



Problem 1

	$p_n = p \in (0, 1)$	$p_n = \alpha n^{-\beta}$	$p_n = \alpha n^{-1}$
\bar{k}	dense	sparse	ultrasparse
\bar{d}	ultrasmall	small	small
\bar{c}	strong	no	no

Table 1: Sparsity, described by the behavior of average degree \bar{k} as $n \rightarrow \infty$, small-worldness, described by the behavior of average distance between nodes \bar{d} as $n \rightarrow \infty$, and clustering, described by the behavior of average clustering, \bar{c} as $n \rightarrow \infty$ for three constructions of Erdős-Rényi graphs based on different edge probabilities.

Table 1 describes the sparsity, small-worldness, and clustering of three $G(n, p)$ graphs defined by some $p = p_n$ which is graph-size-dependent.

Sparsity

To begin, we consider the case of $p_n = p \in (0, 1)$. We know that for $G(n, p_n)$, the $P[(i, j) \in E]$ is independent and depends only on p for all nodes i, j and some set of edges E in G . Thus,

$$\bar{k} = p_n(n - 1) \quad (1)$$

Substitution into Eq. 1 the first connection probability, we get $\bar{k} = p(n - 1)$ and thus, $\bar{k} \sim n$ and is therefor *dense* in our definitions.

Doing the same with the second connection probability, we see that $\bar{k} = \alpha n^{-\beta}(n - 1) \approx \alpha n^{1-\beta}$. As $\beta \in (0, 1)$ by definition, this graph scales as $\bar{k} \ll n$ to $\bar{k} < n$. Thus, by our definition, this is a *sparse* graph.

Finally, substituting our last value of p_n into Eq. 1, we get $\bar{k} = \alpha n^{-1}(n - 1) = \alpha(1 - 1/n)$. As the last term gets dropped in the thermodynamic limit, $\bar{k} \sim \alpha$ and is thus *ultrasparse* in our definitions.

Small-worldness

For constant p , as in the first case of $p_n = p$, we know that the diameter $\bar{d} \sim 1$. As the definition of small world is

$$G \text{ is small world} \iff \bar{d} \ll n^\epsilon \forall \epsilon. \quad (2)$$

$$G \text{ is ultrasmall} \iff \bar{d} \ll \ln^\epsilon(n) \forall \epsilon. \quad (3)$$

As $1 \ll \ln^\epsilon(n)$ as $n \rightarrow \infty$, we classify $G(n, p_n = p \in (0, 1))$ to be ultrasmall world.

For the second case, we consider that $\bar{k} \sim n^\epsilon, \epsilon > 0$ as per the derivation above. Thus, we know that the average distance is

$$\bar{d} \sim \frac{\log(n)}{\log(\bar{k})} \quad (4)$$

. Substituting our value for \bar{k} , we get $\bar{d} \sim \frac{\log(n)}{\log(\alpha n^{1-\beta})} = \frac{\log(n)}{\log(\alpha) + \log(n) - \beta \log(n)}$ the right hand side is therefor $c < \bar{d} < c \log n$ where c is some constant factor. Thus, this does not satisfy Eq. 3 but suffices to satisfy Eq. 2 and is thus a small world.

Finally, we examine the final case in which we know $\bar{k} \propto n^0 > 1$. Thus, from Eq. 4, we know that $\bar{d}/\log(n) \rightarrow 1/\alpha$ and this is thus a small world as well.

Clustering

We consider any clustering that is non-zero in the limit to be "strong" clustering. For a single node i , the clustering coefficient c_i is defined as

$$c_i = t_i/w_i \quad (5)$$

where w_i is the number of pairs of neighbors of a node, and t_i is the number of connected pairs of neighbors. The average clustering \bar{c} which we are looking for is a weighted linear combination of these c_i . However, if we use consider our Erdős-Rényi model to generate graphs, we can see that for any node i , Eq. 5 can be expressed as

$$c_i = p_n^3/p_n^2 \quad (6)$$

From Eq. 6, it is clear that due to the n -dependence for our latter two cases, this c_i term completely vanishes to 0. Only in the case of constant $p_n = p$ does a term remain in the thermodynamic limit and thus that is the only graph realization in which we would witness strong clustering.

