

# Table of Contents

- 1 Modules for this notebook
- 2 Example
- 3 Trees
  - 3.1 Another fact about trees
- 4 How many trees are there?
  - 4.1 Cayley's Formula
  - 4.2 Computing the Prüfer code
  - 4.3 Making a tree from a Prüfer code
- 5 Random Trees
- 6 Graph and Tree Traversal
  - 6.1 Depth First Search (DFS)
  - 6.2 Breadth First Search (BFS)
  - 6.3 Alternative Implementations (Extra: won't do in class)
- 7 Exercises

## CS4423-Networks: Lecture 9 [Draft]

## Week 5, Lecture 2: Trees and Algorithms

Niall Madden, School of Mathematical and Statistical Sciences University of Galway

This Jupyter notebook, and PDF and HTML versions, can be found at <https://www.niallmadden.ie/2425-CS4423/#Week05>

*This notebook was written by Niall Madden, adapted from notebooks by Angela Carnevale.*

### Modules for this notebook

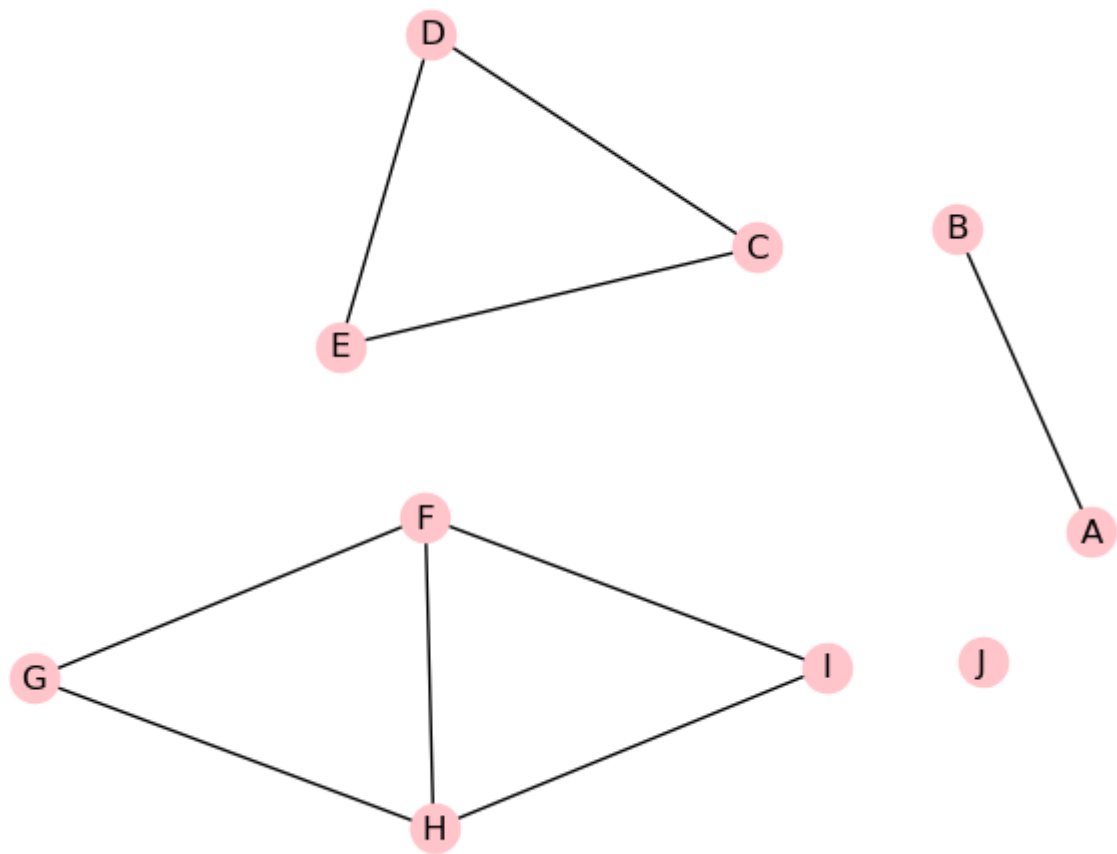
Today, we'll default to light rose-coloured nodes, with has an RGB code of `#ffc5cb`. For more options, see <https://matplotlib.org/stable/users/explain/colors/colors.html>

```
In [1]: import networkx as nx
import numpy as np
opts = { "with_labels": True, "node_color": '#ffc5cb' } # show labels; rose nodes
```

### Example

Short discussion (again) of paths and cycles, and connected componets

```
In [2]: nodes = 'ABCDEFGHIIJ'
edges = ['AB', 'CD', 'DE', 'CE', 'FG', 'FH', 'FI', 'GH', 'HI']
G2 = nx.Graph()
G2.add_nodes_from(nodes)
G2.add_edges_from(edges)
nx.draw_kamada_kawai(G2, **opts)
```



- A cycle in a simple graph provides, for any two nodes on that cycle, (at least) two different paths from one to the other.
- It can be useful to provide alternative routes for connectivity in case one of the edges should fail (e.g. in a electricity networks).
- $(C, D, E, C)$  is a 3-cycle; there are others too.
- The graph is not connected: there are 4 connected components.

## Trees

- A graph is called **acyclic** if it does not contain any cycles.
- A **tree** is a (simple) graph that is **connected** and **acyclic**.

In other words, between any two vertices in a tree there is **exactly one simple path**.

Trees can be characterized in many different ways.

