University of California, Los Angeles
Department of Electrical and Computer Engineering
ECE 232E Large Scale Social and Complex Networks: Design and Algorithms

Project 1: Random Graphs and Random Walks

505430686 Viacheslav Inderiakin

904627828 Mia Levy

805626088 Connor Roberts

804737257 Tameez Latib

April 15, 2021

1. Generating Random Networks

1. Create random networks using Erdos-Renyi (ER) model

(a) Create undirected random networks with $n = 1000$ nodes, and the probability p for drawing an edge between two arbitrary vertices 0.003, 0.004, 0.01, 0.05, and 0.1. Plot the degree distributions. What distribution is observed? Explain why. Also, report the mean and variance of the degree distributions and compare them to the theoretical values.

Answer:

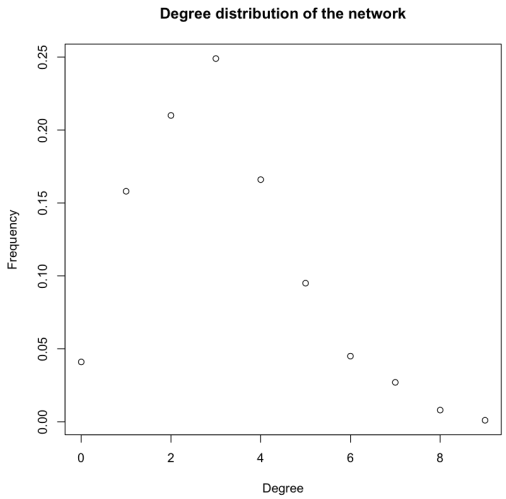
The degree distributions are plotted in Figure 1. The Erdos-Renyi degree distributions follow the Binomial distribution, because with probability p a node will connect to another node in the graph. The equation given in class is

$P_k = \binom{n}{k} p^k (1-p)^{n-k}$

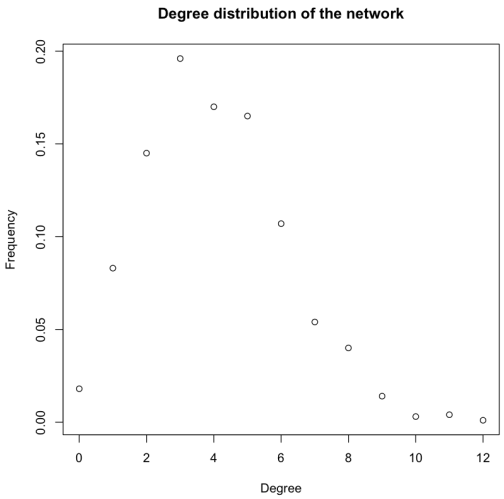
Where P_k is the probability that a randomly chosen node has degree k. As n becomes large and p becomes small this approximately becomes the Poisson distribution, which is simply a property of the Binomial distribution. Therefore, $Mean = np$ and $Var = np(1-p)$. Empirical and theoretical values of mean and variance are provided in table 1.

P=	Test mean	Theoretical mean	Test variance	Theoretical variance
0.003	2.996	3	2.832	2.991
0.004	4.018	4	4.251	3.984
0.01	9.886	10	9.678	9.9
0.05	50.21	50	47.493	47.5
0.1	101.42	100	96.305	90

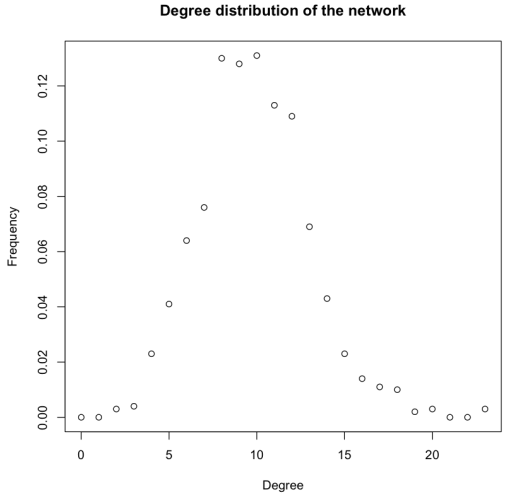
Table 1: Mean and variance of ER networks.



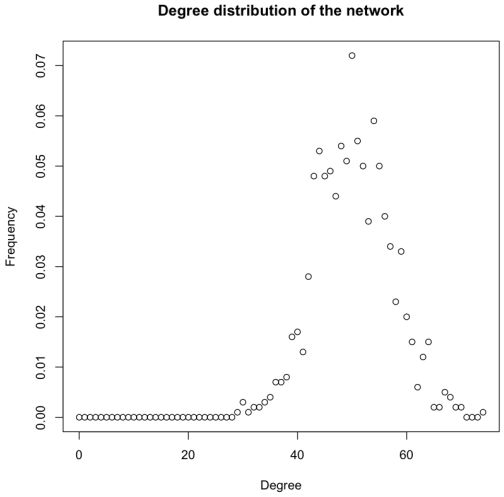
(a) $p = 0.003$



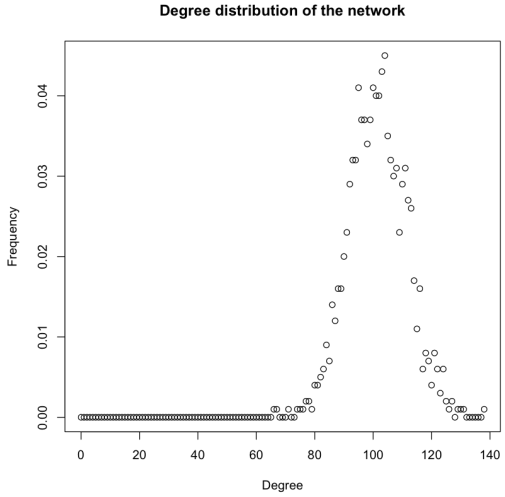
(b) $p = 0.004$



(c) $p = 0.01$



(d) $p = 0.05$



(e) $p = 0.1$

Figure 1: Degree distributions of ER networks for $n = 1000$.

(b) For each p and $n = 1000$, answer the following questions: Are all random realizations of the ER network connected? Numerically estimate the probability that a generated network is connected. For one instance of the networks with that p, find the giant connected component (GCC) if not connected. What is the diameter of the GCC?

Answer: No, not all random realizations of the ER network are connected. To numerically estimate the probability that each ER network is connected, each p is tested 1000 times, and then the fraction of connected graphs is taken as the probability. Then, for one instance for each p, the GCC diameter was found. The obtained results are shown in table 2. It can be noted that while having 3 edge per node is not enough to create a connected net, having 10 edges on average makes the net almost surely connected. We can also see that the size of GCC drops with increasing connectivity. This is because higher p allows to establish short paths to otherwise hardly accessible peripheral vertices.

p=	Estimated Probability Graph is Connected	GCC Diameter
0.003	0.000	18
0.004	0.000	10
0.01	0.957	6
0.05	1.000	3
0.1	1.000	3

Table 2: ER networks diameter and connectivity.

(c) It turns out that the normalized GCC size (i.e., the size of the GCC as a fraction of the total network size) is a highly nonlinear function of p, with interesting properties occurring for values where $p = O(\frac{1}{n})$ and $p = O(\frac{\ln n}{n})$. For $n = 1000$, sweep over values of p from 0 to a p_{max} that makes the network almost surely connected and create 100 random networks for each p. p_{max} should be roughly determined by yourself. Then scatter plot the normalized GCC sizes vs p. Plot a line of the average normalized GCC sizes for each p along with the scatter plot.

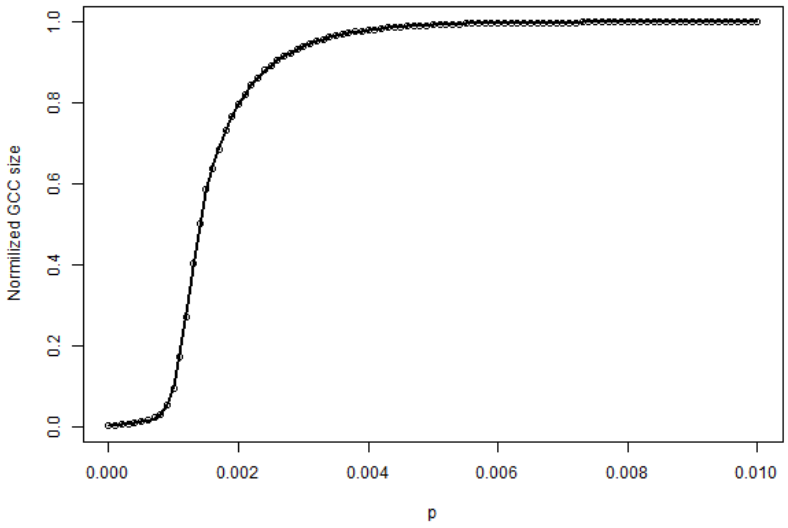


Figure 2: Normalized GCC sizes over p.

i. Empirically estimate the value of p where a giant connected component starts to emerge (define your criterion of "emergence")? Do they match with theoretical values mentioned or derived in lectures?

Answer: We can define an GCC emergence as an event when normalized GCC size overcomes 5%. From the figure 2, it can be noted that GCC emerges at $p = 0.001$. This matches with the theoretical value of $O(1/n)$ where $n = 1000$.

ii. Empirically estimate the value of p where the giant connected component takes up over 99% of the nodes in almost every experiment.

Answer: From figure 2 we can see that this happens at roughly $p \approx 0.005 - 0.006$, which matches the theoretical value of $O(\frac{ln(n)}{n})$ since $\frac{ln(n)}{n} = 0.006$.

(d) Define the average degree of nodes $c = n \times p = 0.5$.

i. Sweep over the number of nodes, n, ranging from 100 to 10000. Plot the expected size of the GCC of ER networks with n nodes and edge-formation probabilities $p = \frac{c}{n}$, as a function of n. What trend is observed?

Answer: As it can be seen from figure 3, the GCC size increases sharply from $n = 0$ to $n = 2000$. After roughly $n = 2000$, the slope starts to flatten out, which could be due to small p limiting connectivity.

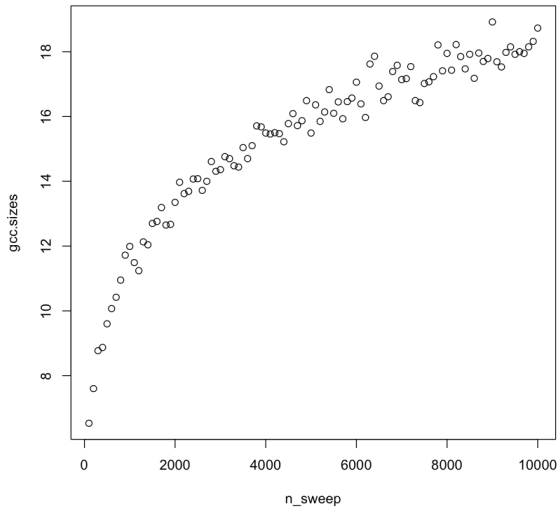


Figure 3: GCC size over n, $c = 0.5$, averaged over 100 trials for each point

ii. Repeat the same for $c = 1$.

Answer: GCC size vs n dependency is shown in figure 4. In this case, the GCC sizes are larger than for $c = 0.5$. This is due to p being larger, which leads to increased connectivity.

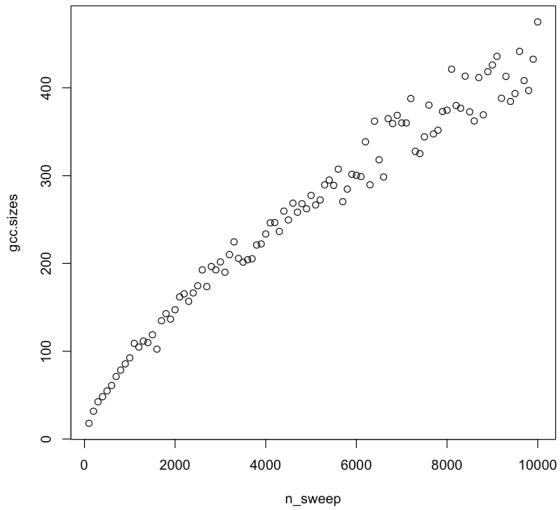


Figure 4: GCC size over n, $c = 1$, averaged over 100 trials for each point

iii. Repeat the same for values of $c = 1.1$, $1, 2$; $1, 3$, and show the results for these three values in a single plot.

Answer: The plots in figure 5 have become more linear as c increases, and the difference in slopes is larger as c is smaller, i.e. there seems to be a larger difference between $c = 1.1$ and $c = 1.2$ than between $c = 1.2$ and $c = 1.3$.

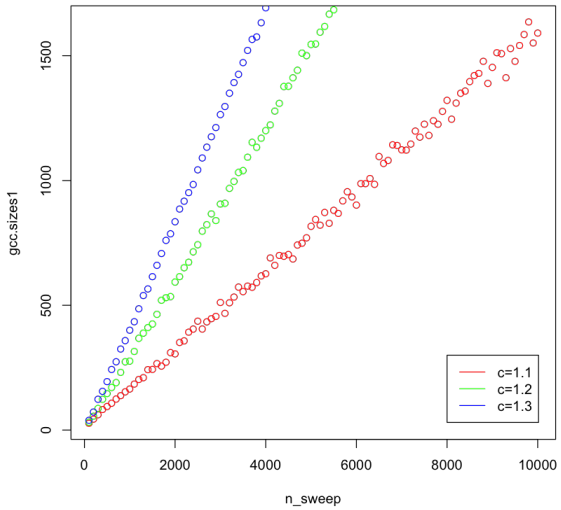


Figure 5: GCC size over n, $c = 1.1$, $c = 1.2$, and $c = 1.3$, averaged over 100 trials for each point

iv. What is the relation between the expected GCC size and n in each case?

Answer: The relationship is approximately linear with n. Approximations of these linear functions are shown in the table 3.

c=	GCC size as a function of n
0.5	$0.007n$ from $n = 0$ to $n = 2000$ $0.0006n$ from $n = 2000$ to $n = 10000$
1	$0.04n$
1.1	$0.16n$
1.2	$0.34n$
1.3	$0.50n$

Table 3: GCC size as a function of n approximations.

2. Create networks using preferential attachment model

(a) Create an undirected network with $n = 1000$ nodes, with preferential attachment model, where each new node attaches to $m = 1$ old nodes. Is such a network always connected?

Answer: yes, any network created using preferential attachment algorithm with $m \geq 1$ is connected. This can be proven with induction

- At step 1 (only 1 node), the network is connected;
- If the network is connected at step t, adding a new vertex with $m \geq 1$ connections also creates a connected network at step $t + 1$.

(b) Use fast greedy method to find the community structure. Measure modularity.

Answer: Modularity = 0.932.

Preferential attachment model favors vertices that already have many connections while selecting where to connect an incoming node. For $m = 1$, this results in emergence of many small clusters where elements are all gathered around the center node. This graph is, therefore, can be divided into clusters well, so modularity is expected to be high (i.e. close to 1).

(c) Try to generate a larger network with 10000 nodes using the same model. Compute modularity. How is it compared to the smaller network's modularity?

Answer: Modularity = 0.977.

Having more nodes in the network creates clusters that are even bigger than ones in 1.2.b), with smaller number of edges connecting them. Thus, fast greedy clusterization performs even better: modularity($n = 10000$) > modularity($n = 1000$).

(d) Plot the degree distribution in a log-log scale for both $n = 1000$; 10000, then estimate the slope of the plot using linear regression.

Answer: The degree distributions for both n-s are provided in figure 6. Using linear regression, we can find that the slopes of the curves are $\beta_{n=1000} = -2.18$ and $\beta_{n=10000} = -2.63$. It can be noted that obtained slopes are higher than theoretical one $\beta_{th} = 3$. This is explained by the fact that β_{th} is derived for a "steady case", which is achieved when the number of vertices $|V| \to \infty$. As $n = 10000$ is more close to this condition than $n = 1000$, $|\beta_{th} - \beta_{n=10000}| < |\beta_{th} - \beta_{n=1000}|$.

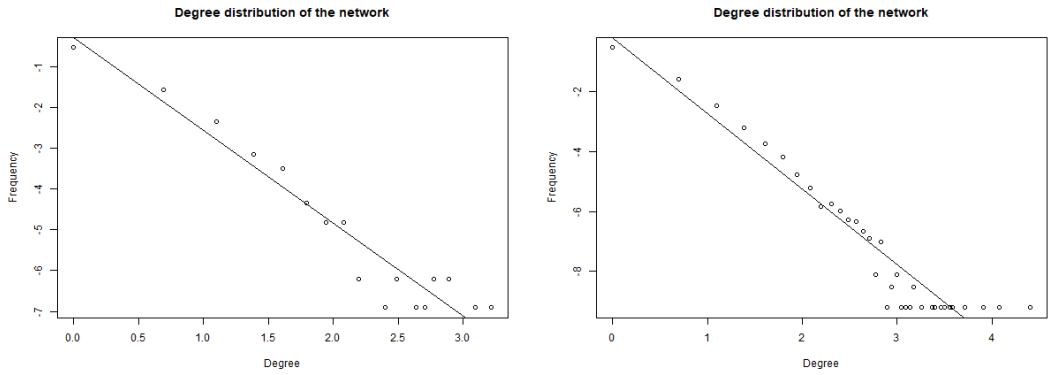


Figure 6: Degree distribution for $n = 1000$ (left) and $n = 10000$ (right).

(e) In the two networks generated in 2(d), perform the following: Randomly pick a node i, and then randomly pick a neighbor j of that node. Plot the degree distribution of nodes j that are picked with this process, in the log-log scale. Is the distribution linear in the log-log scale? If so, what is the slope? How does this differ from the node degree distribution?

Answer: The degree distributions are shown in figure 7. It appears as though this distributions are relatively linear. Comparing the slopes to those found in part d), it appears as though they converge to approximately the same value. However, when running for only 1 graph, as seen in 8, we see the slopes are significantly less than those when we average over multiple graphs. This can be attributed to outliers for individual graphs being averaged out as we record data from multiple graphs.

Slope for n = 1,000 and m = 1 averaged over 5000 graphs: -2.7742
Slope for n = 10,000 and m = 1 averaged over 5000 graphs: -2.6542

Slope for n = 1,000 and m = 1 over one graph: -1.0342
Slope for n = 10,000 and m = 1 over one graph: -1.6467

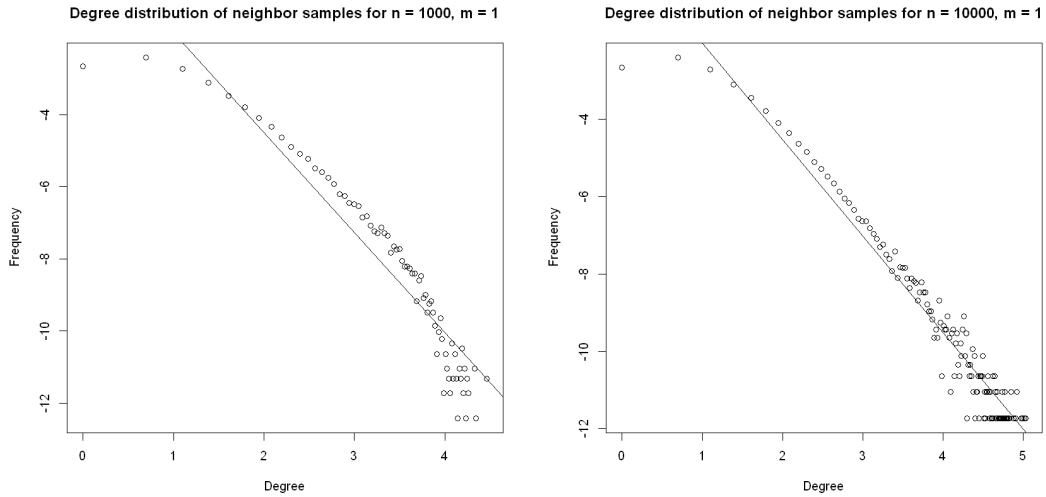


Figure 7: Degree distribution for $n = 1000$ (left) and $n = 10000$ (right) with m = 1 averaged over 5000 graphs.

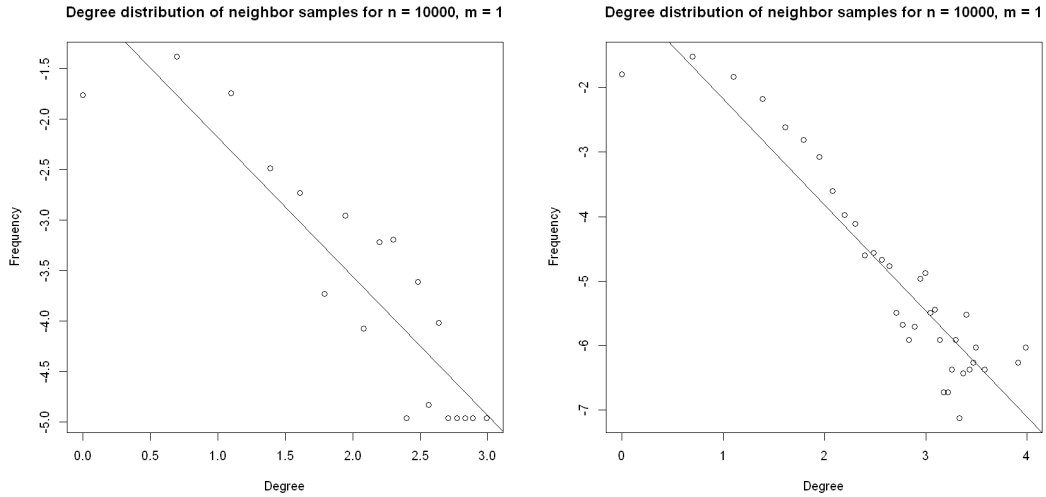


Figure 8: Degree distribution for $n = 1000$ (left) and $n = 10000$ (right) with m = 1 for one graph.

(f) Estimate the expected degree of a node that is added at time step i for $1 \leq i \leq 1000$. Show the relationship between the age of nodes and their expected degree through an appropriate plot.

Answer: In figure 9 we have shown both the theoretical, as well as empirically found, relationship between the age of the nodes and their expected degree. For our empirical

data we averaged over 50 graphs in order to get a better overall representation of our data.

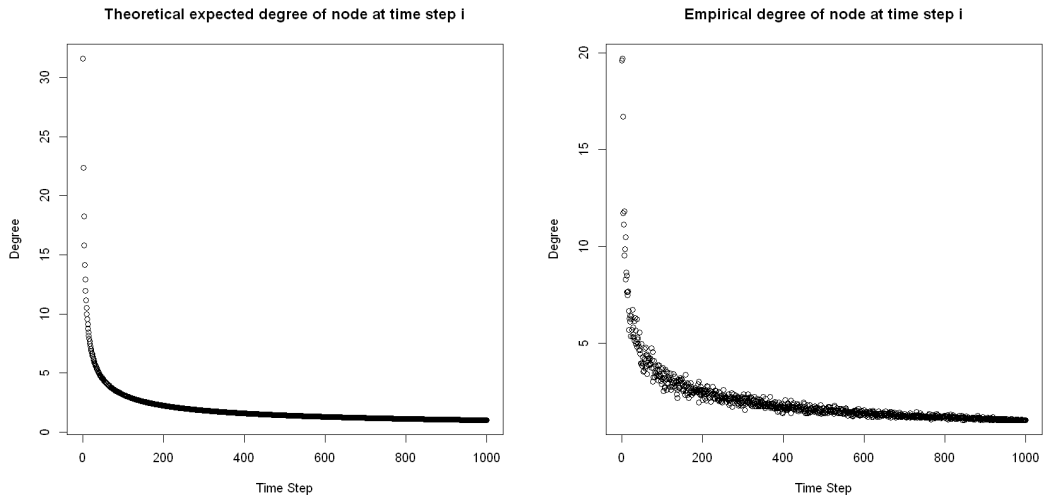


Figure 9: Relationship between expected degree of the nodes and age of the nodes.

(g) Repeat the previous parts for $m = 2$; and $m = 5$. Compare the results of each part for different values of m.

Answer: As it was mentioned before, for any n for $m = 2$ and $m = 5$ networks are connected.

Modularities of clusterizations obtained for newly generated graphs are provided in table 4. It can be noted that fast greedy clusterization quality quickly drops with increasing m for both n-s. This is explained by the increased connectivity of the networks, which increases the number of edges connecting clusters to each other.

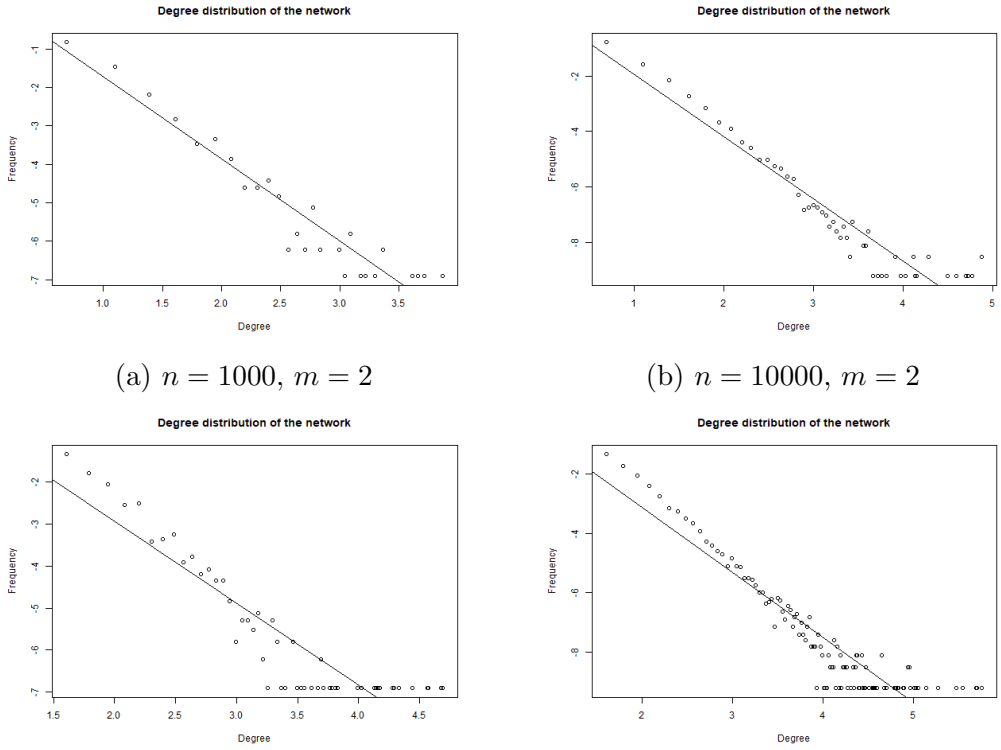
	m = 1	m = 2	m = 5
n = 1000	0.932	0.527	0.277
n = 10000	0.977	0.530	0.272

Table 4: Modularities of PA networks.

