CS4423-W05-2

February 17, 2025

Table of Contents

CS4423-Networks: Lecture 10 [Final]

Week 5, Lecture 2: BFS and Graph Traversal

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 $X = \{0, 1, 2, \dots, n-1\}$, 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).

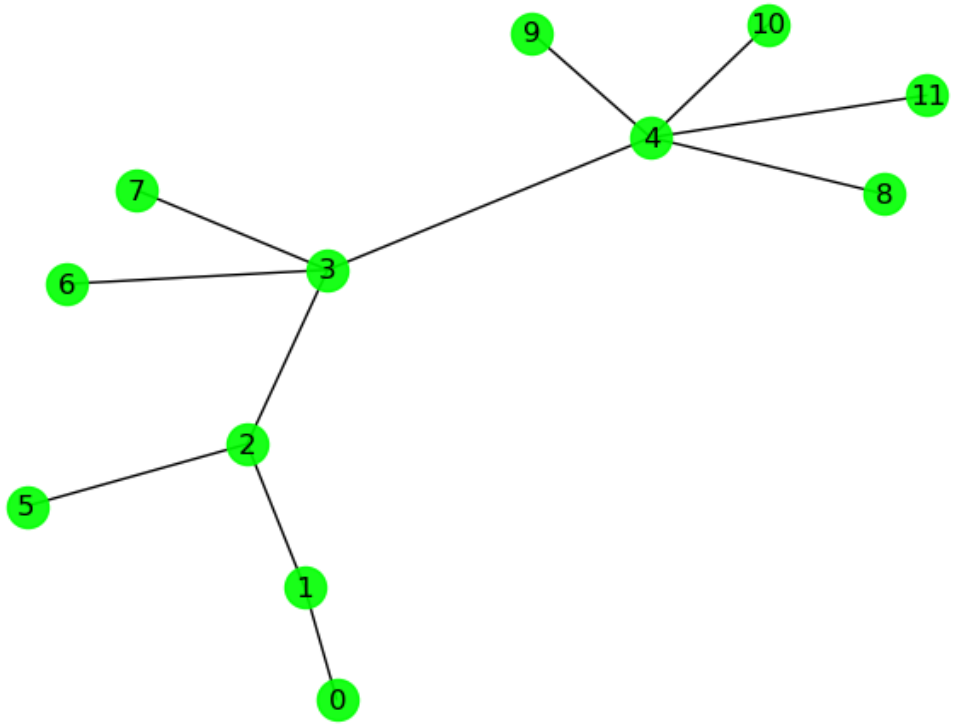
We won't go through it in class, but here is the procedure for turning a code intro a tree as a `Python` function:

[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([1,2,2,3,3,3,4,4,4,4])
nx.draw(T1, **opts)
```

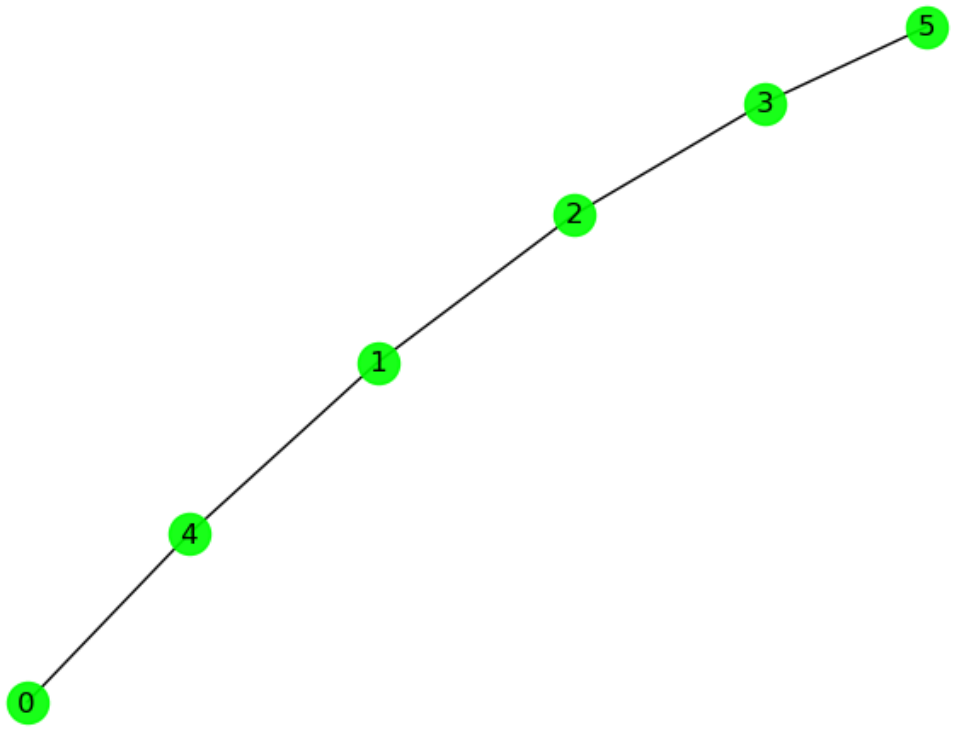


Since we have now shown that there is a bijection between labeled trees and Prüfer codes, we can prove Cayley's Theorem easily: * A tree with n nodes has a Prüfer code of length $n-2$. * There are n choices for each entry in the code. * So there are n^{n-2} possible codes for a tree with n nodes * So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

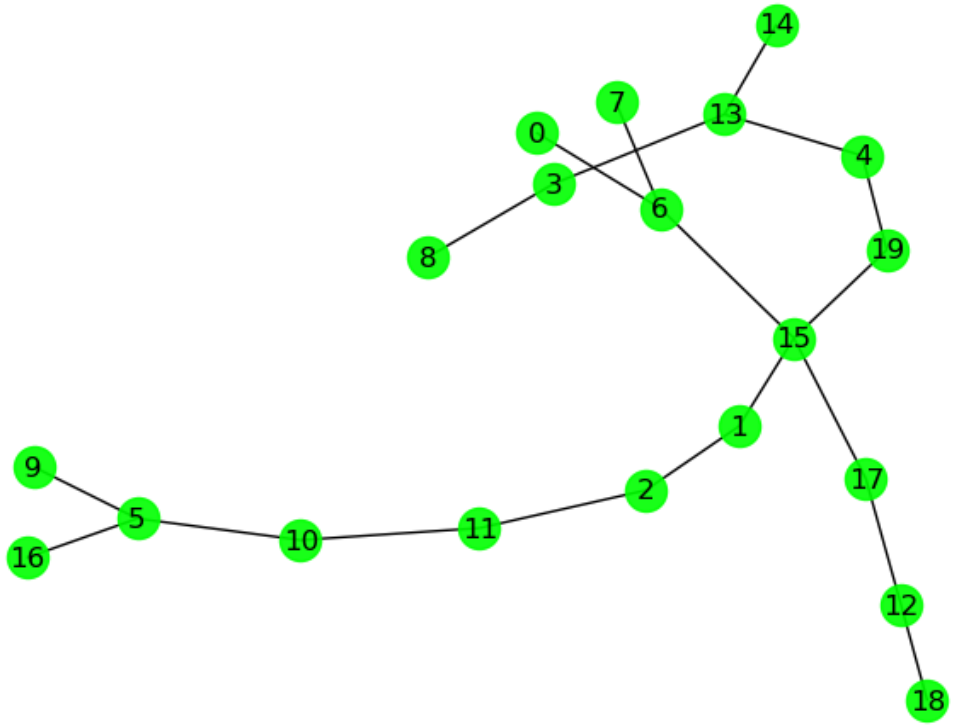


However, are can also construct a random tree on n nodes from a random Prüfer code of length $n-2$.

[5]:
```
n = 20
code = np.random.randint(n, size=n-2)
print(f"code={code}")
```

