CS4423 - Networks

Prof. Götz Pfeiffer
School of Mathematics, Statistics and Applied Mathematics
NUI Galway

Lecture 6: Paths, Trees and Breadth First Search

Sequences of interconnected edges in a graph are called *paths*, leading to notions of *connectivity* and *distance*. A *tree* is a particularly useful kind of connected graph. Many questions on networks concerning distance or connectivity in a graph can be *algorithmically* answered be a versatile strategy called *Breadth First Search (BFS)*.

Start by importing the necessary python libraries into this jupyter notebook.

In [1]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

Paths

The fundamental notion of *connectivity* in a network is closely related to the notion of *paths* in a graph.

Definitions. A **path** in a graph $G = (X, E)$ is a sequence of nodes, where any pair of consecutive nodes in the sequence is (linked by) an edge in E.

Such a path can have repeated nodes. If it doesn't, the path is called a **simple path**.

The **length** of a path is the number of *edges* it involves (that is the number of *nodes* minus 1).

At each vertex $x \in X$, there is a unique path of length 0, the **empty path**, consisting of vertex x only.

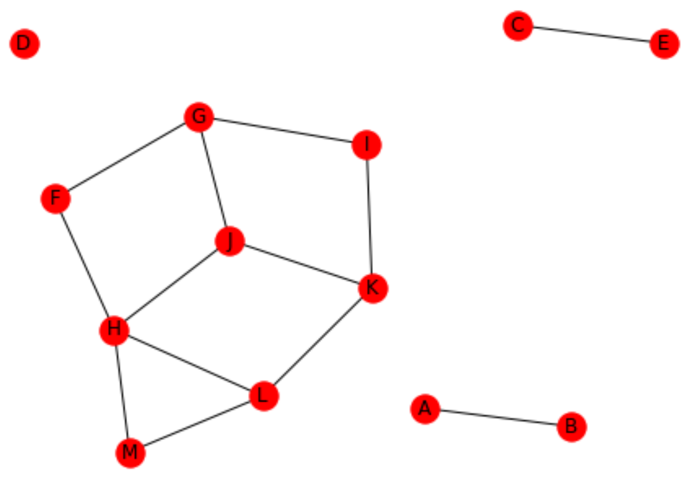
A **cycle** is a path of length at least 3 that is a simple path, except for the first and the last node being the same.

In [2]:
```
nodes = list('ABCDEFGHIJKLM')
edges = [
    'AB', 'CE', 'FG', 'FH', 'GI', 'GJ', 'HJ', 'HL', 'HM',
    'IK', 'JK', 'KL', 'LM'
]
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
```

In [3]:
```
import pygraphviz
from networkx.drawing.nx_agraph import graphviz_layout
pos = graphviz_layout(G, prog="neato")
```

```
In [4]: nx.draw(G, pos, with_labels = True)
```

```
/home/goetz/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py
:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```



(F, G, I) is a path in the graph above, and (H, J, K, L, H) is a cycle

A cycle in a simple graph provides, for any two nodes on that cycle, (at least) two different paths from one to the other.

Note that each edge (and node) of the 1970 Internet graph belongs to a cycle. This makes the other way around the cycle an alternative route in case one of the edges should fail.

In a *directed* network, paths are directed, too. A path from a vertex x to a vertex y is a sequence of vertices $x = x_0, x_1, \ldots, x_k = y$ such that, for any $i = 1, \ldots, k$, there is an edge from x_{i-1} to x_i in the graph.

Connected Components

Communication and transportation networks tend to be connected, as this is their main purpose: to connect.

Definition. A simple graph is **connected** if, for every pair of nodes, there is a path between them.

If a graph is not connected, it naturally breaks into pieces, its **connected components**.

The connected components of the graph below are the node sets $\{A, B\}$, $\{C, E\}$, $\{D\}$, and $\{F, G, H, I, J, K, L, M\}$. Note that a component can consist of a single node only.