The degree distribution for newly generated networks are provided in figure 10, and their slopes are shown in table 5. It can be seen that for higher m β deviates more from theoretical values, as each incoming node introduces more edges which makes it harder to achieve stabilized degree distribution.

	m = 1	m = 2	m = 5
n = 1000	-2.18	-2.13	-1.94
n = 10000	-2.63	-2.24	-2.18

Table 5: Slopes of PA networks degree distributions.



(a) $n = 1000$, $m = 2$ (b) $n = 10000$, $m = 2$

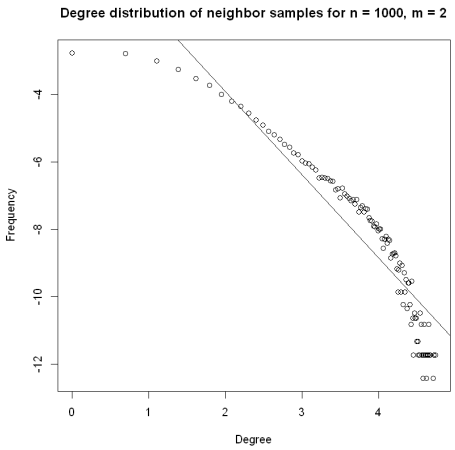
(c) $n = 1000$, $m = 5$ (d) $n = 1000$, $m = 5$

Figure 10: Degree distributions for new networks.

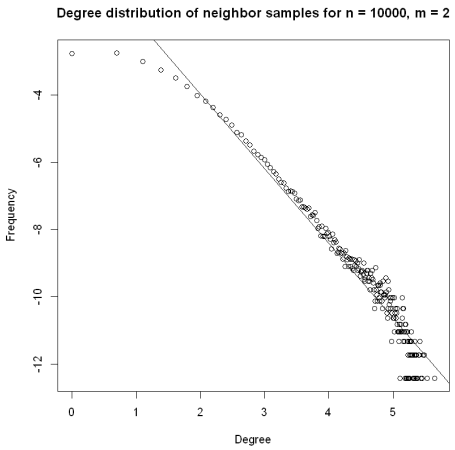
Figure 11 show the relationship between frequency and degree for m = 2 and m = 5 respectively, averaged for 5000 graphs.

	m = 1	m = 2	m = 5
n = 1000	-2.7742	-2.4722	-2.3932
n = 10000	-2.6542	-2.2195	-2.2019

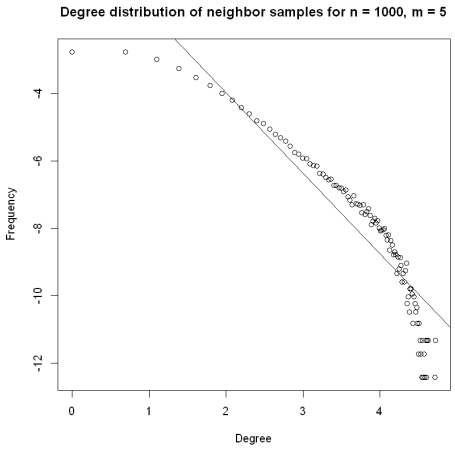
Table 6: Slopes for degree distribution of neighbor of random node



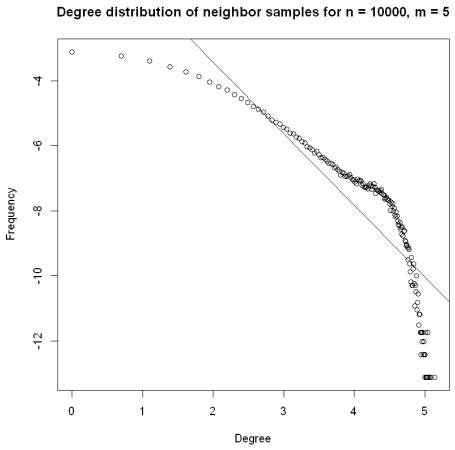
(a) $n = 1000$, $m = 2$



(b) $n = 10000$, $m = 2$



(c) $n = 1000$, $m = 5$



(d) $n = 1000$, $m = 5$

Figure 11: Degree distribution for the 1^{st} neighbour averaged over 5000 graphs.

Figure 12 shows the dependency between node's edge and its degree. We can see that the distribution has exactly the same shape as in the case of $m = 1$ (see figure 9), but with scale increased (which is a result of having more edges in the networks).

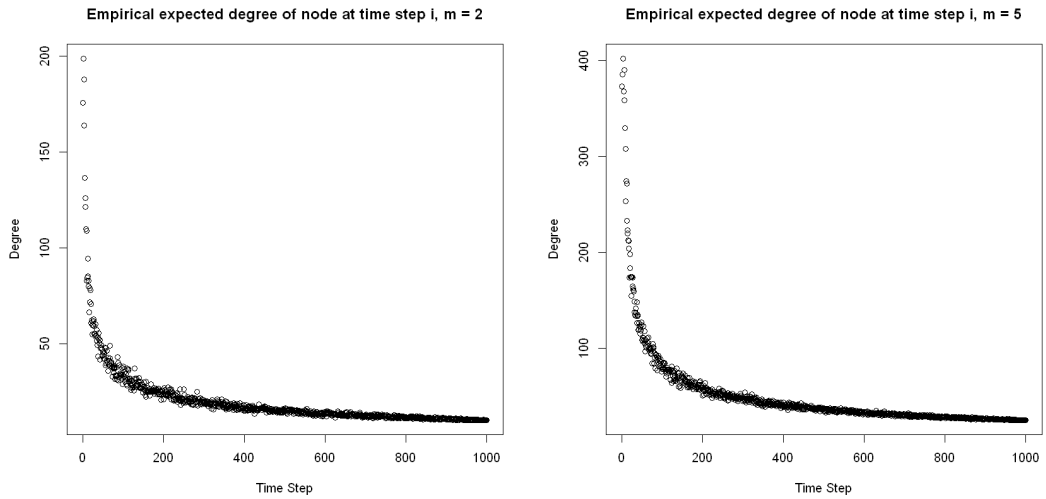


Figure 12: Relationship between expected degree of the nodes and age of the nodes.

(h) Again, generate a preferential attachment network with $n = 1000$, $m = 1$. Take its degree sequence and create a new network with the same degree sequence, through stub-matching procedure. Plot both networks, mark communities on their plots, and measure their modularity. Compare the two procedures for creating random power-law networks.

Answer: Figure 13 shows the graphs generated using preferential attachment as well as the graph generating using the same degree sequence and stub matching. As we can see in the figure, the preferential attachment network is always connected and has clearly defined communities while the degree sequence network has many pairs that are not connected to its center. Additionally, the modularity of the preferential attachment model is significantly more than that of the degree sequence network. Thus, we can conclude that PA algorithm is more suited to create graphs with defined communities than basic Power Law network generation algorithms.

Modularity of preferential attachment = 0.933708
Modularity using same degree sequence = 0.7387829

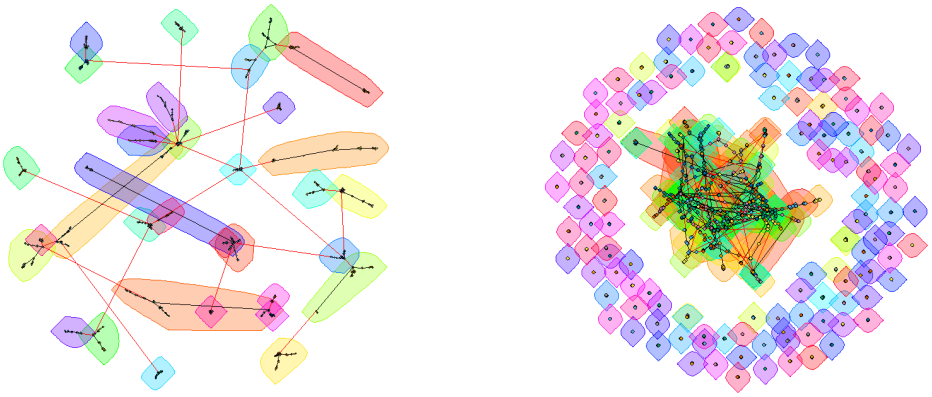


Figure 13: Graphs generated using preferential attachment (left) using the same degree sequence (right) with community structures marked

3. Create a modifed preferential attachment model that penalizes the age of a node

(a) Each time a new vertex is added, it creates m links to old vertices and the probability that an old vertex is cited depends on its degree (preferential attachment) and age. In particular, the probability that a newly added vertex connects to an old vertex is proportional to:

$$P[i] \sim (ck_i^{\alpha} + a)(dl_i^{\beta} + b)$$

where k_i is the degree of vertex i in the current time step, and l_i is the age of vertex i. Produce such an undirected network with 1000 nodes and parameters $m = 1$, α, $\beta = -1$, and $a = c = d = 1$, $b = 0$. Plot the degree distribution. What is the power law exponent?

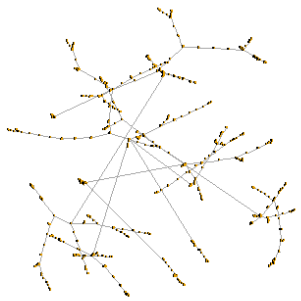


Figure 14: Preferential Attachment graph generated with sample pa with $m = 1$, α, $\beta = -1$, and $a = c = d = 1$, $b = 0$ and $n = 1000$

Answer: Slope of graph is $slope_{age} = -2.49487$, which closely matches the theoretical power law exponent value of $slope_{age} = 3$.

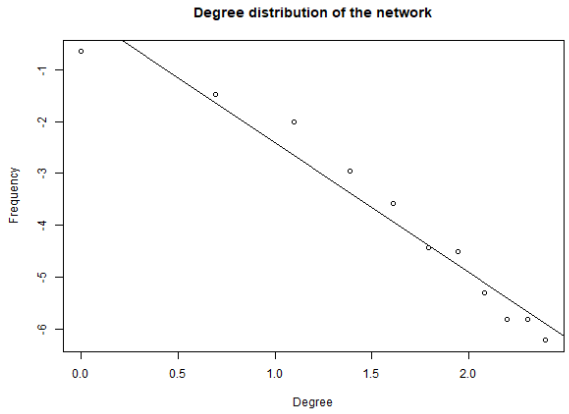


Figure 15: Preferential Attachment graph generated with sample pa with $m = 1$, α, $\beta = -1$, and $a = c = d = 1$, $b = 0$ and $n = 1000$

(b) Use fast greedy method to find the community structure. What is the modularity?

Answer: Modularity is 0.9352005.

This value is close to 1 because similarly to classic PA algorithm with $m = 1$, PA with age penalty allows to create many big clusters gathering around center nodes, which are connected by a very small number of vertices.

2. Random Walk on Networks

1. Random walk on Erdos-Renyi networks

(a) Create an undirected random network with 1000 nodes, and the probability p for drawing an edge between any pair of nodes equal to 0.01.

Answer: A graph was generated using the `sample_gnp()` function in R.

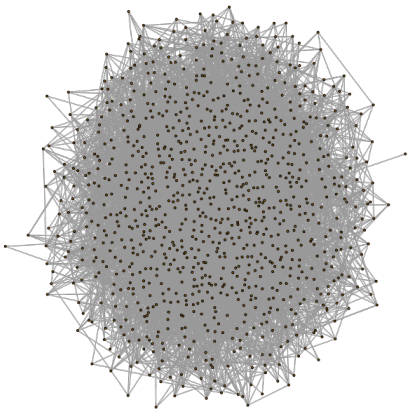


Figure 16: Erdos-Renyi graph generated with sample gnp with $p = 0.01$ and $n = 1000$

(b) Let a random walker start from a randomly selected node (no teleportation). We use t to denote the number of steps that the walker has taken. Measure the average distance (defined as the shortest path length) $\langle s(t) \rangle$ of the walker from his starting point at step t. Also, measure the variance $\sigma^2(t) = \langle (s(t) - \langle s(t) \rangle)^2 \rangle$ of this distance. Plot $\langle s(t) \rangle$ v.s. t and $\sigma^2(t)$ v.s. t. Here, the average $\langle \cdot \rangle$ is over random choices of the starting nodes.

Answer: The mean and variances for 100 trials of random walks are provided in figure 17. We can see that after ≈ 10 steps the mean converges at the value close to the network's diameter. The variance converges as well.

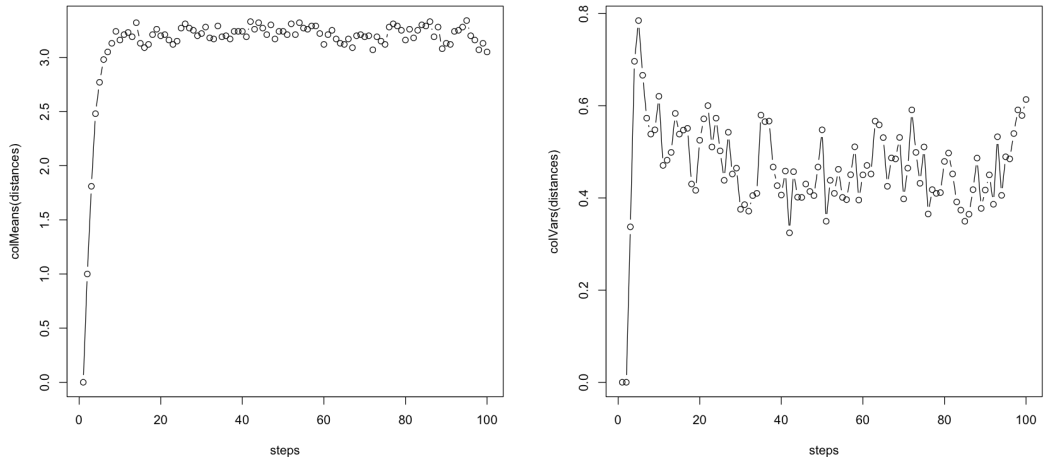


Figure 17: Mean distances (left) and variance of distances (right) over step number

(c) Measure the degree distribution of the nodes reached at the end of the random walk. How does it compare to the degree distribution of graph?

Answer: The degree distribution of the end nodes and the degree distribution of the original graph are very similar. The original graph distribution has $Mean = 10.172$ and $Var = 10.142$. The end node degree distribution has $Mean = 11.510$ and $Var = 10.515$. The original graph has a Binomial degree distribution, and the end node degree distribution is similar.

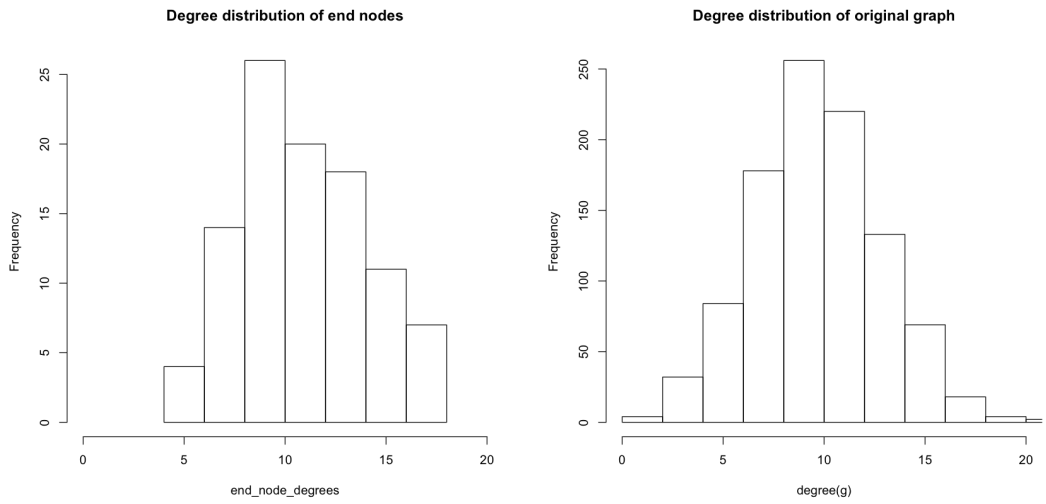


Figure 18: Degree distributions for end nodes of the random walk and the original graph

(d) Repeat 1(b) for undirected random networks with 10000 nodes. Compare the results and explain qualitatively. Does the diameter of the network play a role?

Answer: The diameter for $n = 1000$ is 5, and the diameter for $n = 10000$ is 3. For $n = 1000$, the means reach a steady state at an higher number of steps, roughly 10, and for $n = 10000$ it takes about 5 steps. For $n = 1000$ the steady state mean is about 3.2 and the steady state variance is about 4.5. For $n = 10000$ the steady state mean is about 2.3 and the steady state variance is about 0.25. Thus, the lower is the diameter, the lower are mean, variance and number of steps required for convergence (as there are less "places to explore").

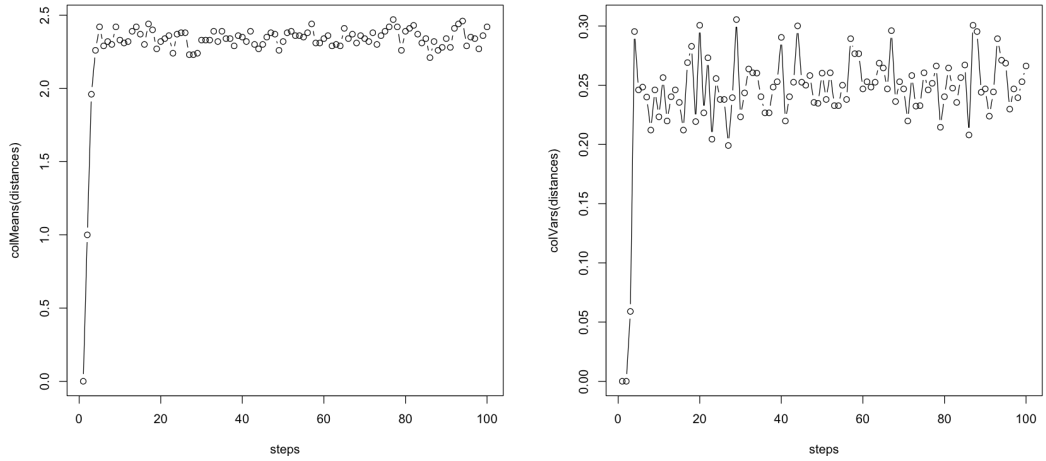


Figure 19: Mean distances (left) and variance of distances (right) over step number, for $n = 10000$

2. Random walk on networks with fat-tailed degree distribution

(a) Generate an undirected preferential attachment network with 1000 nodes, where each new node attaches to $m = 1$ old nodes.

Answer:

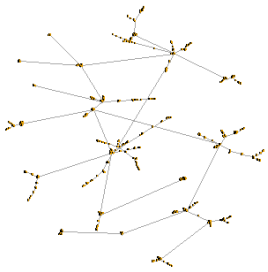


Figure 20: Undirected preferential attachment network with $n = 1000$

(b) Let a random walker start from a randomly selected node. Measure and plot $\langle s(t) \rangle$ v.s. t and $\sigma^2(t)$ v.s. t.

Answer: The mean and variance for a random walk on PA network are provided in figure 21. We can see that there are much more "periphery" nodes to explore, so both mean and variance continue to grow and do not stabilize even for 400 steps.

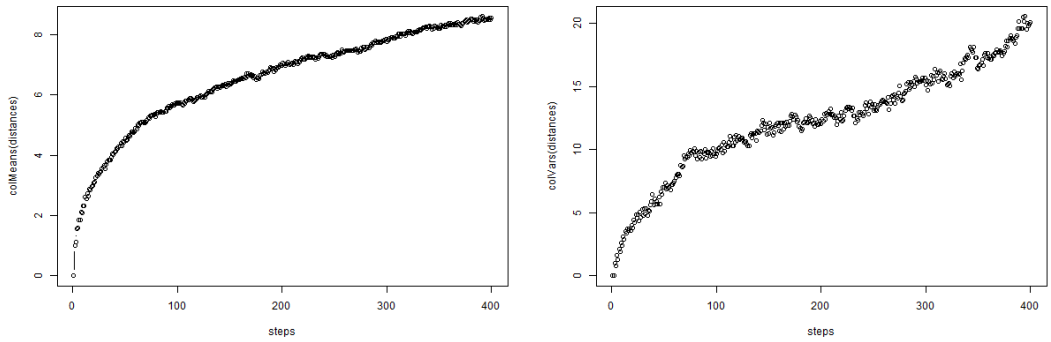


Figure 21: Mean distances (left) and variance of distances (right) over step number, $n = 1000$

(c) Measure the degree distribution of the nodes reached at the end of the random walk on this network. How does it compare with the degree distribution of the graph?

Answer: We see that the two distributions are similar and both behave like fat tailed degree distributions.

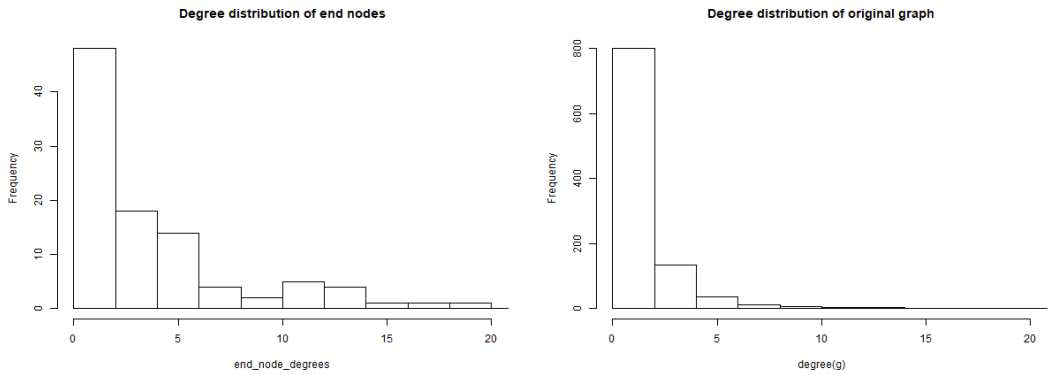


Figure 22: Degree distributions for end nodes of the random walk and the original graph

(d) Repeat 2(b) for preferential attachment networks with 100 and 10000 nodes, and $m = 1$. Compare the results and explain qualitatively. Does the diameter of the network play a role?

Answer: The means and variance of networks with $n = 100$ and $n = 10000$ are provided in figures 23 and 24 correspondingly. We can note that the mean distance saturates for $n = 100$ network, as its diameter is the lowest (13). For $n = 1000$ and $n = 10000$, the diameter is much higher (32 and 56) correspondingly, so we do not observe saturation in our experiments.

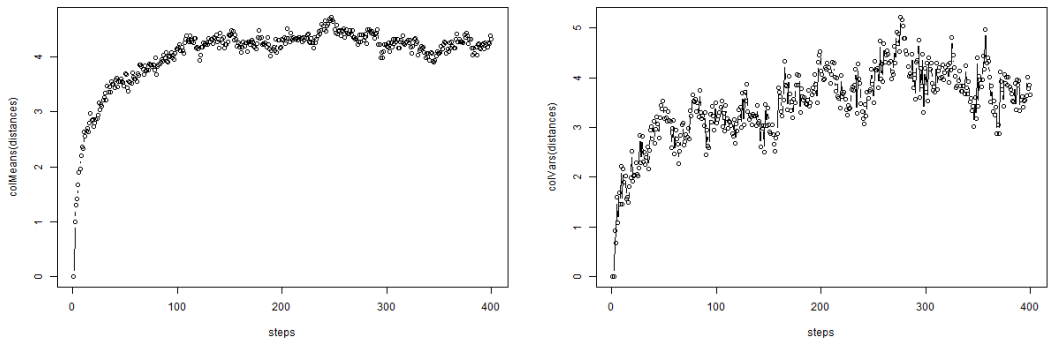


Figure 23: Mean distances (left) and variance of distances (right) over step number, $n = 100$

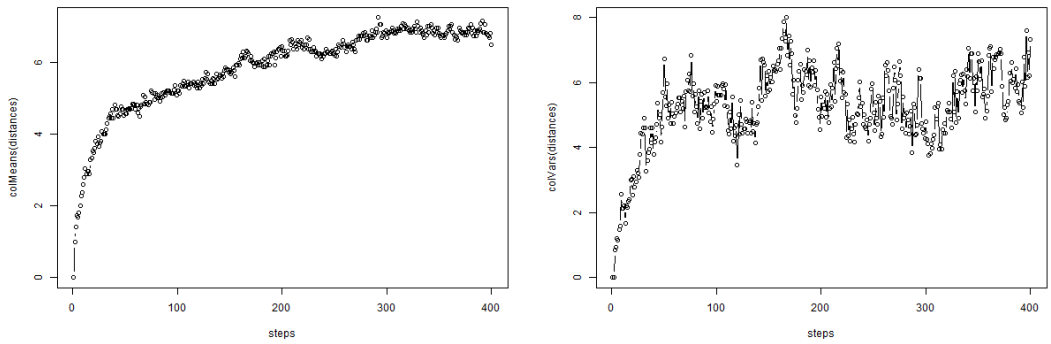


Figure 24: Mean distances (left) and variance of distances (right) over step number, $n = 10000$

3. PageRank

The PageRank algorithm, as used by the Google search engine, exploits the linkage structure of the web to compute global "importance" scores that can be used to in influence the ranking of search results. Here, we use random walk to simulate PageRank.

(a) We are going to create a directed random network with 1000 nodes, using the preferential attachment model. Note that in a directed preferential attachment network, the out-degree of every node is m, while the in-degrees follow a power law distribution. One problem of performing random walk in such a network is that, the very first node will have no outbounding edges, and be a "black hole" which a random walker can never "escape" from. To address that, let's generate another 1000-node random network with preferential attachment model, and merge the two networks by adding the edges of the second graph to the first graph with a shuffling of the indices of the nodes. Create such a network using $m = 4$. Measure the probability that the walker visits each node. Is this probability related to the degree of the nodes?

Answer: The probabilities of visit are calculated for 5 trials of random walk doing 500 steps each and are shown in figure 25 (left). As we can see, probability of visit generally grows with in-degree of the node, as the more links point to the node the higher is a chance to choose one of them. This trend was confirmed by running random walk on 3000 graphs as shown in figure 25 (right).