```
code=[ 6  6 15  3 13  5 13  4 19  5 10 11  2  1 15 12 17 15]
```

[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

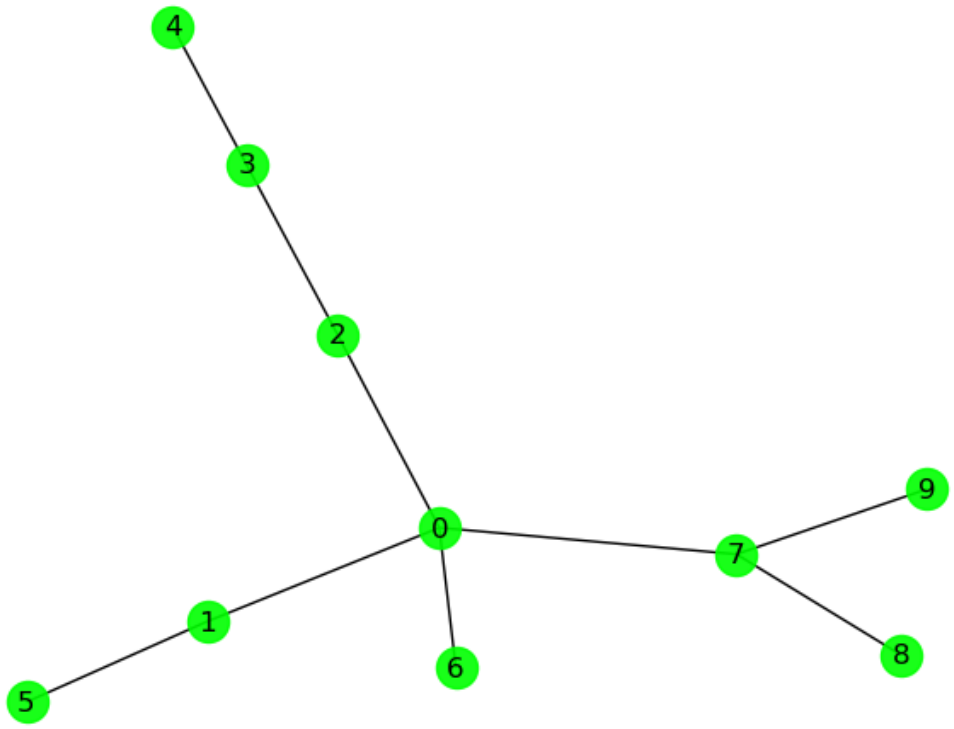
DFS: Given a rooted tree T with root x, visit all nodes in the tree. Start with an empty stack, `S`: * `S.push(x)` * while $S \neq \emptyset$: * `y = S.pop()` * visit(y) * `S.push(y.children)`

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 T3a are {T3a.edges()}")
```

```
Edges of T3a 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]) # add the list T[y] to the end of the list S
    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:

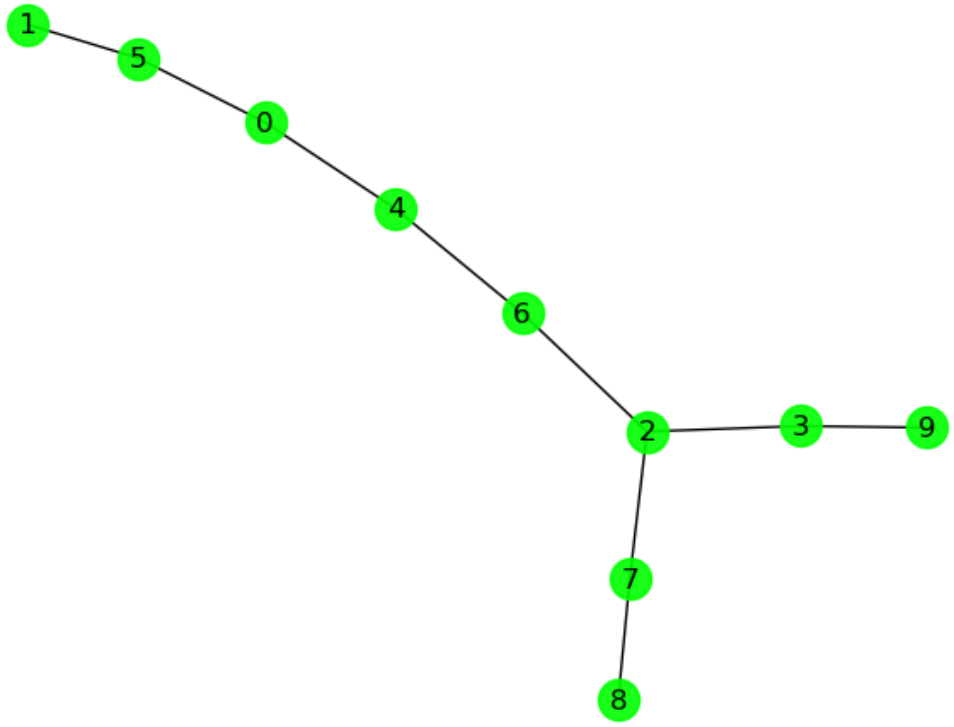
- In a recursive implementation, DFS can make use of the (`Python`) interpreter's **function call stack**.
- BFS can take advantage of the fact that some types of lists in a (`Python`) `for` loops are largely organized as **queues**.

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

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

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