```
In [5]: list(nx.connected_components(G))
```

```
Out[5]: [{'A', 'B'}, {'C', 'E'}, {'D'}, {'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M'}]
```

Note. The relation 'there is a path from x to y on the node set X of a graph is the **transitive closure** of the graph relation 'there is an *edge* between x and y'. It is

- **reflexive** (as each node x is connected to itself by the zero length path starting and ending at x),
- **symmetric** (as a path from x to y can be used backwards as a path from y to x),
- and **transitive** (as a path from x to y and a path from y to z together make up a path from x to z),

hence an **equivalence relation**. The connected components of the graph are the parts (equivalence classes) of the corresponding **partition** of X.

Trees

Trees appear in the analysis of networks and many other applications.

Definition. A graph is called **acyclic** if it does not contain any cycles.

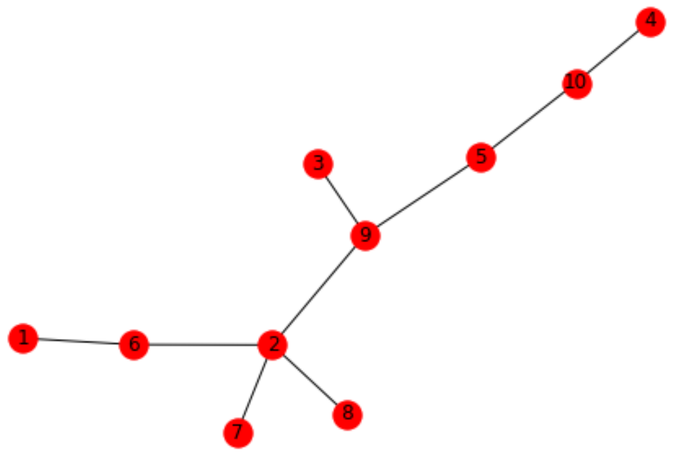
A **tree** is a connected, acyclic graph.

A **forest** is a graph whose connected components are all trees.

Trees are naturally bipartite.

The (bipartite) graph B from the previous lecture is in fact a tree.

In [6]:
```
B = nx.Graph([(1,6), (2,6), (2,7), (2,8), (2,9),
              (3,9), (4,10), (5,9), (5,10)])
nx.draw(B, with_labels=True)
```



Theorem. Let $G = (X, E)$ be a graph of degree $n = |X|$ and size $m = |E|$. Then the following are equivalent:

- G is a tree (i.e. acyclic and connected);
- G is connected and $m = n - 1$;
- G is a minimal connected graph (i.e., removing an edge will disconnect G);
- G is acyclic and $m = n - 1$;
- G is a maximal acyclic graph (i.e., adding an edge will introduce a cycle in G).

Corollary. If $G = (X, E)$ is a tree then, for any vertices $x, y \in X$, there is *exactly one path* from x to y in G.

Shortest Paths

Recall that a **path** in a network $G = (X, E)$ is a sequence $p = (x_0, x_1, \dots, x_k)$ of nodes $x_i \in X$, $i = 0, \dots, k$, such that any pair of consecutive nodes forms an edge in G, i.e., $\{x_{i-1}, x_i\} \in E$ for all $i = 1, \dots, k$. The **length** $l(p)$ of the path p is the number of edges, $l(p) = k$.

In many practical applications it is of interest to find for a pair x, y of nodes, one or all the paths form x to y connecting the two nodes with the fewest number of edges possible. This is a more complex measure on a network than, say, the degree of a node, and we will need a more complex procedure, that is: an algorithm, in order to answer such questions systematically. Let's start with a proper definition.

Definition. Let $G = (X, E)$ be a simple graph and let $x, y \in X$. Let $P(x, y)$ be the set of all paths from x to y. Then the **distance** $d(x, y)$ from x to y is

$$d(x, y) = \min\{l(p) : p \in P(x, y)\},$$

the shortest possible length of a path from x to y, and a **shortest path** from x to y is a path $p \in P(x, y)$ of length $l(p) = d(x, y)$.

The **diameter** $\operatorname{diam}(G)$ of the network G is the length of the longest shortest path between any two nodes,

$$\operatorname{diam}(G) = \max\{d(x, y) : x, y \in X\}$$

.

Breadth First Search

Now we consider the following problem: Given a node $x \in X$, what are the distances $d(x, y)$ for all nodes $y \in X$? A systematic procedure for finding these distances, and the shortest paths through which they are realized, is given by the algorithm which is know in Computer Science as **Breadth First Search** (BFS).

In order to describe the algorithm step by step, let's call a node y a **neighbor** of node x, if $\{x, y\}$ is an edge, and let's denote by

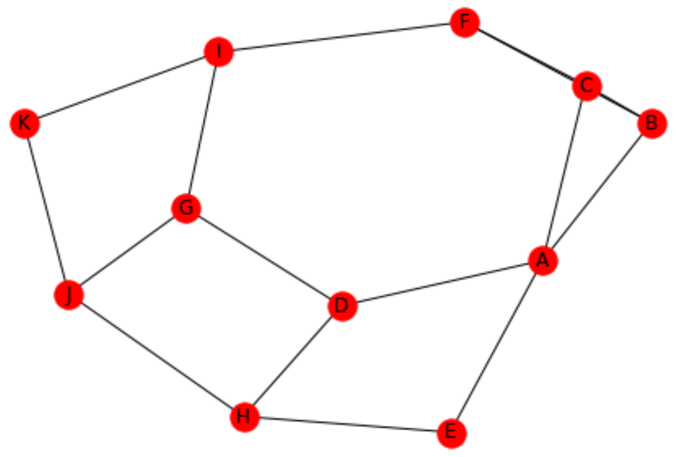
$$N(x) = \{y \in X : \{x, y\} \in E\}$$

the set of all neighbors of node x. The algorithm works through the network layer by layer, starting with the given vertex x at layer 0 and all its friends at layer 1. It then finds the friends of the friends at layer 2, and so on, until every node that can be reached from x by a path has been recorded, taking care that no node gets recorded twice. The layer of a node then corresponds to its distance from the given node x.

In practice, as in the following example, the layer does not need to be made explicit.

