Random Graph

In [1]:

```
%matplotlib inline
import networkx as nx
from netlab import monet
from netlab import monetgen as gen
from netlab import drawing  as dr
import matplotlib.pylab as plt
import math
import numpy as np
plt.rcParams['figure.figsize'] = 8,8
```

Erdos-Renyi Random Graph

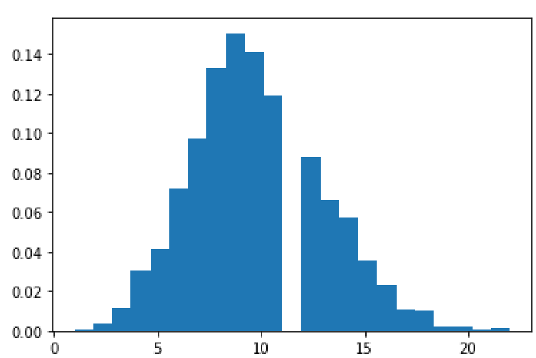
ER Graph $G(n, p)$

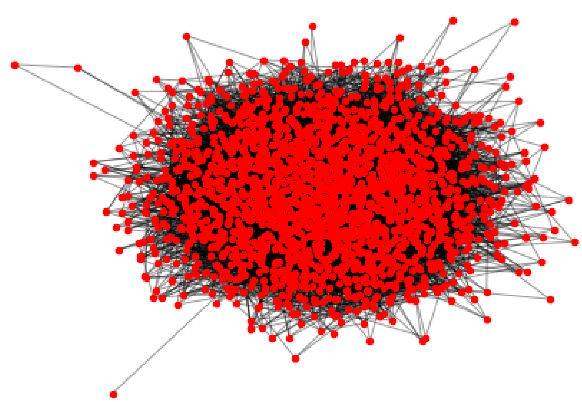
`gnp_random_graph(n, p, seed=None, directed=False)`

In [2]:

```
N = 2000
P = 0.005
G = nx.gnp_random_graph(N,P)
print ('|V| =', G.number_of_nodes())
print ('|E| =', G.number_of_edges())
print ('P =', P)
print ('Expected # of edges =', N*(N-1)/2 * P)
print ('ln(%s)/%s = %s' % (N,N,math.log(N)/N))
dr.degree_histogram(G)
dr.draw_graph(G)
plt.show()
```

```
|V| = 2000
|E| = 9774
P = 0.005
Expected # of edges = 9995.0
ln(2000)/2000 = 0.003800451229771041
```





In [3]:
```
dr.degree_histogram??
```

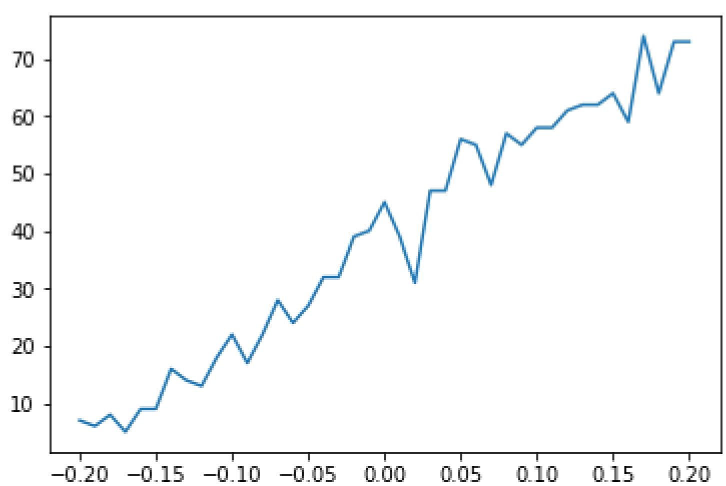
In [4]:
```
dr.draw_graph??
```

Connectivity of ER Random Graph

In [5]:
```
def test_er_graph(N, p, cnt):
    ccnt = 0
    for k in range(cnt):
        G = nx.gnp_random_graph(N,p)
        if nx.is_connected(G):
            ccnt += 1
    return ccnt


N = 200
p = math.log(N)/N
print ('p=log(N)/N=', p)

erange = np.arange(-0.2, 0.21, 0.01)
cnt = 100
num = []
for e in erange:
    num.append(test_er_graph(N, (1.0 + e) * p, cnt))

plt.plot(erange, num)

```

```
p=log(N)/N= 0.02649158683274018
```

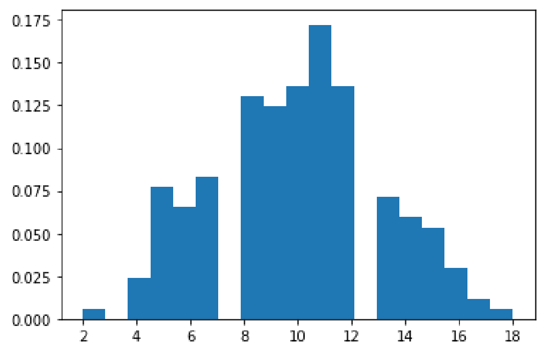
Out[5]:
```
[<matplotlib.lines.Line2D at 0x16f8d681470>]
```

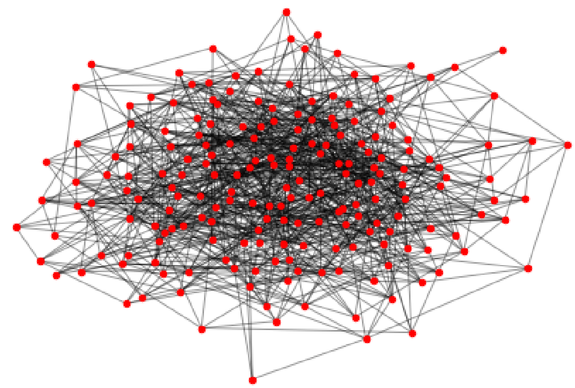


ER Graph $G(n, m)$

In [6]:
```python
# Erdos-Renyi random graph G(n,m)
G = nx.gnm_random_graph(200,1000)
print ('|E| =', G.number_of_edges())
dr.degree_histogram(G)
dr.draw_graph(G)
```

```
|E| = 1000
```