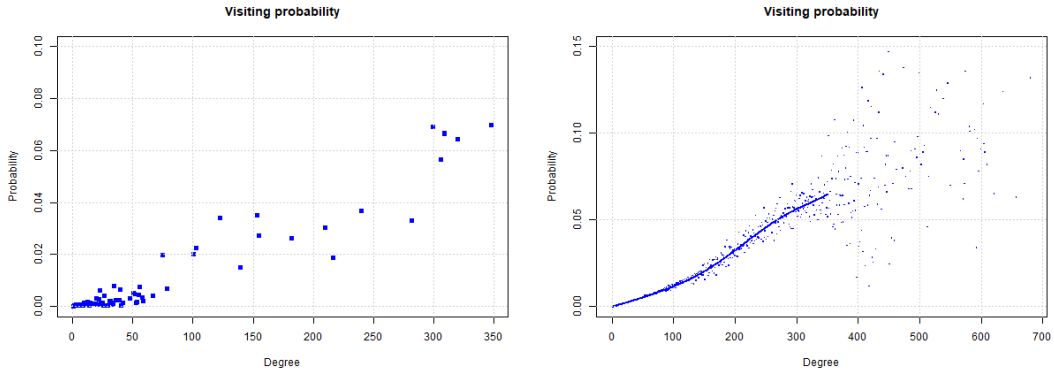


Figure 25: Average probability of being visited for nodes with a given in-degree in the case of random walk. Probabilities are provided for a single graph (on the left) and averaged over 3000 graphs (on the right)

(b) In all previous questions, we didn't have any teleportation. Now, we use a teleportation probability of $alpha = 0.15$. By performing random walks on the network created in 3(a), measure the probability that the walker visits each node. Is this probability related to the degree of the node?

Answer: The probabilities of visit were recomputed on the same graph for a random walker that has $\alpha = 0.15$ teleportation chance. The results are shown in figure 26 (left), and the plot from 2.3.a) is also provided for comparison. It can be noted that random teleportation works in favor to the nodes with low in-degree (≤ 25) by teleporting walker to the nodes that hardly can be accessed otherwise. This, however, distracts walker from the "center" high-degree nodes, so their probabilities of visit drop. From figure 26 (right), we can see that these trends persists in probabilities averaged over 3000 graphs.

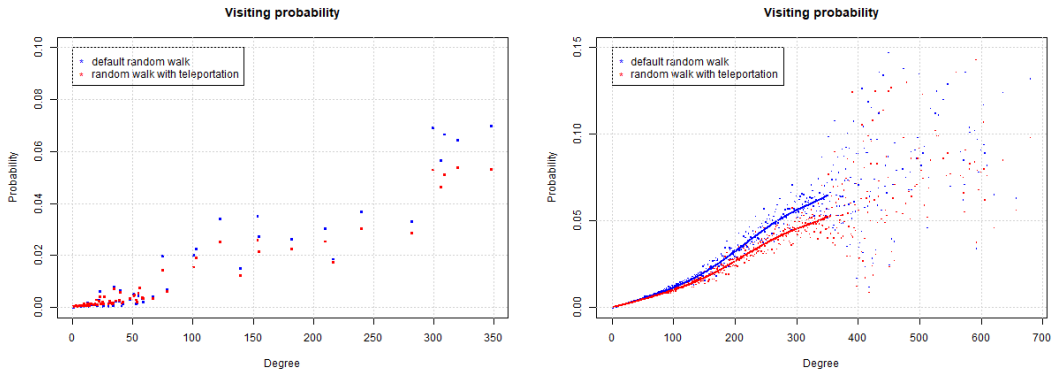


Figure 26: Average probability of being visited for nodes with a given in-degree in the case of random walk with teleportation. Probabilities are provided for a single graph (on the left) and averaged over 3000 graphs (on the right). Classic random walk probabilities are plotted for comparison.

4. Personalized PageRank

While the use of PageRank has proven very effective, the web's rapid growth in size and diversity drives an increasing demand for greater exibility in ranking. Ideally, each user should be able to define their own notion of importance for each individual query.

(a) Suppose you have your own notion of importance. Your interest in a node is proportional to the node's PageRank, because you totally rely upon Google to decide which website to visit (assume that these nodes represent websites). Again, use random walk on network generated in question 3 to simulate this personalized PageRank. Here the teleportation probability to each node is proportional to its PageRank (as opposed to the regular PageRank, where at teleportation, the chance of visiting all nodes are the same and equal to $\frac{1}{N}$). Again, let the teleportation probability be equal to $\alpha = 0.15$. Compare the results with 3(a).

Answer: The probabilities of visit were recomputed on the same graph for a random walker that has $\alpha = 0.15$ teleportation chance and teleportation weights proportional to nodes' PageRank scores. The results are shown in figure 26 (left), and the plot from 2.3.a) is also provided for comparison. It can be noted that, similarly to teleportation with uniform weights, weighted teleportation works in favor of nodes with low in-degree at the cost of highly visited nodes. However, in the case of weighted teleportation vertices with high in-degree receive lower drop in the number of visits. This is because while a random walker gets a chance to be teleported to "peripheral" nodes, the chances of remaining in the "center" section of the graph after teleportation are higher. From figure 26 (right), we can see that these trends persists in probabilities averaged over 3000 graphs, as "random walk with pagerank" curve lies much closer to "random walk" without teleportation.

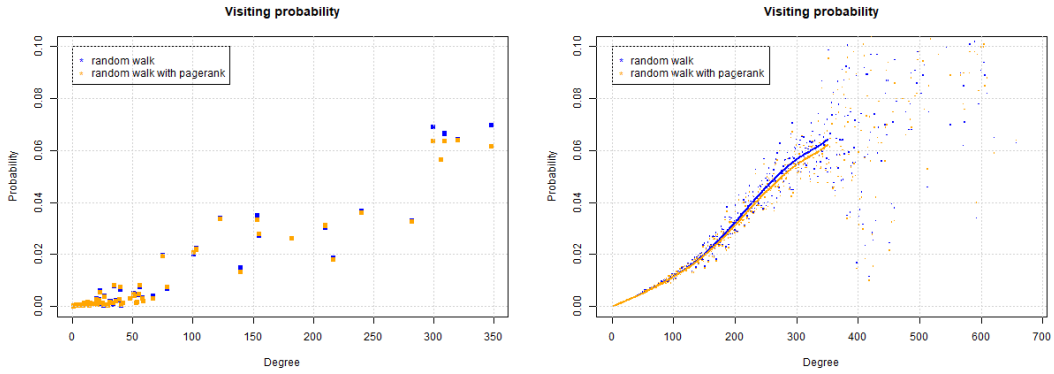


Figure 27: Average probability of being visited for nodes with a given in-degree in the case of random walk with weighted teleportation. Probabilities are provided for a single graph (on the left) and averaged over 3000 graphs (on the right). Random walk with unweighted teleportation probabilities are plotted for comparison.

(b) Find two nodes in the network with median PageRanks. Repeat part 4(a) if teleportations land only on those two nodes (with probabilities 1/2, 1/2). How are the PageRank values affected?

Answer: The probabilities of visit were recomputed on the same graph for a random walker that has $\alpha = 0.15$ teleportation chance and can teleport only to the trusted

nodes. The results are shown in figure 28 (left), along with probabilities of random walker with PageRank teleportation. We can see that trusted nodes distract the random walker from vertices with high in-degree, so their probabilities are lowered than they were in case of PageRank weights and more closer to the case of uniform teleportation chance. Meanwhile, the nodes that got boosted with 0.075 teleportation chance each have low in-degree. As there is many nodes with similar degrees, the high number of visits for these particular vertices gets smoothed out and is not noticeable in average probability vs. degree plot. As a result, average probabilities of visit in the case of trusted nodes teleportation are close to or lower than those in case of PageRank teleportation. We can also see that this in the figure 28 (right) plotted for 3000 graphs, as purple curve lies under the orange one.

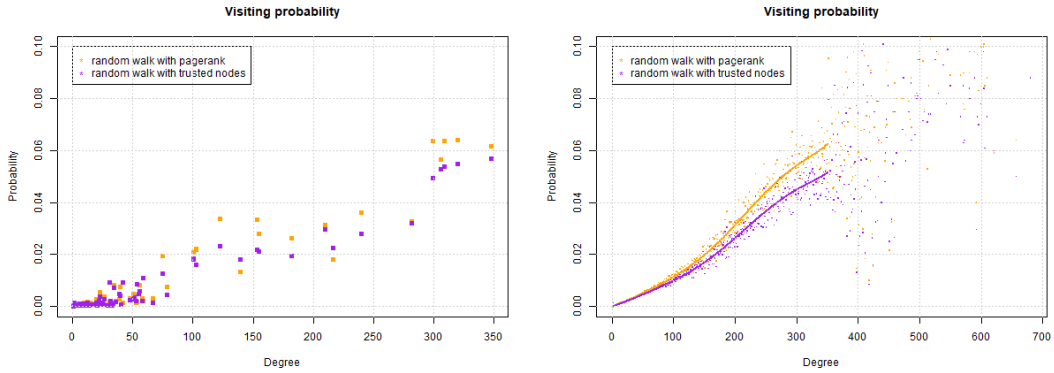


Figure 28: Average probability of being visited for nodes with a given in-degree in the case of random walk with teleportation to a set of trusted nodes. Probabilities are provided for a single graph (on the left) and averaged over 3000 graphs (on the right). Random walk with weighted teleportation probabilities are plotted for comparison.

(c) More or less, 4(b) is what happens in the real world, in that a user browsing the web only teleports to a set of trusted web pages. However, this is against the assumption of normal PageRank, where we assume that people's interest in all nodes are the same. Can you take into account the effect of this self-reinforcement and adjust the PageRank equation?

Answer: Let $G = (V, E)$, $|V| = n$ to be a graph generated using procedure described in question 2.3.a). Let's also assume that in the case with no teleportation node-node incidence matrix was A_G such that

$$A_G = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & otherwise \end{cases}$$

Let $T = \{t_1, t_2, ..., t_m\}$ to be a set of trusted nodes, α to be a teleportation probability. Then, in the case of the conditions described in the problem, node-node transition matrix will become

$$A'_G = \begin{cases} 1, (v_i, v_j) \in E \text{ or } j \in T \\ 0, \ otherwise \end{cases}$$

Correspondingly, node-node transition probability matrix will become

$$P'_G = \begin{cases} \frac{1-\alpha}{k_i},\ (v_i, v_j) \in E \\ \frac{\alpha}{|T|},\ j \in T \\ 0,\ otherwise \end{cases}$$

where k_i - out-degree of node i.

1_1

April 15, 2021

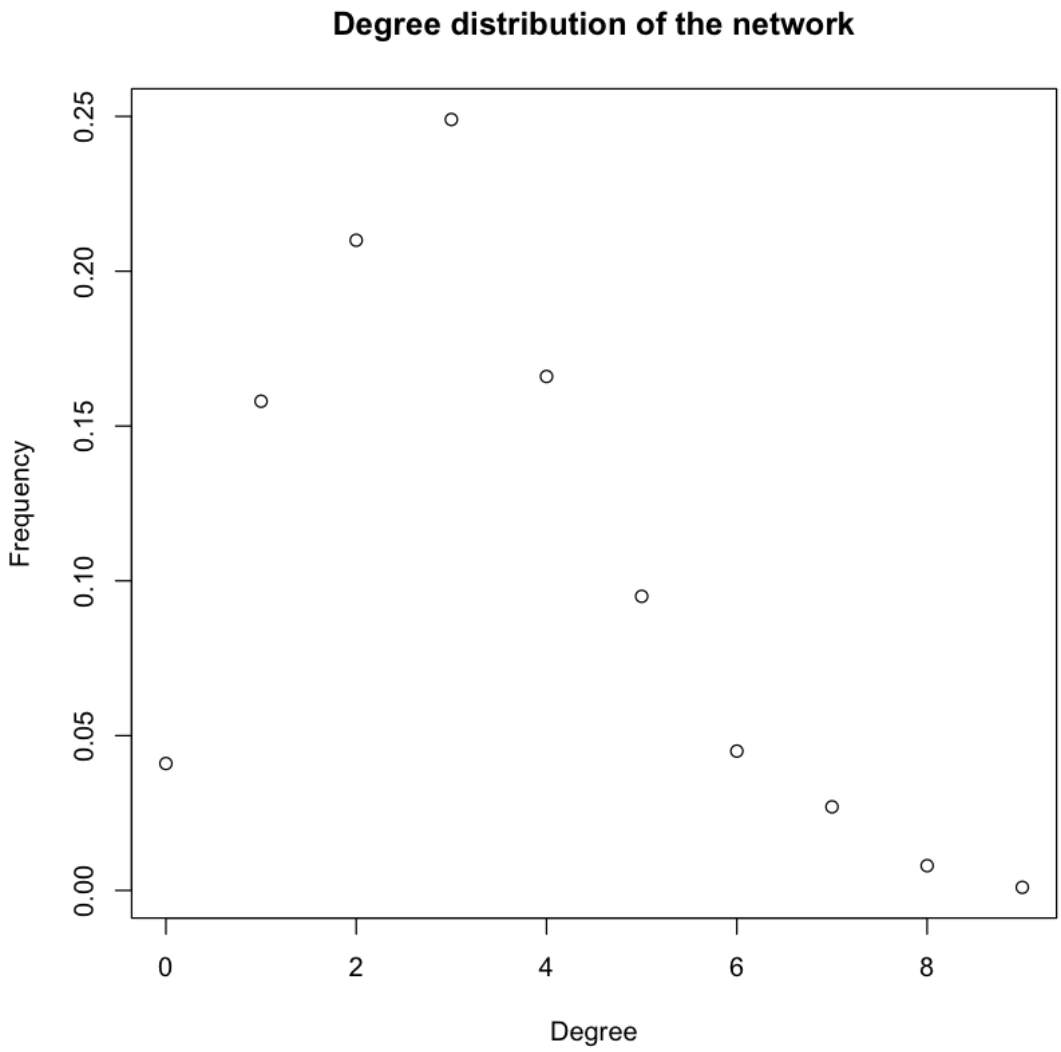
[2]:
```
library('igraph')
library('Matrix')
library('pracma')
if (!require("ITNr")) install.packages("ITNr")
library ("ITNr")
```

Problem 1.1a: Create undirected random networks with n = 1000 nodes, and the probability p for drawing an edge between two arbitrary vertices 0.003, 0.004, 0.01, 0.05, and 0.1. Plot the degree distributions. What distribution is observed? Explain why. Also, report the mean and variance of the degree distributions and compare them to the theoretical values. Hint Useful function(s): sample_gnp , degree , degree_distribution , plot

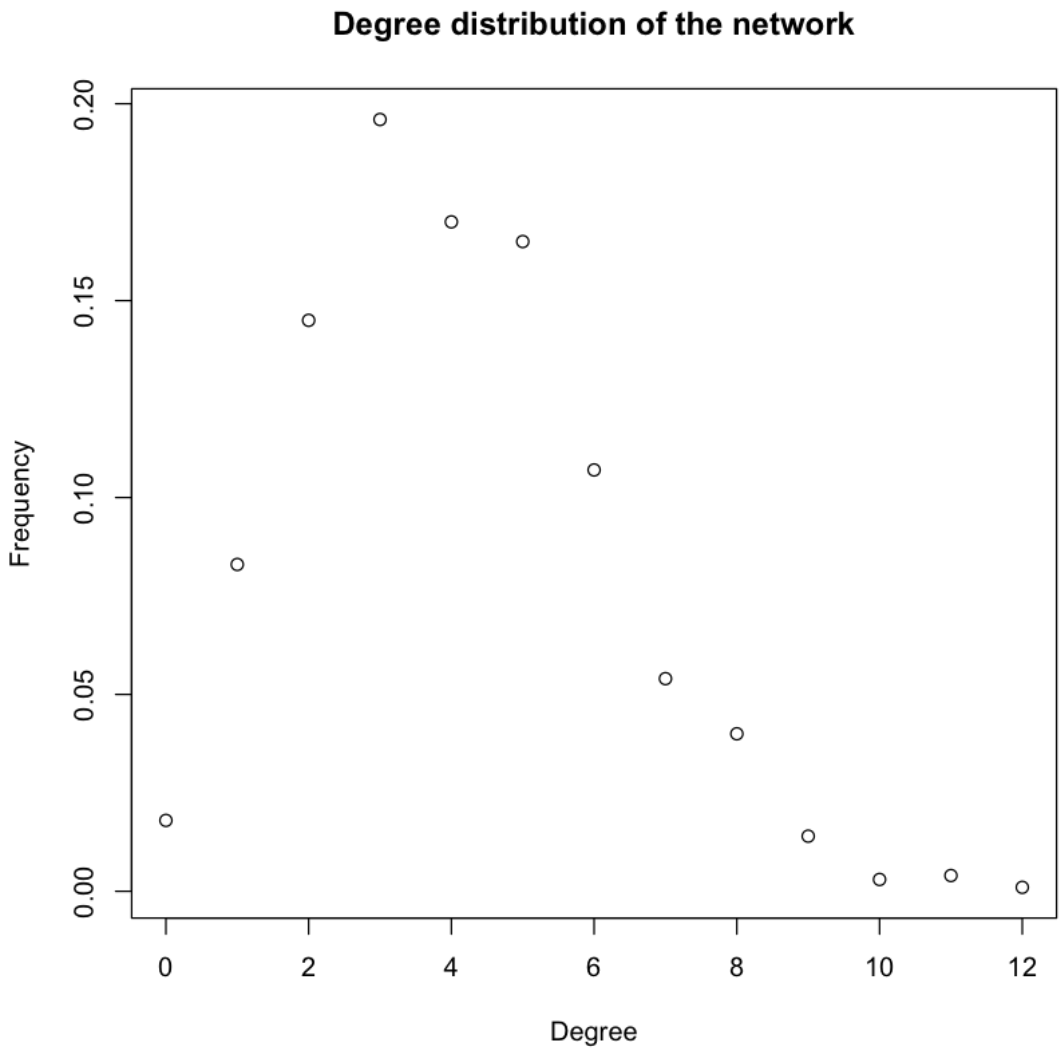
[8]:
```
# Create undirected random networks, sample_gnp uses Erdos-Renyi (I think?)
g1 = sample_gnp(n=1000, p=0.003, directed=FALSE)
g2 = sample_gnp(n=1000, p=0.004, directed=FALSE)
g3 = sample_gnp(n=1000, p=0.01, directed=FALSE)
g4 = sample_gnp(n=1000, p=0.05, directed=FALSE)
g5 = sample_gnp(n=1000, p=0.1, directed=FALSE)
```

[15]:
```
# Plot the degree distributions as well as their mean and variance
for(g in list(g1, g2, g3, g4, g5)) {
    plot(seq_along(degree.distribution(g)) - 1, degree.
 ↪distribution(g),main="Degree distribution of the␣
 ↪network",xlab="Degree",ylab="Frequency")
    dist = degree(g)
    mean.result = mean(dist)
    print(mean.result)
    variance.result = var(dist)
    print(variance.result)
}
```

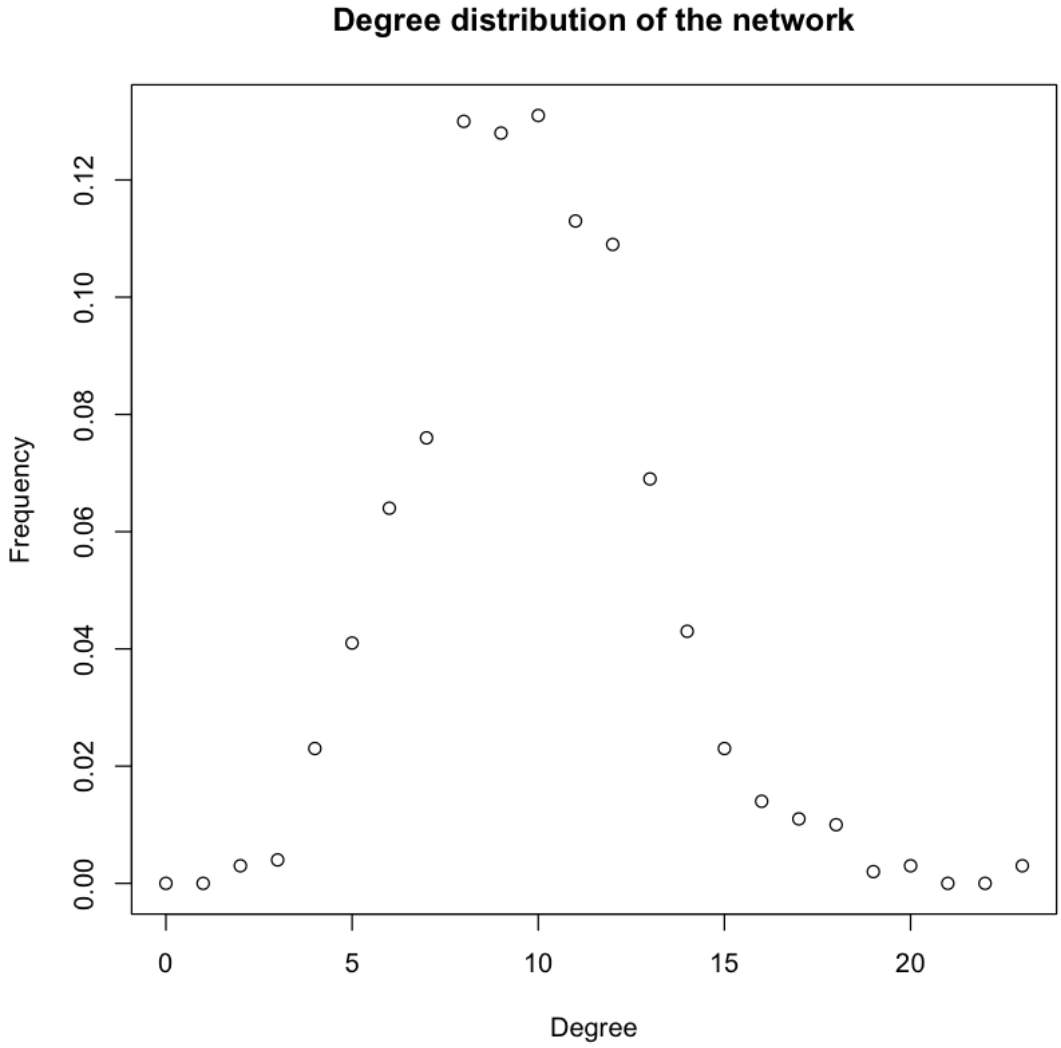
```
[1] 2.996
[1] 2.832817
```



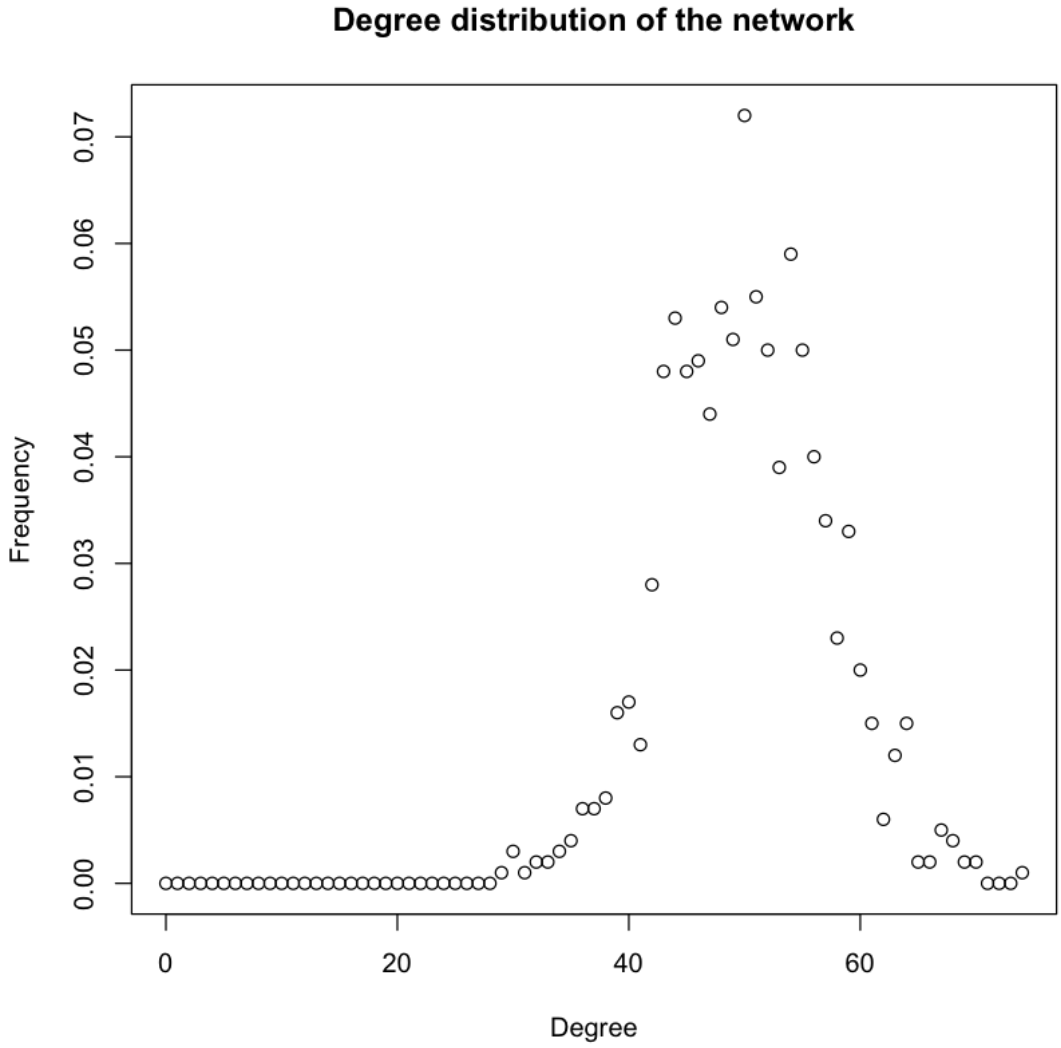

```
[1] 4.018
[1] 4.251928
```



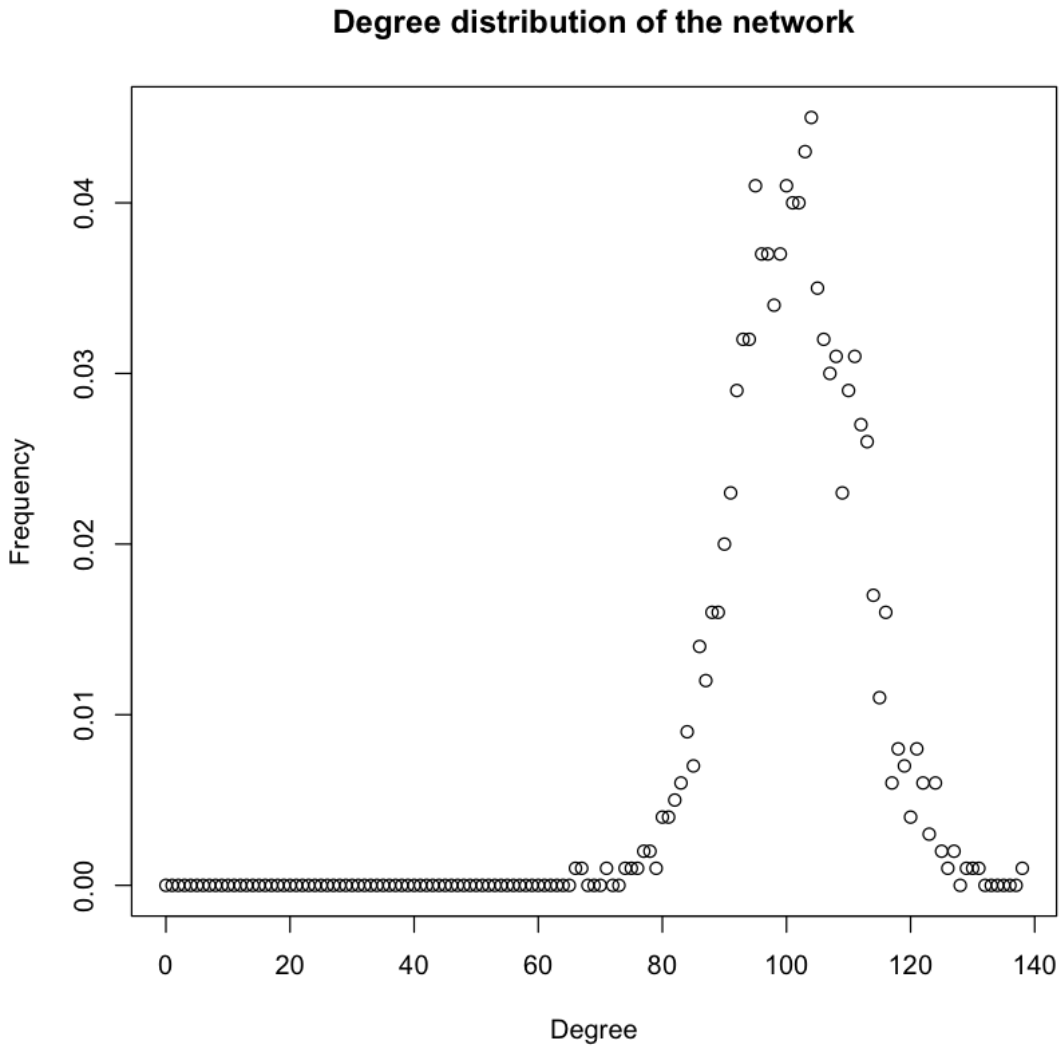

```
[1] 9.886
[1] 9.678683
```




```
[1] 50.21
[1] 47.49339
```




```
[1] 101.42
[1] 96.30591
```