```
In [7]: G = nx.read_adjlist("bfs.adj")
```

In [8]:
```
nx.draw(G, with_labels=True)
```



In [9]:
```
for x in G:
    G.nodes[x]['seen'] = False
```

In [10]:
```
G.nodes['A']
```

Out[10]:
```
{'seen': False}
```

In [11]:
```
seen = []
```

In [12]:
```
seen.append('A')
G.nodes['A']['seen'] = True
seen
```

Out[12]:
```
['A']
```

In [13]:
```
list(G.neighbors('A'))
```

Out[13]:
```
['B', 'C', 'D', 'E']
```

In [14]:
```
for x in G.neighbors('A'):
    seen.append(x)
    G.nodes[x]['seen'] = True
```

In [15]:
```
seen
```

Out[15]:
```
['A', 'B', 'C', 'D', 'E']
```

In [16]:
```
node = 'B'
for x in G.neighbors(node):
    if not G.nodes[x]['seen']:
        seen.append(x)
        G.nodes[x]['seen'] = True
seen
```

Out[16]:
```
['A', 'B', 'C', 'D', 'E', 'F']
```

In [17]:
```
node = 'K'
for x in G.neighbors(node):
    if not G.nodes[x]['seen']:
        seen.append(x)
        G.nodes[x]['seen'] = True
seen
```

Out[17]:
```
['A', 'B', 'C', 'D', 'E', 'F', 'I', 'J']
```

... and so on, until there are no more nodes to be processed.

When this process is formulated as an algorithm, we use a **queue** (a first-in first-out buffer) to keep track of the node whose neighbors are currently under consideration. A queue is an array of values that comes with two basic operations:

- one can **push** a value to the end of the queue, and
- one can **pop** a value off the top of the queue (provided the queue is not empty).

It can be shown that this version of the algorithm in the common case of a sparse network has complexity $O(n)$, which is as good as one could hope for.

Breadth First Search. Given a simple graph $G = (X, E)$ and a vertex $x \in X$, determine $d(x, y)$ for all nodes $y \in X$.