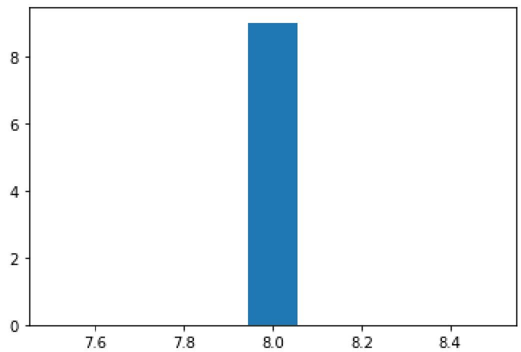
In [7]:
```python
help(nx.gnm_random_graph)
```

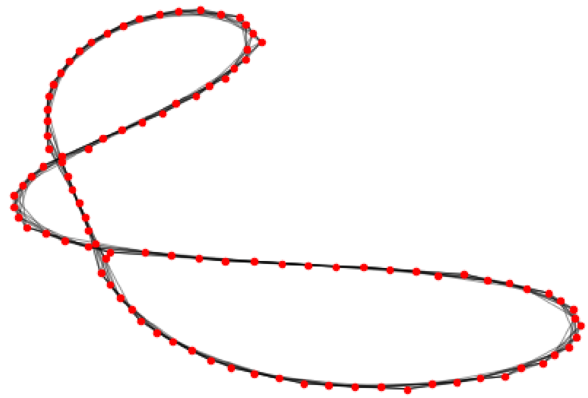
```
Help on function gnm_random_graph in module networkx.generators.random_graphs:

gnm_random_graph(n, m, seed=None, directed=False)
    Returns a $G_{n,m}$ random graph.

    In the $G_{n,m}$ model, a graph is chosen uniformly at random from the set
    of all graphs with $n$ nodes and $m$ edges.

    This algorithm should be faster than :func:`dense_gnm_random_graph` for
    sparse graphs.

    Parameters
    ----------
    n : int
        The number of nodes.
    m : int
        The number of edges.
    seed : int, optional
        Seed for random number generator (default=None).
    directed : bool, optional (default=False)
        If True return a directed graph

    See also
    --------
    dense_gnm_random_graph
```

Watts-Strogatz Random Graph

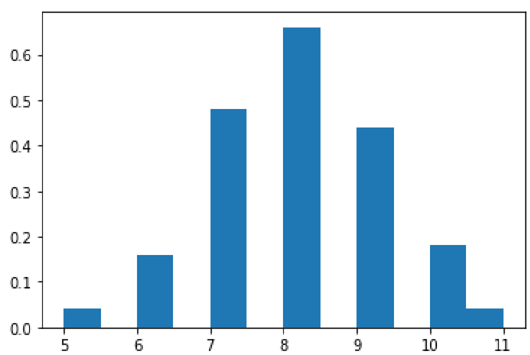
In [8]:
```python
for prob_rewire in [0.0, 0.2, 0.4, 0.6, 0.8]:
    G = nx.watts_strogatz_graph(100,8, prob_rewire)
    print ('probability of re-wire =', prob_rewire)
    dr.degree_histogram(G)
    dr.draw_graph(G)
    plt.show()
```

