Engineering Tripos Part IIA
SF5 Networks, friendship, and disease: Interim Rerpot 1

Ruitong Sun [rs2177]

1 Graph definition

A `network`[1] class is created with attributes number of nodes n, adjacency list, adjacency matrix. Methods are create to compute key properties of a network object, including neighbors of node i, edge list, edge counts, component that contains node i. Refer to Appendix A for codes.

2 Random Graph Sampling

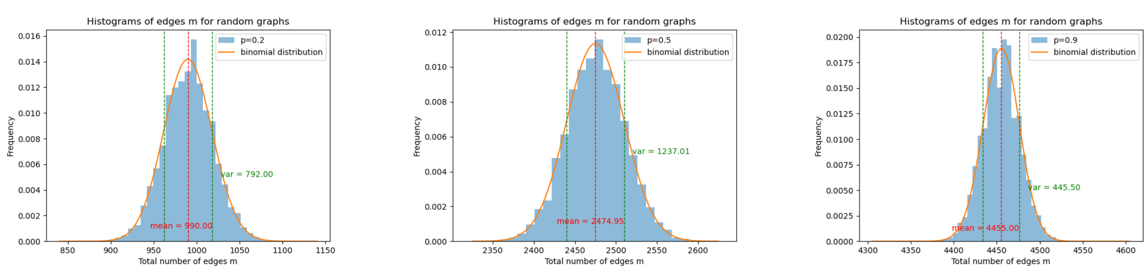
2.1 Naive Sampling

A naive approach to generate a random graph $G(n, p)$ is to iterate over all $\frac{1}{2}n)n-1)possiblepairsofnodes(i,j)andsamplewhe$

It can be validated empirically that number of edges m of any random graph $G(n, p)$ follows a binomial distribution:

$$P(m) = \binom{\binom{n}{2}}{m} p^m (1-p)^{\binom{n}{2}-m}$$

By plotting histograms of m for 10000 sampling trials, number of nodes n=100, and several values of p= 0.2, 0.5, 0.9, as shown in Figure 1.



(a) p=0.2 (b) p=0.5 (c) p=0.9

Figure 1: Histogram of number of edges m for random graph $G(100, p)$

It is observed that the histograms match the theoretical binomial distribution. The expected values and variances of m observed are also consistent with its binomial expressions:

$$\mathrm{E}[m] = \binom{n}{2} p; \qquad \mathrm{Var}[m] = \binom{n}{s} p(1-p)$$

2.2 Degree Distribution

Similar to total number of edges m, number of edges k connected to a random node follows a binomial distribution $k \sim \text{Binomial}(n-1, p)$, with mean $(n-1)p$, variance $(n-1)p(1-p)$.

[1] In this paper, "network" and "graph" are used interchangeably.

Consider $G(n, \frac{\lambda}{n-1})$ for some constant λ. Both the expected degree and its variance would be constant at λ, independent of n. This seems to be consistent with characteristics of Poisson distribution. It can be shown that the Binomial degree distribution does tend to a Poisson distribution in the limit $n \to \infty$, using generating functions[2]:

$$G(z) = \sum_{k=0}^{n-1} z^k \mathrm{P}(k) = \sum_{k=0}^{n-1} \binom{n-1}{k} z^k (\frac{\lambda}{n-1})^k (1 - \frac{\lambda}{n-1})^{n-1-k} \tag{1}$$

$$= [\frac{z\lambda}{n-1} + (1 - \frac{\lambda}{n-1}]^n = [1 - \frac{\lambda(z-1)}{n-1}]^n \tag{2}$$

$$\lim_{n\to\infty} G(z) = \lim_{n\to\infty} [1 - \frac{\lambda(z-1)}{n-1}]^n = e^{\lambda(z-1)} \tag{3}$$

The resulting generating function coincides with that of a Poisson distribution $\mathrm{Po}(\lambda)$.

2.3 2-stage Sampling

An more efficient 2-stage sampling algorithm is proposed over the naive one in Appendix B.2:

1. Sample a value for number of edges m based on its Binomial distribution;
2. Populate the m edges randomly into the network.

2.4 Time complexity

The time complexity of the 2 algorithms are measured sampling $G(n, \frac{\lambda}{n-1})$, with $n = 64, 128, ..., 1204$, each for 100 trials. By plotting in log-log scale in Figure 1, the slope shows that naive sampling has a quadratic complexity, which scales with $\mathcal{O}(n^2)$, while 2-stage sampling is linear with $\mathcal{O}(n)$.