```
1. [Initialize.] Suppose that $X = \{x_1, x_2, \dots, x_n\}$ and that $x = x_j$. Set $d_i \leftarrow \perp$ (undefined) for
   $i = 1, \dots, n$. Set $d_j \leftarrow 0$ and initialize a queue $Q \leftarrow (x_j)$.
2. [Loop.] While $Q \neq \emptyset$:
    • pop node $x_k$ off $Q$
    • for each neighbor $x_l$ of $x_k$ with $d_l = \perp$:
        ▪ push $x_l$ onto $Q$ and set $d_l \leftarrow d_k + 1$.
3. [Stop.] Return the array $(d_1, \dots, d_n)$.
```

In [18]:
```python
from queue import Queue
```

In [19]:
```python
for x in G:
    G.nodes[x]['d'] = -1 # undefined

x = 'B'
G.nodes[x]['d'] = 0
q = Queue()
q.put(x)
```

In [20]:
```python
list(q.queue)
```

Out[20]:
```
['B']
```

In [21]:
```python
while not q.empty():
    x = q.get()
    for y in G.neighbors(x):
        if G.nodes[y]['d'] < 0: # undefined?
            G.nodes[y]['d'] = G.nodes[x]['d'] + 1
            q.put(y)
    print(x, ": ", list(q.queue))
```

```
B :  ['A', 'C', 'F']
A :  ['C', 'F', 'D', 'E']
C :  ['F', 'D', 'E']
F :  ['D', 'E', 'I']
D :  ['E', 'I', 'G', 'H']
E :  ['I', 'G', 'H']
I :  ['G', 'H', 'K']
G :  ['H', 'K', 'J']
H :  ['K', 'J']
K :  ['J']
J :  []
```

In [22]:
```
print(list(map(lambda x: G.nodes[x]['d'], G)))
```
```
[1, 0, 1, 2, 2, 1, 3, 3, 2, 4, 3]
```

In [23]:
```
print(list(G.nodes.data('d')))
```
```
[('A', 1), ('B', 0), ('C', 1), ('D', 2), ('E', 2), ('F', 1), ('G', 3), ('H', 3
), ('I', 2), ('J', 4), ('K', 3)]
```

Variants. BFS is an extremely versatile algorithm, which applies in many different situations and can be adapted to produce additional information on a network.

For example, BFS run on a node x in a network $G = (X, E)$ determines the **connected component** of X in G (as the set of all nodes that get a distance value assigned).

With little more work (and an additional array) BFS can produce a **spanning tree** (or **shortest path tree**). Here, whenever node x_l is pushed onto Q, it is assigned the current node x_k (in the additional array) as its predecessor on a shortest path from x_j to x_l. The subgraph of the network consisting of these edges is a tree. As a tree, it has exactly one path between the given node x and any of its vertices y and, by construction, this path is a shortest path between x and y.

In [24]:
```
for x in G:
    G.nodes[x]['d'] = -1 # undefined

x = 'A'
G.nodes[x]['d'] = 0
q = Queue()
q.put(x)