Problem 1.1b: For each p and n = 1000, answer the following questions: Are all random realizations of the ER network connected? Numerically estimate the probability that a generated network is connected. For one instance of the networks with that p, find the giant connected component (GCC) if not connected. What is the diameter of the GCC? Hint Useful function(s): is_connected , clusters , diameter

[30]:
```
# Numerically estimate the probability that a generated network is connected
# Create num_trials realizations of each graph to test
probs = c(0.003,0.004,0.01,0.05,0.1)
num_trials = 1000
for(pr in probs){
    count_connected <- 0
    # try num_trials
```

```
    for(i in seq(1,num_trials,1)){
        g <- sample_gnp(n=1000, p=pr, directed=FALSE)
        if (is.connected(g)){
            count_connected <- count_connected + 1
        }
    }
    print(sprintf("Probability of each connection: %5.3f",pr))
    print(sprintf("Estimated probability network is connected: %5.
↪3f",count_connected/num_trials))

    g_components <- clusters(g)
    # which is the largest component
    ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
    gcc <- induced.subgraph(g, which(g_components$membership == ix))

    print(sprintf("Connected: %s", is.connected(g)))
    print(sprintf("GCC Diameter: %5.3f",diameter(gcc)))

}
```

```
[1] "Probability of each connection: 0.003"
[1] "Estimated probability network is connected: 0.000"
[1] "Connected: FALSE"
[1] "GCC Diameter: 18.000"
[1] "Probability of each connection: 0.004"
[1] "Estimated probability network is connected: 0.000"
[1] "Connected: FALSE"
[1] "GCC Diameter: 10.000"
[1] "Probability of each connection: 0.010"
[1] "Estimated probability network is connected: 0.957"
[1] "Connected: TRUE"
[1] "GCC Diameter: 6.000"
[1] "Probability of each connection: 0.050"
[1] "Estimated probability network is connected: 1.000"
[1] "Connected: TRUE"
[1] "GCC Diameter: 3.000"
[1] "Probability of each connection: 0.100"
[1] "Estimated probability network is connected: 1.000"
[1] "Connected: TRUE"
[1] "GCC Diameter: 3.000"
```

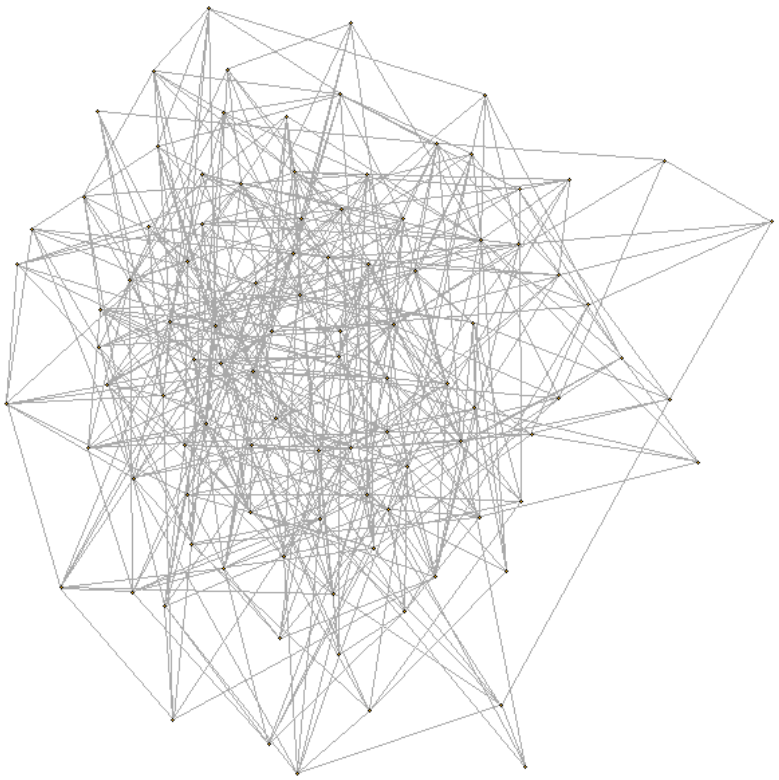
[18]:
```
temp <- sample_gnp(n=100, p=0.1, directed=FALSE)
```

[19]:
```
is.connected(temp)
diameter(temp)
```

TRUE

4

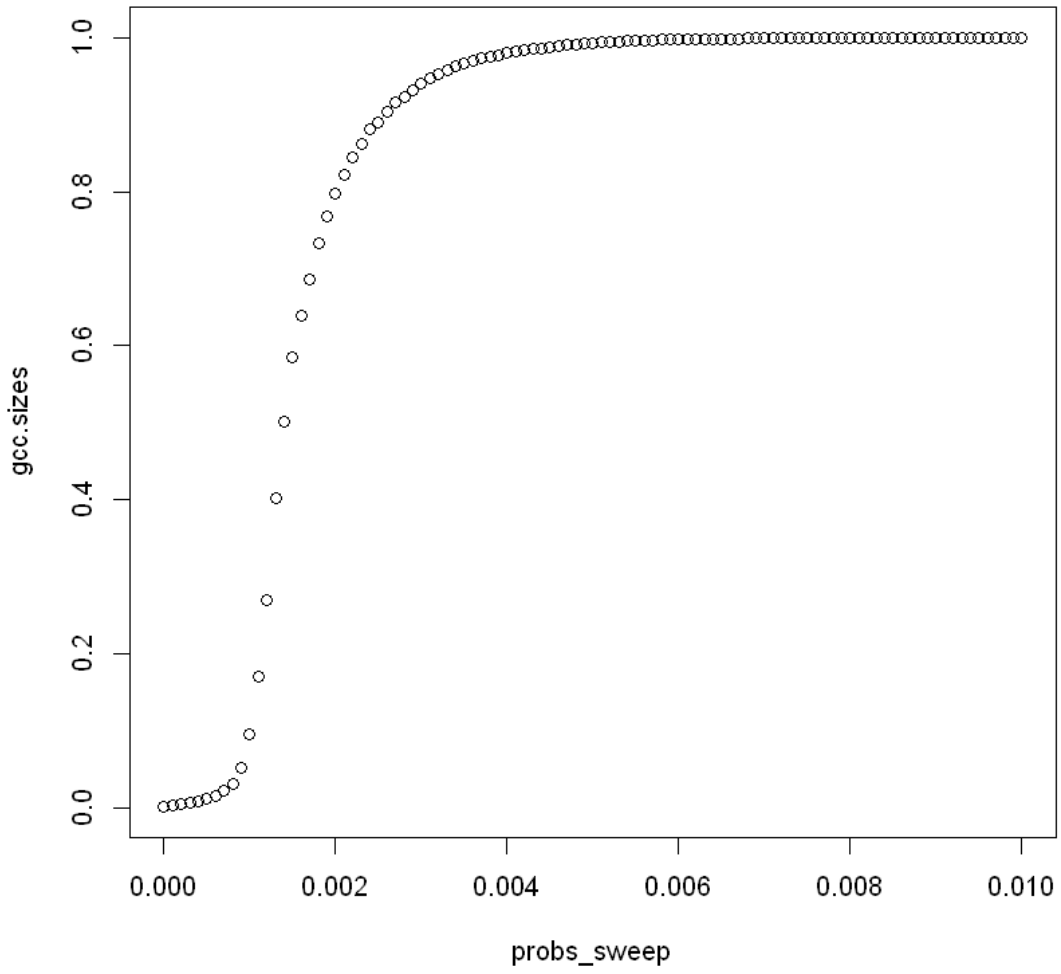
[20]:
```
plot(temp, vertex.size=1, vertex.label=NA)
```



Problem 1.1c: It turns out that the normalized GCC size (i.e., the size of the GCC as a fraction of the total network size) is a highly nonlinear function of p, with interesting properties occurring for values where p = O(1 n) and p = O(ln n n). For n = 1000, sweep over values of p from 0 to a pmax that makes the network almost surely connected and create 100 random networks for each p. pmax should be roughly determined by yourself. Then scatter plot the normalized GCC sizes vs p. Plot a line of the average normalized GCC sizes for each p along with the scatter plot. i. Empirically estimate the value of p where a giant connected component starts to emerge (define your criterion of "emergence")? Do they match with theoretical values mentioned or derived in

lectures? ii. Empirically estimate the value of p where the giant connected component takes up over 99% of the nodes in almost every experiment.

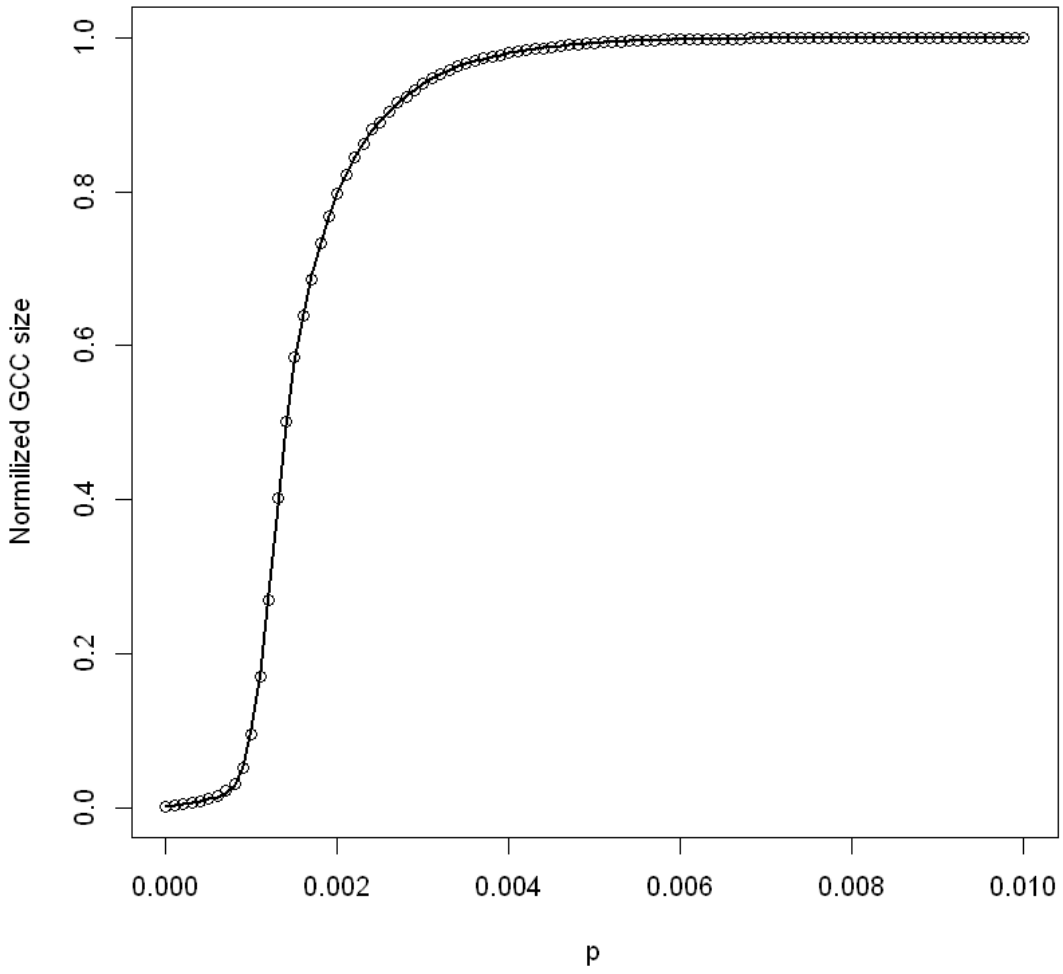
[21]:
```
# 1/n = 0.001 =, ln(n)/n = 0.006
# So sweep p from 0.0000 to 0.0100 in increments of 0.0001
# Normalized GCC is diameter / n

p_min = 0
p_max = 0.01
p_inc = 0.0001
num_trials = 100

# This array includes all the probabilities we are sweeping
probs_sweep = seq(p_min, p_max, p_inc)

# Save all the mean gcc sizes here
gcc.sizes = vector(mode="numeric",length=length(probs_sweep))

i=0
for(pr in probs_sweep){
    # Advance the position in the gcc.sizes vector
    i=i+1
    # Accumulate sum here
    sum = 0
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=1000, p=pr, directed=FALSE)
        # Get the GCC
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc)/1000)
    }
    gcc.sizes[i] = sum / num_trials
}

plot(probs_sweep, gcc.sizes)
```



[28]:
```
lo1 <- smooth.spline(probs_sweep, gcc.sizes, spar=0.3)
line1 <- predict(lo1)



plot(probs_sweep, gcc.sizes,
     xlab="p",
     ylab="Normilized GCC size",
)

lines(predict(lo1), lwd=2)
```

```r
# save plot
png(file="plots/1_1_c.png", width=600, height=450)

lo1 <- smooth.spline(probs_sweep, gcc.sizes, spar=0.3)
line1 <- predict(lo1)



plot(probs_sweep, gcc.sizes,
     xlab="p",
     ylab="Normilized GCC size",
)

lines(predict(lo1), lwd=2)


dev.off()
```

png: 2



Problem 1.1d:

i. Define the average degree of nodes c = n × p = 0.5. Sweep over the number of nodes, n, ranging from 100 to 10000. Plot the expected size of the GCC of ER networks with n nodes and edge-formation probabilities p = c/n, as a function of n. What trend is observed?

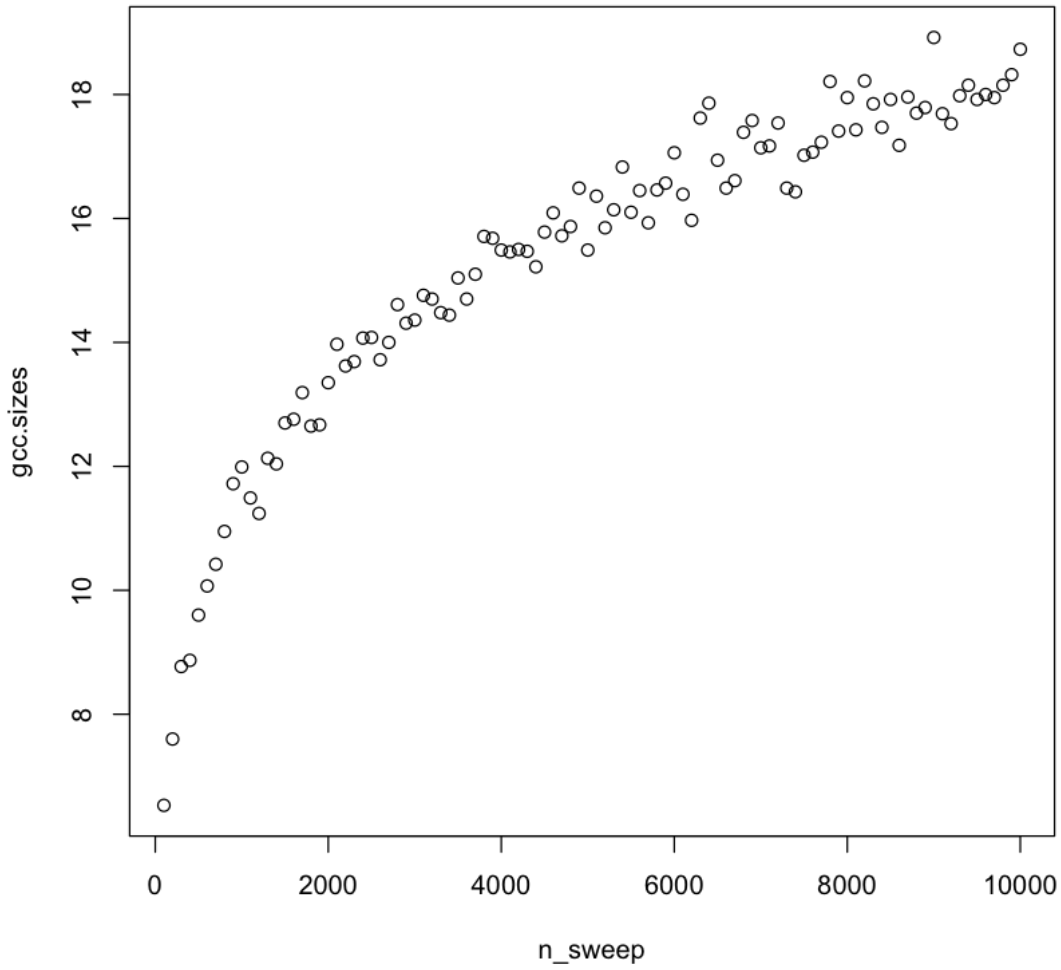
ii. Repeat the same for c = 1.

iii. Repeat the same for values of c = 1.1, 1.2, 1.3, and show the results for these three values in a single plot.

iv. What is the relation between the expected GCC size and n in each case?

[37]:
```
# Sweep over n = 100 to 10000
```

```
# Plot expected (average over num_trials) size of the GCC of gs with n nodes␣
 ↪and p=c/n as a function of n

c = 0.5
n_sweep = seq(100,10000,100) # Start at 100, end at 1000, increments of 100
gcc.sizes = vector(mode="numeric",length=length(n_sweep))
num_trials = 100

i=0
for(n in n_sweep){
    # Advance the position in the gcc.sizes vector
    i=i+1
    # Accumulate sum here
    sum = 0
    p=c/n
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=n, p=p, directed=FALSE)
        # Get the GCC
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc))
    }
    gcc.sizes[i] = sum / num_trials
}

plot(n_sweep, gcc.sizes)
```

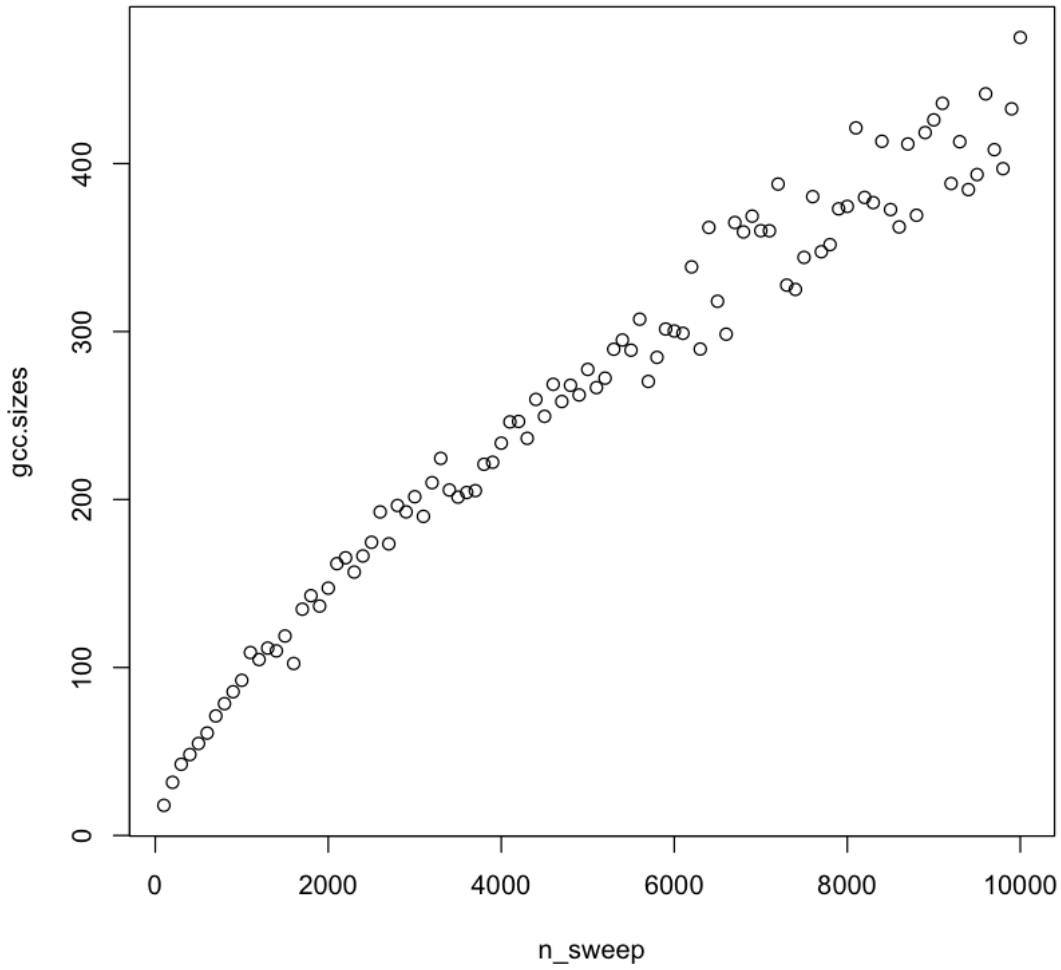


[31]:
```
g <- sample_gnp(n=1000, p=0.01, directed=FALSE)
diameter(g)
```

5

[38]:
```
c = 1
n_sweep = seq(100,10000,100) # Start at 100, end at 1000, increments of 100
gcc.sizes = vector(mode="numeric",length=length(n_sweep))
num_trials = 100

i=0
for(n in n_sweep){
    # Advance the position in the gcc.sizes vector
```

```
    i=i+1
    # Accumulate sum here
    sum = 0
    p=c/n
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=n, p=p, directed=FALSE)
        # Get the GCC
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc))
    }
    gcc.sizes[i] = sum / num_trials
}

plot(n_sweep, gcc.sizes)
```

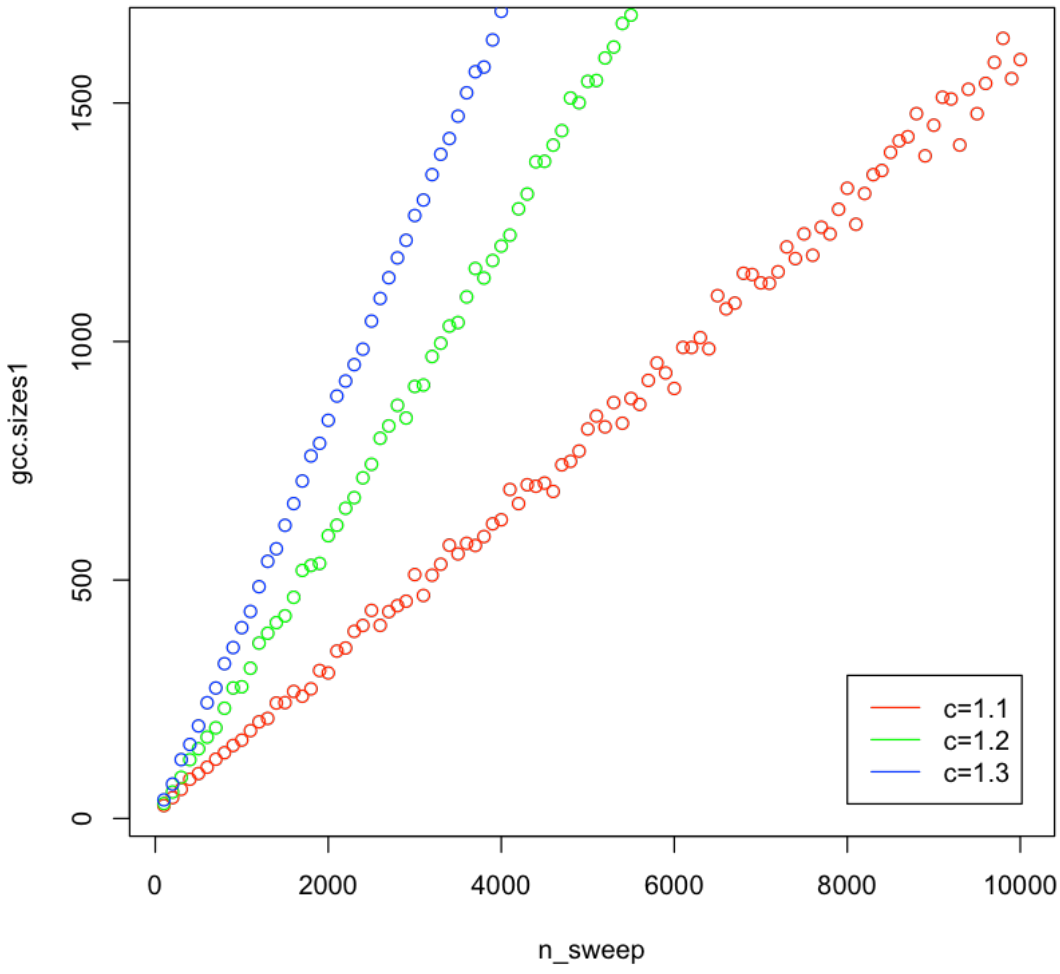


[42]:
```r
# excuse my 3 for loops please i got scared
n_sweep = seq(100,10000,100) # Start at 100, end at 1000, increments of 100
num_trials = 100

gcc.sizes1 = vector(mode="numeric",length=length(n_sweep))
c1 = 1.1
i=0
for(n in n_sweep){
    # Advance the position in the gcc.sizes vector
    i=i+1
    # Accumulate sum here
    sum = 0
    p=c1/n
```

```
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=n, p=p, directed=FALSE)
        # Get the GCC
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc))
    }
    gcc.sizes1[i] = sum / num_trials
}

gcc.sizes2 = vector(mode="numeric",length=length(n_sweep))
c2 = 1.2
i=0
for(n in n_sweep){
    # Advance the position in the gcc.sizes vector
    i=i+1
    # Accumulate sum here
    sum = 0
    p=c2/n
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=n, p=p, directed=FALSE)
        # Get the GCC
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc))
    }
    gcc.sizes2[i] = sum / num_trials
}

gcc.sizes3 = vector(mode="numeric",length=length(n_sweep))
c3=1.3
i=0
for(n in n_sweep){
    # Advance the position in the gcc.sizes vector
    i=i+1
    # Accumulate sum here
    sum = 0
    p=c3/n
    for(trial in seq(1,num_trials,1)){
        g <- sample_gnp(n=n, p=p, directed=FALSE)
        # Get the GCC
```

```
        g_components <- clusters(g)
            # which is the largest component
        ix <- which.max(g_components$csize) # like np.argmax(...)
            # get the subgraph correspondent to just the giant component
        gcc <- induced.subgraph(g, which(g_components$membership == ix))
        sum = sum + (vcount(gcc))
    }
    gcc.sizes3[i] = sum / num_trials
}
```

[48]:
```
plot(n_sweep, gcc.sizes1, col='red')
points(n_sweep,gcc.sizes2, col='green')
points(n_sweep,gcc.sizes3, col='blue')
legend(8000,300,legend=c("c=1.1","c=1.2","c=1.
↪3"),col=c('red','green','blue'),lty=1:1)
```



[ ]:

1_2

April 15, 2021

1 Import libs

[2]:
```
library('igraph')
library('Matrix')
library('pracma')
```

```
Warning message:
"package 'igraph' was built under R version 3.6.3"
Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union

Warning message:
"package 'Matrix' was built under R version 3.6.3"Warning message:
"package 'pracma' was built under R version 3.6.3"
Attaching package: 'pracma'

The following objects are masked from 'package:Matrix':

    expm, lu, tril, triu
```

2 P 1.2 - Create networks using preferential attachment model

a) Create an undirected network with $n = 1000$ nodes, with preferential attachment model, where each new node attaches to $m = 1$ old nodes. Is such a network always connected?

