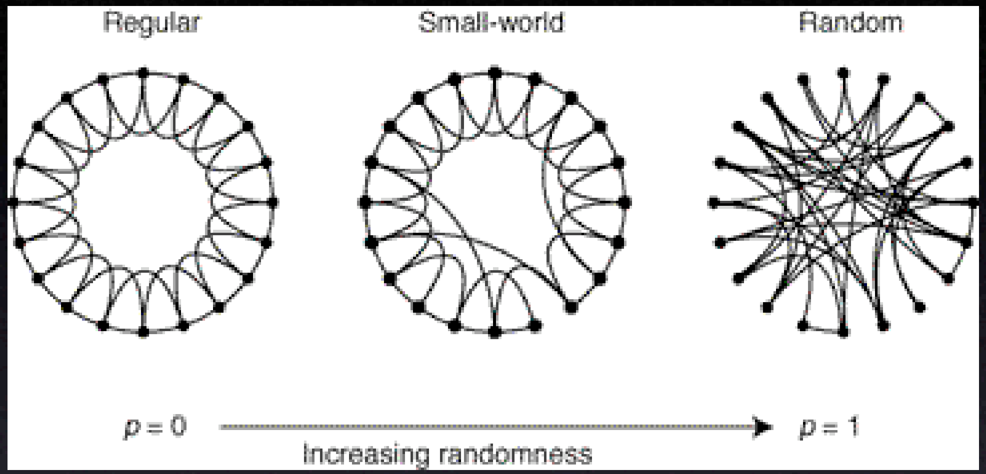
Small-world networks

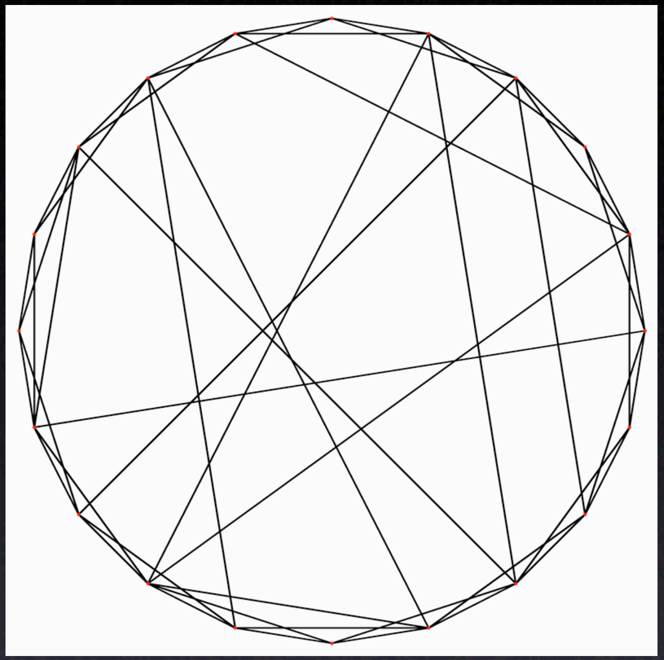
Myers/Sethna: Computational Methods for Nonlinear Systems

- motivated by phenomenon of "six degrees of separation"
- studied at Cornell by Duncan Watts and Steve Strogatz
  - Nature 393, 440-442 (1998)
  - simple model of networks with regular short-range bonds and random long-range bonds
  - examination of path lengths and clustering in model and in real-world networks
- Course exercise
  - calculation of shortest path lengths in randomly wired graphs
  - scaling of continuum limit
  - application to real network data
  - calculation of node and edge betweenness
  - provided with simple visualization tool



from Watts and Strogatz (1998)

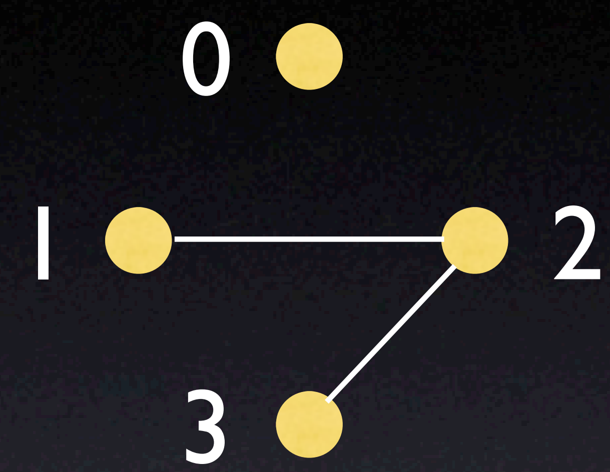
Computing for small-world networks: data structures



- network = graph (a set of nodes connected by edges)
- interested here in *undirected graphs* (edge is symmetric in two connecting nodes
- data structures for undirected graph?
  - some use adjacency matrix
    - a_{ij} = 1 if nodes i,j connected; 0 otherwise
  - we will use a neighbor dictionary
    - dictionary maps *key* to *value*
    - neighbor_dict[i] = [j_0, j_1, j_2, ...]
    - i.e., for a node i, we store a list [j_0, j_1, j_2, ...] of nodes that i is connected to
    - neighbor dictionary is directed (asymmetric), so we need to duplicate connections
      - if i points to j, then j must point to i
    - add a new entry to the dictionary when a new node is added, append to an existing entry when an existing node is connected to