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

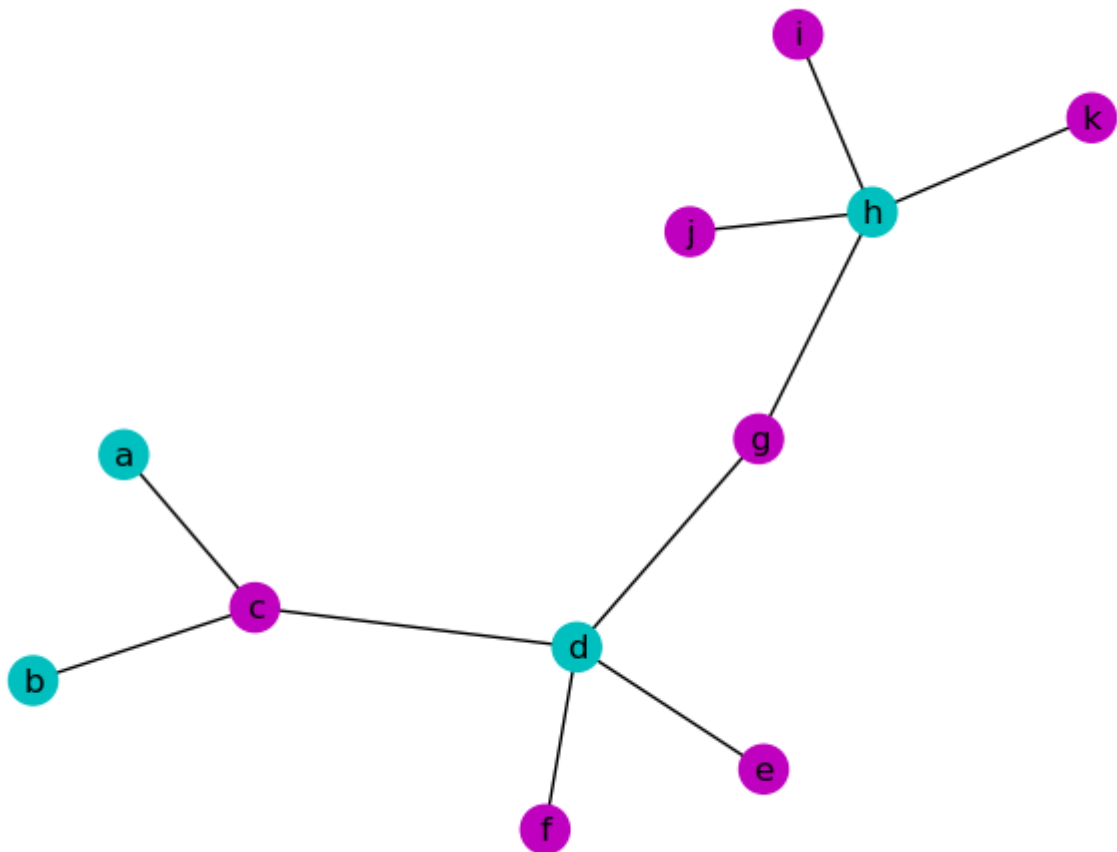
- $G$  is a tree (i.e. acyclic and connected);
- $G$  is connected and  $m = n - 1$ ;
- $G$  is a minimally connected graph (i.e., removing any edge will disconnect  $G$ );
- $G$  is acyclic and  $m = n - 1$ ;

- $G$  is a maximally acyclic graph (i.e., adding any edge will introduce a cycle in  $G$ ).
- There is a unique path between each pair of nodes in  $G$ .

## Another fact about trees

**All trees are bipartite.** There are a few ways of thinking about this. One is the a graph is bipartite if has no cycles of odd-length. Since a tree has no cycles - it must be bipartite!

```
In [3]: G3 = nx.Graph(["ac","bc","cd","de", "df", "dg","gh", "hi", "hj", "hk"])
top,bottom = nx.bipartite.sets(G3)
G3_colours = ['c' if node in top else 'm' for node in G3.nodes()]
nx.draw(G3, node_color=G3_colours, with_labels=True)
```



## How many trees are there?

1. There is one tree with a single node.
2. There is also just one tree with two nodes.
3. We can easily see that there are 3 trees with 3 nodes (see notes on the board).
4. After that, it gets a little harder to count!

## Cayley's Formula

**Theorem (Cayley's Formula).** There are exactly  $n^{n-2}$  distinct (labelled) trees on the  $n$ -element vertex set  $X = \{0, 1, 2, \dots, n-1\}$ , if  $n > 1$ .

We'll later see why this is true. But let's see what the numbers look like:

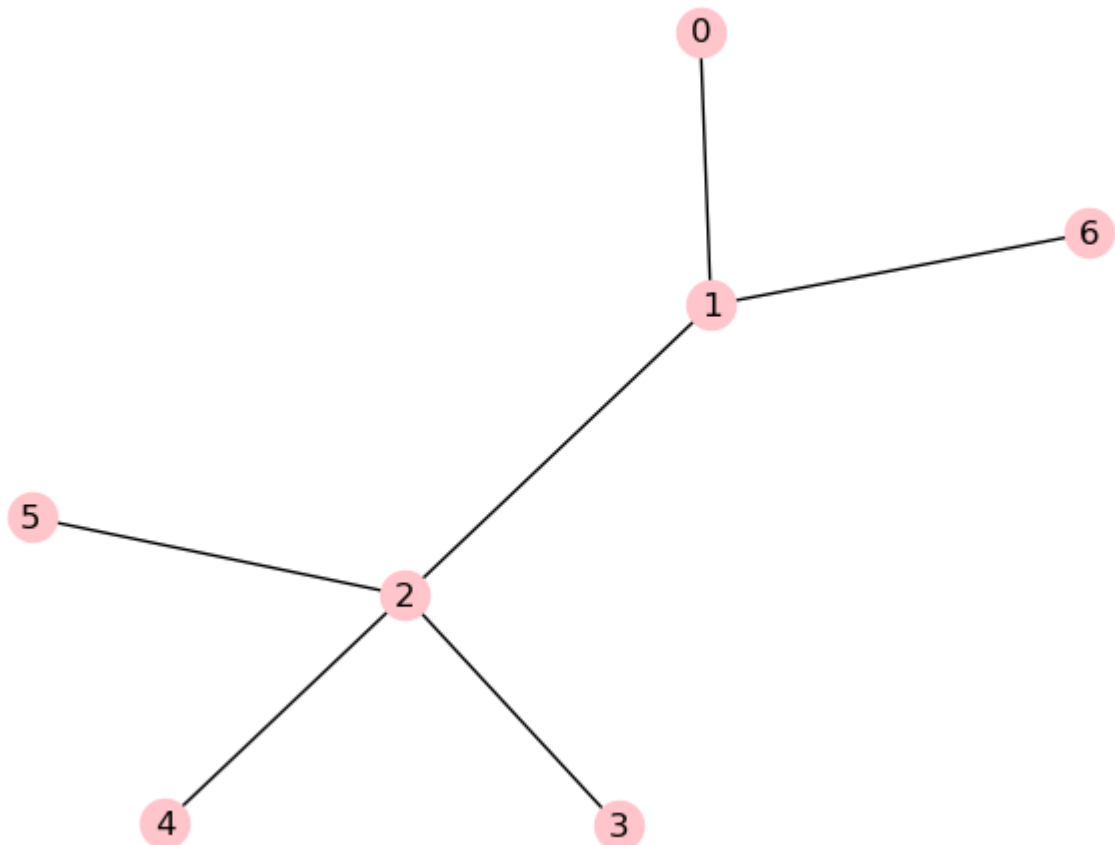
```
In [4]: domain = range(2, 10)
print(np.array([domain, [n**(n-2) for n in domain]]))
```

[	2	3	4	5	6	7	8	9]
[	1	3	16	125	1296	16807	262144	4782969]]

To see why this is true, we'll learn about **Prüfer Codes**.