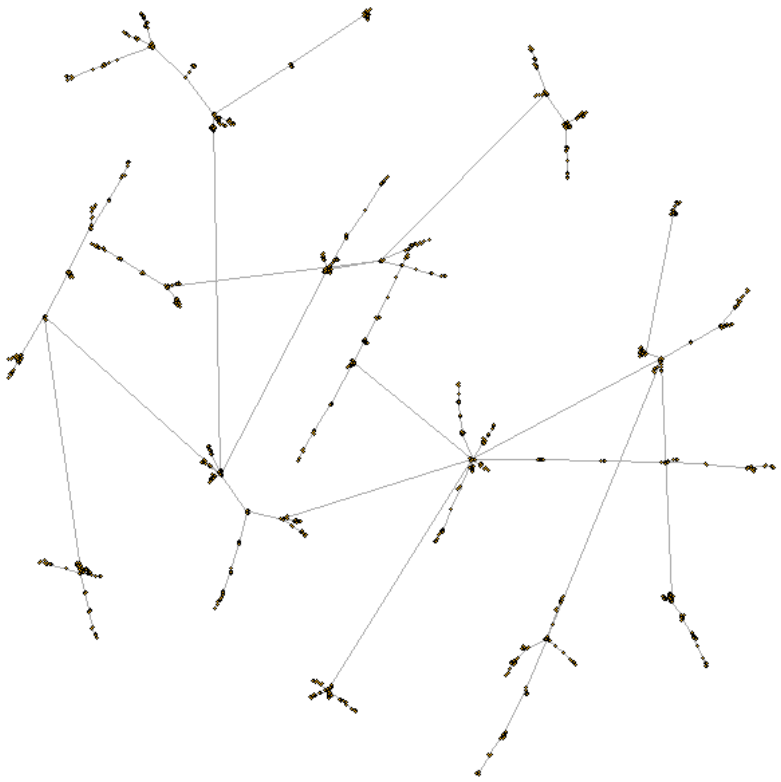
[2]:
```
# g1 = barabasi.game(1000, m=1, directed=F)
g_1m_1000n <- sample_pa(1000, m=1, directed=FALSE)
is.connected(g_1m_1000n)
```

TRUE

Note: yes, PA model is always connected by construction. Can be proven with induction. Also was mentioned in TA discussion section.

b) Use fast greedy method to find the community structure. Measure modularity.

[5]:
```
# let's plot the graph for reference
plot(g_1m_1000n, vertex.size=1, vertex.label=NA)
```



[5]:
```
clusters_1m_1000n <- cluster_fast_greedy(g_1m_1000n)
print(modularity(clusters_1m_1000n))
```

```
[1] 0.9300622
```

We expect several really big cluster with low probability of having several other very small ones. This is because of PA model construction rules: nodes that already have high degree have higher

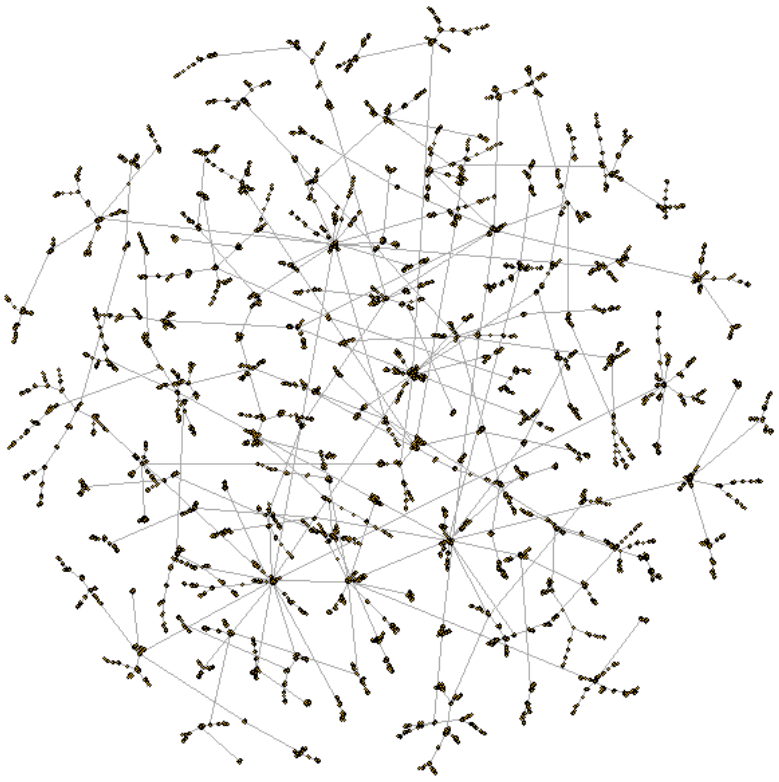
probability to increase degree even more, which results in several nodes getting most of the connections.

We also expect modularity close to 1 because fast greedy algorithm resembles PA model construction so should perfrom really well.

c) Try to generate a larger network with 10000 nodes using the same model. Compute modularity. How is it compared to the smaller network's modularity?

[35]:
```
g_1m_10000n <- sample_pa(10000, m=1, directed=FALSE)
plot(g_1m_10000n, vertex.size=1, vertex.label=NA)
clusters2 <- cluster_fast_greedy(g_1m_10000n)
print(modularity(clusters2))
```

```
[1] 0.9784681
```



We can expect even more clusters whith bigger size (by construction). They will be even easier to notice (since they are larger).

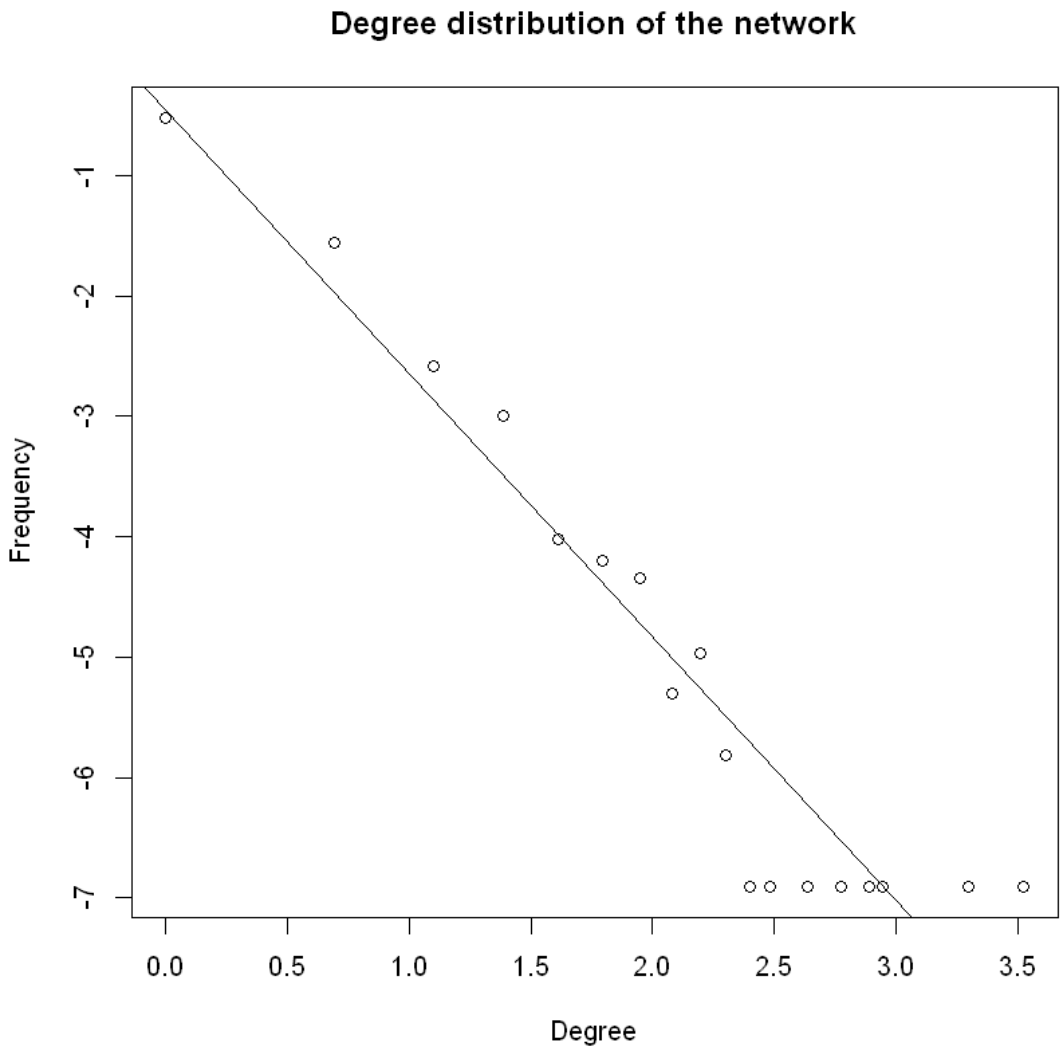
d) Plot the degree distribution in a log-log scale for both n = 1000; 10000, then estimate the slope of the plot using linear regression.

Let's consider $n = 1000$ first.

[50]:
```
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_1m_1000n)) - 1
distribution <- degree.distribution(g_1m_1000n)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
```

[51]:
```
# train linear regression model on the data
model = lm(Y ~ X)
```

[52]:
```
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
```



[53]:
```
# save plot
png(file="plots/1_2_d_1m_1000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()
```

png: 2

[15]:
```r
# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

```
Call:
lm(formula = Y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-1.20183 -0.30582 -0.02107  0.35762  1.26512

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.4639     0.3808  -1.218    0.241
X            -2.1861     0.1661 -13.158 5.36e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6237 on 16 degrees of freedom
Multiple R-squared:  0.9154,        Adjusted R-squared:  0.9101
F-statistic: 173.1 on 1 and 16 DF,  p-value: 5.36e-10
```

As we can see, the result is a little bit less than 3. The reason behind deviation from 3 is that Preferential attachment model follows power law degree distribution only in stable case, i.e. when there is infinitely many nodes $t \to +\infty$.

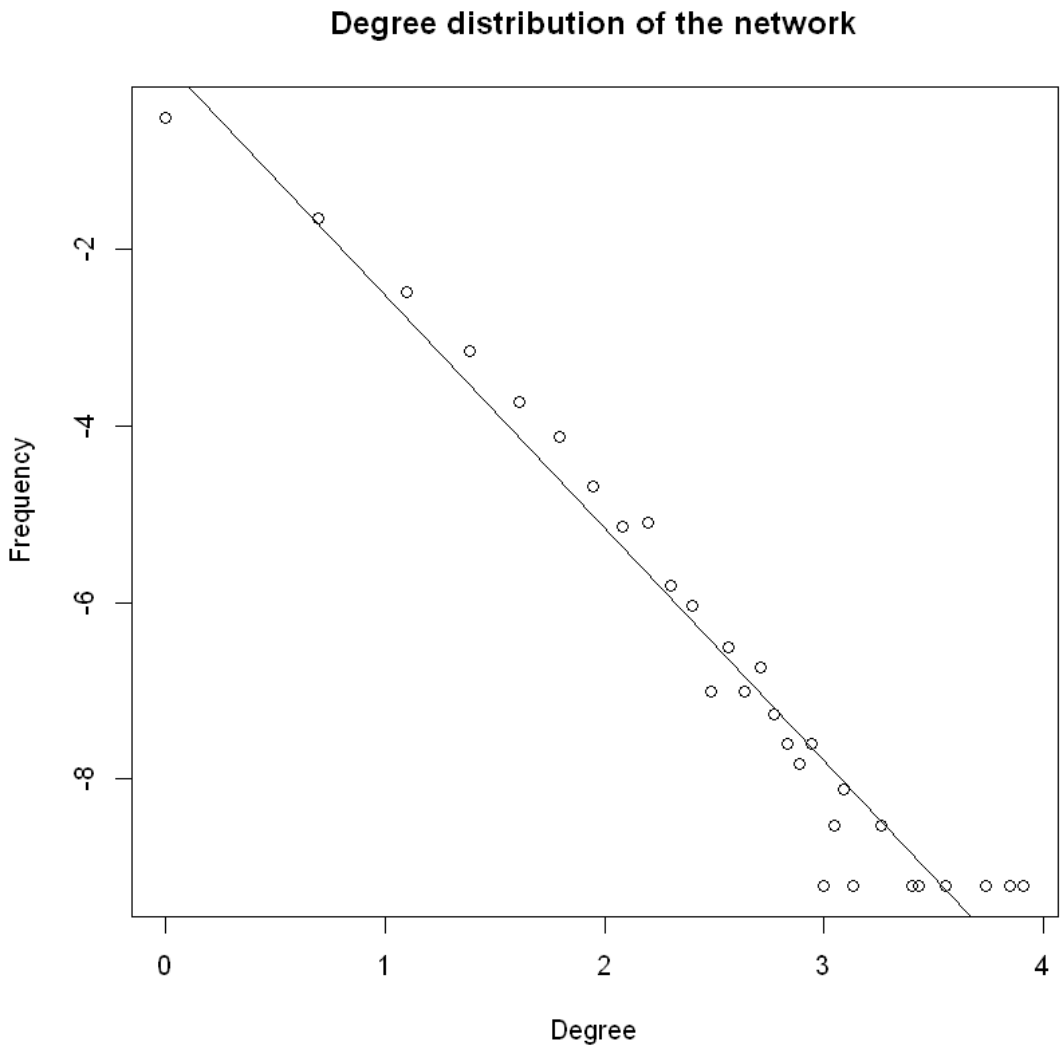
Now let's consider $n = 10000$. We can expect coefficient closer to 3 as the number of nodes increased.

[45]:
```r
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_1m_10000n)) - 1
distribution <- degree.distribution(g_1m_10000n)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
```

```
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
```

[46]:
```
# train linear regression model on the data
model = lm(Y ~ X)
```

[47]:
```
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
```



[49]:
```
# save plot
png(file="plots/1_2_d_1m_10000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()
```

png: 2

[40]:
```
# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

```
Call:
lm(formula = Y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-1.43378 -0.27222  0.05488  0.32421  0.98113

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.1188     0.2881   0.412    0.683
X            -2.6355     0.1060 -24.856   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5318 on 28 degrees of freedom
Multiple R-squared:  0.9566,        Adjusted R-squared:  0.9551
F-statistic: 617.8 on 1 and 28 DF,  p-value: < 2.2e-16
```

As we can see, $b_{10000} \approx 2.63 > b_{1000} \approx 2.18$, which confirms our expectations.

e) In the two networks generated in **d)**, perform the following:

Randomly pick a node i, and then randomly pick a neighbor j of that node. Plot the degree distribution of nodes j that are picked with this process, in the log-log scale. Is the distribution linear in the log-log scale? If so, what is the slope? How does this differ from the node degree distribution?

[30]:
```
create_transition_matrix = function (g){

    # WARNING: make sure your graph is connected (you might input GCC of your␣
 ↪graph)

    vs = V(g)
    n = vcount(g)
    adj = as_adjacency_matrix(g)
    adj[diag(rowSums(adj) == 0)] = 1  # handle if the user is using the␣
 ↪function for networks with isolated nodes by creating self-edges
    z = matrix(rowSums(adj, , 1))

    transition_matrix = adj / repmat(z, 1, n)  # normalize to get probabilities

    return(transition_matrix)
}
```

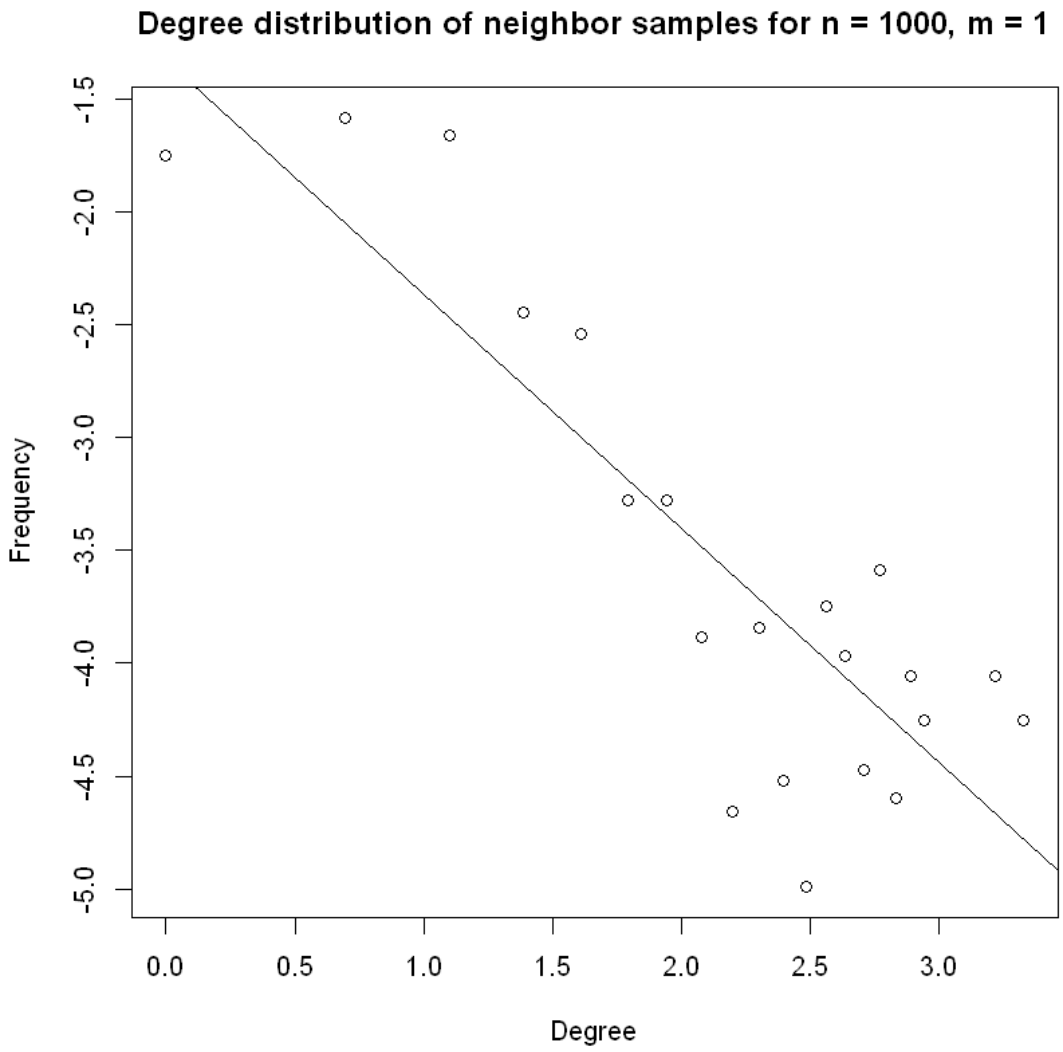
[32]:
```
generate_mask <- function(vector_w_zeros) {
    # function that generates mask, which removes zero values from the vector
    # find which indices we need to remove
    to_remove <- which(vector_w_zeros == 0)
    # initialize mask
    mask <- c(ones(1, length(vector_w_zeros)))
    # mark elements we need to remove in the mask
    for (i in 1:length(vector_w_zeros)) {
        if (is.element(i, to_remove)) {
            mask[i] <- 0
        }
    }
    # convert mask to boolean
    mask <- mask > 0.5
    return(mask)
}
```

[105]:
```
numOfExamples = 10000
numOfGraphs = 1

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(1000, m=1, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 1000, m = 1",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 28


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
     Min       1Q   Median       3Q      Max
-1.08819 -0.34072  0.09347  0.45386  0.80400

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.3328     0.3382  -3.941 0.000876 ***
X[mask]      -1.0342     0.1447  -7.146 8.59e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5486 on 19 degrees of freedom
Multiple R-squared:  0.7288,        Adjusted R-squared:  0.7145
F-statistic: 51.06 on 1 and 19 DF,  p-value: 8.595e-07
```

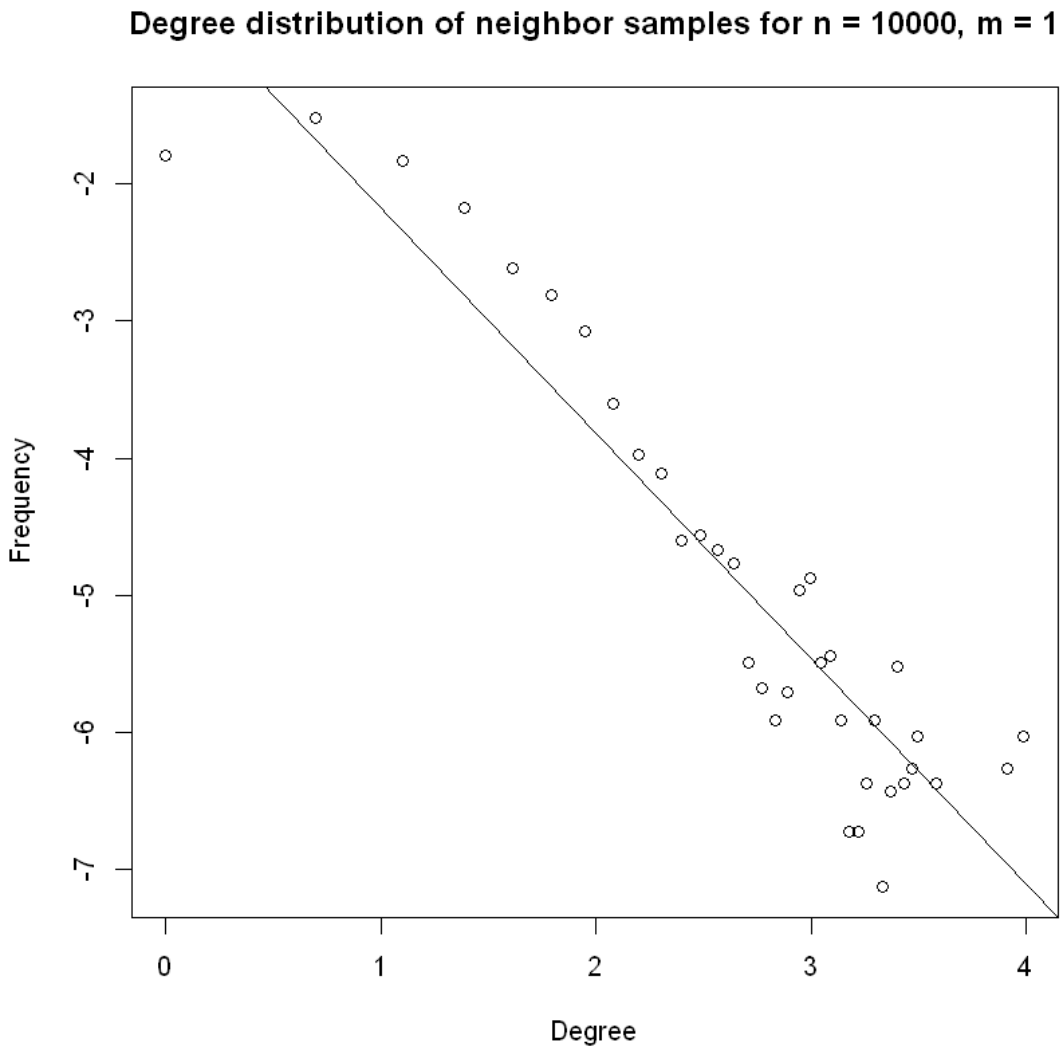


[104]:
```
numOfExamples = 10000
numOfGraphs = 1

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(10000, m=1, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 10000, m = 1",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 54


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
     Min       1Q   Median       3Q      Max
-1.26786 -0.38523  0.06429  0.43328  1.05930

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.5231     0.3045  -1.718   0.0949 .
X[mask]      -1.6467     0.1079 -15.266   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5705 on 34 degrees of freedom
Multiple R-squared:  0.8727,        Adjusted R-squared:  0.8689
F-statistic:   233 on 1 and 34 DF,  p-value: < 2.2e-16
```