Problem 2

All code for this section is attached but can also be found on my Github¹.

Log-Binning Distribution

See function `distributionBin()`

Log-Binning Function

See function `functionBin()`

Graph Generation

See function `simpleSF()`

Measuring Graph Properties

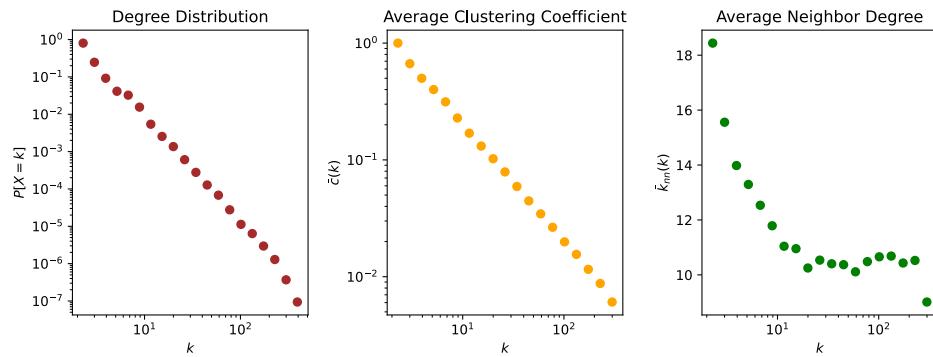


Figure 1: For the ensemble of $N = 10$ graphs of size $n = 10^4$ generated using the `simpleSF()` function: (left) Degree Distribution $P[X = k]$ or $P(k)$ plotted on a log-log scale as a logarithmically binned distribution, (center) Average Clustering Coefficient $\bar{c}(k)$ plotted against degree on a log-log scale, binned logarithmically, and (right) Average Neighbor Degree $\bar{k}_{nn}(k)$ plotted against degree as a logarithmically binned function on a semi-log plot.

¹<https://github.com/sagarkumar16/complex-networks-2/blob/main/HW1/HW1.ipynb>

Figure 1 shows the measurements of $P(k)$, $\bar{c}(k)$, and $\bar{k}_{nn}(k)$ for the ensemble of graphs created by the simple Scale-Free graph generation algorithm. For each measurement, all values from each of the graphs was concatenated into a single sequence of values or single sequence of input-output pairs. Thus, the results are not the measurements for any single of the graphs output nor is it the average of all of them, but rather, a concatenation of all of them.

The results are consistent with the literature on scale-free graphs. When logarithmically binned, the degree distribution shows a clear power law, as does the average clustering. Average neighbor degree seems to show some signs of "disassortivity" at first, but quickly becomes quite neutral, suggesting that this apparent negative degree assortativity is likely to be a finite-size effect.

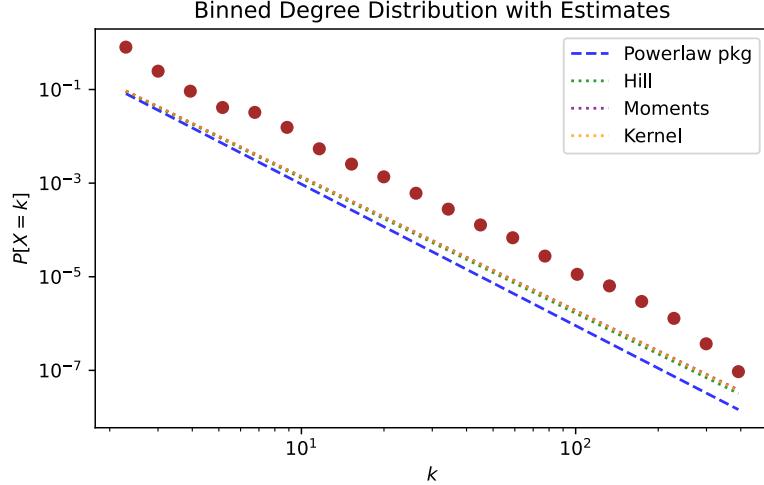


Figure 2: The degree distribution as shown in Fig. 1 plotted alongside estimates from the `powerlaw` package, as well as the Hill, Moments, and Kernel estimates from *Tail Index Estimation for Degree Sequences of Complex Networks*²