Let's look at an example: a tree of order  $n = 7$

```
In [5]: T4 = nx.Graph()
T4.add_nodes_from(range(0,7))
T4.add_edges_from([(0,1),(1,2),(2,3),(2,4),(2,5),(1,6)])
nx.draw(T4, **opts)
```



## Computing the Prüfer code

How to determine the Prüfer code of a tree  $T$  (destructively):

- Start with a tree,  $T$  with nodes labeled  $0, 1, \dots, n - 1$ , and empty list **a**.
1. Find the **leaf**  $x$  with the smallest label (a "*leaf*" is a node of degree 1. Every tree must have at least 2).
  2. Append the label of its unique neighbour,  $y$  to the list **a**
  3. Remove  $x$  (and the edge  $x - y$ ) from  $T$ .
  4. Repeat Steps 1-3 until  $T$  has only 2 nodes left. We now have the code as a list of length  $n - 2$ .

So the graph above has Prüfer code  $\{1, 2, 2, 2, 1\}$

We'll write some code to compute the Prüfer code of a tree.

Since the algorithm is recursive, we first write a function that does Steps 1-3:

- Find the **leaf**  $x$  with the smallest label
- Set  $y$  to be its neighbour.
- Delete  $x$  from  $T$
- Return  $y$

One of the steps involves finding the neighbour of  $x$ . A minor technical issue is that the method `T.neighbours(x)` returns a iterable. To get its one and only item, we'll use the `next()` function (there are a few other ways to do this, including converting it to a list)`.

```
In [6]: # Does Steps 1-3 of the Algorithm
def pruefer_node(tree):
    for x in tree: # go through nodes in order
        if tree.degree(x) == 1: # first one of degree 1 (is a leaf)
            y = next(tree.neighbors(x)) # y is its only neighbour
            tree.remove_node(x)
            return(y)
```

Since our function destroys the list, we'll make a copy before we start. Also, since we know the list has length  $n - 2$ , we just call this function  $n - 2$  times, adding the value returned to the list:

```
In [7]: n = T4.order()
T = T4.copy()
a = [] # empty list
for k in range(n-2):
    y = pruefer_node(T)
    a+= [y]
print(a)
```

```
[1, 2, 2, 2, 1]
```

If you prefer list comprehension:

```
In [8]: T = T4.copy()
a = [pruefer_node(T) for k in range(n-2)]
print(a)
```

```
[1, 2, 2, 2, 1]
```

Let's wrap this up as a python function

```
In [9]: def pruefer_code(tree):
        return [pruefer_node(tree) for k in range(tree.order() - 2)]
```

Test it:

```
In [10]: T = T4.copy()
code = pruefer_code(T)
code
```

```
Out[10]: [1, 2, 2, 2, 1]
```

## Making a tree from a Prüfer code

Maybe surprisingly, the tree can be reconstructed from its Prüfer code. This is based on the following fact and shows that the map from trees to codes is a bijection!

**Fact:** The degree of node  $x$  is 1 plus the number of entries  $x$  in the Prüfer code of  $T$ .

### Example

```
In [11]: d = n*[1] # list of n 1's/  
for k in code:  
    d[k] += 1  
print(f"degree list: {d}")
```

degree list: [1, 3, 4, 1, 1, 1, 1]

```
In [12]: print(f'Check: {[T4.degree[x] for x in T4]}')
```

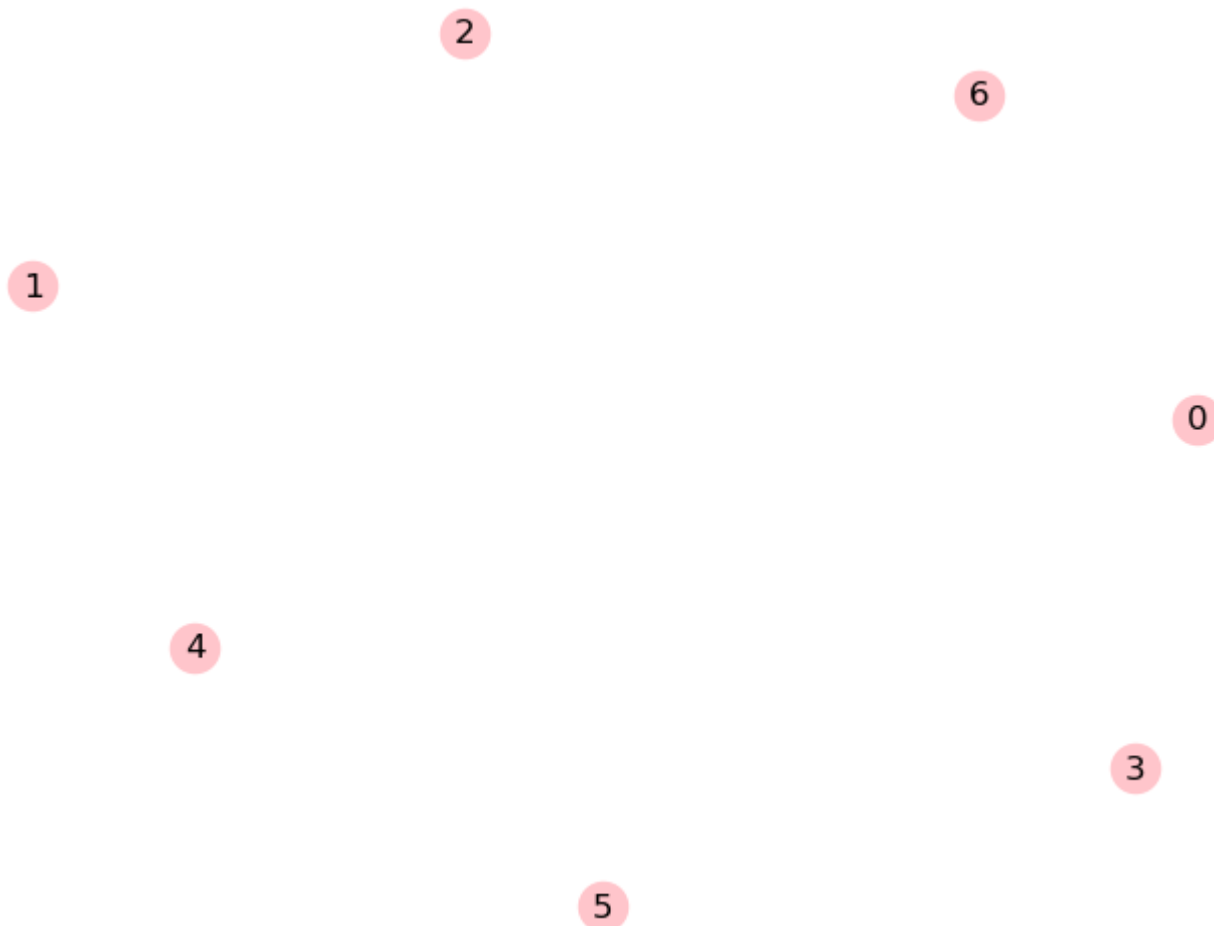
Check: [1, 3, 4, 1, 1, 1, 1]