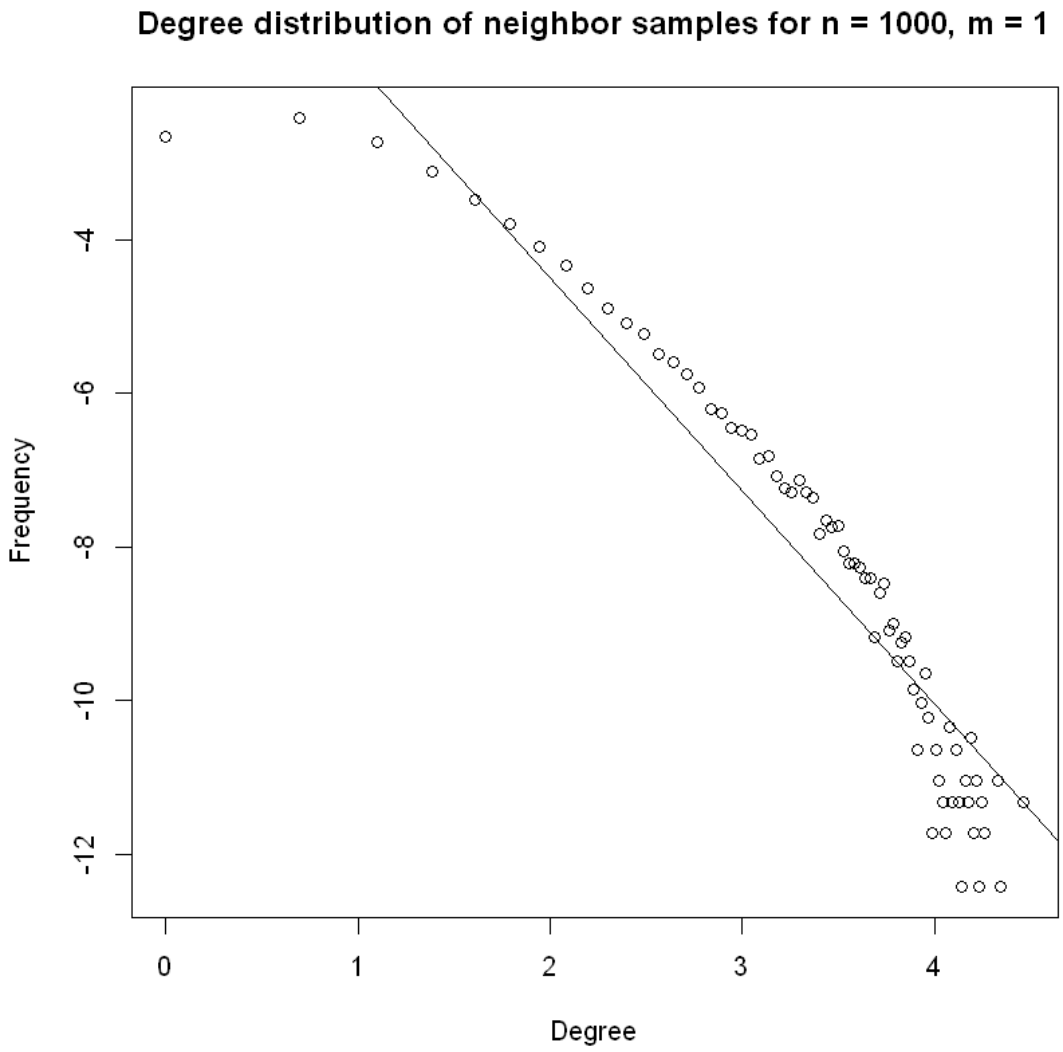


[93]:
```
numOfExamples = 50
numOfGraphs = 5000
g2 <- sample_pa(1000, m=1, directed=FALSE)
v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(1000, m=1, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 1000, m = 1",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 87


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-3.7191 -0.5189  0.2982  0.6694  0.9518

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.0556     0.3971   2.659  0.00966 **
X[mask]      -2.7742     0.1144 -24.249  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8963 on 72 degrees of freedom
Multiple R-squared:  0.8909,        Adjusted R-squared:  0.8894
F-statistic:   588 on 1 and 72 DF,  p-value: < 2.2e-16
```

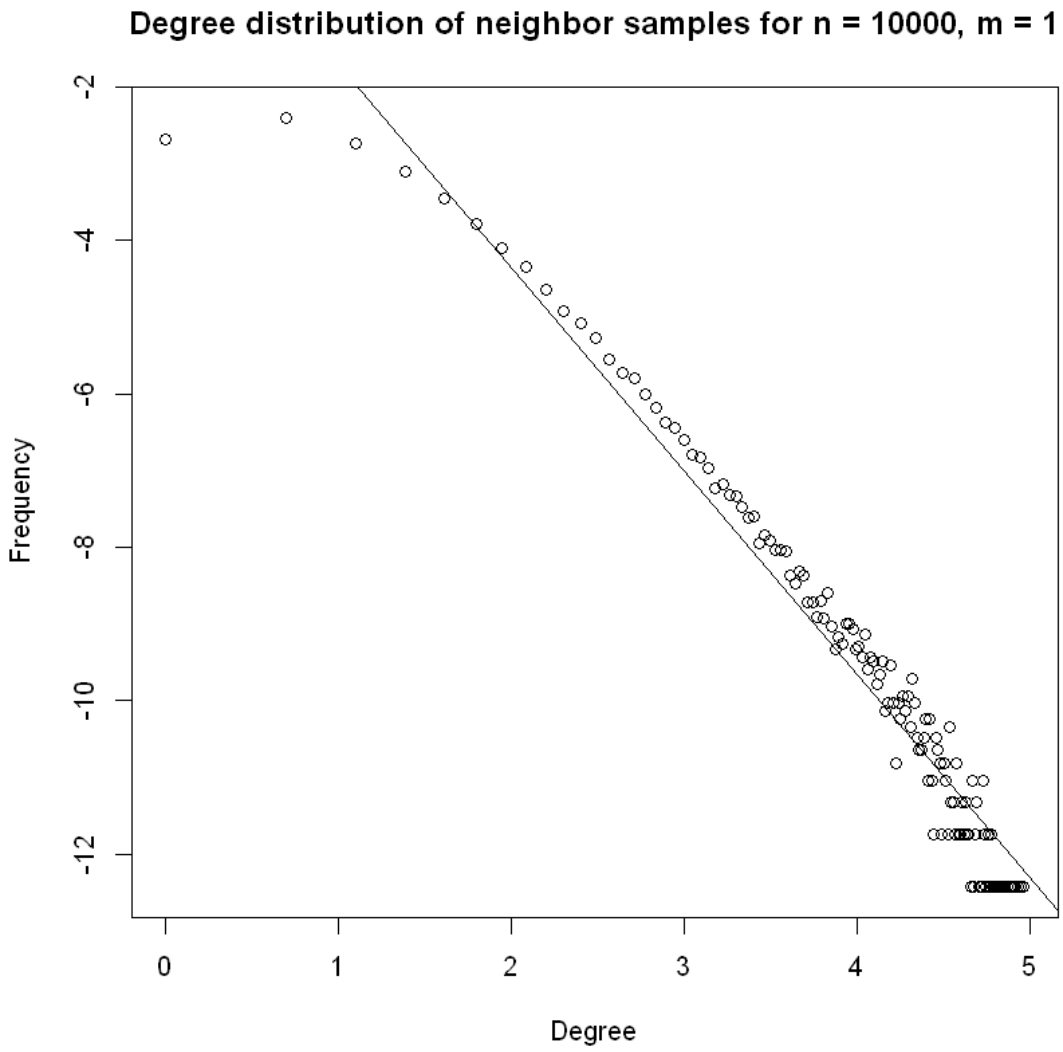


[99]:
```
numOfExamples = 50
numOfGraphs = 5000

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(10000, m=1, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

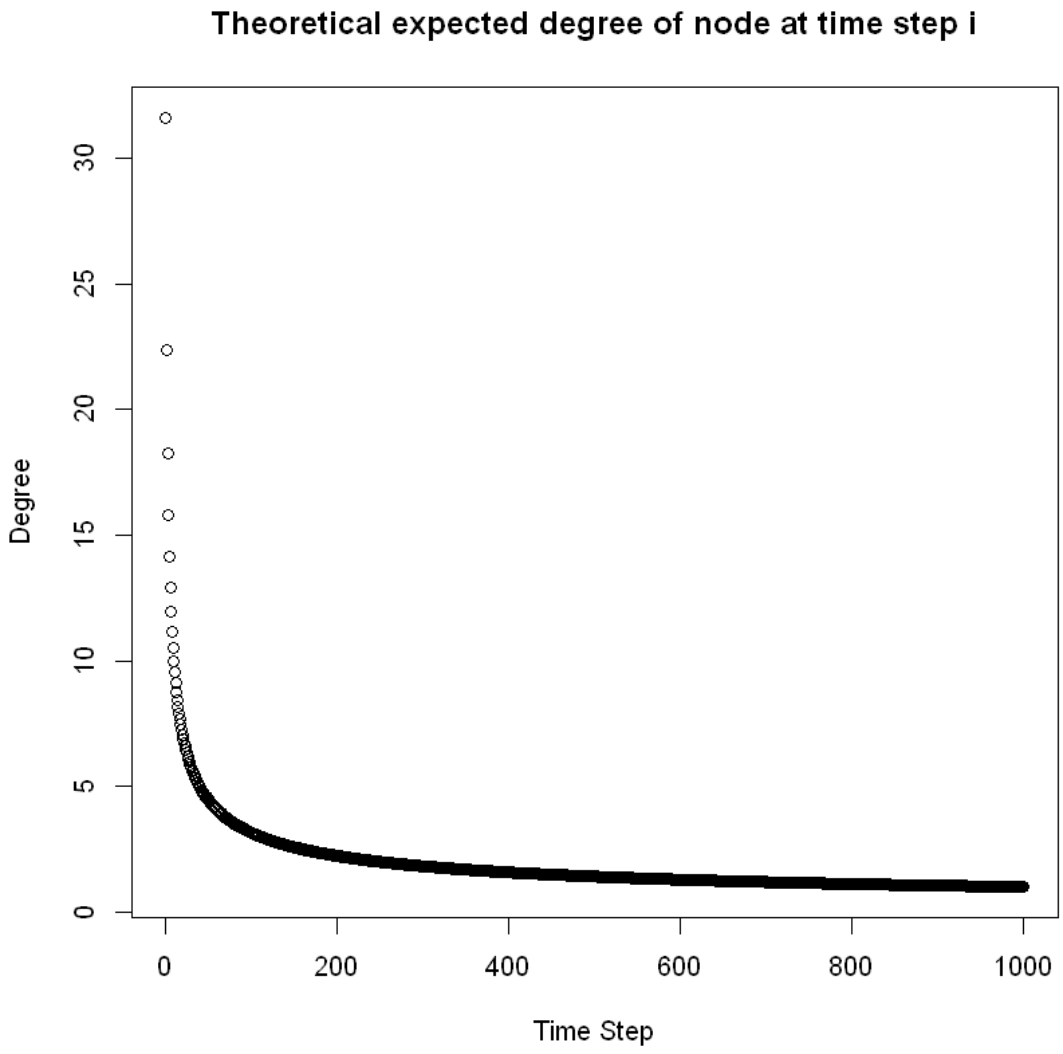
```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 10000, m = 1",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 143


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-3.6413 -0.3461  0.1853  0.3942  0.7870

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.9511     0.2070   4.596 9.85e-06 ***
X[mask]      -2.6542     0.0511 -51.939  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.56 on 134 degrees of freedom
Multiple R-squared:  0.9527,        Adjusted R-squared:  0.9523
F-statistic:  2698 on 1 and 134 DF,  p-value: < 2.2e-16
```



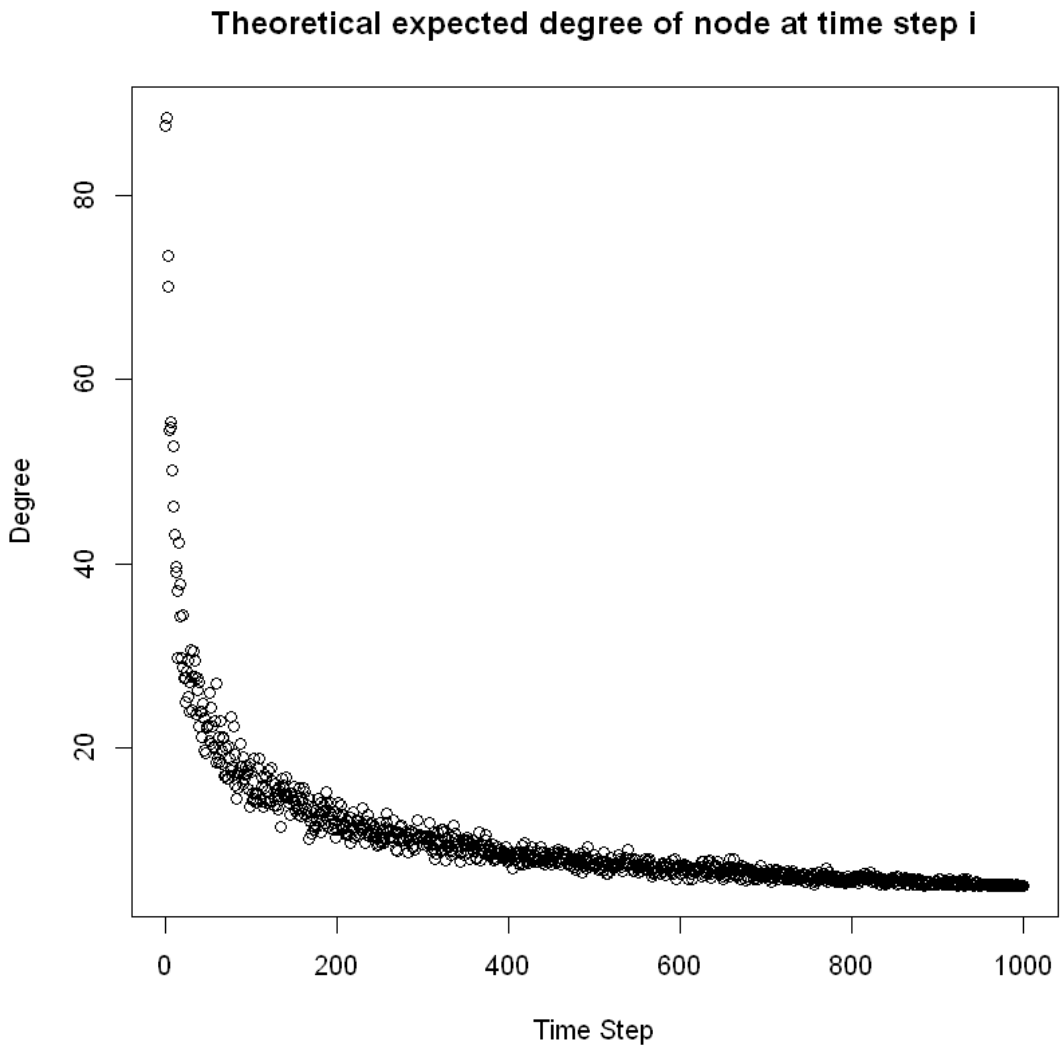
f) Estimate the expected degree of a node that is added at time step i for $1 \leq i \leq 1000$. Show the relationship between the age of nodes and their expected degree through an appropriate plot.

[133]:
```
X = matrix(data = 0, nrow = 1, ncol = 1)
K = matrix(data = 0, nrow = 1, ncol = 1)
for(i in 1:1000){
    X[i] = i
    K[i] = sqrt((1000)/i)
    }
plot(X, K,
     main="Theoretical expected degree of node at time step i",
     xlab="Time Step",
     ylab="Degree",)
```



[134]:
```
empK = matrix(data = 0, nrow = 1000, ncol = 1)
totalEmpK = matrix(data = 0, nrow = 1000, ncol = 1)
g <- sample_pa(1000, m=1, directed=FALSE)
for(j in 1:50){
    g <- sample_pa(1000, m=1, directed=FALSE)
    for(i in 1:1000){

         empK[i] = degree(g, v = i)

        }
    totalEmpK = totalEmpK + empK
}
totalEmpK = totalEmpK/10
```

```
plot(X, totalEmpK,
     main="Theoretical expected degree of node at time step i",
     xlab="Time Step",
     ylab="Degree",)
```



g) Repeat the previous parts for $m = 2$; and $m = 5$. Compare the results of each part for different values of m.

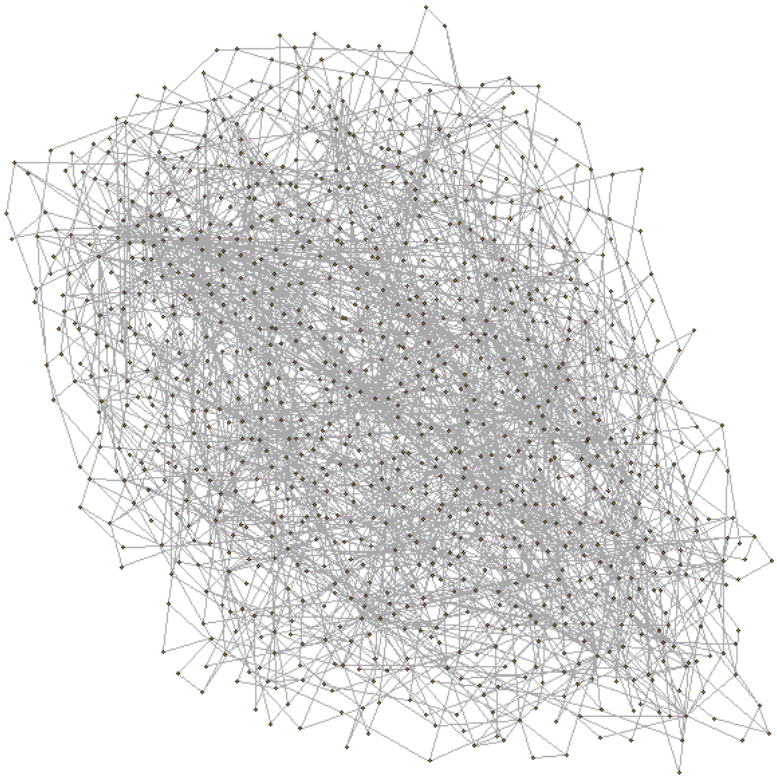
[9]:
```
g_2m_1000n <- sample_pa(1000, m=2, directed=FALSE)
g_5m_1000n <- sample_pa(1000, m=5, directed=FALSE)

is.connected(g_2m_1000n)
is.connected(g_5m_1000n)
```

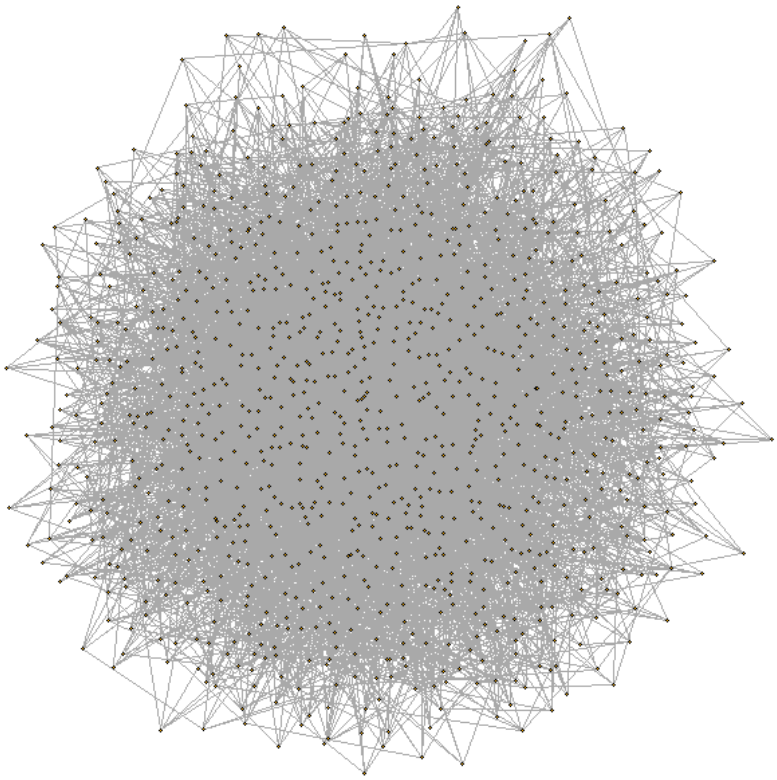
TRUE

TRUE

Obviously, both the $m = 2$ and $m = 5$ graphs are connected.

[10]:
```
# let's plot the graph for reference
plot(g_2m_1000n, vertex.size=1, vertex.label=NA)
plot(g_5m_1000n, vertex.size=1, vertex.label=NA)
```



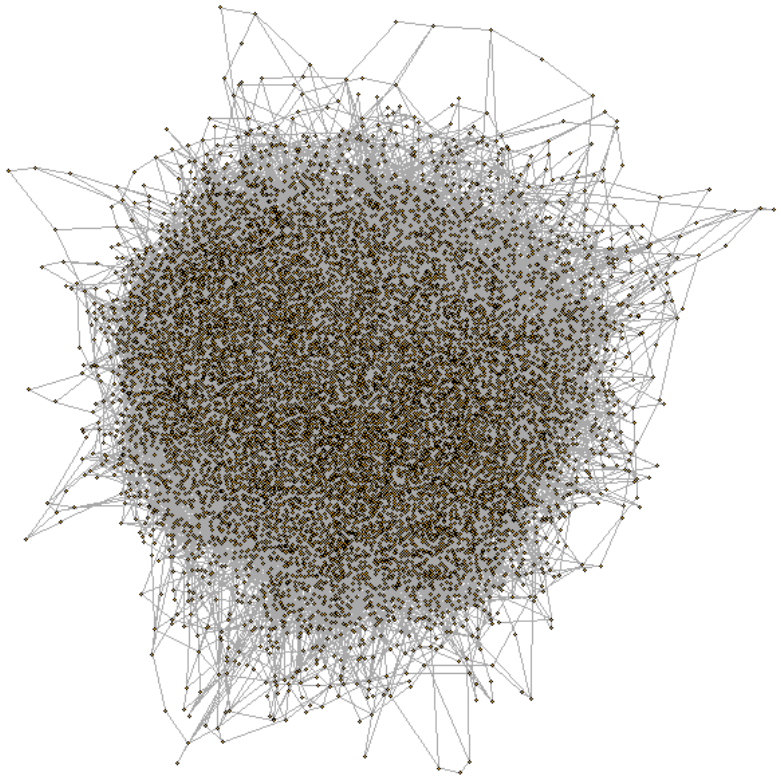


[11]:
```
clusters_2m_1000n <- cluster_fast_greedy(g_2m_1000n)
clusters_5m_1000n <- cluster_fast_greedy(g_5m_1000n)
print(modularity(clusters_2m_1000n))
print(modularity(clusters_5m_1000n))
```

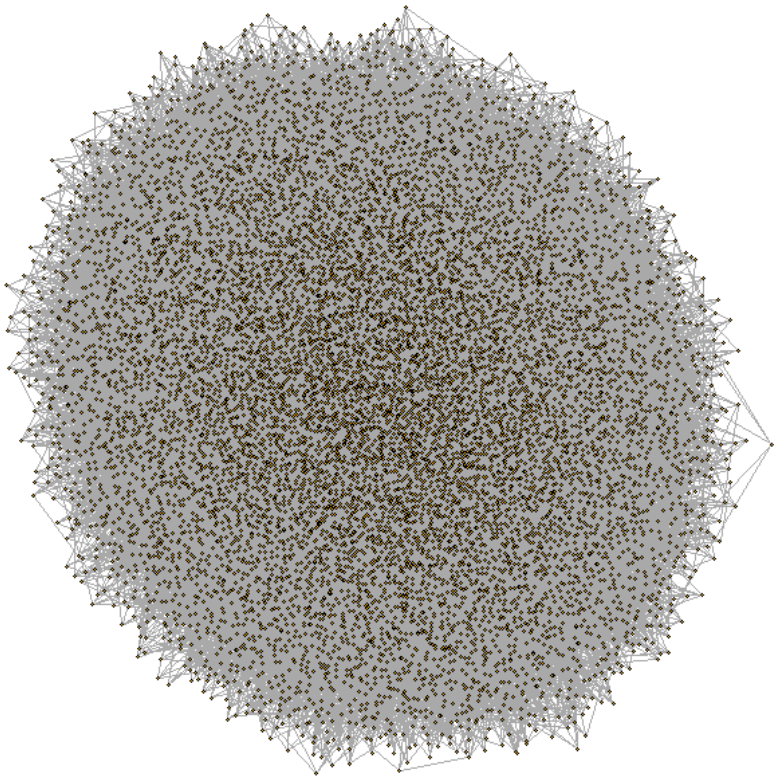
```
[1] 0.5271631
[1] 0.2771637
```

It can be seen that with the increasing m, the network changes its structure. Instead of a collection of big clusters, the network itself becomes one gigantic cluster, where elements are strongly connected (not fat-tailed distribution). This makes harder for the fast greedy algorithm to minimize the number of edges lying in-between clusters. This results in lowered modularity.

[19]:
```r
g_2m_10000n <- sample_pa(10000, m=2, directed=FALSE)
g_5m_10000n <- sample_pa(10000, m=5, directed=FALSE)
# let's plot the graph for reference
plot(g_2m_10000n, vertex.size=1, vertex.label=NA)
plot(g_5m_10000n, vertex.size=1, vertex.label=NA)
# compute modularity
clusters_2m_10000n <- cluster_fast_greedy(g_2m_10000n)
clusters_5m_10000n <- cluster_fast_greedy(g_5m_10000n)
print(modularity(clusters_2m_10000n))
print(modularity(clusters_5m_10000n))
```