Estimator	γ
powerlaw	3.022
Hill	2.890
Moments	2.860
Kernel	2.865

Table 2: Summary of rounded estimations for the exponent, γ , on a power-law fit of the degree distribution

Figure 2 shows estimates from the `powerlaw` package, as well as Hill, Moments, and Kernel estimates of the same degree distribution plotted in Figure 1. Table 2 provides the γ estimates, rounded after 3 decimal points. Clearly, there is some spread in these estimates. Interestingly, the `powerlaw` package, unlike the other estimators, places this graph in the regime of $\gamma > 3$ and visually, seems to diverge further from the true value than the other estimates which have been proven to be optimal and consistent. Nevertheless, all estimators are in consensus that this model does indeed have a diverging second moment in the infinite limit and is thus scale-free.

Appendix: Code

```

1 #%% raw
2 !pip install powerlaw
3 #%%
4 import numpy as np
5 import math
6 import matplotlib.pyplot as plt
7 from collections import defaultdict
8 from statistics import mean
9 from scipy.sparse import dok_matrix, csr_matrix, triu, tril
10 from random import randint
11 import powerlaw as pl
12 import tail_estimation as te
13 from tqdm.notebook import tqdm
14 from utils import count_faster
15 #%% md
16 ## Functions
17 #%%
18 ##### Problem 2.1
19 def distributionBin(x: 'input data',
20                     B: 'number of bins'
21                     ):
22     """
23     :param x: (list) list of real-valued integers interpreted as the input data sampled from a
24     presumed heavy-tail distribution
25     :param B: (positive int) number of log-scaled bins to create a histogram from x
26
27     distributionBin: Tuple, int -> Tuple, Tuple[Float]
28
29     distributionBin takes a list of input data :x: assumed to be sampled from some heavy-tailed
30     distribution and bins in :B: log
31     sized bins in such a way that the output function *integrates* to 1. The output is a list of bin-
32     midpoints X and the binned
33     estimated distribution values Y.
34     """
35
36
37     xmin = min(x)
38     xmax = max(x)
39
40     # creating the B+1 bin edges
41     bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)
42
43     # using numpy's histogram function get distributions
44     density, _ = np.histogram(x, bins=bin_edges, density=True)
45
46     # obtaining bin midpoints for cleaner absciss
47     log_be = np.log10(bin_edges)
48     xout = 10**((log_be[1:] + log_be[:-1])/2)

```

```

46
47     return xout, density
48 #%%
49 ##### Problem 2.2
#####
50 def functionBin(x: 'preimage',
51                 y: 'function output for some function y | y_i=y(x)',
52                 B: 'number of bins'
53                 ):
54
55     """
56     :param x: (list) list of real-valued numbers interpreted as the preimage of the function y(x)
57     :param y: (list) list of real-valued numbers interpreted as the values of function y applied to
58     each input value in x
59     :param B: (positive int) number of log-scaled bins to create a histogram
60
61     functionBin: Tuple, Tuple, int -> Tuple, Tuple
62
63     functionBin takes an ordered list of sampled input values :x: and their corresponding outputs
64     under some function :y: (
65         i.e.  $y(x)$ ) and bins the input values into :B: log sized bins, averaging the output values within
66     each bin. The output is
67         the bin-midpoints X and the binned outputs.
68
69     """
70
71     xmin = min(x)
72     xmax = max(x)
73
74     # creating the B+1 bin edges
75     bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)
76
77     # obtaining bin midpoints for cleaner absciss
78     log_be = np.log10(bin_edges)
79     bm = 10**((log_be[1:] + log_be[:-1])/2)
80
81     # creating (input, output) pairs and sorting by input value
82     fpairs = list(zip(x,y))
83     fpairs.sort(key=lambda x: x[0])
84
85     # creating (label, boundary) pairs using midpoint and right boundary for each bin
86     mid_redge_pairs = list(zip(bm, bin_edges[1:]))
87
88     # dictionary of values where key corresponds to the bin and values are those which fit in the bin
89     bin_out_list = defaultdict(list)
90
91     idx = 0
92     for mid, redge in tqdm(mid_redge_pairs, desc='Binning Function'):

