute distances (from source)?

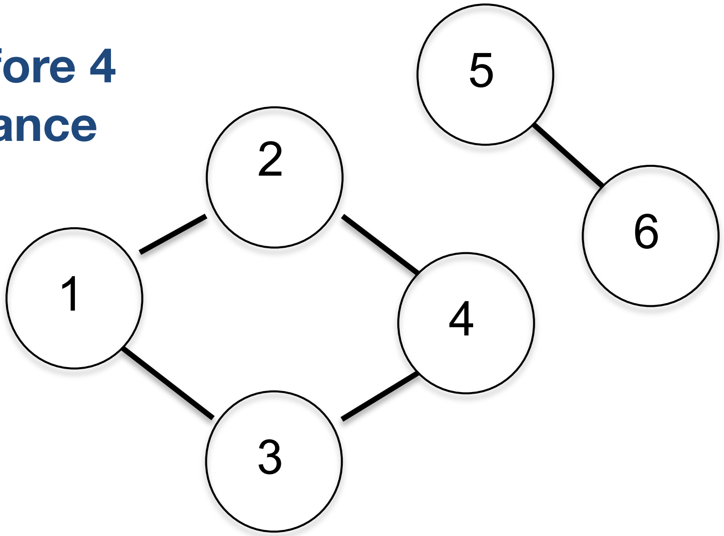
- BFS iterates through the graph by layer
  - We know nodes 2 and 3 will be searched before 4
  - When a node is added to the queue, it's distance is one greater than its neighbor being removed from queue
  - Just need to maintain a list of distances which are filled in as nodes are added to the queue



Breadth-first search

- How can we use BFS to compute distances (from source)?

- BFS iterates through the graph by layer
  - We know nodes 2 and 3 will be searched before 4
  - When a node is added to the queue, it's distance is one greater than its neighbor being removed from queue
  - Just need to maintain a list of distances which are filled in as nodes are added to the queue

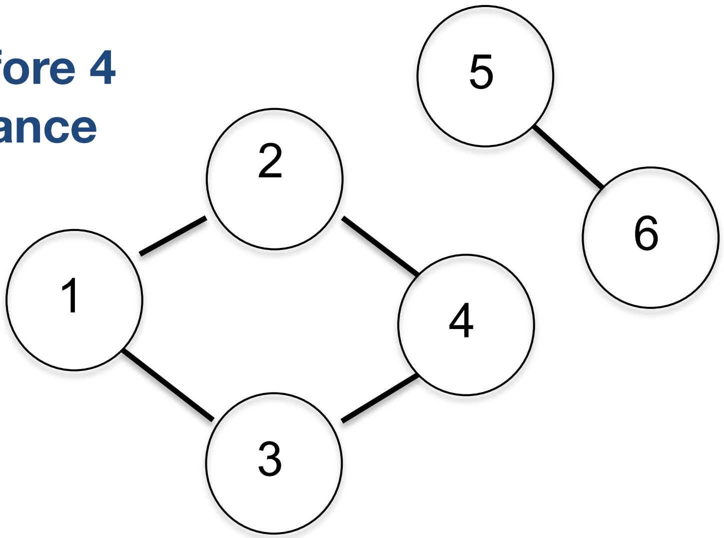



```
d = [-1000 for i in nodes] #initialize distances to -1000
L = [nodes,z,d]
L[2][s-1]=0 #Source node has distance zero
```

Breadth-first search

- How can we use BFS to compute distances (from source)?

- BFS iterates through the graph by layer
  - We know nodes 2 and 3 will be searched before 4
  - When a node is added to the queue, it's distance is one greater than its neighbor being removed from queue
  - Just need to maintain a list of distances which are filled in as nodes are added to the queue

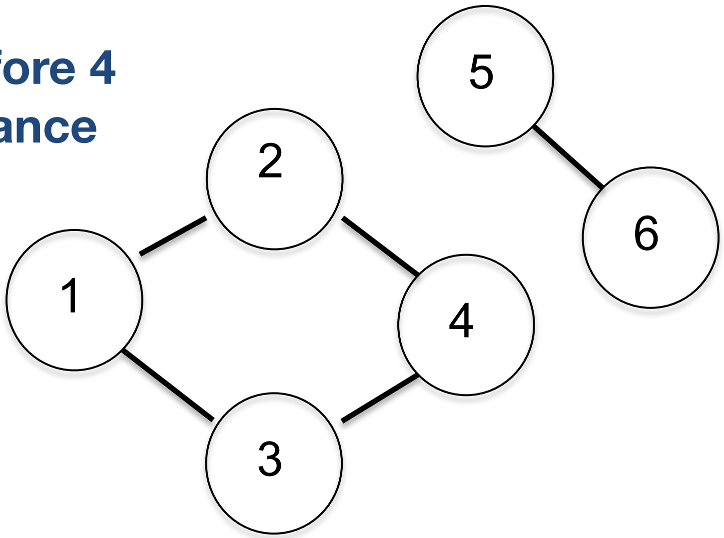



```
while len(Q)>0:
    n = Q.pop(0)
    for v in G.adj[x].keys(): #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            L[2][v-1] = L[2][n-1]+1 #Set distance of node v
            Q.append(v)
```

Breadth-first search

- **How can we use BFS to compute distances (from source)?**

- **BFS iterates through the graph by layer**
  - **We know nodes 2 and 3 will be searched before 4**
  - **When a node is added to the queue, it's distance is one greater than its neighbor being removed from queue**
  - **Just need to maintain a list of distances which are filled in as nodes are added to the queue**




```
while len(Q)>0:
    n = Q.pop(0)
    for v in G.adj[x].keys(): #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            L[2][v-1] = L[2][n-1]+1 #Set distance of node v
            Q.append(v)
```

Running this code: `In [5]: L[2]`

```
Out[5]: [0, 1, 1, 2, -1000, -1000]
```

Breadth-first search

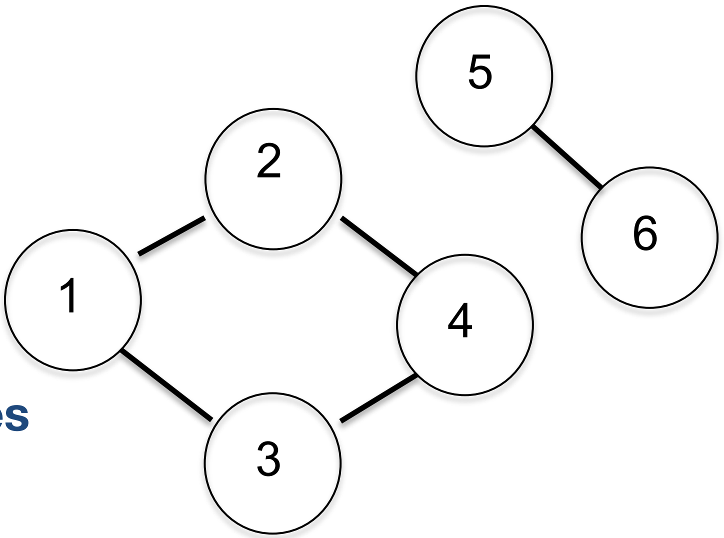
- **Distance calculation doesn't effect cost estimate, still O(M+N)**

- **But, is our implementation actually efficient?**

- **For large networks, the key steps involve queue management**

- **The** append **operation requires O(1) operations**

- **However,** pop **from the front of the queue requires shifting all other elements in the list,** Q



- **The** collections **module contains a *dequeue* datatype**
- **From online documentation:**

list-like container with fast appends and pops on either end