CS4423-W05-2

February 13, 2025

Table of Contents

CS4423-Networks: Lecture 10 [Draft]

Week 5, Lecture 2: BFS and Graph Diameter

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.

0.0.1 Reminders:

Dates and Deadlines
* Assignment 1: 5pm Tuesday 25th February * **Class Test: 14:00, Thursday 6th March** (Week 8) * Assignment 2: Week 10 or 11 (will discuss in class)

0.0.2 Modules for this notebook

Today, we'll default to a light gree colour for nodes. It is specified in RGBA mode: three HEX digits specifying the mix of Red, Green and Blue, and an Alpha channel determining opacity. For more see

[1]:
```
import networkx as nx
import numpy as np
opts = { "with_labels": True,  "node_color": ('#0f0',.9) } # show labels; nodes␣
 ↪are opaic green
```

0.1 Again we ask: How many trees are there?

In Lecture 9 we learned about Cayley's Formula: There are exactly n^{n-2} distinct (labelled) trees on the n-element vertex set $X = \{0, 1, 2, \ldots, n-1\}$, if $n > 1$.

And we learned about Prüfer codes: * A tree on n nodes can be represented uniquely by a list if length $n-2$ where entries in the list are node labels: that is, each is an integer in the range 0 to $n-1$. * Every Prüfer code generates a unique tree (and we have a bijection between tress and codes).

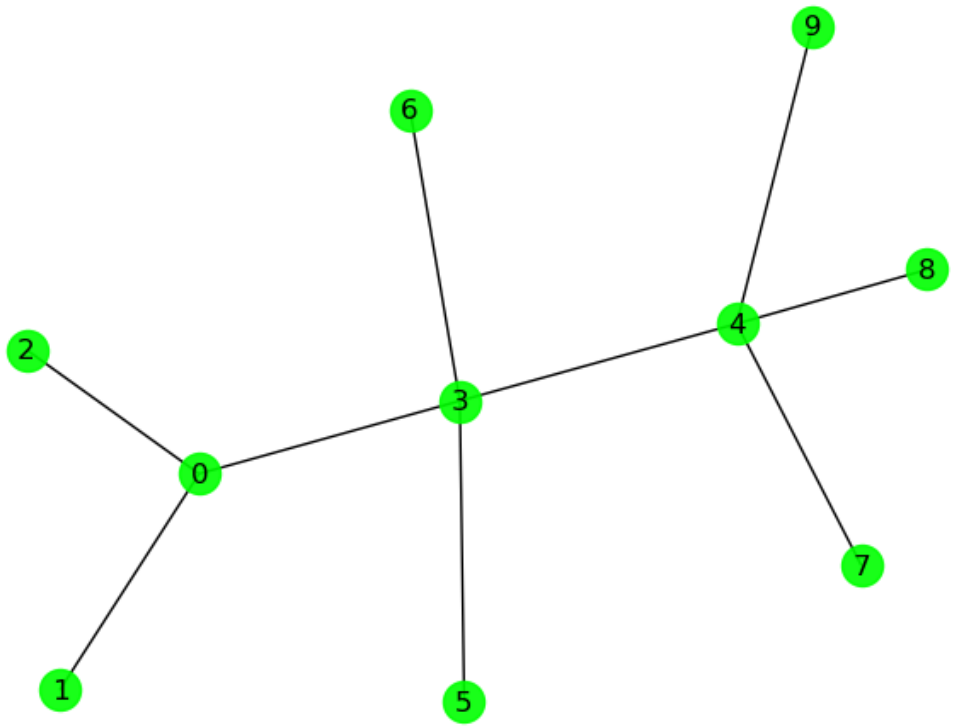
We won't go through it in class, but here is the procedure for turning a code intro a tree as a `Python` function:

[2]:
```
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:

[3]:
```
T1 = pruefer_to_tree([0,0,3,3,3,4,4,4])
nx.draw(T1, **opts)
```

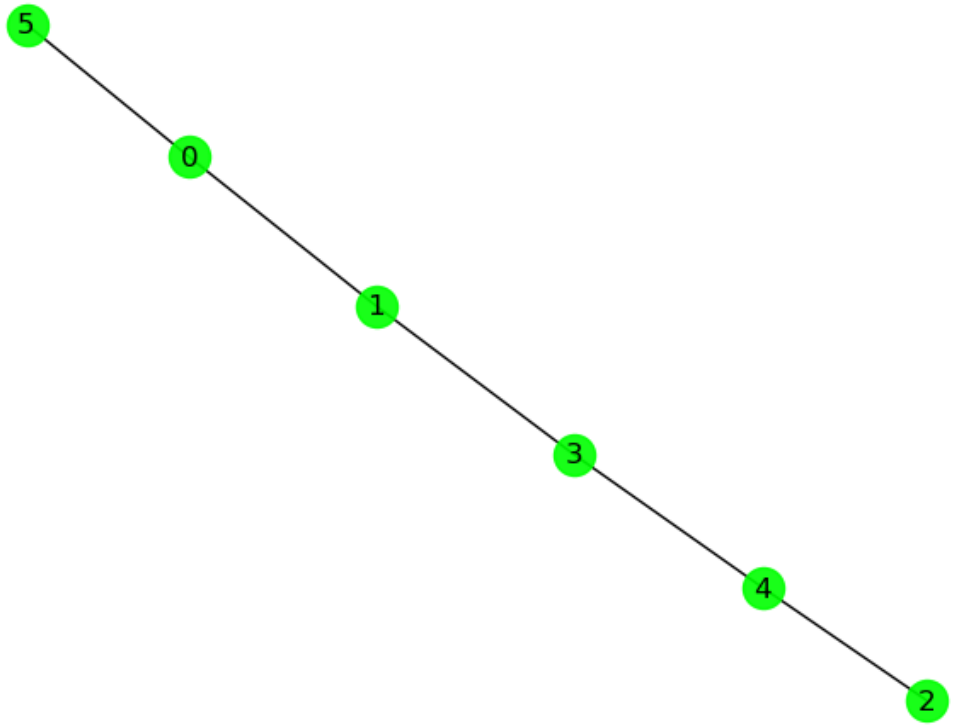


Since we have now shown that there is a bijection between labeled trees and Prüfer codes, we can prove Cayley's Theorem easily: * A tree with n nodes has a Prüfer code of length $n-2$. * There are n choices for each entry in the code. * So there are n^{n-2} possible codes for a tree with n nodes * So there are n^{n-2} possible trees with n nodes.

0.2 Random Trees

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

[4]:
```
n = 6
T2a = nx.random_labeled_tree(n)
nx.draw(T2a, **opts)
```

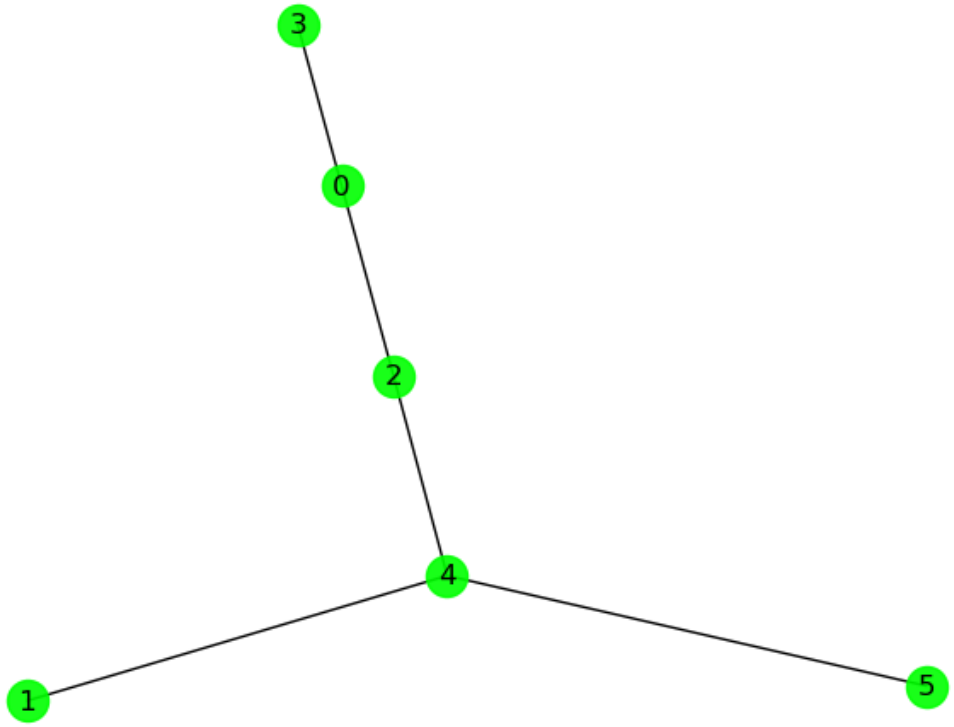


However, are can also construct a random tree on n nodes from a random Prüfer code of length $n-2$.

[5]:
```
code = np.random.randint(n, size=n-2)
print(f"code={code}")
```

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

[6]:
```
T2b = pruefer_to_tree(code)
nx.draw(T2b, **opts) # not intended to be the same as the one above
```



0.3 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: * Depth First Search: DFS * Breadth First Search: BFS

0.3.1 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

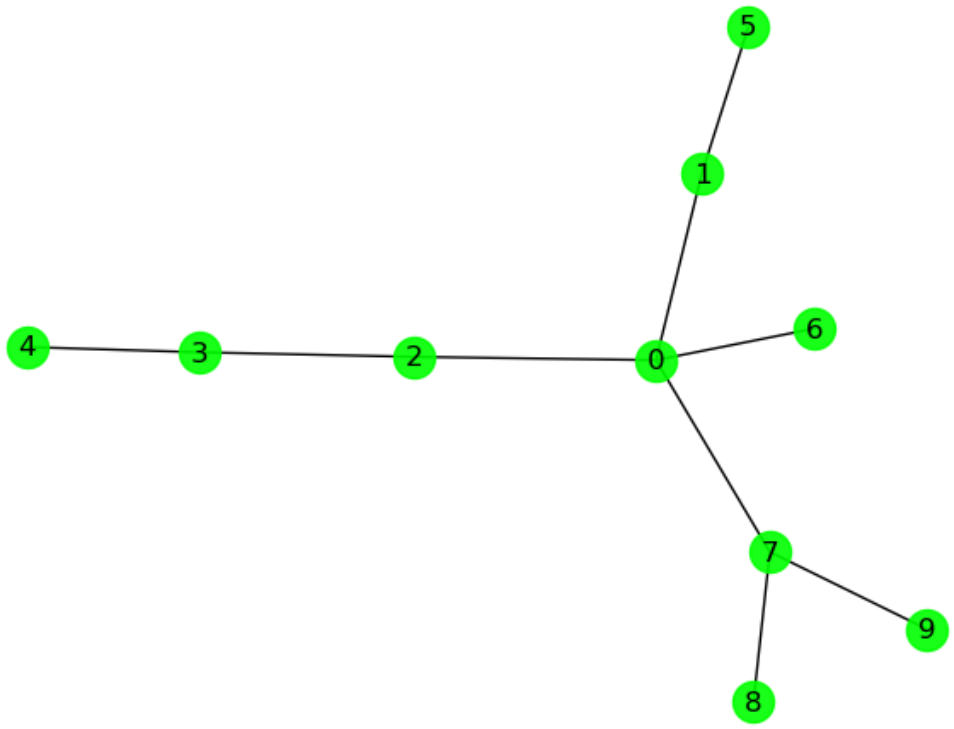
DFS: Given a rooted tree T with root x, visit all nodes in the tree. Start with an empty stack, `S`: * `S.push(x)` * while $S \neq \emptyset$: * `y = S.pop()` * visit(y) * `S.push(y.children)`

Let's create a tree to try this:

[7]:
```
T3a = nx.Graph()
T3a.add_nodes_from(range(10))
T3a.add_edges_from([(0,1), (0,2), (2,3), (3,4), (1,5), (0,6),(0,7),(7,8),(7,9)])
```