```

```

90     while (fpairs[idx][0] < redge) & (idx < len(fpairs)-1):
91         bin_out_list[mid].append(fpairs[idx])
92         idx += 1
93
94     # adding the last value
95     bin_out_list[list(bin_out_list.keys())[-1]].append(fpairs[-1])
96
97     xout = list(bin_out_list.keys())
98
99     # y value is the average of each bin
100    yout = [mean([i[1] for i in b]) for b in bin_out_list.values()]
101
102    return xout, yout
103 #%%
104 ##### Problem 2.3
#####
105 def simpleSF(n: 'graph size' = 10**4,
106             s: 'number of samples' = 10,
107             sparse: 'choice of numpy vs. scipy sparse matrix' = False
108             ):
109
110     """
111         :param n: (positive int) total number of nodes in the desired graph after completing its
112             percolation
113         :param s: (positive int) total number of graphs created
114         :param sparse: (boolean) False maps output to dense numpy ndarrays, True maps them to
115             scipy sparse csr_matrix
116
117         simpleSF takes a value for the size of the graph :n: and the number of samples :s: and
118             percolates :s: scale-free graphs of
119             size :n: by randomly selecting an edge E := (i,j) and attaching new incoming nodes to both i
120             and j. The output is a list of
121             :s: (:n: x :n:) matrices which act as the adjacency matrix of each sample graph. Choosing scipy
122             sparse matrix allows for
123             larger graph sizes.
124     """
125
126     outlist = list()
127     pbar = tqdm(range(s), desc=f"Generating Samples")
128
129     for i in pbar:
130
131         # edgelist
132         edgelist = {(0,1)}
133
134         # selecting a random edge and making the incident nodes the fruits of an incoming cherry
135         # to create a triangle

```

```

132     # process repeats until t == n
133     t = 2
134     while t < n:
135         m = randint(0, len(edgelist)-1)
136         i,j = list(edgelist)[m]
137         edgelist = edgelist | {(i, t), (j, t)}
138         t += 1
139
140     # mapping corresponding matrix values for all pairs in edgelist from 0 -> 1
141     if sparse:
142         A = dok_matrix((n, n), dtype=int)
143         for e in edgelist:
144             A[e] = 1
145
146         # currently only have upper triangle, so making the matrix symmetric about the main
147         # diagonal
148         # triu and tril automatically change it from dok_matrix to csr_matrix
149         A = triu(A) + tril(A.T)
150
151     else:
152         A = np.zeros((n,n))
153         for e in edgelist:
154             A[e] = 1
155
156         # currently only have upper triangle, so making the matrix symmetric about the main
157         # diagonal
158         A = np.triu(A) + np.tril(A.T)
159
160     return outlist
161
162
163 %% md
164 ## Tests
165 %%
166 import networkx as nx
167 import itertools
168 %%
169 ##### Problem 2.4
170 #####
171 # creating graph ensemble G
172 G = simpleSF(sparse=True)
173
174 # setting number of bins
175 num_bins = 20
176
177 # preparing the plots