```
[1] 0.529787
[1] 0.2728815
```



Modularity remains the same for larger networks. Seemingly, at this point the size does not affect clusterization quality much.

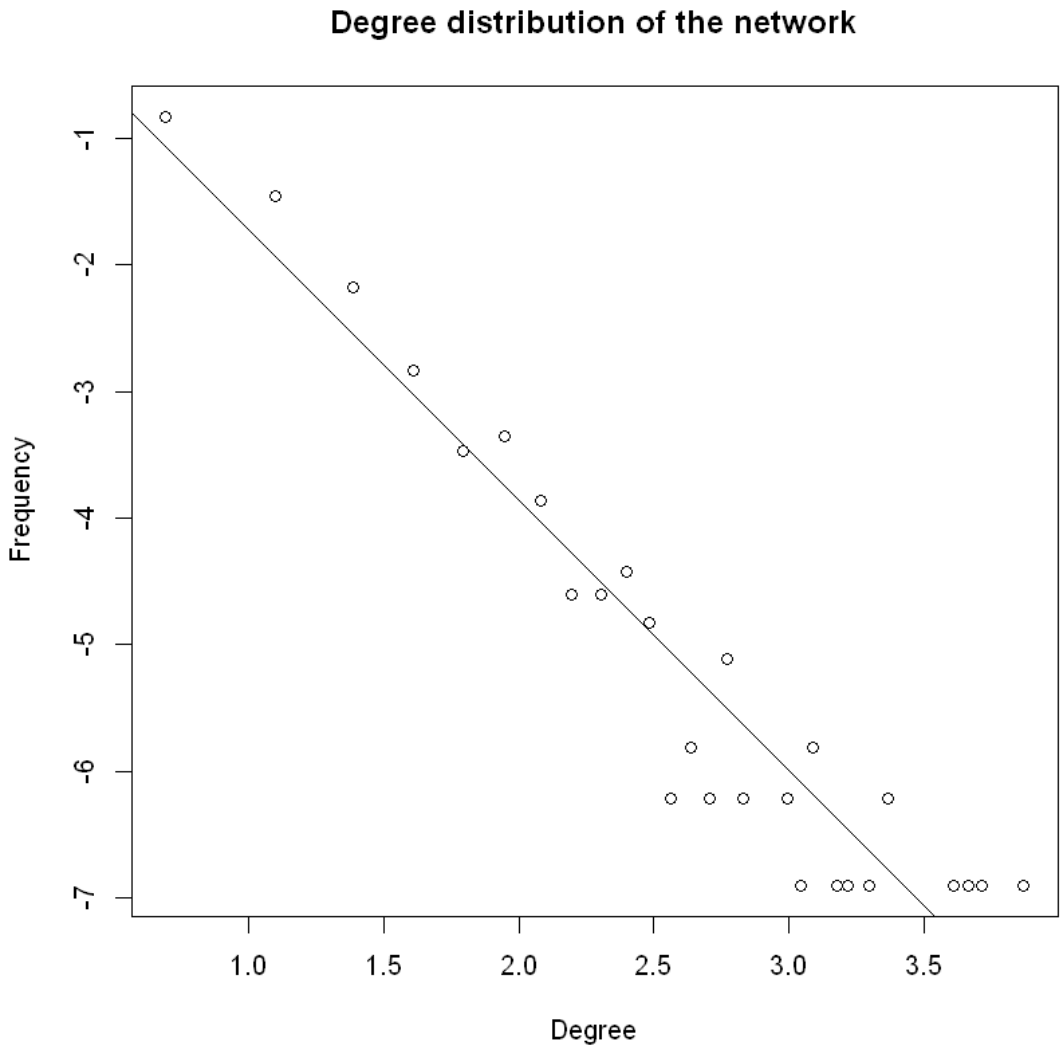
[59]:
```
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_2m_1000n)) - 1
distribution <- degree.distribution(g_2m_1000n)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
```

```
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
# train linear regression model on the data
model = lm(Y ~ X)
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

# save plot
png(file="plots/1_2_g_2m_1000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()

# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

png: 2

```
Call:
lm(formula = Y ~ X)

Residuals:
    Min      1Q  Median      3Q     Max
-1.1504 -0.3870  0.1653  0.3894  0.9430

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)   0.4074     0.3493   1.166    0.254
X            -2.1332     0.1277 -16.701 4.52e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5342 on 25 degrees of freedom
Multiple R-squared:  0.9177,        Adjusted R-squared:  0.9145
F-statistic: 278.9 on 1 and 25 DF,  p-value: 4.524e-15
```



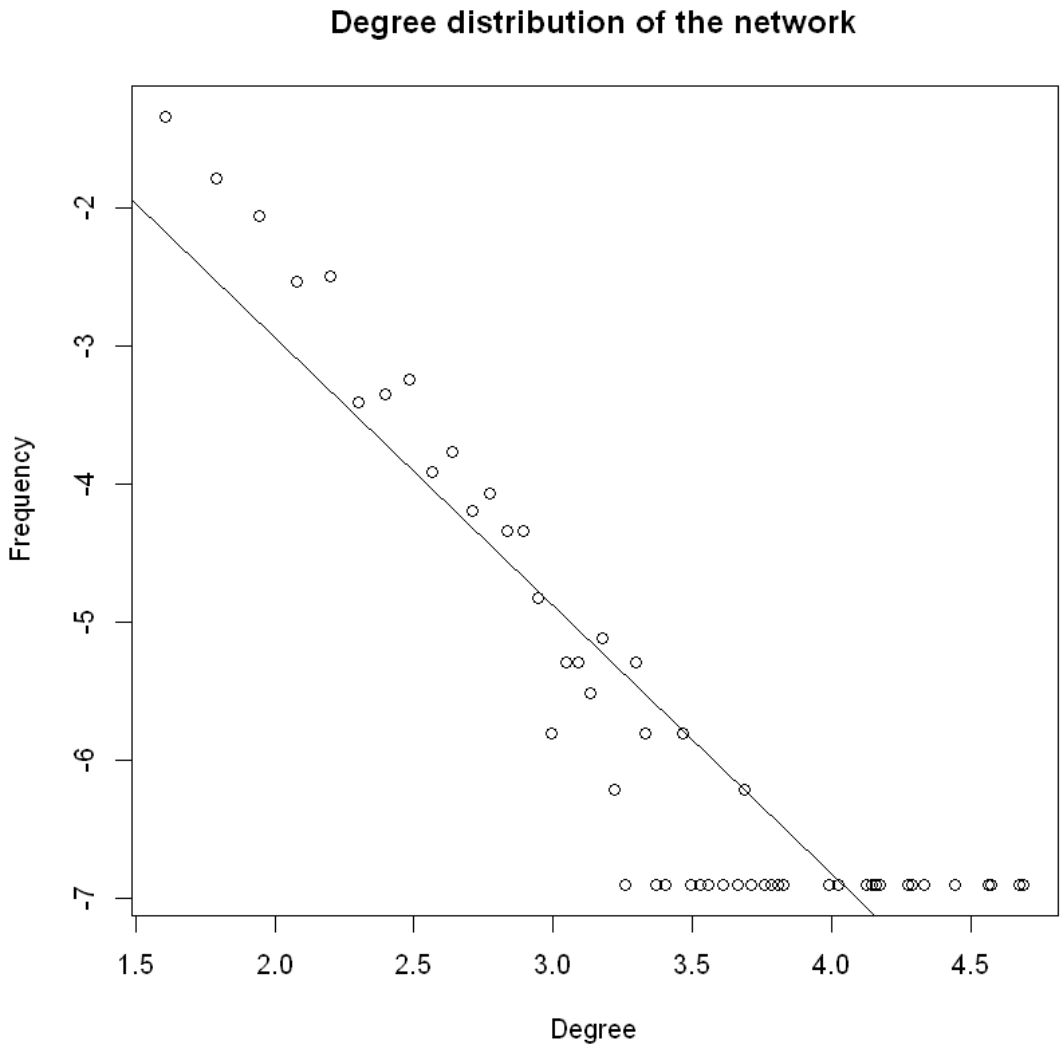
[56]:
```
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_5m_1000n)) - 1
distribution <- degree.distribution(g_5m_1000n)
```

```
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
# train linear regression model on the data
model = lm(Y ~ X)
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

# save plot
png(file="plots/1_2_g_5m_1000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()

# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

png: 2

```
Call:
lm(formula = Y ~ X)
```

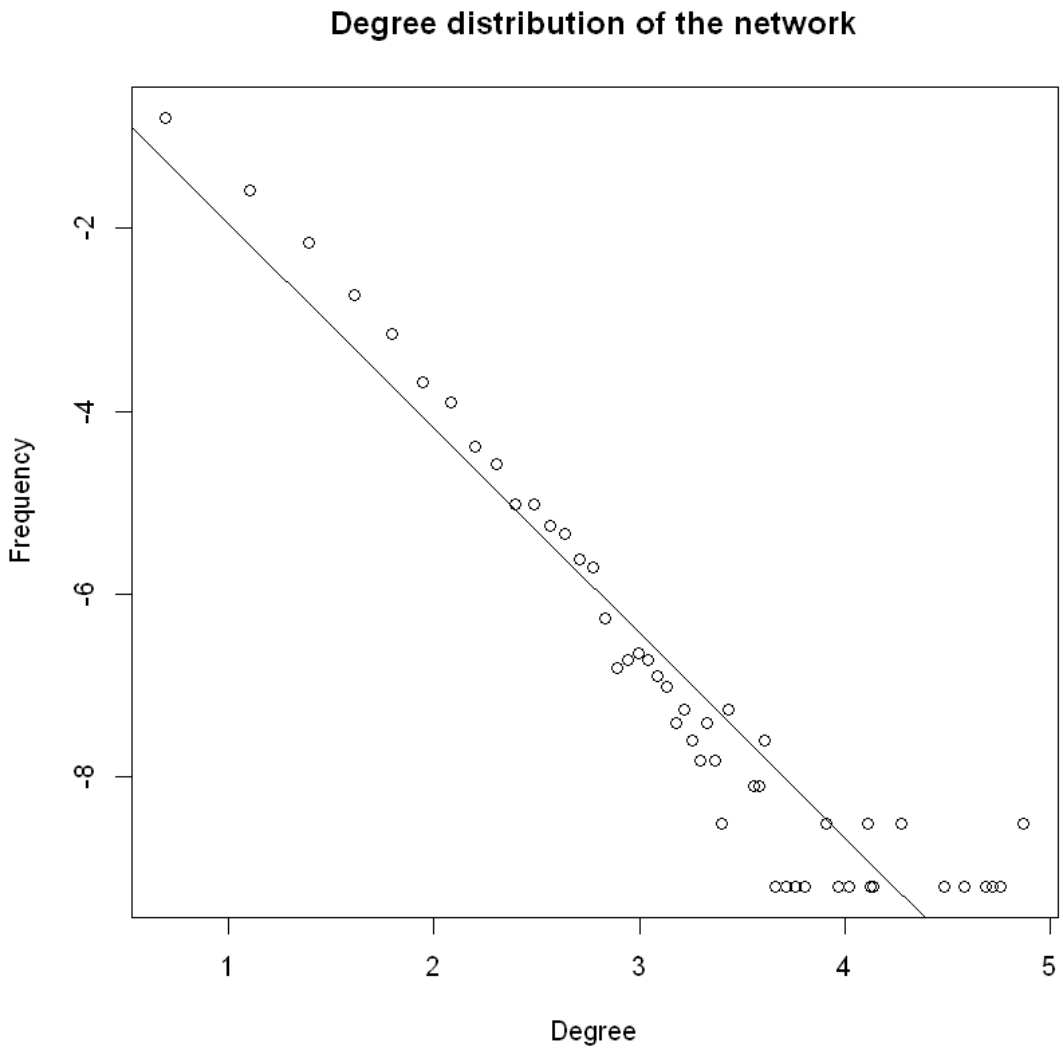
```
Residuals:
    Min      1Q  Median      3Q     Max
-1.5207 -0.4664  0.1197  0.4640  1.2626

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.9400     0.4303   2.184   0.0337 *
X            -1.9419     0.1240 -15.654   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6923 on 50 degrees of freedom
Multiple R-squared:  0.8305,        Adjusted R-squared:  0.8272
F-statistic: 245.1 on 1 and 50 DF,  p-value: < 2.2e-16
```



[57]:
```r
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_2m_10000n)) - 1
distribution <- degree.distribution(g_2m_10000n)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
# train linear regression model on the data
model = lm(Y ~ X)
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

# save plot
png(file="plots/1_2_g_2m_10000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()

# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

png: 2

```
Call:
lm(formula = Y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-1.29758 -0.43686  0.00464  0.44254  2.10942

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.2930     0.3358   0.873    0.387
X            -2.2398     0.1002 -22.353   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6829 on 48 degrees of freedom
Multiple R-squared:  0.9124,        Adjusted R-squared:  0.9105
F-statistic: 499.6 on 1 and 48 DF,  p-value: < 2.2e-16
```



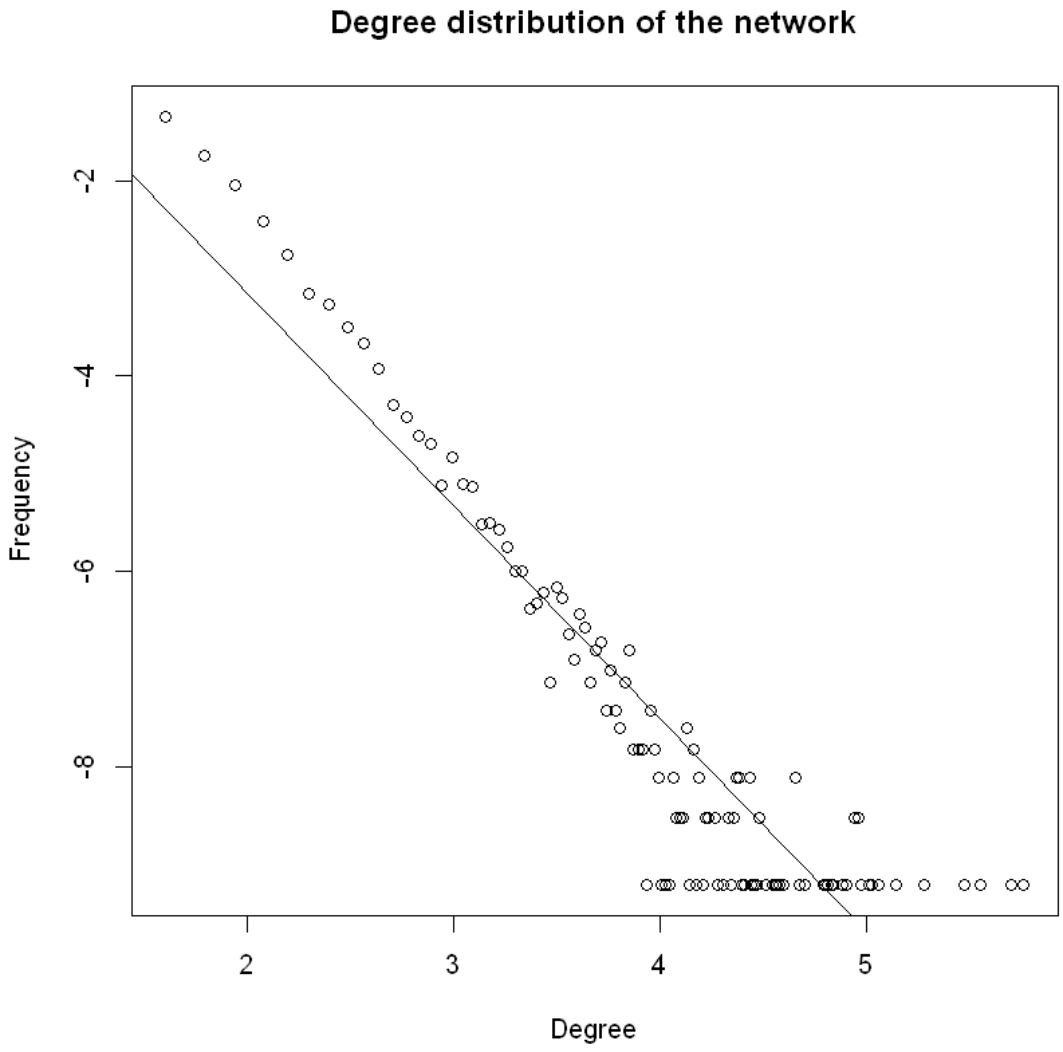
[58]:
```r
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_5m_10000n)) - 1
distribution <- degree.distribution(g_5m_10000n)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
```

```
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
# train linear regression model on the data
model = lm(Y ~ X)
# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

# save plot
png(file="plots/1_2_g_5m_10000n.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()

# print model summary
# coefficient we are looking for is X estimate.
summary(model)
```

png: 2

```
Call:
lm(formula = Y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-1.84214 -0.48879  0.04245  0.39906  2.14833

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.21920    0.34198   3.565 0.000545 ***
X           -2.18408    0.08419 -25.941  < 2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7646 on 107 degrees of freedom
Multiple R-squared:  0.8628,        Adjusted R-squared:  0.8615
F-statistic:   673 on 1 and 107 DF,  p-value: < 2.2e-16
```



Trends we are observing are similar to ones in the question **d)**, in other words, $b(n = 10000) > b(n = 1000)$. However, for any n:

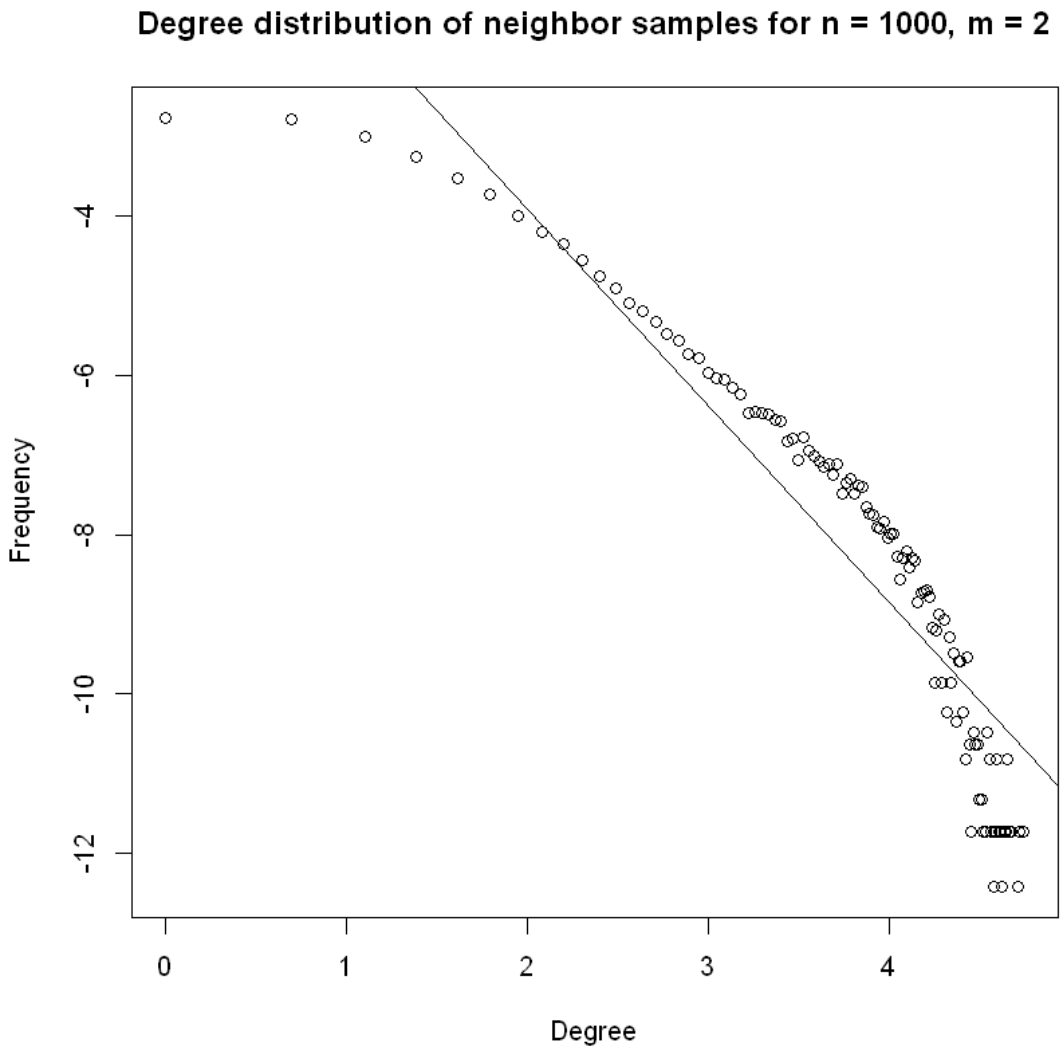
$$b(m = 5) < b(m = 2) < b(m = 1)$$

We can suppose that for higher values of m convergense to steady state is slower.

[95]:
```
numOfExamples = 50
numOfGraphs = 5000

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(1000, m=2, directed=FALSE)
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
```

```
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}

mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 1000, m = 2",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 115


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-3.7932 -0.5802  0.3797  0.7374  1.0921

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.0341     0.3741   2.764  0.00671 **
X[mask]      -2.4722     0.0973 -25.407  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9481 on 108 degrees of freedom
Multiple R-squared:  0.8567,        Adjusted R-squared:  0.8553
F-statistic: 645.5 on 1 and 108 DF,  p-value: < 2.2e-16
```

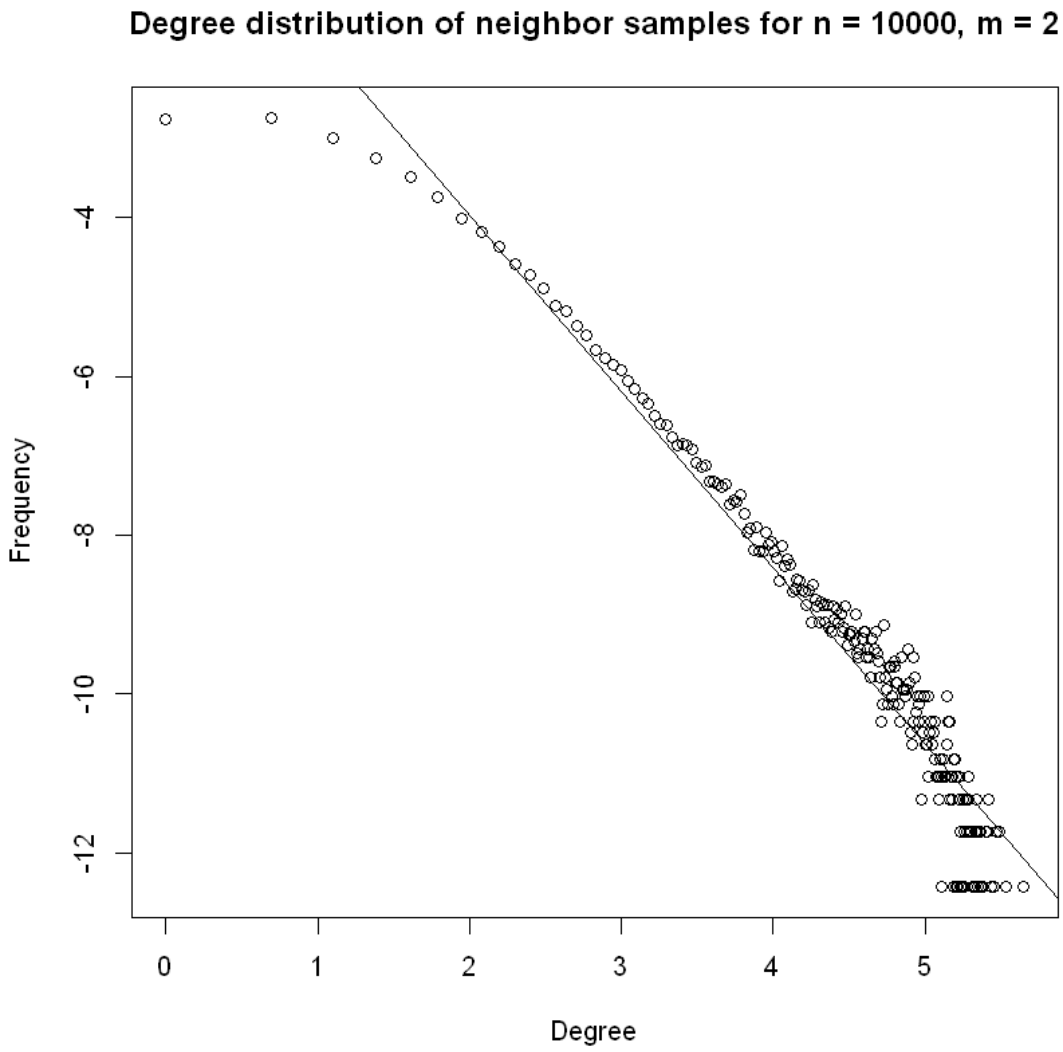


[96]:
```
numOfExamples = 50
numOfGraphs = 5000

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(10000, m=2, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 10000, m = 2",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 285


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-3.2297 -0.1351  0.1545  0.2832  0.9504

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.47036    0.16863   2.789  0.00574 **
X[mask]     -2.21953    0.03713 -59.781  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5353 on 224 degrees of freedom
Multiple R-squared:  0.941,        Adjusted R-squared:  0.9408
F-statistic:  3574 on 1 and 224 DF,  p-value: < 2.2e-16
```