```
nx.draw(T3a, **opts)
print(f"Edges of T6 are {T3a.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

[8]:
```
T = T3a.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 []
```

0.3.2 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**](https://en.wikipedia.org/wiki/Queue_(abstract_data_type), 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 $Q \neq \emptyset$: * `y = Q.pop(0)` * visit(y) * `Q.push(y.children)`

Let's test it:

[9]:
```
T = T3a.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.

0.3.3 Alternative Implementations (Extra: will skim in class)

(This bit will be skimmed in class; can jump to Section 4).

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 explicitly:

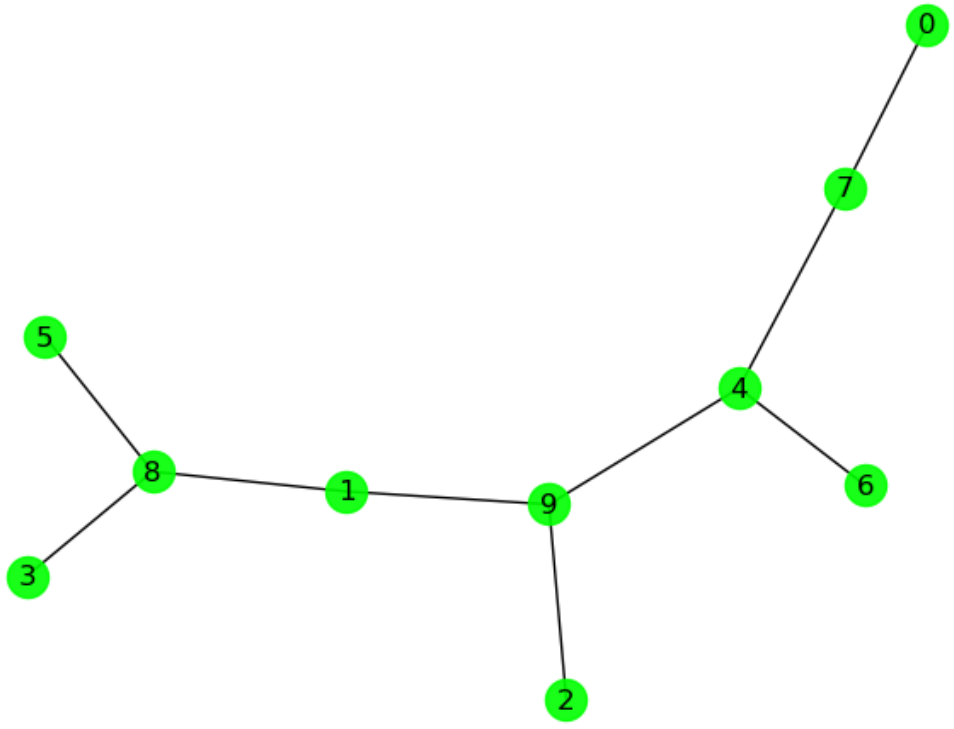
- In a recursive implementation, DFS can make use of the (`Python`) interpreter's **function call stack**.
- BFS can take advantage of the fact that some types of lists in a (`Python`) `for` loops are largely organized as **queues**.

Node attributes In `networkx` one can assign **attributes** to nodes, such as the node's colour.

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]`.

[10]:
```
n = 10
T3b = nx.random_labeled_tree(n)
nx.draw(T3b, **opts)
```



[11]:
```
TT = T3b.copy()
for x in TT:
    TT.nodes[x]['seen'] = False
```