*How to compute a tree from a Prüfer code  $a$*  (Note that  $a$  is a list of length  $n - 2$ , with all entries numbers 0 to  $n - 1$ ).

1. Set  $G$  to be a graph with node list  $[0, 1, 2, \dots, n-1]$  (and no edges yet).
2. Compute the list of node degrees  $d$  from the code.
3. For  $k = 0, 1, \dots, n - 2$ 
  - Set  $y = a[k]$
  - Set  $x$  to be the node with smallest degree in  $d$
  - Add the edge  $(x, y)$  to  $G$
  - Set  $d[x] = d[x] - 1$  and  $d[y] = d[y] - 1$  (i.e., decrease the degrees of both  $x$  and  $y$  by 1).
4. Finally, connect the remaining 2 nodes of degrees 1 by an edge.

*Tip:* if  $d$  is a list,  $d.index(1)$  returns the index of the first entry of  $d$  that has the value 1.

```
In [13]: T4a = nx.empty_graph( T4.order() )  
nx.draw(T4a, **opts)
```



```
In [14]: code
```

Out[14]: [1, 2, 2, 2, 1]

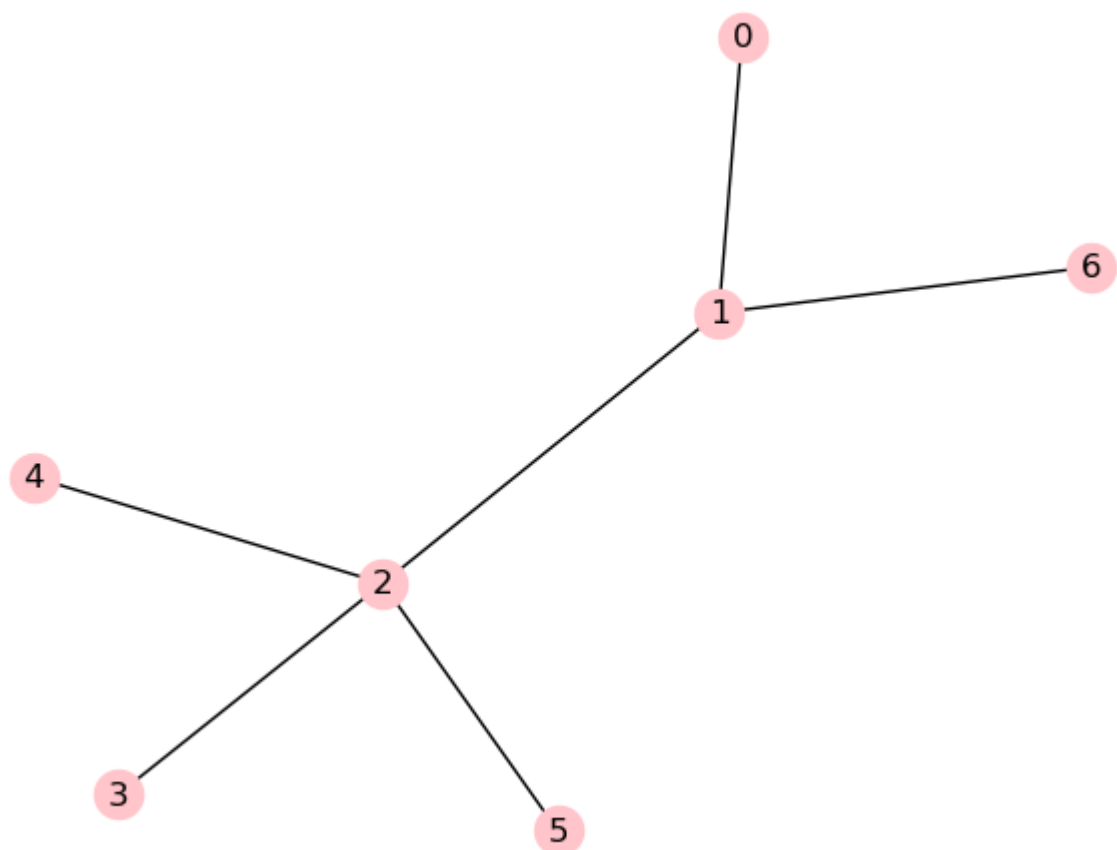
```
In [15]: d = n*[1] # list of n 1's/
for k in code:
    d[k] += 1
# repeat n-2 times:
for k in range(n-2):
    y = a[k]
    x = d.index(1) # firstly
    T4a.add_edge(x, y)
    d[x] -= 1; d[y] -= 1
    print(f'Degrees = {d} : adding edge {x}-{y}')
```

```
Degrees = [0, 2, 4, 1, 1, 1, 1] : adding edge 0-1
Degrees = [0, 2, 3, 0, 1, 1, 1] : adding edge 3-2
Degrees = [0, 2, 2, 0, 0, 1, 1] : adding edge 4-2
Degrees = [0, 2, 1, 0, 0, 0, 1] : adding edge 5-2
Degrees = [0, 1, 0, 0, 0, 0, 1] : adding edge 2-1
```

Add the final edge, by find the index to the remaining two 1's. We can find the first with `x=d.index(1)`, and the second with `y=d.index(1, x+1)` (could also use list comprehension, of course: see below).

```
In [16]: x = d.index(1)
y = d.index(1, x+1)
T4a.add_edge(x,y)
```

```
In [17]: nx.draw(T4a, **opts)
```