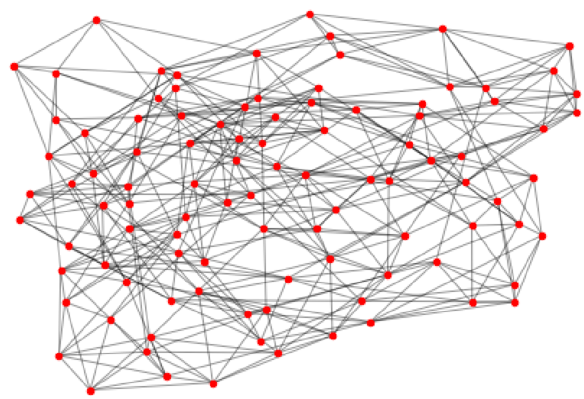
```
probability of re-wire = 0.0
```



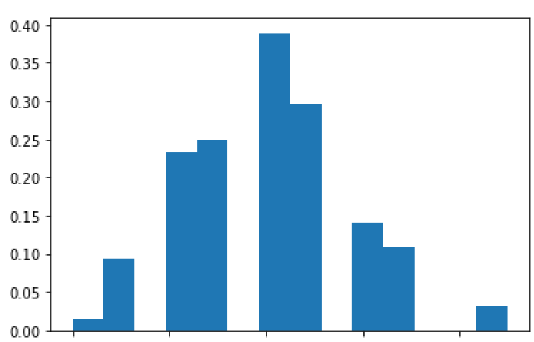


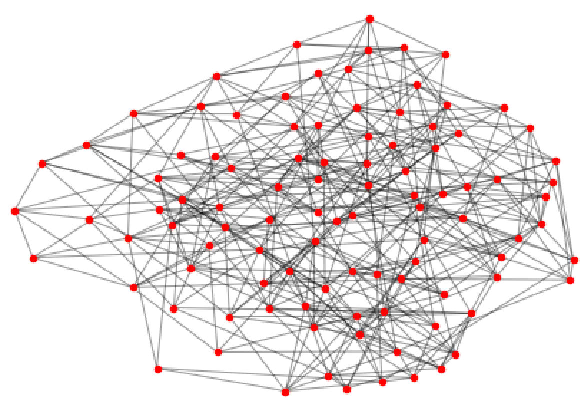
```
probability of re-wire = 0.2
```



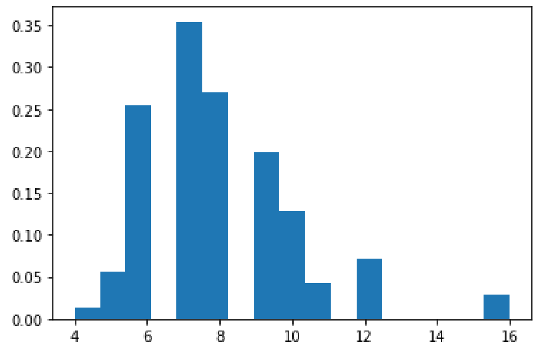


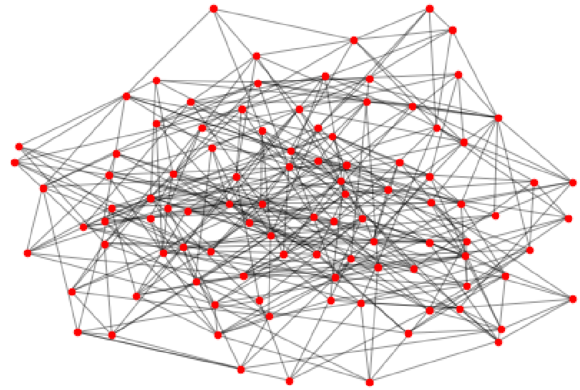
```
probability of re-wire = 0.4
```



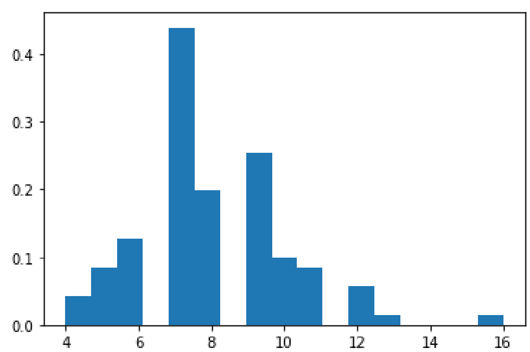


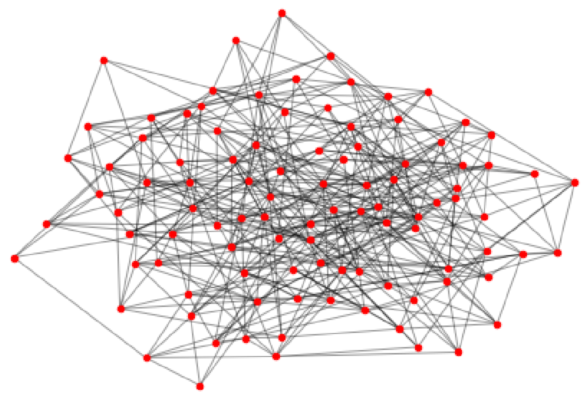
```
probability of re-wire = 0.6
```





```
probability of re-wire = 0.8
```





In [9]:
```
help(nx.watts_strogatz_graph)
```

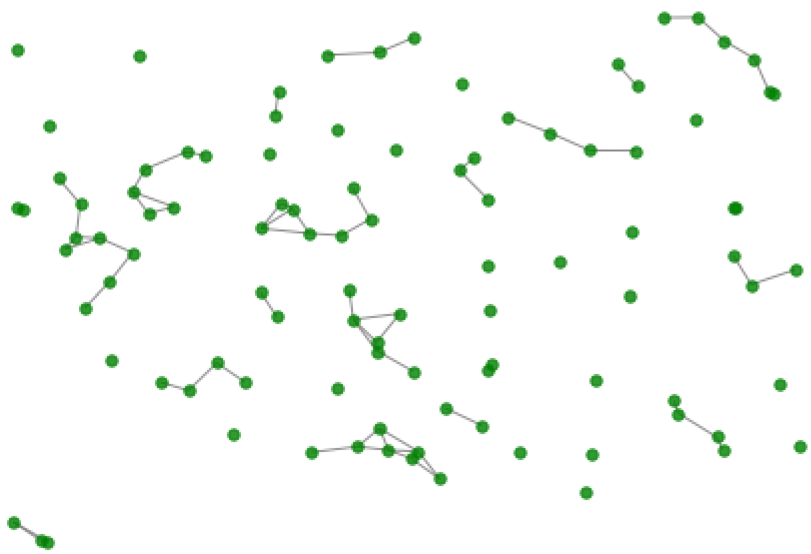
```
Help on function watts_strogatz_graph in module networkx.generators.random_grap
hs:

watts_strogatz_graph(n, k, p, seed=None)
    Return a Watts-Strogatz small-world graph.

    Parameters
    ----------
    n : int
        The number of nodes
    k : int
        Each node is joined with its `k` nearest neighbors in a ring
        topology.
    p : float
        The probability of rewiring each edge
    seed : int, optional
        Seed for random number generator (default=None)

    See Also
    --------
    newman_watts_strogatz_graph()
    connected_watts_strogatz_graph()

    Notes
    -----
    First create a ring over $n$ nodes [1]_.  Then each node in the ring is joi
ned
    to its $k$ nearest neighbors (or $k - 1$ neighbors if $k$ is odd).
    Then shortcuts are created by replacing some edges as follows: for each
    edge $(u, v)$ in the underlying "$n$-ring with $k$ nearest neighbors"
    with probability $p$ replace it with a new edge $(u, w)$ with uniformly
    random choice of existing node $w$.

    In contrast with :func:`newman_watts_strogatz_graph`, the random rewiring
    does not increase the number of edges. The rewired graph is not guaranteed
    to be connected as in :func:`connected_watts_strogatz_graph`.

    References
    ----------
    .. [1] Duncan J. Watts and Steven H. Strogatz,
       Collective dynamics of small-world networks,
       Nature, 393, pp. 440--442, 1998.
```

Unit Disk Graph Model

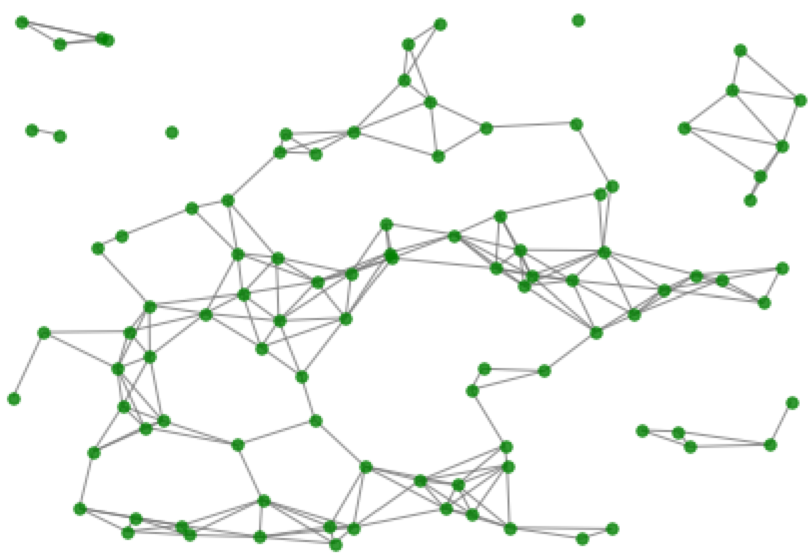
In [10]:

```python
for R in [100,200,300,400]:
    G = gen.unit_disk_graph(100,R=R,D=[1500,1500])
    print ('range =', R)
    # dr.degree_histogram(G)
    dr.draw_monet(G)
    plt.show()
```