```
print(TT.nodes())
```

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

Check a specific attribute:

[12]:
```
print(TT.nodes('seen'))
```

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

Implement DFS Implement DFS recursively on a tree with root `x` as a function:

[13]:
```
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)
```

Test it:

[14]:
```
TT = T3b.copy()
nx.set_node_attributes(TT, False, 'seen') # same as for loop above
dfs(TT, 0)
```

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

Implement BFS Implement BFS on a tree recursively

[15]:
```
TT = T3b.copy()
nx.set_node_attributes(TT, False, 'seen') # same as for loop above
```

[16]:
```
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, 8, 1, 5, 9, 2, 4, 6, 7, 0,
```

0.4 Graph Diameter

- A natural problem arising in many practical applications is the following: Given a pair of nodes x, y, find one or all the paths from x to y with the **fewest number of edges** possible.

- This is a somewhat complex measure on a network (compared to, say, statistics on node degrees). And we will need a more complex procedure, that is, an algorithm, in order to solve such problems 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** $\mathrm{diam}(G)$ of the network G is the length of the longest shortest path between any two nodes,

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

Examples (done on board): what are the diameters of these graphs? 1. K_5 2. $k_{3,3}$ 3. P_5 4. C_8

0.4.1 Breadth First Search for Distance

We now consider the following problem: Given a node $x \in X$ in a graph G, what are the distances $d(x, y)$ for all nodes $y \in X$?

We know that it is possible to answer this question by looking at sums of powers of the adjacency matrix. But that is *extremely* expensive. Also, it does not give you the paths (automatically).

Better: use *BFS*.

- BFS provides a systematic procedure for finding these distances, and the shortest paths through which they are realized.

- We will start by describing how BFS works for **graph** traversal.

In order to describe the algorithm step by step, let's recall that a node y a **neighbour** (or friend) 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 neighbours 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**.

We'll exploit the fact that the layer of a node then corresponds to its distance from the given node x.

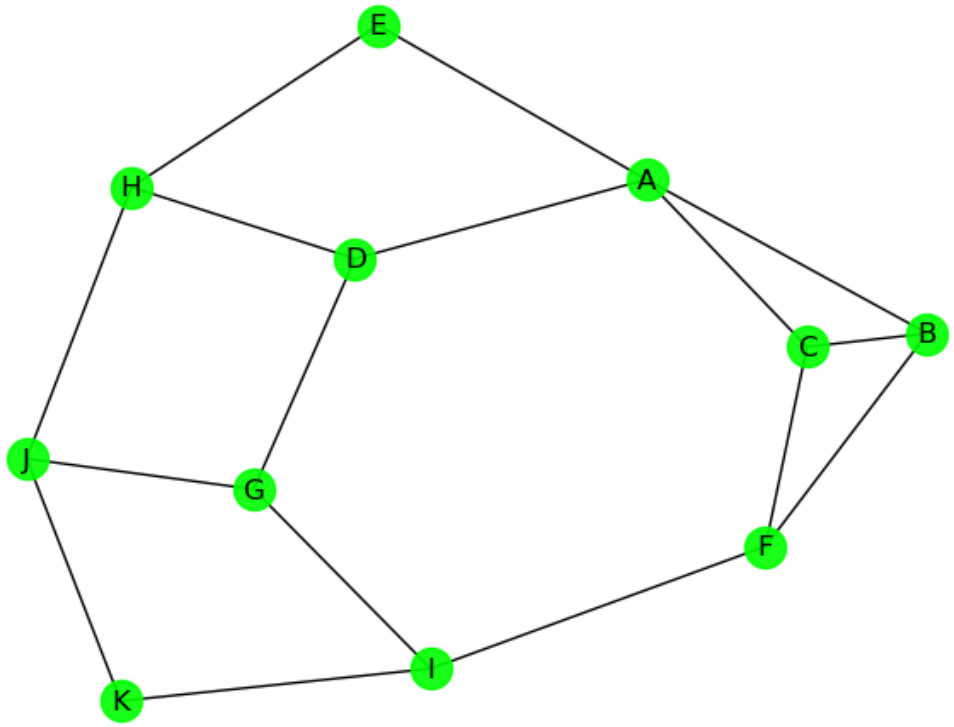
In practice, for simple graph traversal, the layers do not need to be made explicit.

We need an example of a network to work with. For a chance, let's load one from an adjacency file. *Syntax*: for each line in the file, the first listed node is a neighbour of all the others in that row.

[17]:
```
!cat bfs.adj
```