for e in G.edges:
    G.edges[e]['seen'] = False
```

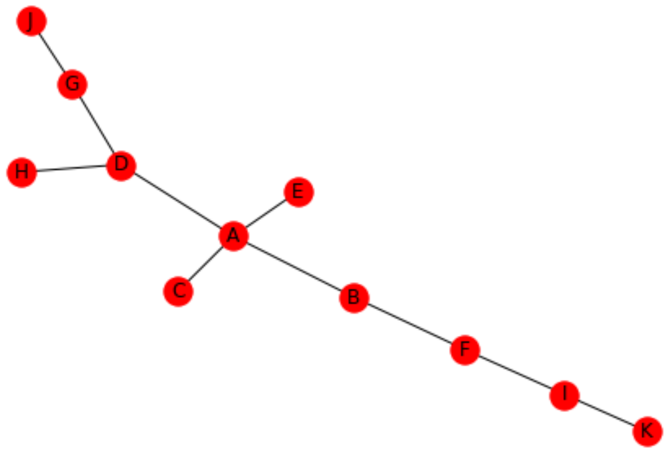
In [ ]:

In [25]:
```
print(list(G.edges.data()))
```
```
[('A', 'B', {'seen': False}), ('A', 'C', {'seen': False}), ('A', 'D', {'seen':
False}), ('A', 'E', {'seen': False}), ('B', 'C', {'seen': False}), ('B', 'F',
{'seen': False}), ('C', 'F', {'seen': False}), ('D', 'G', {'seen': False}), ('
D', 'H', {'seen': False}), ('E', 'H', {'seen': False}), ('F', 'I', {'seen': Fa
lse}), ('G', 'I', {'seen': False}), ('G', 'J', {'seen': False}), ('H', 'J', {'
seen': False}), ('I', 'K', {'seen': False}), ('J', 'K', {'seen': False})]
```

In [26]:
```
while not q.empty():
    x = q.get()
    for y in G.neighbors(x):
        if G.nodes[y]['d'] < 0: # undefined?
            G.nodes[y]['d'] = G.nodes[x]['d'] + 1
            q.put(y)
            G.edges[x, y]['seen'] = True
    print(x, ": ", list(q.queue))
```
```
A :  ['B', 'C', 'D', 'E']
B :  ['C', 'D', 'E', 'F']
C :  ['D', 'E', 'F']
D :  ['E', 'F', 'G', 'H']
E :  ['F', 'G', 'H']
F :  ['G', 'H', 'I']
G :  ['H', 'I', 'J']
H :  ['I', 'J']
I :  ['J', 'K']
J :  ['K']
K :  []
```

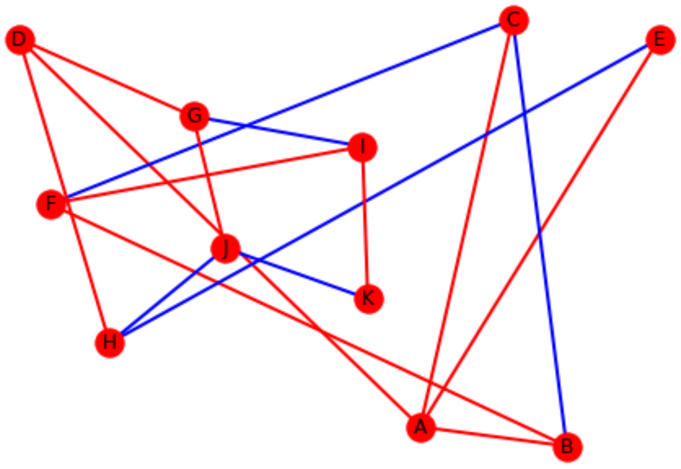
```
In [27]: sub = [e for e in G.edges if G.edges[e]['seen']]
```

```
In [28]: nx.draw(G.edge_subgraph(sub), with_labels=True)
```



Or, one could highlight the spanning tree inside the graph by using, say, red as color for the spanning edges (and blue for the rest).

```
In [29]: colors = ['red' if G.edges[e]['seen'] else 'blue' for e in G.edges]
         nx.draw(G, pos, edge_color = colors, with_labels = True, width=2.0)
```



Of course, in order to find distances, or shortest paths between **all pairs** of nodes x and y in a network, one can perform BFS for each of the vertices $x \in X$ in turn.

The algorithm and its variants also works on directed networks, but the results then will have to be interpreted in the context of directed networks.

More about BFS can be found in [Newman, Section 10.3].

Exercises.

1. Compute the distances $d(x, y)$ for all vertices x and y in the above graph `G`.
2. Hence determine the diameter $\text{diam}(G)$.
3. Construct a (simple) graph H with edges

```
1-9, 9-3, 9-12, 9-15, 9-2, 9-13, 5-11, 5-14, 5-3, 11-14,
11-4, 14-12, 14-4, 12-15, 15-7, 2-6, 2-7, 13-10, 4-7, 7-8
```

4. Using BFS, construct a spanning tree of H, starting with vertex 1.
5. Compute a matrix $D = (d_{ij})$ with entries

$$d_{ij} = d(i, j),$$

   the distance between nodes i and j in H.

```
In [ ]:
```