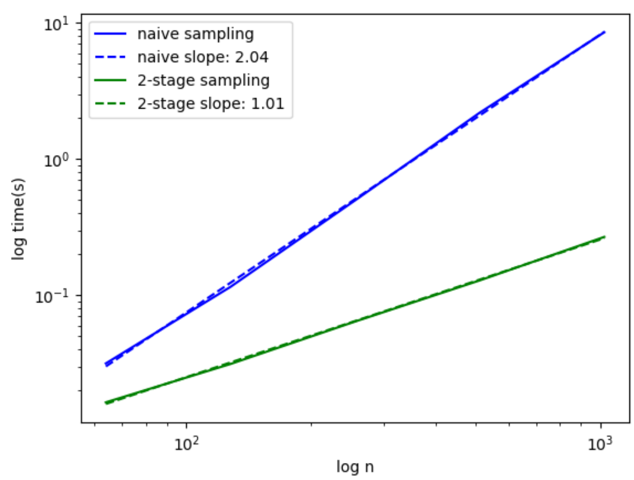


Figure 2: Time complexity for naive and 2-stage sampling algorithm, with fitting slop.

3 Component

The function `find_comp` in Appendix A aims to find all nodes that can be reached from node 1 (i.e. component that contains node 1). For $n = 4096$, plot the average component size (with sample size = 50) for each p varies from 0 to 0.001. At $p \approx 1/(n-1)$, a clear change from small components (approximately 0% of total network) to much larger ones, until approaching 100% at higher p values.

[2]Step(1)-(2) utilizes the definition of binomial expansions. Step(2)-(3) is based on limit of exponential.

A Appendix: Definition

The initialization codes below omits test to ensure that input arguments ofr

```
class Network(object):
    def __init__(self, num_nodes=None, adj_m=None):
        # Attributes:
            # number of nodes <num_nodes>;
            # adjacency list <adj_ls>;
            # adjacency matrix <adj_m>.

        if adj_m is not None:
            self.adj_m = adj_m
            self.num_nodes = adj_m.shape[0]
            self.adj_ls = np.array([set(np.nonzero(self.adj_m[i])[0]) for i in
            range(self.num_nodes)])

        elif num_nodes is not None:
            self.num_nodes = num_nodes
            self.adj_ls = np.empty(num_nodes, dtype=object)
            self.adj_ls.fill(set())
            self.adj_m = np.zeros((num_nodes,num_nodes), dtype = bool)

        else:
            raise ValueError("Missing argument for graph definition.")


    def add_edge(self, i, j):
        self.adj_ls[i].append(j)
        self.adj_ls[j].append(i)
        self.adj_m[i][j] = 1
        self.adj_m[j][i] = 1

    def neighbors(self, i):
        return self.adj_ls[i]

    def edge_list(self):
        return [(i,j) for i in self.adj for j in self.adj[i] if i<j]

    def edge_count(self):
        # Must divide by 2 to avoid repeated counting of edges
        return np.count_nonzero(self.adj_m) / 2

    def find_comp(self, i):
        """Find the component that contains node i for a given network object"""
        c = set()
        q = [i]
        while len(q) > 0:
            j = q.pop()
            c.add(j)
            q += self.neighbors(j) - c  # python type overloading
        return c
```

B Appendix: Sampling

B.1 Naive Sampling

```
def rm_graph_gen(n, p):
    """Naive sample a random network G(n,p) from Bernoulli distribution with nodes n
    and success rate p."""

    adj_m = np.zeros((n, n), dtype=int)
    for i in np.ndindex(n, n):
        if i[0] < i[1]:
            adj_m[i] = np.random.binomial(1, p)

            # To copy the adjacency information from one direction to its opposite direction
            adj_m[i[::-1]] = adj_m[i]
    rm_graph = Network(adj_m=adj_m)
    return rm_graph
```

B.2 2-stage Sampling

```
def rm_graph_gen2(n, p, m_only=False):
    """2-stage graph genertion sampling based on binomial edge distribution m"""

    nC2 = comb(n, 2)
    m = np.random.binomial(nC2, p)

    adj_m = np.zeros((n, n), dtype=int)

    m_idx = 0
    while m_idx < m:
        # Randomly select a pair of nodes (i,j)
        i = np.random.randint(0, n-1)
        j = np.random.randint(i+1, n)
        # Connect the nodes (i,j) if not already connected
        if adj_m[i][j] == 0:
            adj_m[i][j] = 1
            adj_m[j][i] = 1
            m_idx +=1
    rm_graph = Network(adj_m=adj_m)
    return rm_graph
```