```
A B C D E
B C F
C F
D G H
E H
F I
G I J
H J
K I J
```

[18]:
```
G4 = nx.read_adjlist("bfs.adj")
nx.draw(G4, **opts)
```



We set the `seen` attribute to `False`:

[19]:
```
nx.set_node_attributes(G4, False, 'seen') # same as for loop above
print( G4.nodes['A'] ) # check
```

```
{'seen': False}
```

Initialise an empty queue, then add `A` to it, and set its `seen` attribute to `True`:

[20]:
```
Q = []
Q.append('A')
G4.nodes['A']['seen'] = True
print(f"Q={Q}")
```

```
Q=['A']
```

Now check $N(A)$

[21]:
```
list(G4.neighbors('A'))
```

[21]:
```
['B', 'C', 'D', 'E']
```

Add neighbours of A to `Q`:

[22]:
```
for y in G4.neighbors('A'):
    Q.append(y)
    G4.nodes[y]['seen'] = True

## No neighbours of A have been seen yet,
## but we'll need to add this check to the generic step of the algorithm

print(Q)
```

```
['A', 'B', 'C', 'D', 'E']
```

[23]:
```
node = 'B'
for y in G4.neighbors(node):
    if not G4.nodes[y]['seen']:
        Q.append(y)
        G4.nodes[y]['seen'] = True
print(Q)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

[24]:
```
node = 'C'
for y in G4.neighbors(node):
    if not G4.nodes[y]['seen']:
        Q.append(y)
        G4.nodes[y]['seen'] = True
print(Q)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

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

Here is how to do it in a loop:

[25]:
```
# 1. initialize
nx.set_node_attributes(G4, False, 'seen') # same as for loop above

G4.nodes['A']['seen'] = True
Q = ['A']

# 2. loop
for node in Q:
    for y in G4.neighbors(node):
        if not G4.nodes[y]['seen']:
            Q.append(y)
            G4.nodes[y]['seen'] = True

# 3. output result
print(f"Q = {Q}")
```

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

When this process is formulated as an algorithm, we use an explicit **queue** (FIFO buffer) to keep track of the node whose neighbors are currently under consideration.

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.

0.5 BFS for Distance

Breadth First Search for Distance. 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 $X = \{x_0, x_1, \ldots, x_{n-1}\}$ and that $x = x_j$. Set $d_i \leftarrow \perp$ (undefined) for $i = 0, \ldots, n-1$. 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 $(d_0, \ldots, d_{n-1})$.

Note how in this version of BFS, in contrast to the simple version, a node is visited (setting its d attribute) immediately when it is pushed onto Q, rather than later when it pops off Q.

[26]:
```
nx.set_node_attributes(G4, None, 'd')
x = 'B'              #starting BFS at vertex B
G4.nodes[x]['d'] = 0 # and setting its distance to 0
Q = []
Q.append(x)
print(Q)
```

```
['B']
```

[27]:
```
while len(Q)>0:
    x = Q.pop(0)
    for y in G4.neighbors(x):
        if G4.nodes[y]['d'] is None: # checking if the distance is undefined
            G4.nodes[y]['d'] = G4.nodes[x]['d'] + 1 # if so, define using␣
 ↪previous
            Q.append(y)
    print(f"{x} : {Q}")
```

```
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 : []
```

[28]:
```
print([G4.nodes[x]['d'] for x in G4])
```

```
[1, 0, 1, 2, 2, 1, 3, 3, 2, 4, 3]
```

0.5.1 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.

[29]:
```
nx.set_node_attributes(G4, None, 'd')
x = 'A' # start with vertex A
G4.nodes[x]['d'] = 0    # set its distance to 0
Q = [] # initialise a queue Q
Q.append(x)  # push x in Q

nx.set_edge_attributes(G4, False, 'seen')
```

[30]:
```
while len(Q)>0:
    x = Q.pop(0) # pop a vertex from the queue
```