Finished here Wednesday

Turn the entire procedure into a python function:

```
In [18]: def pruefer_to_tree(code):
```

```

# initialize graph and defects
n = len(code) + 2
tree = nx.empty_graph(n)
d = n*[1]
for y in code:
    d[y] += 1

# add edges
for y in code:
    x = d.index(1)
    tree.add_edge(x, y)
    d[x]-=1; d[y]-=1;
# final edge
e = [x for x in tree if d[x] == 1]
tree.add_edge(*e)
return tree

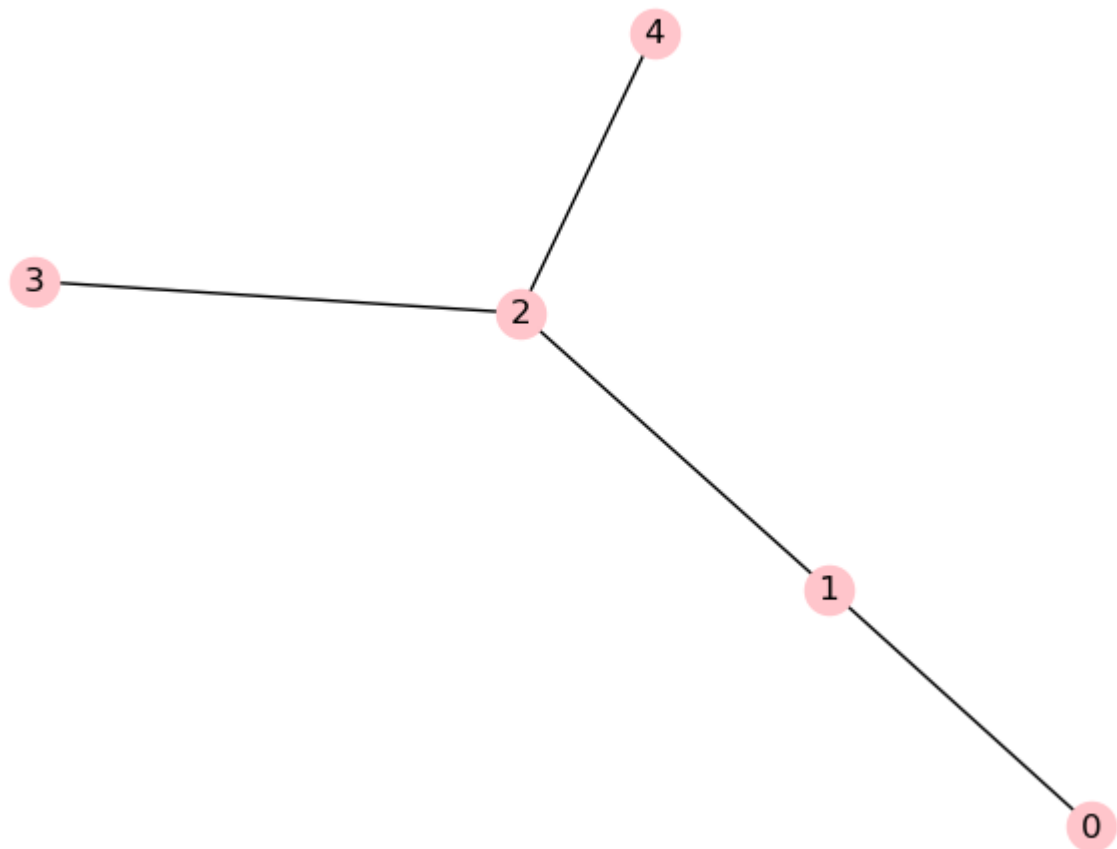
```

Let's check it works:

```

In [19]: T4b = pruefer_to_tree([1,2,2])
         nx.draw(T4b, **opts)

```



Since we have now shown that there is a bijection between labeled trees and Prüfer codes, we can prove Cayley's Theorem easily:

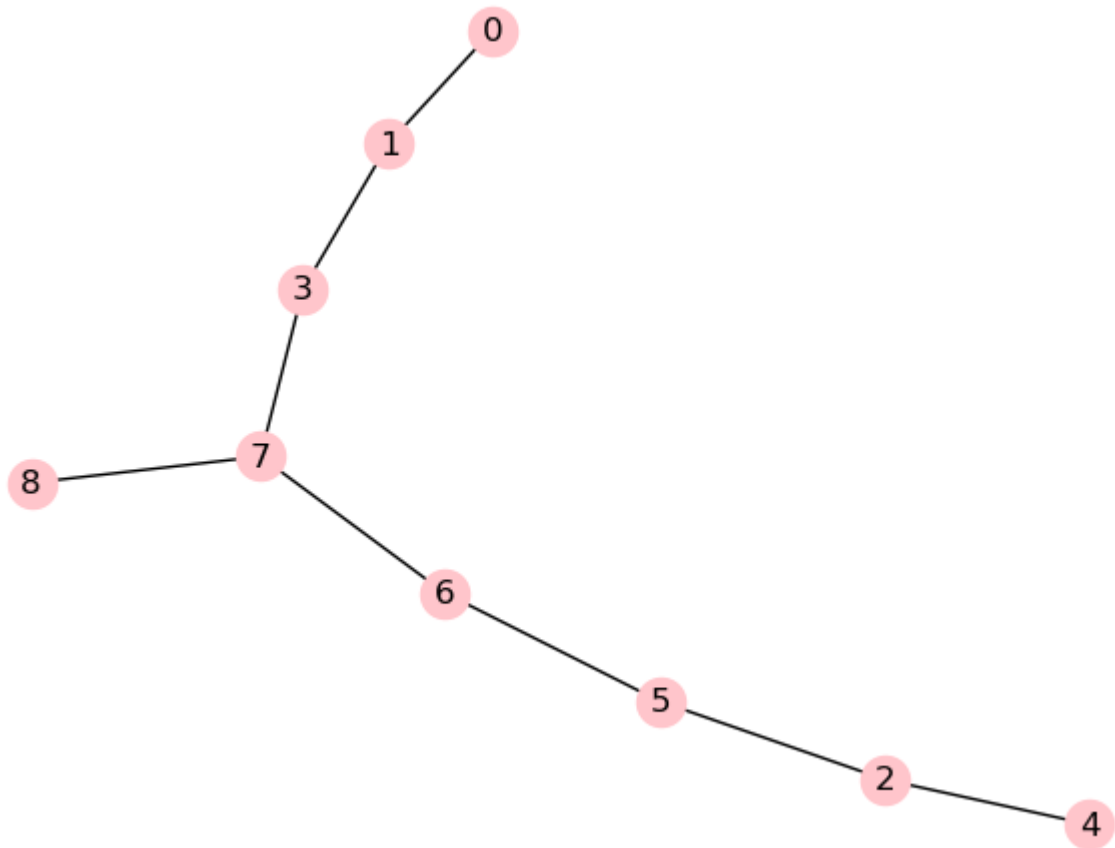
- A tree with  $n$  nodes has a Prüfer code of length  $n - 2$ .
- There are  $n$  choices for each entry in the code.
- So there are  $n^{n-2}$  possible codes for a tree with  $n$  nodes
- So there are  $n^{n-2}$  possible trees with  $n$  nodes.