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

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

Finished here Thursday

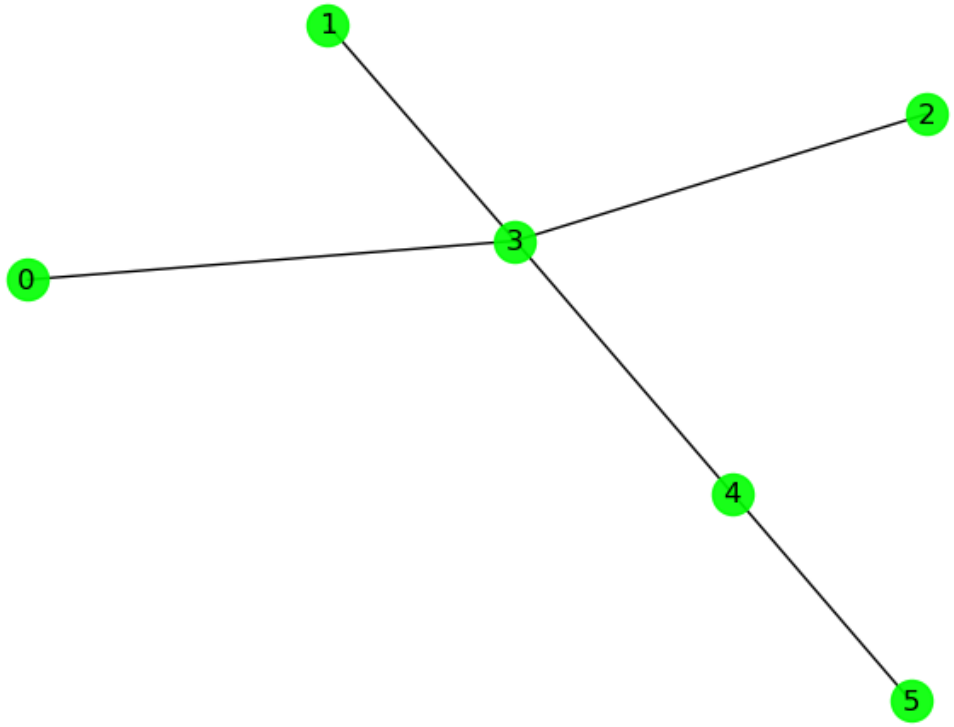
0.5 Code Corner

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.

0.5.1 Pruefer codes in 'networkx'

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

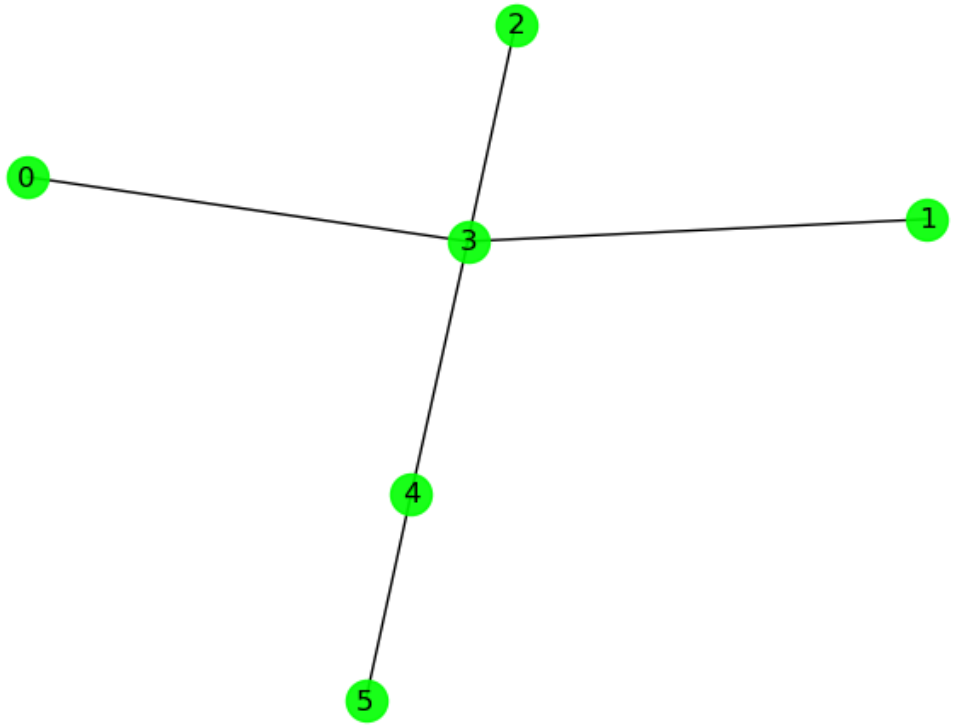
[17]:
```
edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
TCCa = nx.Graph(edges)
nx.draw(TCCa, **opts)
```



[18]:
```
code = nx.to_prufer_sequence(TCCa) # get the Pruefer code
print(code)
```

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

[19]:
```
TCCb = nx.from_prufer_sequence(code) # get tree from Pruefer code
nx.draw(TCCb, **opts)
```



0.5.2 Setting node attributes

These are the same:

[20]:
```
TT = TCCb.copy()
nx.set_node_attributes(TT, False, 'seen')
```

[21]:
```
for x in TT:
    TT.nodes[x]['seen'] = False
```

0.6 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. G is a graph with n nodees and a diameter of 1. Is G the complete graph on n nodes?