Computing for small-world networks: object-oriented programming

- object-oriented programming
  - definition of new datatypes, along with associated behavior
  - encapsulate details of internal implementation (e.g., neighbor dictionary vs. adjacency matrix) without modifying external interface
- python `class` keyword allows definition of new class of objects




```
class UndirectedGraph:
   def __init__(self):
      self.neighbor_dict = {}

   def AddNode(self, node):
      # code to add a node

   def AddEdge(self, node1, node2):
      # code to add an edge connecting two nodes

   def HasNode(self, node):
      # return True if graph has specified node

   # etc.
```

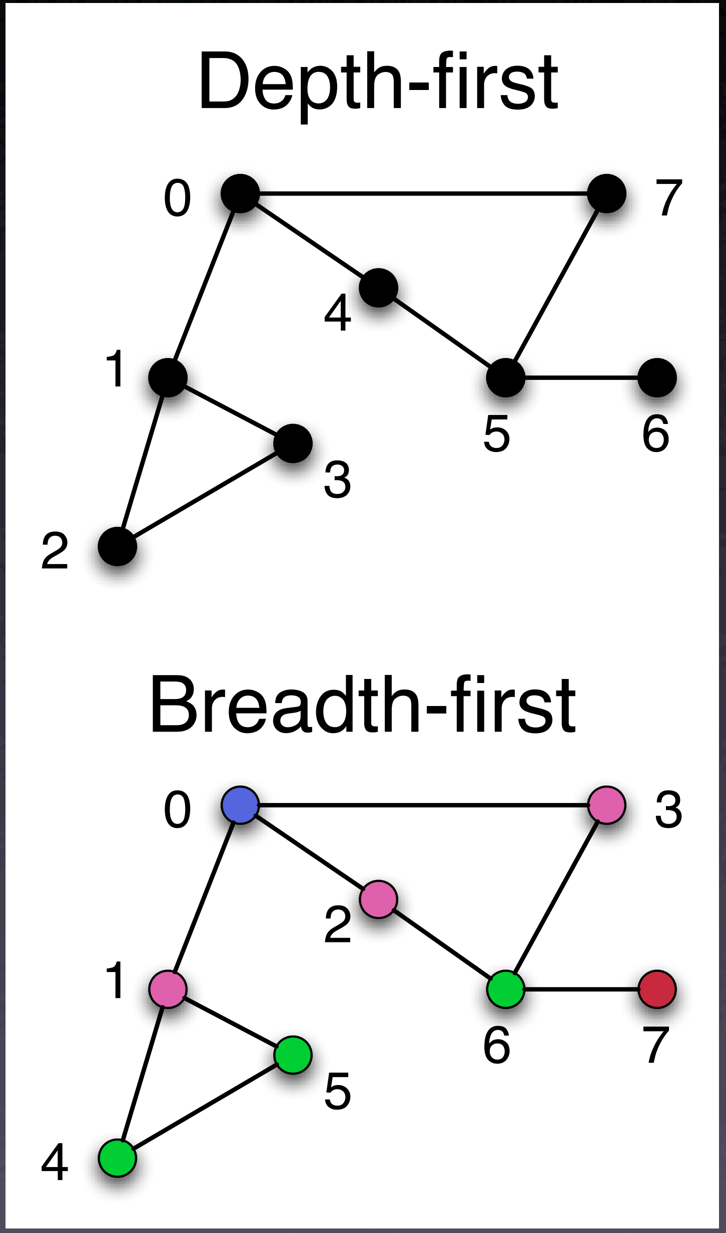
```
>>> g = UndirectedGraph()
>>> g.AddNode(0)
>>> g.AddEdge(1,2)
>>> g.AddEdge(2,3)
>>> g.HasNode(4)
False
```

```
"self" refers to the
particular object instance we
are working with, in this
case the graph "g"

g.AddNode(0) is shorthand for
UndirectedGraph.AddNode(g,0)
```

Computing for small-world networks: graph traversal algorithms

- graph traversal
  - iterating through a graph (i.e., over its nodes and edges) and calculating some quantity of interest
    - average shortest path: shortest path between all pairs of nodes in a graph
    - node and edge betweenness: what fraction of shortest paths each node or edge participates in
    - connected clusters (percolation)
  - traversing nodes and edges, marking nodes as visited so they get visited only once
    - most common: breadth-first and depth-first
- breadth-first search
  - involves iterating through the neighbors of all the nodes in the current shell, and adding to the next shell all subsequent neighbors which have not already been visited



Small-world networks: exercise and demo

- demo
  - create and display small-world networks for various parameters
  - compute average shortest path lengths
  - perform scaling collapse of path lengths (continuum limit analysis of Watts and Newman)
  - examine shortest path length and clustering coefficient
  - compute and display edge and node betweenness (using algorithm of Girvan and Newman)