## Random Trees



We can ask `networkx` to produce a **random tree** with a given number of nodes:

```
In [20]: n = 8  
T5 = nx.random_tree(9)  
nx.draw(T5, **opts)
```

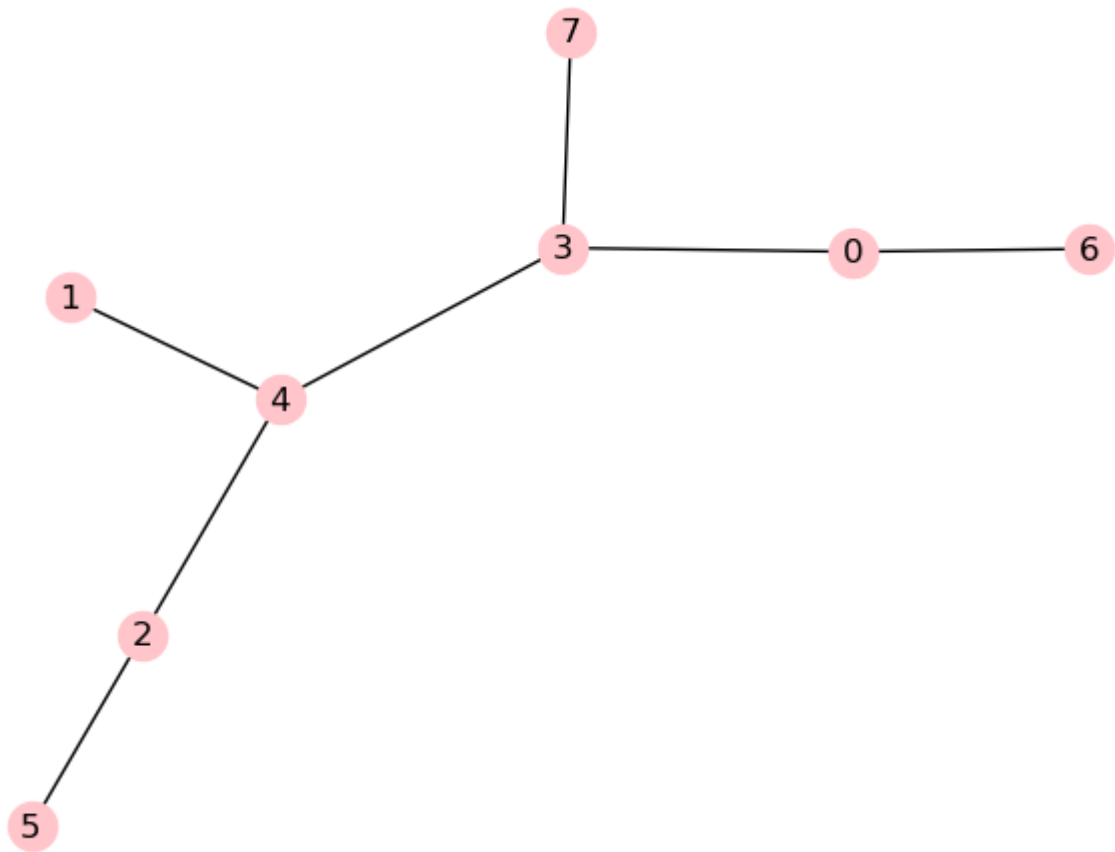


However, we can also construct a random tree on  $n$  nodes from a random Prüfer code of length  $n - 2$ .

```
In [21]: code = np.random.randint(n, size=n-2)  
print(f"code={code}")
```

```
code=[4 2 4 3 0 3]
```

```
In [22]: T5a = pruefer_to_tree(code)  
nx.draw(T5a, **opts)
```



## Graph and Tree Traversal

Often one has to search through a network to check properties of nodes (e.g., finding the node with largest degree). For large unstructured networks, this could be challenging. Fortunately, there are simple and efficient algorithms:

- DFS
- BFS

### Depth First Search (DFS)

DFS works by starting at a root node, and travelling as far along one of its branches as it can, then returning the the last unexplored branch.

The main data structure we'll need is a **stack**, also called a "*Last In First Out (LIFO) queue*". It has two operations:

- `S.push(x)` : pushes `x` onto the top of the stack (We'll use the `extend()` method)
- `y=S.pop()` : pops/removes the item from the top of the stack and stores it in `y`

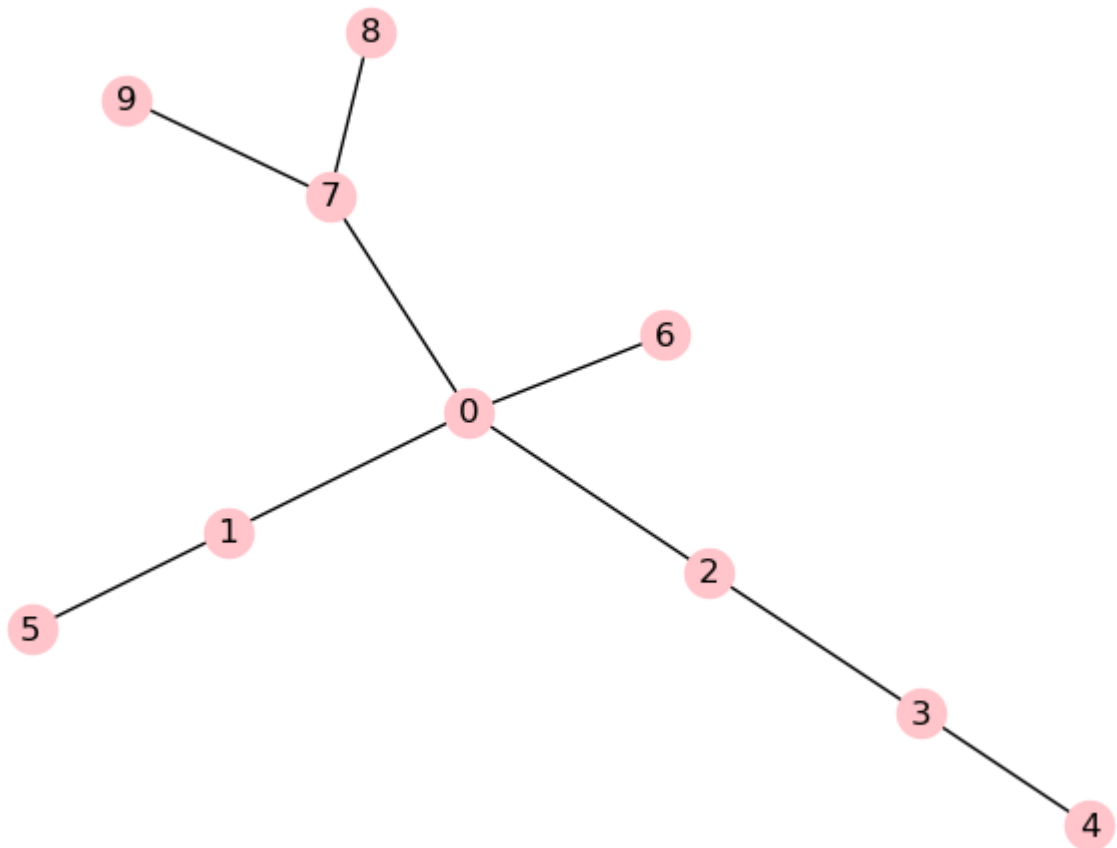
**DFS:** Given a rooted tree  $T$  with root  $x$ , visit all nodes in the tree. Start with an empty stack,  $S$  :