```
    for y in G4.neighbors(x):
        if not G4.nodes[y]['d']: # undefined?
            G4.nodes[y]['d'] = G4.nodes[x]['d'] + 1 # set distance
            Q.append(y) # push in queue
            G4.edges[x, y]['seen'] = True # set relevant edge to seen
    print(x, ": ", Q)
```

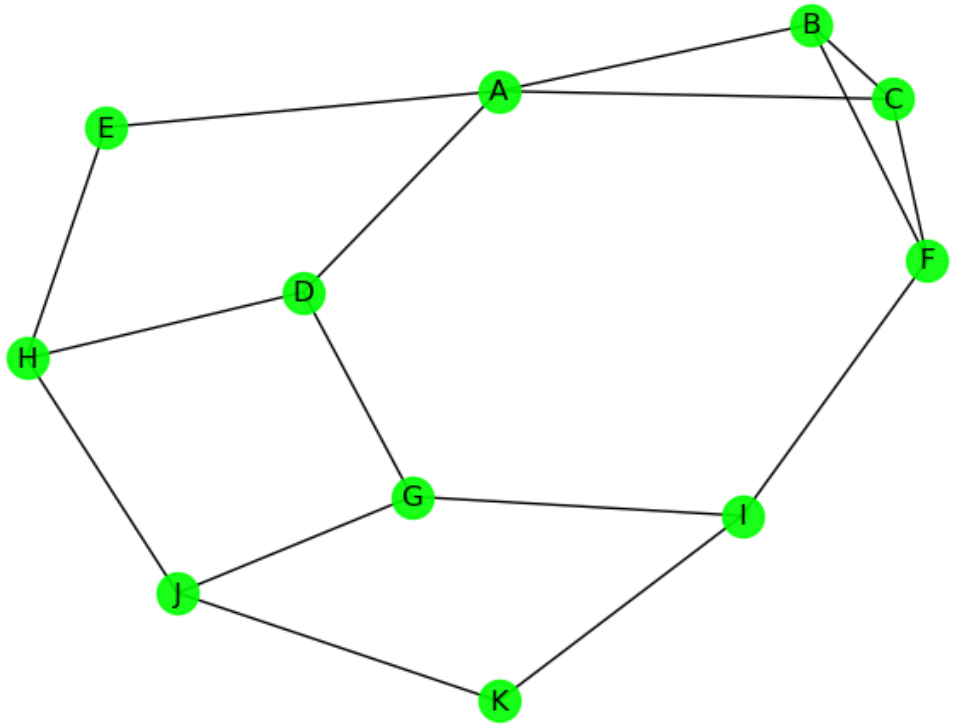
```
A :  ['B', 'C', 'D', 'E']
B :  ['C', 'D', 'E', 'A', 'F']
C :  ['D', 'E', 'A', 'F']
D :  ['E', 'A', 'F', 'G', 'H']
E :  ['A', 'F', 'G', 'H']
A :  ['F', 'G', 'H']
F :  ['G', 'H', 'I']
G :  ['H', 'I', 'J']
H :  ['I', 'J']
I :  ['J', 'K']
J :  ['K']
K :  []
```

[31]:
```
print(G4.edges())
```

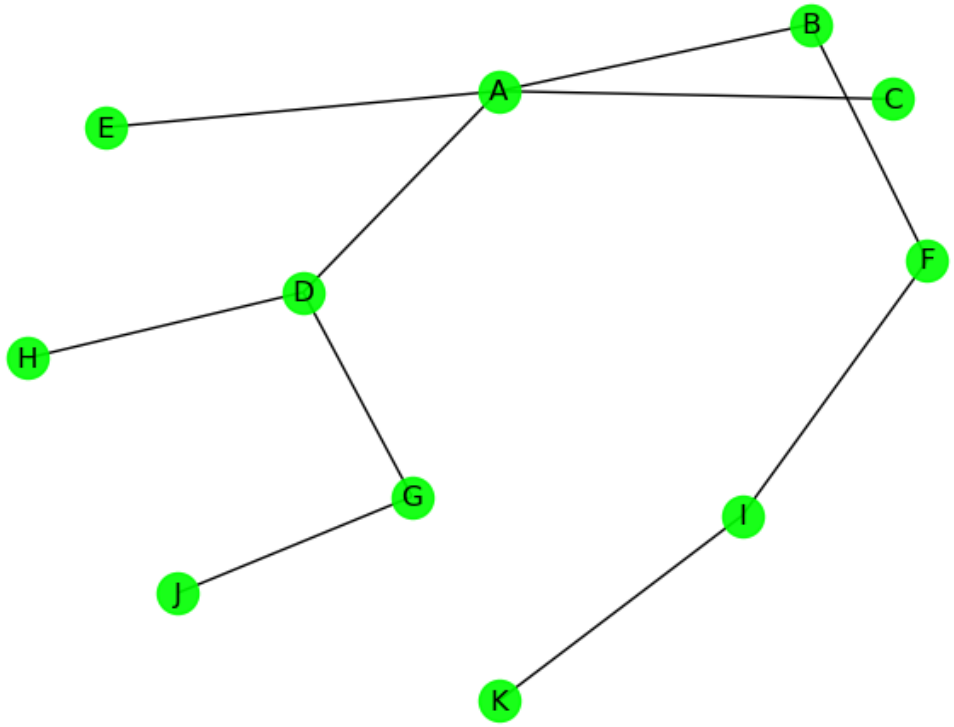
```
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('A', 'E'), ('B', 'C'), ('B', 'F'), ('C',
'F'), ('D', 'G'), ('D', 'H'), ('E', 'H'), ('F', 'I'), ('G', 'I'), ('G', 'J'),
('H', 'J'), ('I', 'K'), ('J', 'K')]
```

[32]:
```
sub = [e for e in G4.edges if G4.edges[e]['seen']]
# subset of edges 'seen' while visiting the graph
```

[33]:
```
pos = nx.spring_layout(G4)