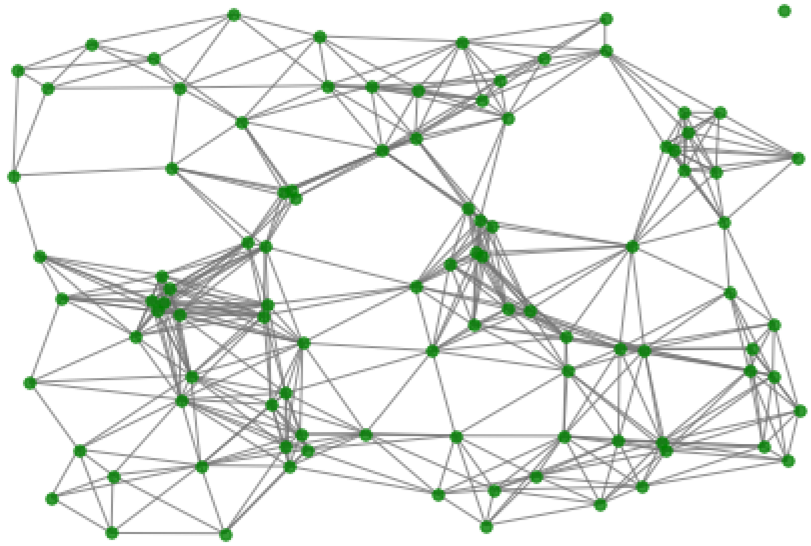
```
range = 100
```



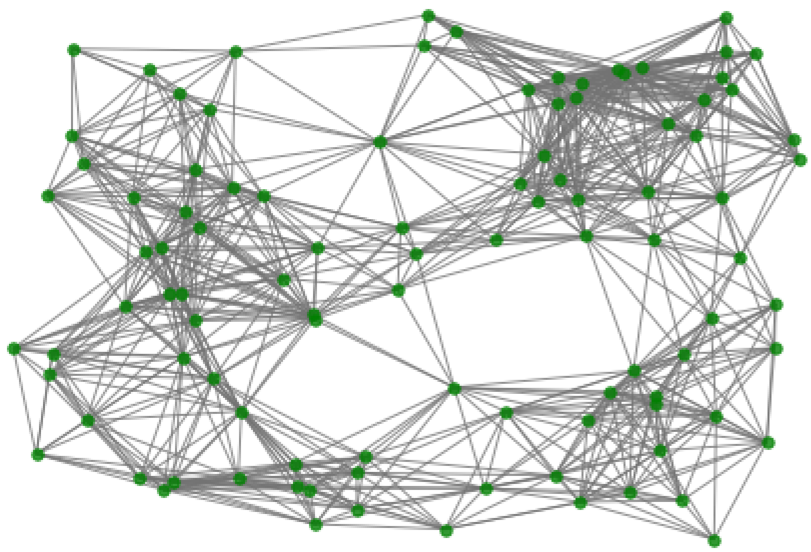
```
range = 200
```



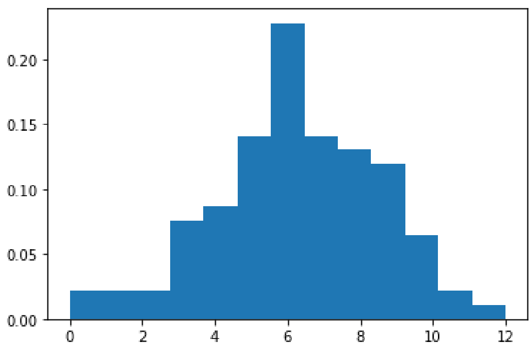
```
range = 300
```

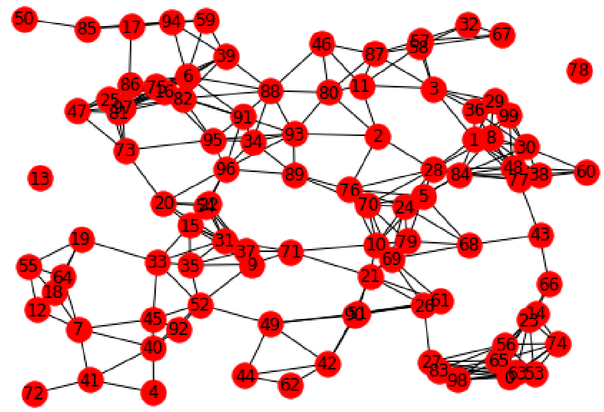


range = 400



In [11]:
```
G = gen.unit_disk_graph(100,R=150,D=[1000,1000])
px = nx.get_node_attributes(G, 'longitude').values()
py = nx.get_node_attributes(G, 'latitude').values()
p = np.column_stack([list(px), list(py)])
pos = p.astype(np.float32)          # change the list to NumPy array
layout = dict(zip(G, pos))  # combine the node list and the position list to
layout
dr.degree_histogram(G)
plt.axes([1,0,1,1])
plt.axis('off')
nx.draw_networkx(G, pos=layout)
plt.show()
```





In [12]:
```
print ('|V| =', G.number_of_nodes())
print ('|E| =', G.number_of_edges())
```

```
|V| = 100
|E| = 315
```

In [13]:
```
gen.unit_disk_graph??
```

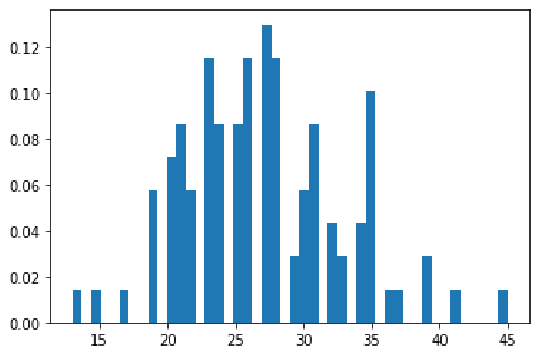
In [14]:
```
dr.draw_monet??
```

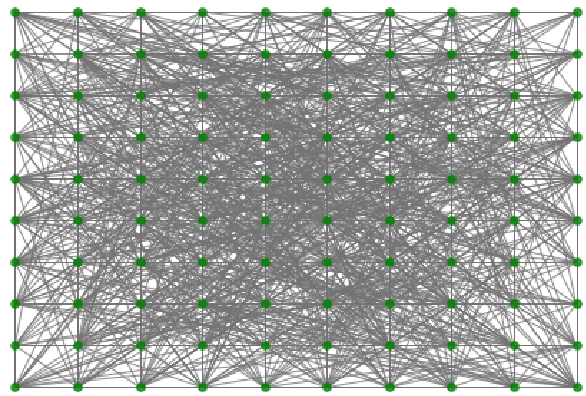
Kleinberg Graph

In [15]:

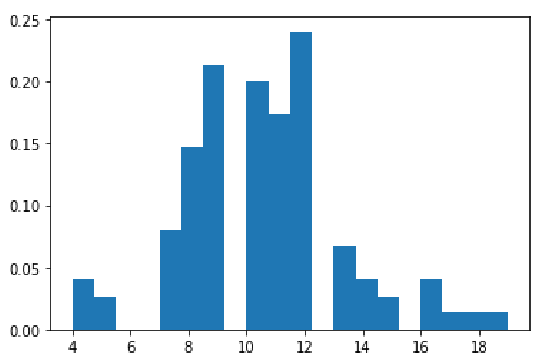
```python
from netlab import drawing as dr
from netlab import monet
from netlab import monetgen as gen

for pow in [1, 2, 3, 4]:
    G = gen.kleinberg_graph(power=pow)
    print ('power =', pow)
    dr.degree_histogram(G)
    dr.draw_monet(G)
    plt.show()
```

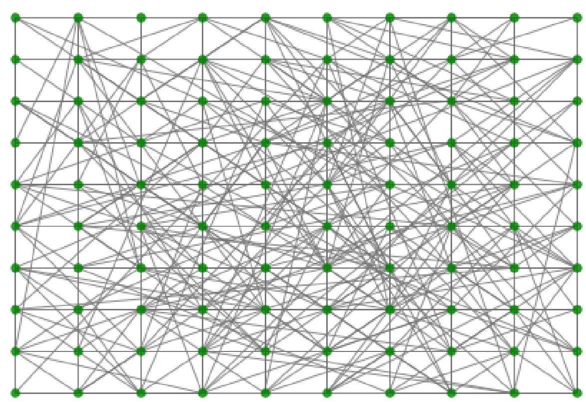
```
power = 1
```



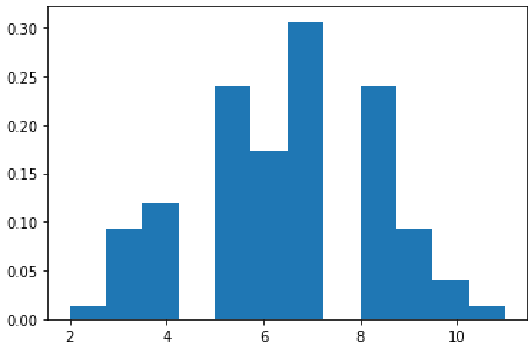


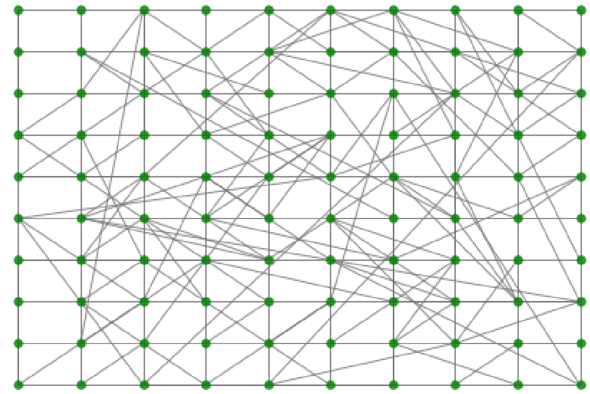
```
power = 2
```



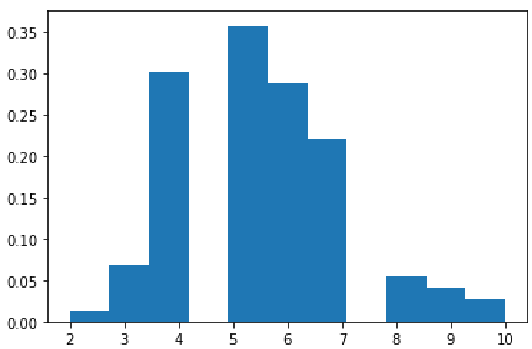


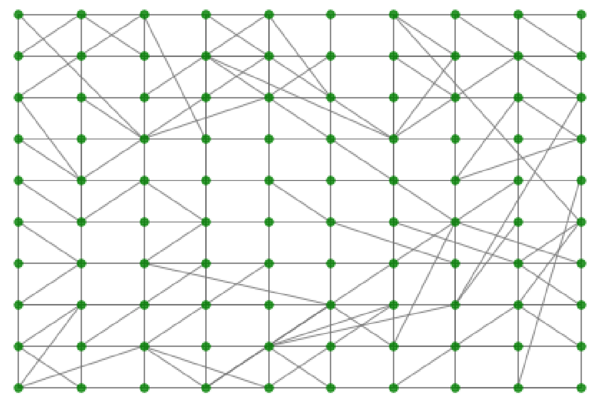
```
power = 3
```





```
power = 4
```