```

```

178
179 fig, axs = plt.subplots(1, 3)
180
181 ##### plotting degree distribution #####
182 degrees = list()
183 for g in G:
184     # compressing ensemble by concatenating all degrees of all graphs into one degree sequence
185     degrees.extend(g.sum(axis=0).tolist()[0])
186
187
188 # binning into :num_bins: bins using function from 2.1
189 dist_x, dist_y = distributionBin(degrees, num_bins)
190
191 # plotting
192 axs[0].loglog(dist_x, dist_y, 'o', color='brown')
193 axs[0].set_title('Degree Distribution')
194 axs[0].set_ylabel(r'$P[X=k]$')
195 axs[0].set_xlabel(r'$k$')
196
197 ##### plotting avg. clustering #####
198
199 # using networkx to get average clustering and average neighbor degree
200 indexed_node_degrees = list()
201 indexed_clustering = list()
202 indexed_average_degree = list()
203
204 for g in G:
205     # this needs to be np.from_numpy_matrix if not sparse
206     graph = nx.from_scipy_sparse_array(g)
207     for node in graph.nodes:
208         indexed_node_degrees.append(graph.degree(node))
209         indexed_clustering.append(nx.clustering(graph, nodes=[node])[node])
210         indexed_average_degree.append(nx.average_neighbor_degree(graph, nodes=[node])[node])
211
212 # Using the function from 2.2 to plot these values
213 c_x, c_y = functionBin(indexed_node_degrees, indexed_clustering, num_bins)
214 kk_x, kk_y = functionBin(indexed_node_degrees, indexed_average_degree, num_bins)
215
216 # plotting clustering on a loglog scale
217 axs[1].loglog(c_x, c_y, 'o', color='orange')
218 axs[1].set_title('Average Clustering Coefficient')
219 axs[1].set_xlabel(r'$k$')
220 axs[1].set_ylabel(r'$\bar{c}(k)$')
221
222 # plotting average neighbor degree on a semi-log scale
223 axs[2].plot(kk_x, kk_y, 'o', color='green')
224 axs[2].set_xscale('log')
225 axs[2].set_title('Average Neighbor Degree')

```

```

226 axs[2].set_xlabel(r'$k$')
227 axs[2].set_ylabel(r'$P(k)$')
228
229 fig.set_size_inches(10, 4, forward=True)
230 fig.tight_layout()
231
232 fig.savefig('HW1_fig1_pk_ck_knnk.svg')
233 #%%
234 ## Powerlaw package fit
235
236 pl_fit = pl.Fit(degrees)
237 gamma_pl = pl_fit.power_law.alpha
238 print(f"Fit after x = {pl_fit.power_law.xmin}")
239 print(f"Estimated gamma: {gamma_pl}")
240 #%%
241 ## Hill, Moments, and Kernel estimators
242
243 # exporting the data to a dat file
244 import struct
245
246 # sending degree counts to a .dat file that can be used with the package
247 deg_data = count_faster(degrees)
248 deg_data.sort(key=lambda x: x[0])
249 deg_data = np.array(deg_data, dtype=np.int64)
250 np.savetxt('synthetic_scale_free_degree.dat', deg_data, fmt='%i')
251 #%%
252 !python tail_estimation.py synthetic_scale_free_degree.dat HW1_Estimator_plots.pdf
253 #%%
254
255 hill_gamma = 2.889644431436195
256 moments_gamma = 2.8602879353057453
257 kernel_gamma = 2.8648694044336103
258
259 # Plotting Distribution with all of its estimates
260 plt.loglog(dist_x, dist_y, 'o', color='brown')
261 plt.plot(dist_x, [k**(-1*pl_fit.power_law.alpha) for k in dist_x], linestyle='dashed', color='blue',
262           label='Powerlaw pkg',
263           alpha=0.8)
264 plt.plot(dist_x, [k**(-1*hill_gamma) for k in dist_x], linestyle='dotted', color='green', label='Hill',
265           alpha=0.8)
266 plt.plot(dist_x, [k**(-1*moments_gamma) for k in dist_x], linestyle='dotted', color='purple',
267           label='Moments', alpha=0.8)
268 plt.plot(dist_x, [k**(-1*kernel_gamma) for k in dist_x], linestyle='dotted', color='orange', label=
269           'Kernel', alpha=0.8)
270 plt.legend()
271 plt.title('Binned Degree Distribution with Estimates')
272 plt.ylabel(r'$P[X=k]$')
273 plt.xlabel(r'$k$')
274 plt.tight_layout()

```

```
271  
272 plt.savefig('HW1_fig2_degree_dist_estimates.svg')  
273 %%  
274 %%  
275 %%  
276
```