nx.draw(G4, **opts, pos=pos)
```

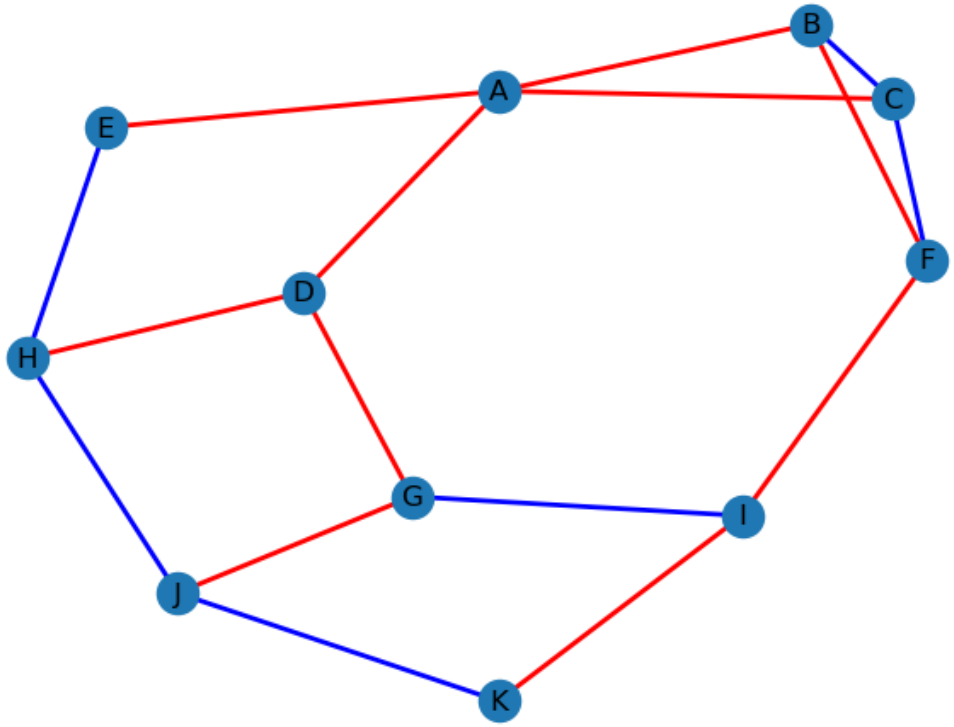


[34]:
```
nx.draw(G4.edge_subgraph(sub), **opts, pos=pos)
```



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).

[35]:
```
colors = ['red' if G4.edges[e]['seen'] else 'blue' for e in G4.edges]
nx.draw(G4, edge_color = colors, with_labels = True, width=2.0, pos=pos)
```



- 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 nodes $x \in X$ in turn.
- As an exercise in a future assignment, you will see more in detail an implementation of BFS aimed at constructing a spanning tree.
- 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].

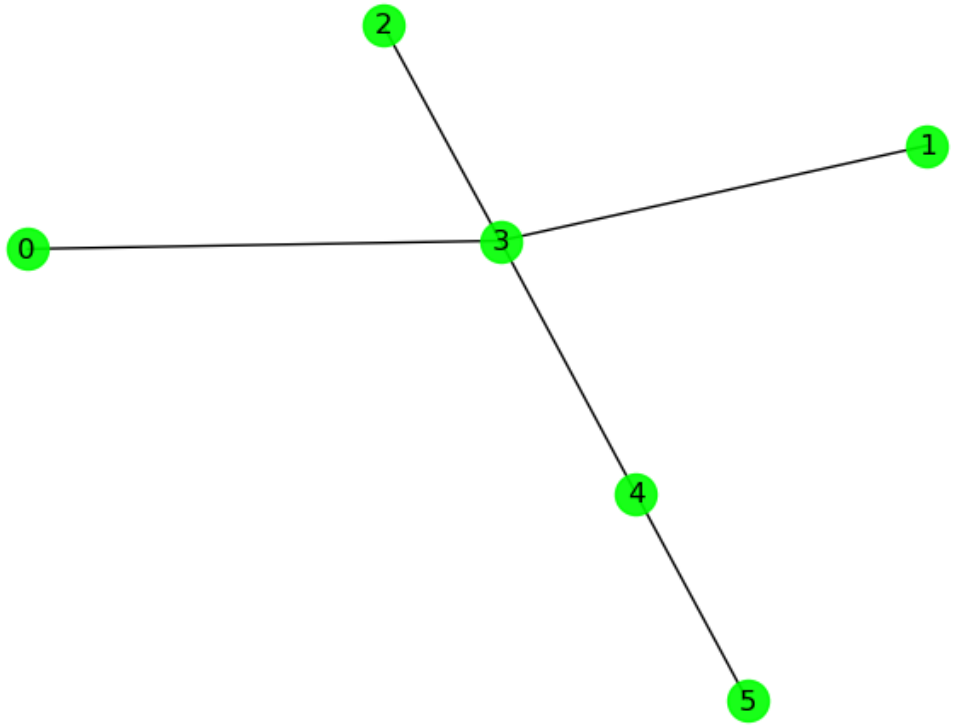
0.6 Code Corner

Here we summarise any new `Python` or `networkx` functions/syntax we met today, or some functions that might be useful. This section is not covered in class.

0.6.1 Pruefer codes in 'networkx'

Make a tree (for a change, just by defining the edges)

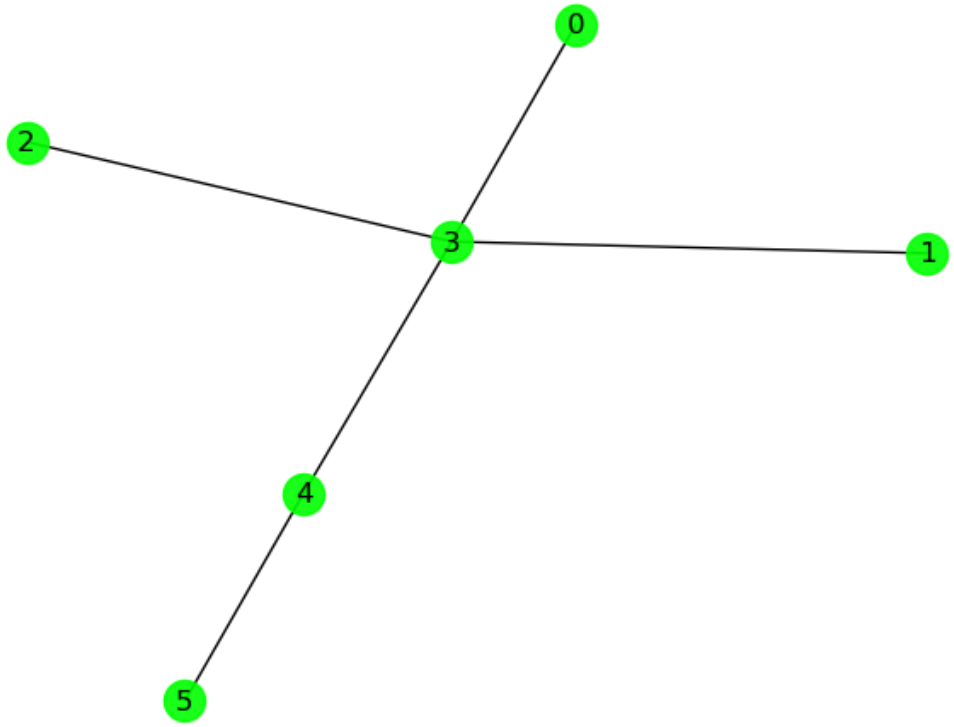
[36]:
```
edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