- `S.push(x)`
- while  $S \neq \emptyset$ :
  - `y = S.pop()`
  - `visit(y)`
  - `S.push(y.children)`

Let's create a tree to try this:

```
In [23]: T6 = nx.Graph()
T6.add_nodes_from(range(10))
T6.add_edges_from([(0,1), (0,2), (2,3), (3,4), (1,5), (0,6),(0,7),(7,8),(7,9)])
nx.draw(T6, **opts)
print(f"Edges of T6 are {T6.edges()}")
```

Edges of T6 are [(0, 1), (0, 2), (0, 6), (0, 7), (1, 5), (2, 3), (3, 4), (7, 8), (7, 9)]



Now try the algorithm

```
In [24]: T = T6.copy()
x = 0
S = [x]
while len(S) > 0:
    y = S.pop()
    S.extend(T[y])
    T.remove_node(y)
    print(y, S)
```

```
0 [1, 2, 6, 7]
7 [1, 2, 6, 8, 9]
9 [1, 2, 6, 8]
8 [1, 2, 6]
6 [1, 2]
2 [1, 3]
3 [1, 4]
4 [1]
1 [5]
5 []
```

**Breadth First Search (BFS)**

*BFS* works by starting at a root node, and explores all the neighbouring nodes ("Level 1") first. Next it searches their neighbours ("Level 2"), etc.

The main data structure we'll need is a **queue**, also called a "*First In First Out (FIFO) queue*". It has two operations:

- `Q.extend(l)` : adds the items in the list `l` to the end of `Q`
- `y=S.pop(0)` : pops/removes the *first* item from queue, and stores it in `y`

**BFS:** Given a rooted tree  $T$  with root  $x$ , visit all nodes in the tree. Start with an empty list/queue, `Q` :

- `Q.push(x)`
- while `Q ≠ ∅`:
  - `y = Q.pop(0)`
  - `visit(y)`
  - `Q.push(y.children)`

Let's test it:

```
In [25]: T = T6.copy()
x = 0
Q = [x]
while len(Q) > 0:
    y = Q.pop(0)
    Q.extend(T[y])
    T.remove_node(y)
    print(y, Q)
```

```
0 [1, 2, 6, 7]
1 [2, 6, 7, 5]
2 [6, 7, 5, 3]
6 [7, 5, 3]
7 [5, 3, 8, 9]
5 [3, 8, 9]
3 [8, 9, 4]
8 [9, 4]
9 [4]
4 []
```

Many questions on networks concerning distance and connectivity can be answered by a versatile strategy involving a subgraph which is a tree, and then searching that. Such a tree is called a **spanning tree** of the underlying graph.

## Alternative Implementations (Extra: won't do in class)

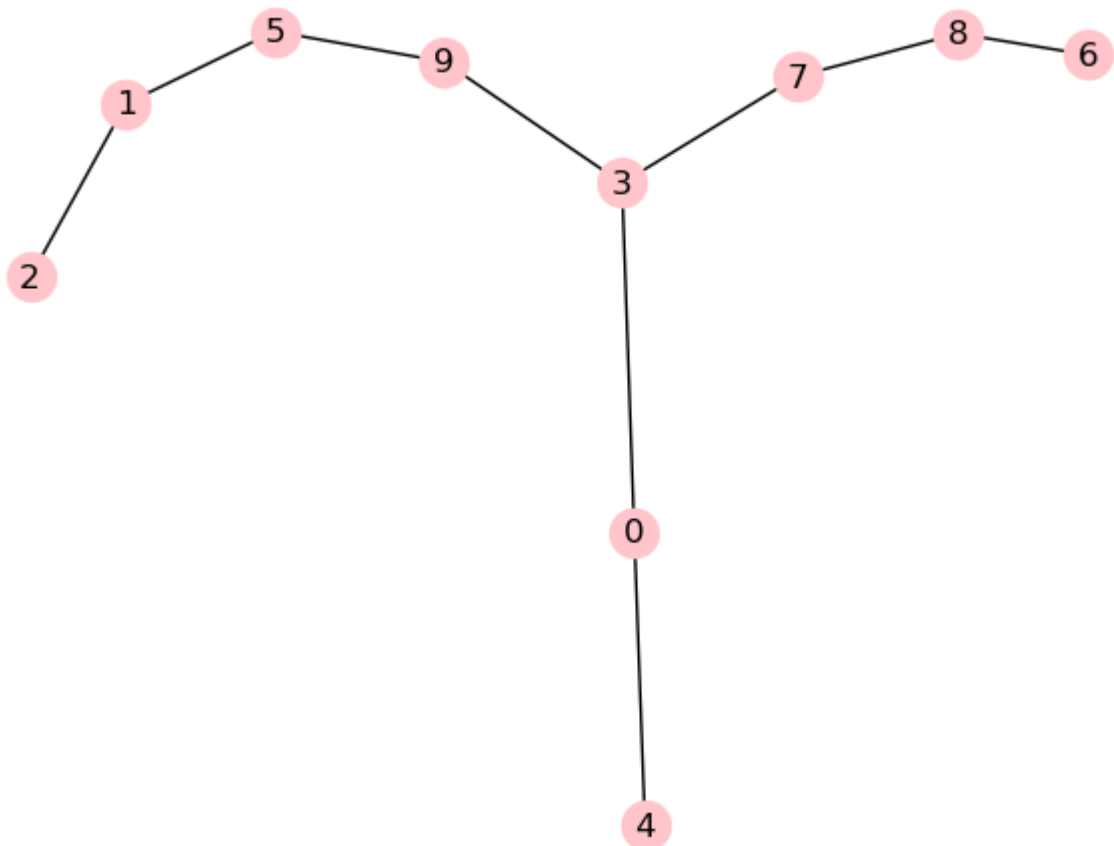
Both DFS and BFS are more like strategies, rather than specific algorithms. Different problems might require different implementations. Sometimes, the stack, or the queue don't have to be made explicit:

- In a recursive implementation, DFS can make use of the ( python ) interpreter's **function call stack**.
- BFS can take advantage of the fact that some types of lists in a ( python ) `for` loops are largely organized as **queues**.

In order to keep track of which nodes have already been visited, we maintain for each node an attribute `"seen"` that is initially `False`, and becomes `True` when the DFS/BFS visits the node.

In `networkx`, the attributes of a node `x` in a graph `G` are kept in a dictionary `G.nodes[x]`.

```
In [26]: n = 10
T6a = nx.random_tree(n)
nx.draw(T6a, **opts)
```



```
In [27]: TT = T6a.copy()
for x in TT:
    TT.nodes[x]['seen'] = False
TT.nodes['seen']
```

```
Out[27]: NodeDataView({0: False, 1: False, 2: False, 3: False, 4: False, 5: False, 6: False,
7: False, 8: False, 9: False}, data='seen')
```

- DFS on a tree:

```
In [28]: def dfs(tree, x):
print(x, end=', ')
tree.nodes[x]['seen'] = True
for z in tree[x]:
    if not tree.nodes[z]['seen']:
        dfs(tree, z)
```

```
In [29]: dfs(TT, 3)
```

```
3, 0, 4, 7, 8, 6, 9, 5, 1, 2,
```

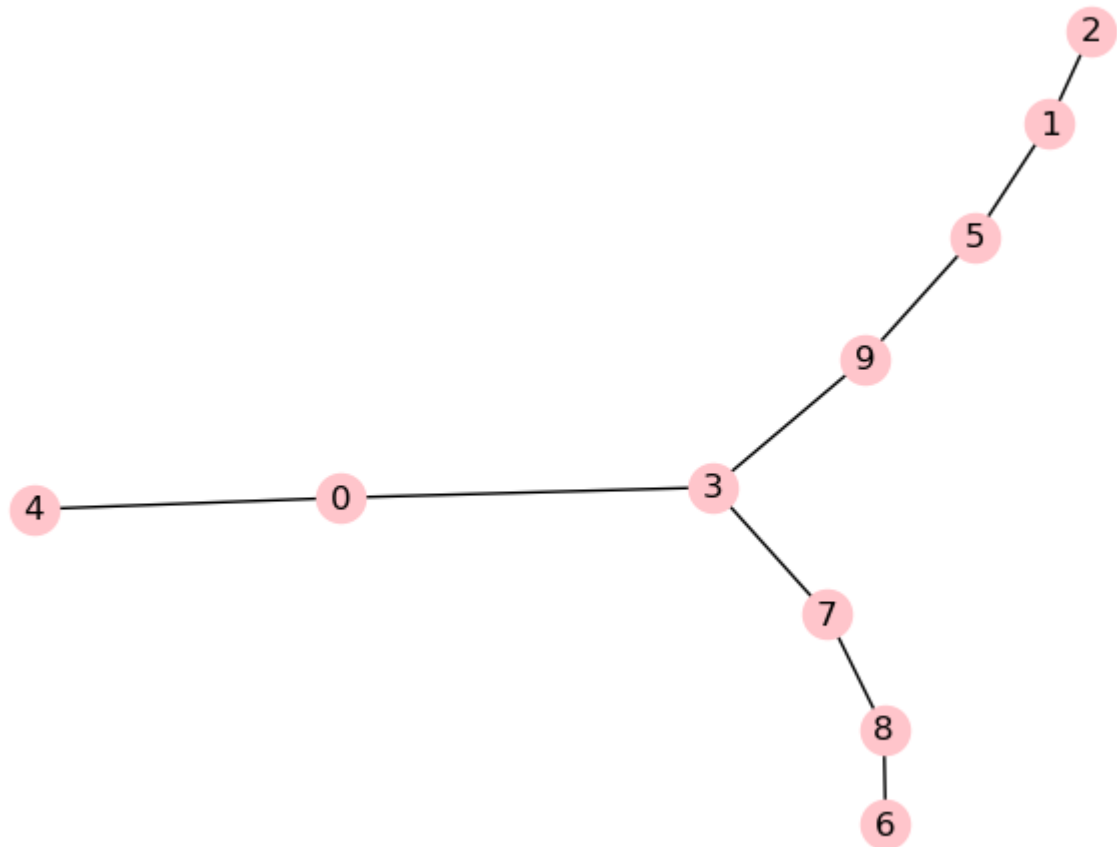
- BFS on a tree:

```
In [30]: TT = T6a.copy()
for x in TT:
    TT.nodes[x]['seen'] = False
```

```
In [31]: Q = [3]
TT.nodes[3]['seen'] = True
for y in Q:
    print(y, end=', ')
    for z in TT[y]:
        if not TT.nodes[z]['seen']:
            Q.append(z)
            TT.nodes[z]['seen'] = True
```

3, 0, 7, 9, 4, 8, 5, 6, 1, 2,

```
In [32]: nx.draw(TT, **opts)
```



## Exercises

1. A tree  $T$  uniquely determines its Prüfer code, and hence the two nodes that remain after (destructively) computing the code. What are those two nodes, in terms of properties of  $T$ , or its Prüfer code?
2. A. What tree has Prüfer code  $[0, 1, 2, \dots, n-3]$ ?  
 B. What tree has Prüfer code  $[0, 0, 0, \dots, 0]$ ?  

$$\underbrace{\hspace{1.5cm}}_{n-2 \text{ zeros}}$$
  
 C. What tree has Prüfer code  $[0, 1, 2, \dots, n-3]$ ?
3. Give the Prüfer for the tree with nodes  $\{0, 1, 2, 3, 4, 5\}$  and edges  $0-1, 0-2, 1-3, 1-4, 2-5$