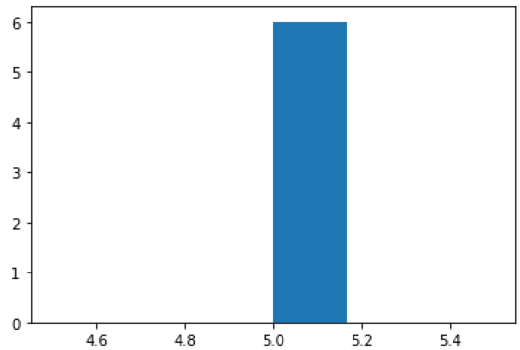


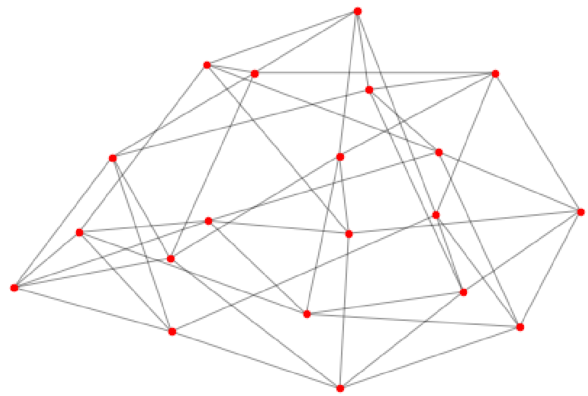
In [16]:
```
gen.kleinberg_graph??
```

Random k-Regular Graph

In [17]:
```
G = nx.random_regular_graph(5,20)
print ('|E| =', G.number_of_edges())
dr.degree_histogram(G)
dr.draw_graph(G)
```

```
|E| = 50
```





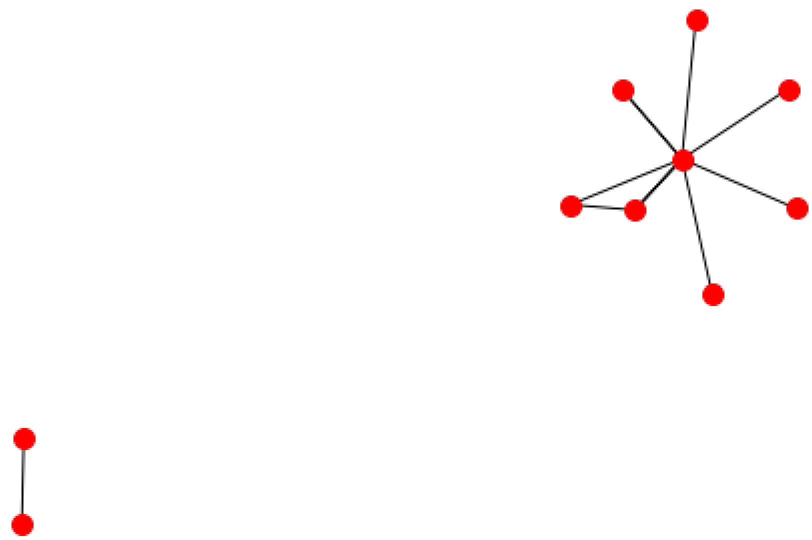
In [35]:
```
nx.random_regular_graph??
```

Configuration Model

In [19]:
```
def powerlaw_deg(n):
    a = list(map(int, nx.utils.powerlaw_sequence(n)))
    s = sum(a)
    while (s % 2 != 0):
        a = list(map(int, nx.utils.powerlaw_sequence(n)))
        s = sum(a)
    return a

pdeg = powerlaw_deg(10)
print(pdeg)
```

```
[39, 2, 32, 1, 1, 1, 2, 5, 4, 1]
```

In [20]:
```
aseq = [2, 1, 1, 2, 1, 1, 22, 4, 3, 1]
G = nx.configuration_model(aseq)   # return a multi-grfaph
nx.draw(G, node_size=100)
plt.show()
# try to draw G using networkx, which is not good for multi-graph
```



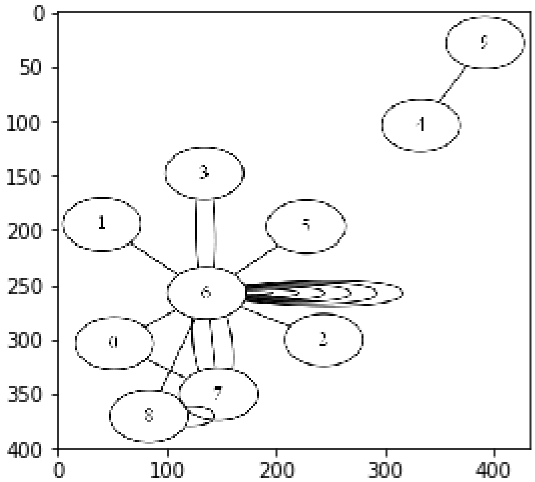
In [34]:
```
nx.configuration_model??
```

In [22]:
```
def deg_stats(D):
    N = len(D)
    s1 = 0.0
    s2 = 0.0
    for d in D:
        s1 += d
        s2 += d ** 2

    d1_bar = s1 / N
    d2_bar = s2 / N
    dvar = (d2_bar - d1_bar) / d1_bar
    mean_multi = dvar ** 2 / 2
    mean_loops = dvar / 2
    return s1, mean_multi, mean_loops

D = dict(nx.degree(G)).values()
print (D)
print (deg_stats(D))

```

```
dict_values([2, 1, 1, 2, 1, 1, 22, 4, 3, 1])
(38.0, 81.11357340720224, 6.36842105263158)
```

In [23]:

```
# only if pygraphviz is available
dr.draw_multi(G)
```



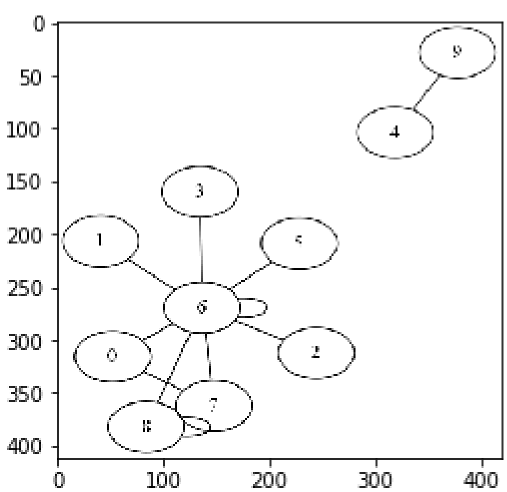
In [24]:

```
dr.draw_multi??
```

In [25]:

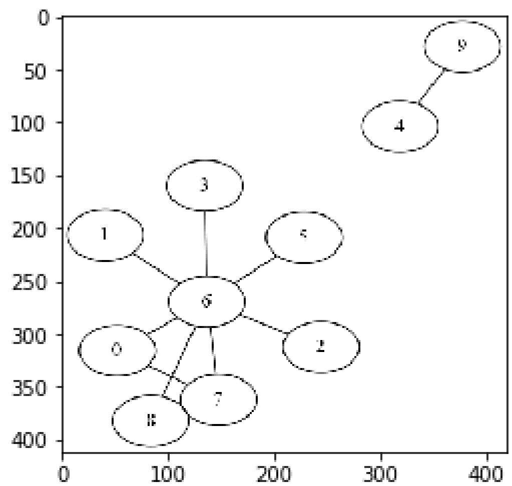
```
# only if pygraphviz is available
G = nx.Graph(G)     # to remove multi-edges
D1 = dict(nx.degree(G)).values()
#print (z)
print (D1)
print (deg_stats(D1))
dr.draw_multi(G)
```

```
dict_values([2, 1, 1, 1, 1, 1, 9, 2, 3, 1])
(22.0, 6.946280991735534, 1.8636363636363633)
```



In [26]:
```
G.remove_edges_from(G.selfloop_edges())    # remove self-edges
D2 = dict(nx.degree(G)).values()
print (D2)
print (deg_stats(D2))
dr.draw_multi(G)
```

```
dict_values([2, 1, 1, 1, 1, 1, 7, 2, 1, 1])
(18.0, 3.265432098765433, 1.2777777777777778)
```



To verify expected multi-edges and self-loops

In [27]:

```
z = powerlaw_deg(100)
G = nx.configuration_model(z)   # return a multi-grfaph
D0 = dict(nx.degree(G)).values()
G = nx.Graph(G)    # to remove multi-edges
D1 = dict(nx.degree(G)).values()
G.remove_edges_from(G.selfloop_edges())    # remove self-edges
D2 = dict(nx.degree(G)).values()
print (deg_stats(D0))
print (deg_stats(D1))
print (deg_stats(D2))
nx.draw(G, node_size=100)
plt.show()
```

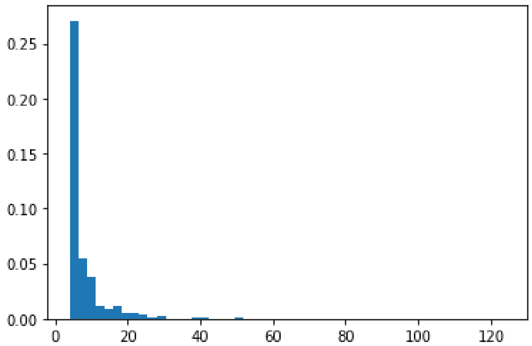
```
(774.0, 1986.4539691124332, 31.515503875968992)
(408.0, 115.83315311418689, 7.61029411764706)
(394.0, 102.74647633280942, 7.167512690355331)
```

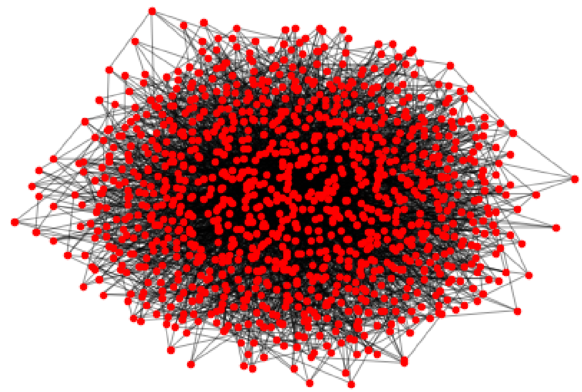
Barabasi Albert Preferential Attachment Model

In [28]:

```
G = nx.barabasi_albert_graph(1090, 4)
print ('avg degree =', float(2 * G.number_of_edges()) / G.number_of_nodes())
dr.degree_histogram(G)
dr.draw_graph(G)
plt.show()
```

```
avg degree = 7.970642201834862
```





In [29]:

```
dr.degree_histogram??
```

In [33]:

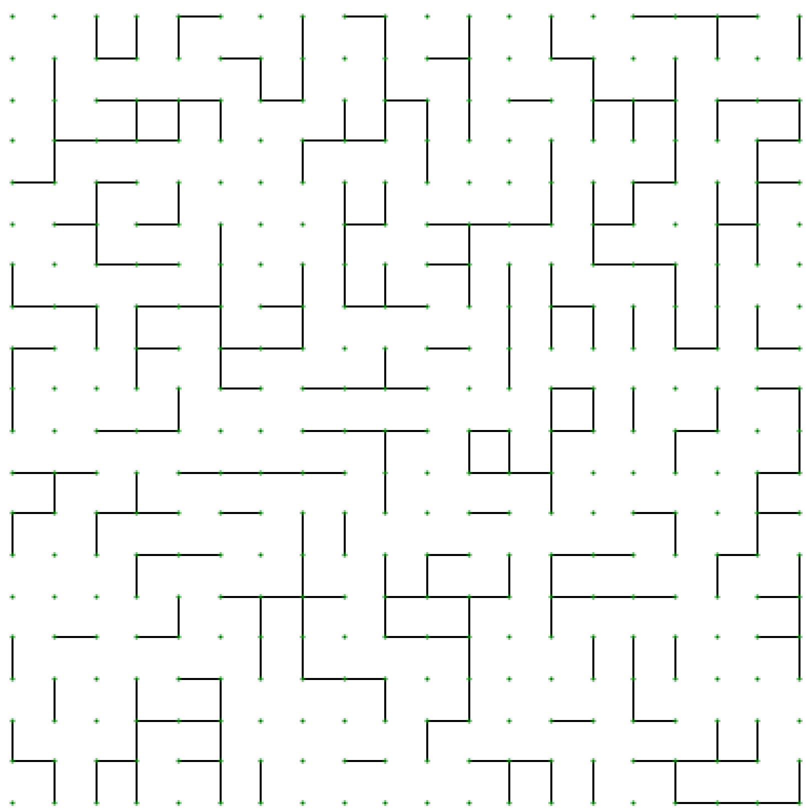
```
nx.barabasi_albert_graph??
```

Percolation Theory

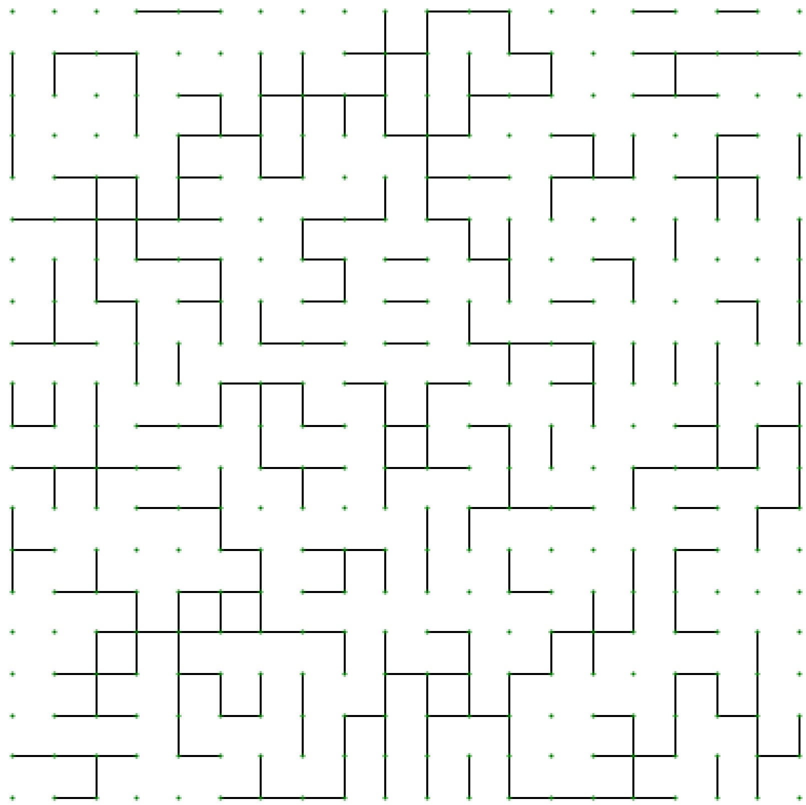
In [31]:

```
import numpy as np
# bond percolation
plt.rcParams['figure.figsize'] = 6,6
for p in np.arange(0.2, 0.4, 0.05):
    print ('probability =', p)
    G = gen.percolation_graph(rows=20, cols=20, prob=p)
    G.draw(node_size=1, edge_color='0', alpha=1); plt.show()
```

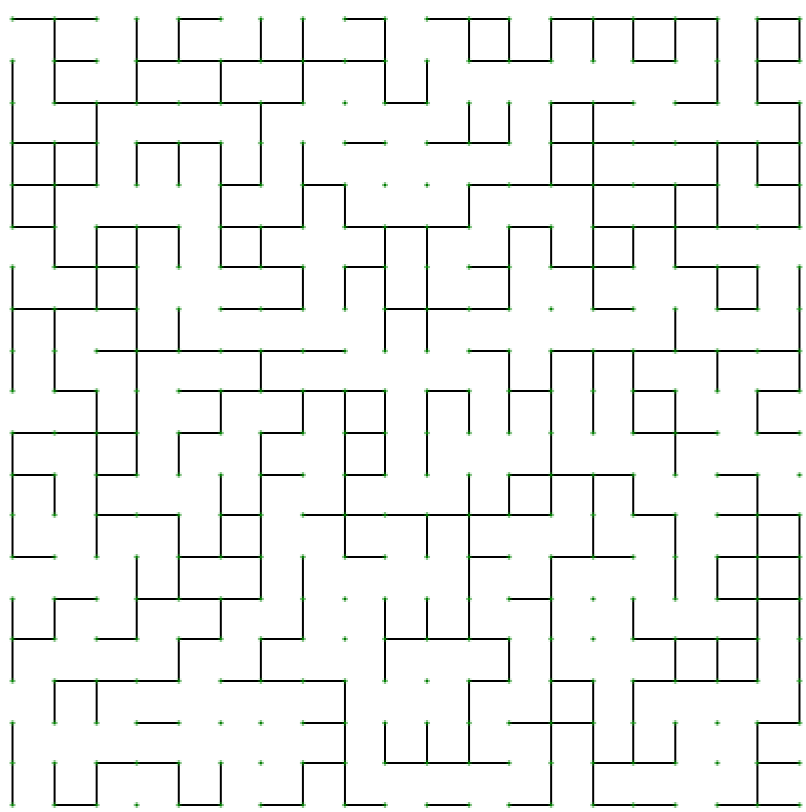
```
probability = 0.2
```



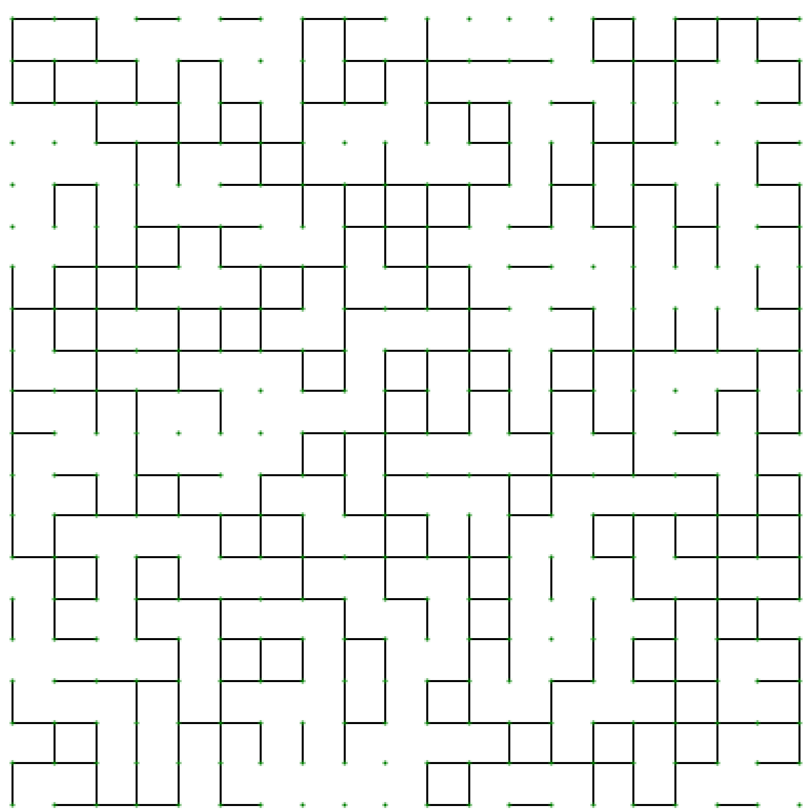
```
probability = 0.25
```



probability = 0.3



probability = 0.35



In [32]:
```
gen.percolation_graph??
```