TCCa = nx.Graph(edges)
nx.draw(TCCa, **opts)
```



[37]:
```
code = nx.to_prufer_sequence(TCCa) # get the Pruefer code
print(code)
```

```
[3, 3, 3, 4]
```

[38]:
```
TCCb = nx.from_prufer_sequence(code) # get tree from Pruefer code
nx.draw(TCCb, **opts)
```



0.6.2 Setting node attributes

These are the same:

[39]:
```
TT = TCCb.copy()
nx.set_node_attributes(TT, False, 'seen')
```

[40]:
```
for x in TT:
    TT.nodes[x]['seen'] = False
```

Finished here Thursday

0.7 Exercises

1. Find the diameters of the following graphs:
    1. $K_{m,n}$ for $m, n > 0$
    2. K_n for $n > 0$
    3. P_n, for $n > 1$
    4. C_n. for $n > 2$
    5. The Petersen Graph
2. (Q1(b)+(d) from 2023/2024 Exam). Consider the graph on the nodes $a, b, c, \ldots, h$, with edges $a - b$, $a - c$,

$b - c$, $b - d$, $b - e$,
$c - e$,
$d - e$, $d - f$,
$e - f$, $e - g$, and
$g - h$.

- Use BFS to find the shortest distance between a and all other nodes;
- Use BFS, starting at a, to construct a spanning tree of the graph.