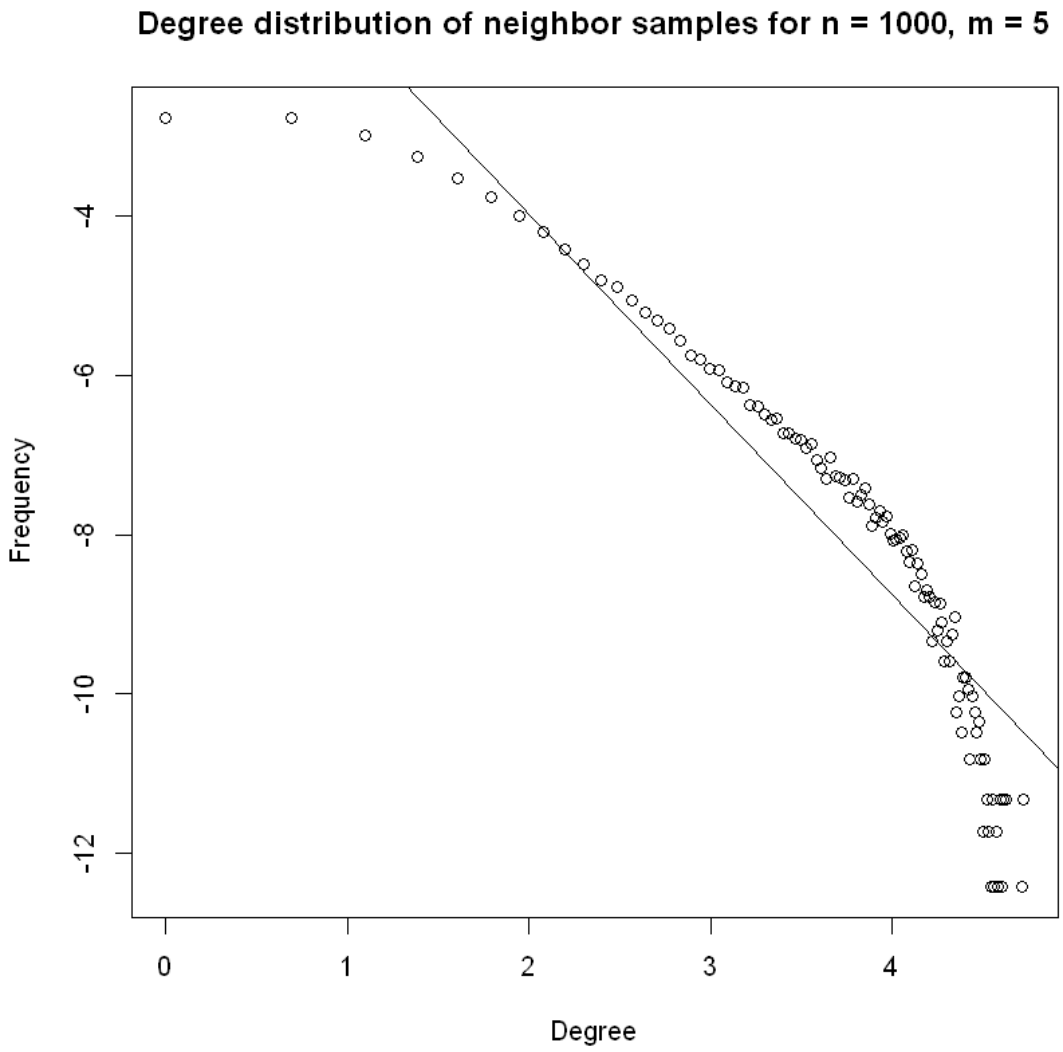


[92]:
```
numOfExamples = 50
numOfGraphs = 5000
g2 <- sample_pa(1000, m=5, directed=FALSE)
v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(1000, m=2, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 1000, m = 5",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 113


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-3.5804 -0.4104  0.4022  0.6874  0.9680

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.82100    0.37826    2.17   0.0323 *
X[mask]     -2.39322    0.09975  -23.99   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9431 on 102 degrees of freedom
Multiple R-squared:  0.8495,        Adjusted R-squared:  0.848
F-statistic: 575.7 on 1 and 102 DF,  p-value: < 2.2e-16
```

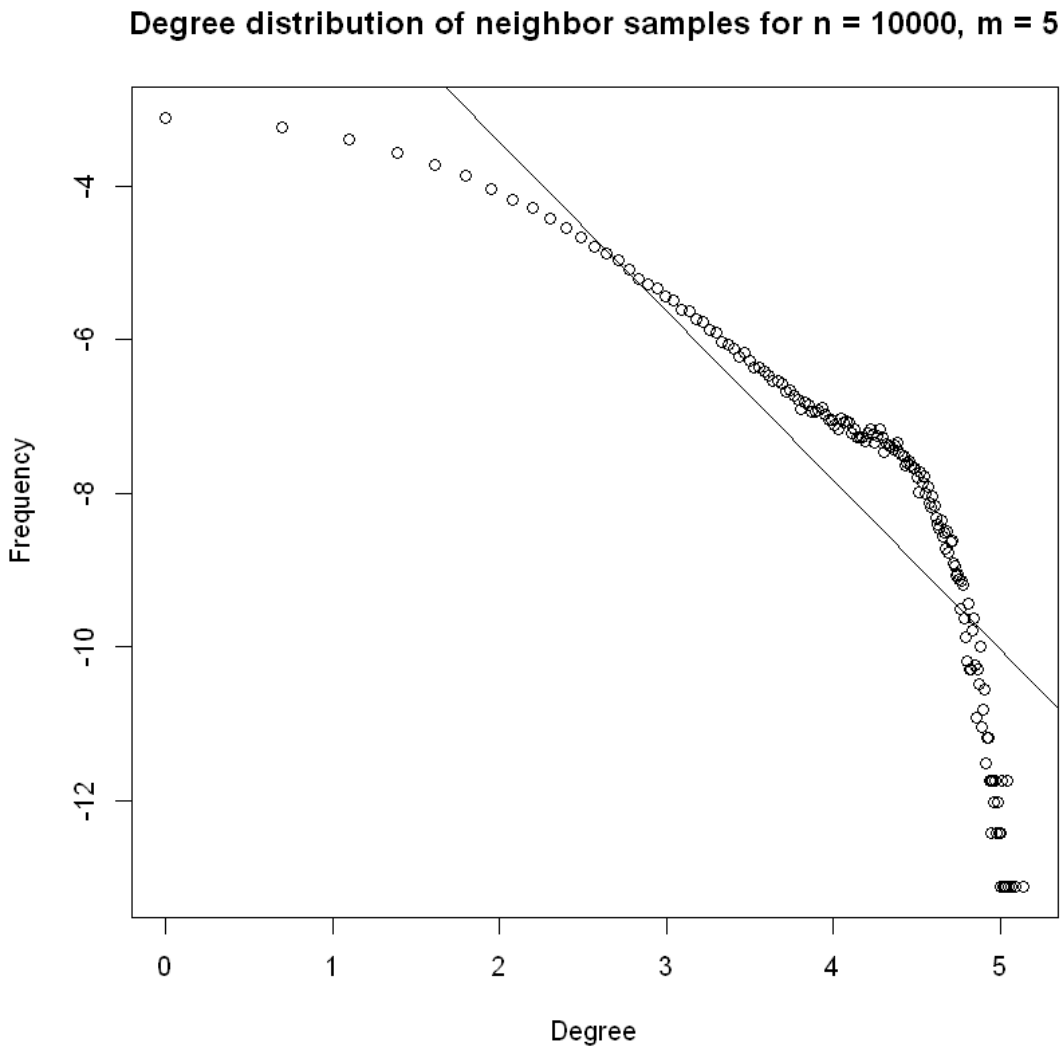


[98]:
```
numOfExamples = 50
numOfGraphs = 10000

v = 1
num = 0
transition_matrix = create_transition_matrix(g2)
PMF = transition_matrix[1, ]
degreeSecondGraph = matrix(data = 0, nrow = numOfGraphs*numOfExamples, ncol = 1)



for(k in 1:numOfGraphs){
    g2 <- sample_pa(1000, m=5, directed=FALSE)
```

```
    for(i in 1:numOfExamples){
              v = sample(1:vcount(g2), 1)
              j = neighbors(g2, v, mode = c("all"))
              #print(j)
              #print(v)
              size = length(j)
              #print("Size: ")
              #print(size)
              num = sample(1:size, 1)
              if (num == 0) {
                 num = 1
                 }
              #print("Num: ")
              #print(num)

              degreeSecondGraph[i*k] = degree(g2,j[num])
             }

}
print(max(degreeSecondGraph))
X = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)
freqSecond = matrix(data = 0, nrow = max(degreeSecondGraph), ncol = 1)

for(i in 1:(numOfExamples*numOfGraphs)){
    freqSecond[degreeSecondGraph[i]] = freqSecond[degreeSecondGraph[i]] + 1
    X[i] = i
}
inc = 0
#print(freqSecond)
for(i in 1:max(degreeSecondGraph)){
    if(freqSecond[i] != 0){
        inc = inc + 1
        freqSecond[inc] = freqSecond[i]
        X[inc] = i
        }
    }
freqSecond = freqSecond/(numOfExamples*numOfGraphs)
#print(freqSecond)
Y <- log(freqSecond)
X <- log(idx <- seq(from = 1, to = length(Y), by = 1))


for (i in 1:length(Y)){
    if(Y[i] == -Inf){
        Y[i] = 0
    }
}
```

```r
mask <- generate_mask(Y)


model = lm(Y[mask] ~ X[mask])
plot(X[mask], Y[mask],
     main="Degree distribution of neighbor samples for n = 10000, m = 5",
     xlab="Degree",
     ylab="Frequency",)
abline(model)
summary(model)
```

```
[1] 171


Call:
lm(formula = Y[mask] ~ X[mask])

Residuals:
    Min      1Q  Median      3Q     Max
-4.0872 -0.5597  0.4818  0.9087  1.3400

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.9795     0.4447   2.203   0.0291 *
X[mask]      -2.2019     0.1059 -20.792   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.261 on 157 degrees of freedom
Multiple R-squared:  0.7336,        Adjusted R-squared:  0.7319
F-statistic: 432.3 on 1 and 157 DF,  p-value: < 2.2e-16
```

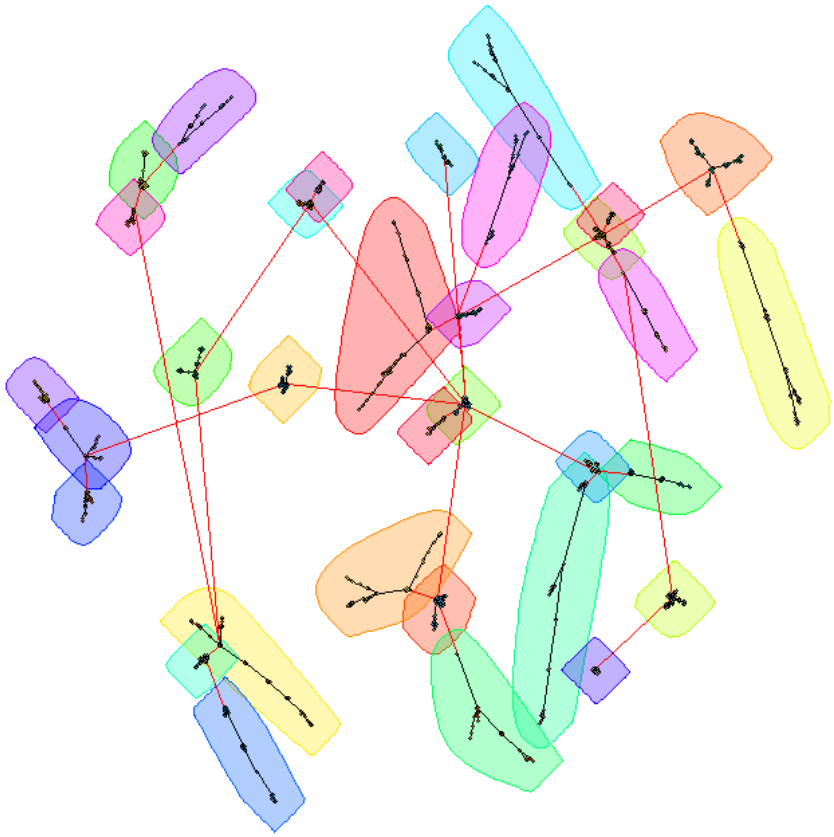


h) Again, generate a preferential attachment network with $n = 1000$, $m = 1$. Take its degree sequence and create a new network with the same degree sequence, through stub-matching procedure. Plot both networks, mark communities on their plots, and measure their modularity. Compare the two procedures for creating random power-law networks.

Hint In case that fastgreedy community detection fails because of self-loops, you may use "walktrap" community detection.

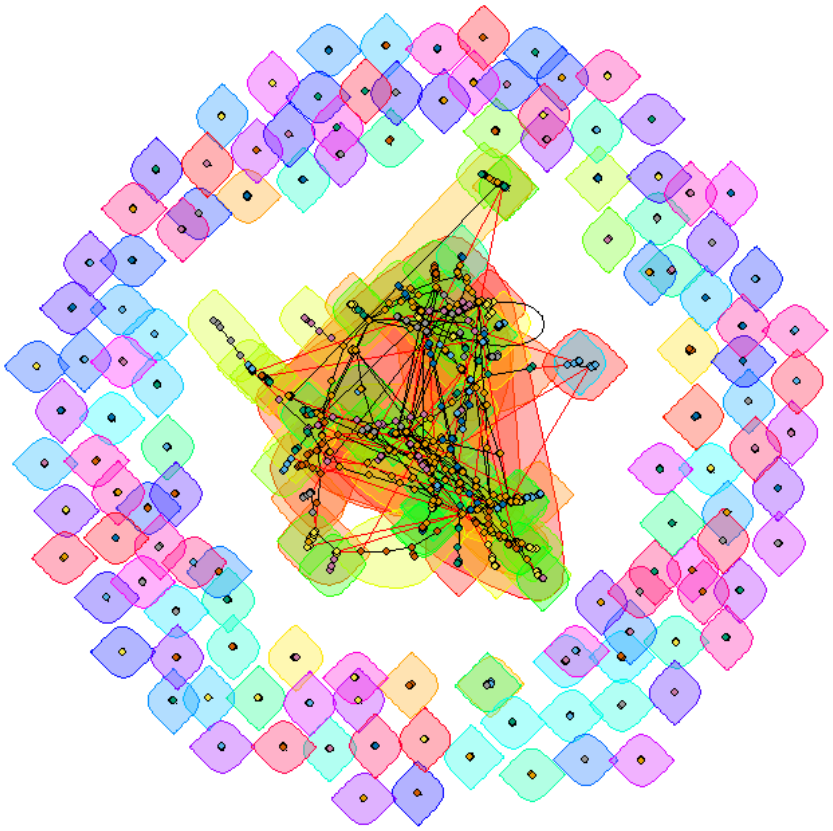
[3]:
```
g <- sample_pa(1000, m=1, directed=FALSE)
clusters <- cluster_fast_greedy(g)
print(modularity(clusters))
plot(clusters,g, vertex.size=1, vertex.label=NA)
```

```
[1] 0.9337886
```



[4]:
```
degSeq = matrix(data = 0, nrow = 1, ncol = 1)
for(i in 1:1000){

         degSeq[i] = degree(g, v = i)

        }
g1 <- sample_degseq(degSeq,  method = c("simple"))
clusters2 <- cluster_walktrap(g1)
print(modularity(clusters2))
plot(clusters2,g1, vertex.size=2, vertex.label=NA)
```

```
[1] 0.7429708
```



1_3

April 15, 2021

1 Install packages

[6]:
```
install.packages('igraph')
install.packages("poweRlaw")

library('Matrix')
library('pracma')
library('igraph')
library('poweRlaw')
```

```
Warning message:
"package 'igraph' is in use and will not be installed"Warning message:
"package 'poweRlaw' is in use and will not be installed"Warning message:
"package 'Matrix' was built under R version 3.6.3"Warning message:
"package 'pracma' was built under R version 3.6.3"
Attaching package: 'pracma'

The following objects are masked from 'package:Matrix':

    expm, lu, tril, triu
```

2 Preferential Attachment 1.3a

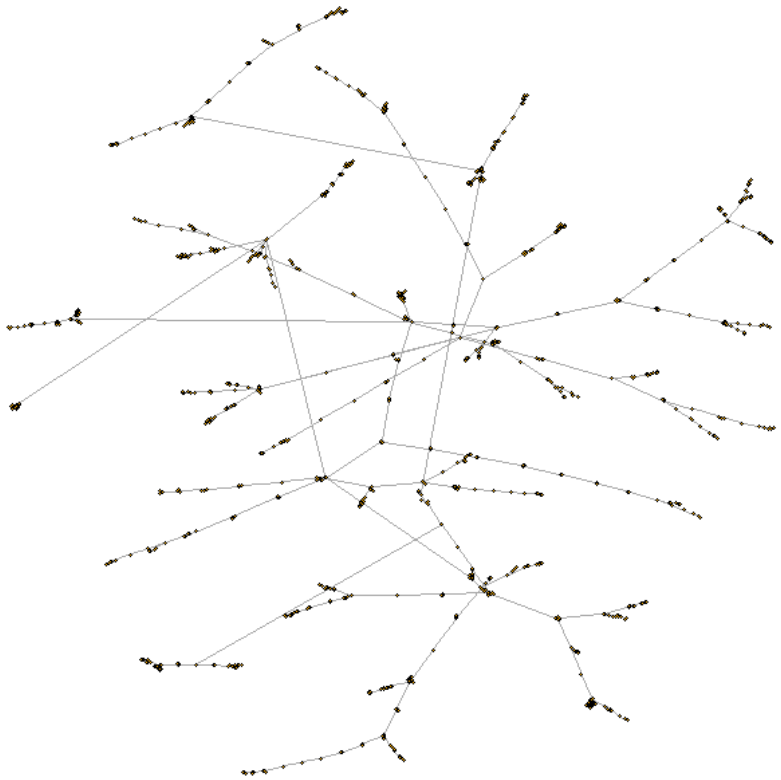
Produce such an undirected network with 1000 nodes and parameters m = 1, = 1, = −1, and a = c = d = 1, b = 0. Plot the degree distribution. What is the power law exponent?

[3]:
```
# create graph
g_pa_1000 <- sample_pa_age(1000, pa.exp=1, aging.exp=-1,  m=1, directed=FALSE)
```

[11]:
```
#plot graph

plot(g_pa_1000, vertex.size=1, vertex.label=NA)
png(file="plots/1_3_a_graph.png", width=600, height=450)

plot(g_pa_1000, vertex.size=1, vertex.label=NA)
```

```
dev.off()
```

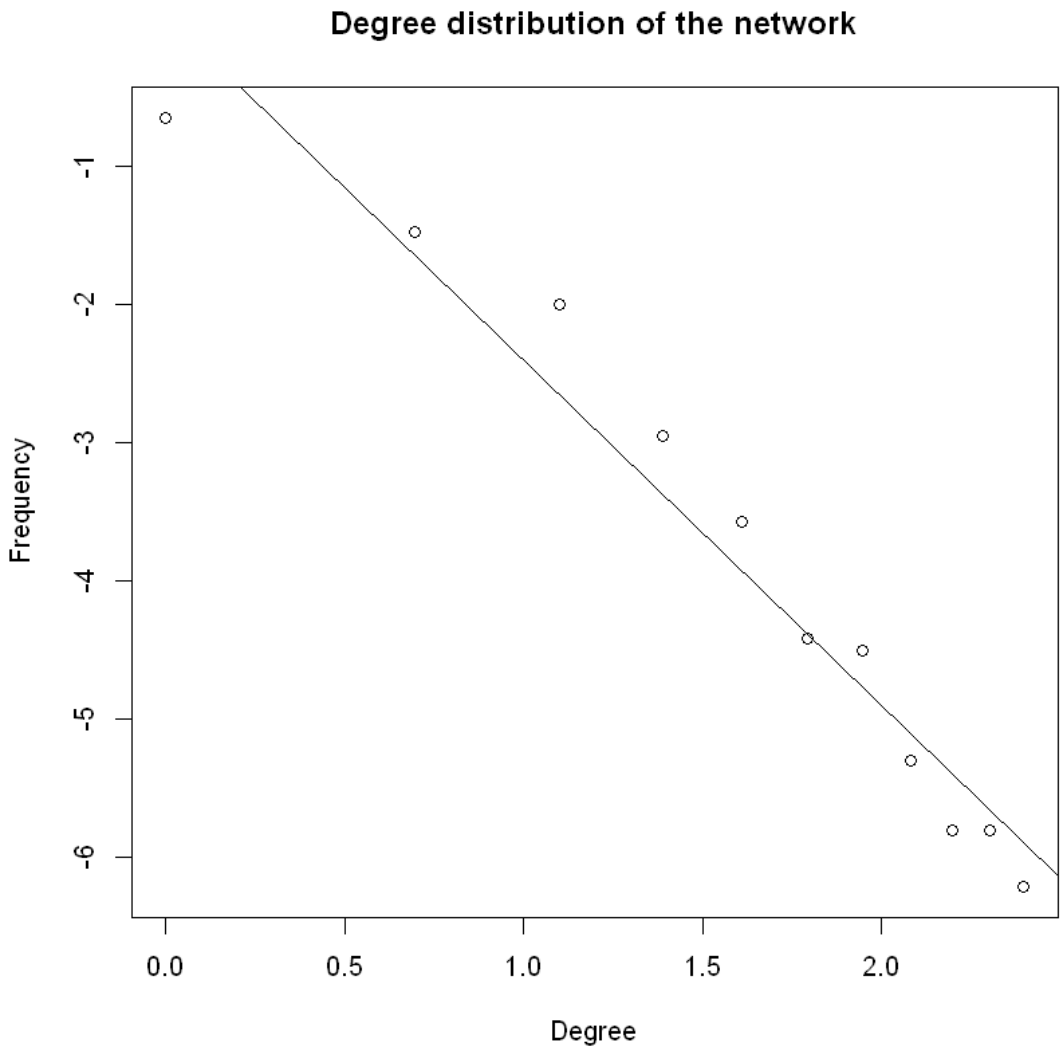
png: 2



[7]:
```
# plot degree dist
# get degrees present in a net along with frequencies with which they appear
degrees <- seq_along(degree.distribution(g_pa_1000)) - 1
distribution <- degree.distribution(g_pa_1000)
# convert them to collections and then to matrices
X <- matrix(c(degrees), byrow=TRUE, nrow=1)
Y <- matrix(c(distribution), byrow=TRUE, nrow=1)
# delete entries with zero frequencies from both matrices
# this allows to avoid - infinity values after log scaling and makes sense
```

```
# as these data is not actually present in the net
indices = which(Y!=0,arr.ind = T)
X <- X[indices]
Y <- Y[indices]
# log scale data
X <- log(X)
Y <- log(Y)
# select how many elements you want to delete from the end to avoid outliners
delete <- 0
# calculate len of the desired array
len <- size(X)[2] - delete
# get the slices of both matrices
X <- X[0:len]
Y <- Y[0:len]
model = lm(Y ~ X)

# plot for reference
plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

# save plot
png(file="plots/1_3_a.png", width=600, height=450)

plot(X, Y,
     main="Degree distribution of the network",
     xlab="Degree",
     ylab="Frequency",)
abline(model)

dev.off()
```

png: 2



[9]:
```
summary(model)
```

```
Call:
lm(formula = Y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-0.73719 -0.25647 -0.03783  0.30717  0.65323

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.08518    0.32119   0.265    0.797
```

```
X           -2.49487    0.18432 -13.535 2.74e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4343 on 9 degrees of freedom
Multiple R-squared:  0.9532,        Adjusted R-squared:  0.948
F-statistic: 183.2 on 1 and 9 DF,  p-value: 2.744e-07
```

3 1.3b: Community Structure

[10]:
```
#Community structure
clusters_pa_1000 <- cluster_fast_greedy(g_pa_1000)
print(modularity(clusters_pa_1000))
```

```
[1] 0.9352005
```

[ ]:
```

```

2_1

April 15, 2021

[1]:
```
library('igraph')
library('Matrix')
library('pracma')
```

```
Attaching package: ‘igraph’

The following objects are masked from ‘package:stats’:

    decompose, spectrum

The following object is masked from ‘package:base’:

    union


Attaching package: ‘pracma’

The following objects are masked from ‘package:Matrix’:

    expm, lu, tril, triu
```

[2]:
```
#install.packages("gsubfn")
```

[3]:
```
#install.packages("resample")
```

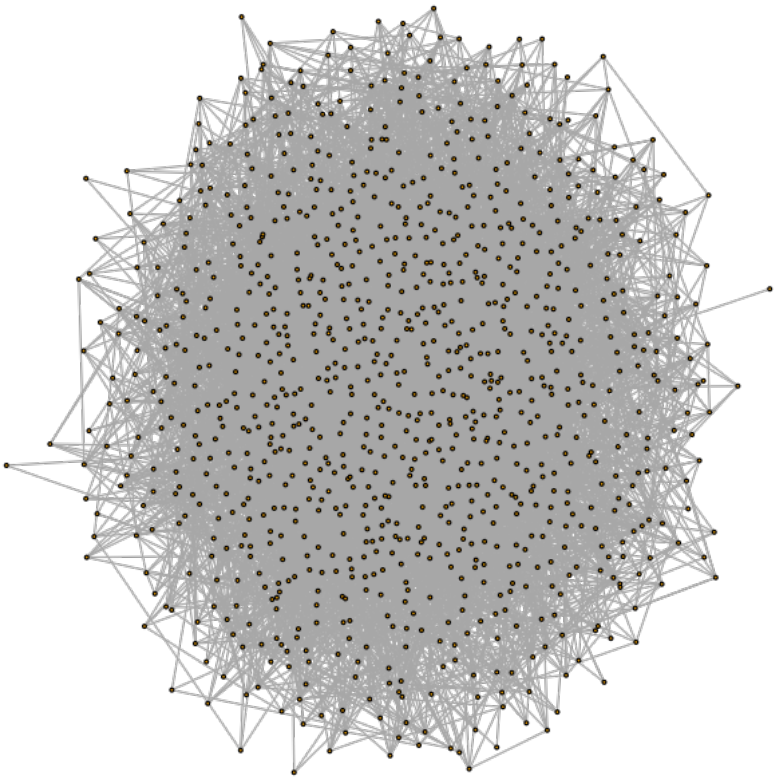
[4]:
```
library(gsubfn)
library('resample')
```

```
Loading required package: proto
Warning message in system2("/usr/bin/otool", c("-L", shQuote(DSO)), stdout =
TRUE):
"running command ''/usr/bin/otool' -L '/Users/mmlevy/opt/anaconda3/envs/my-r-
env/lib/R/library/tcltk/libs//tcltk.so'' had status 1"
Attaching package: ‘resample’

The following object is masked from ‘package:gsubfn’:
```

```
cat0
```

(a) Create an undirected random network with 1000 nodes, and the probability p for drawing an edge between any pair of nodes equal to 0.01.

[5]:
```
# Create graph
g <- sample_gnp(n=1000, p=0.01, directed=F)
# Plot graph
plot(g, vertex.size=1, vertex.label=NA)
```



(b) Let a random walker start from a randomly selected node (no teleportation). We use t to denote the number of steps that the walker has taken. Measure the average distance (defined

as the shortest path length) hs(t)i of the walker from his starting point at step t. Also, measure the variance 2

(c) = h(s(t) − hs(t)i) 2 i of this distance. Plot hs(t)i v.s. t and 2

(d) v.s. t. Here, the average h · i is over random choices of the starting nodes.

[6]:
```
# Transition matrix from discussion notebook
create_transition_matrix = function (g){

    # WARNING: make sure your graph is connected (you might input GCC of your␣
 ↪graph)

    vs = V(g)
    n = vcount(g)
    adj = as_adjacency_matrix(g)
    adj[diag(rowSums(adj) == 0)] = 1  # handle if the user is using the␣
 ↪function for networks with isolated nodes by creating self-edges
    z = matrix(rowSums(adj, , 1))

    transition_matrix = adj / repmat(z, 1, n)  # normalize to get probabilities

    return(transition_matrix)
}
```

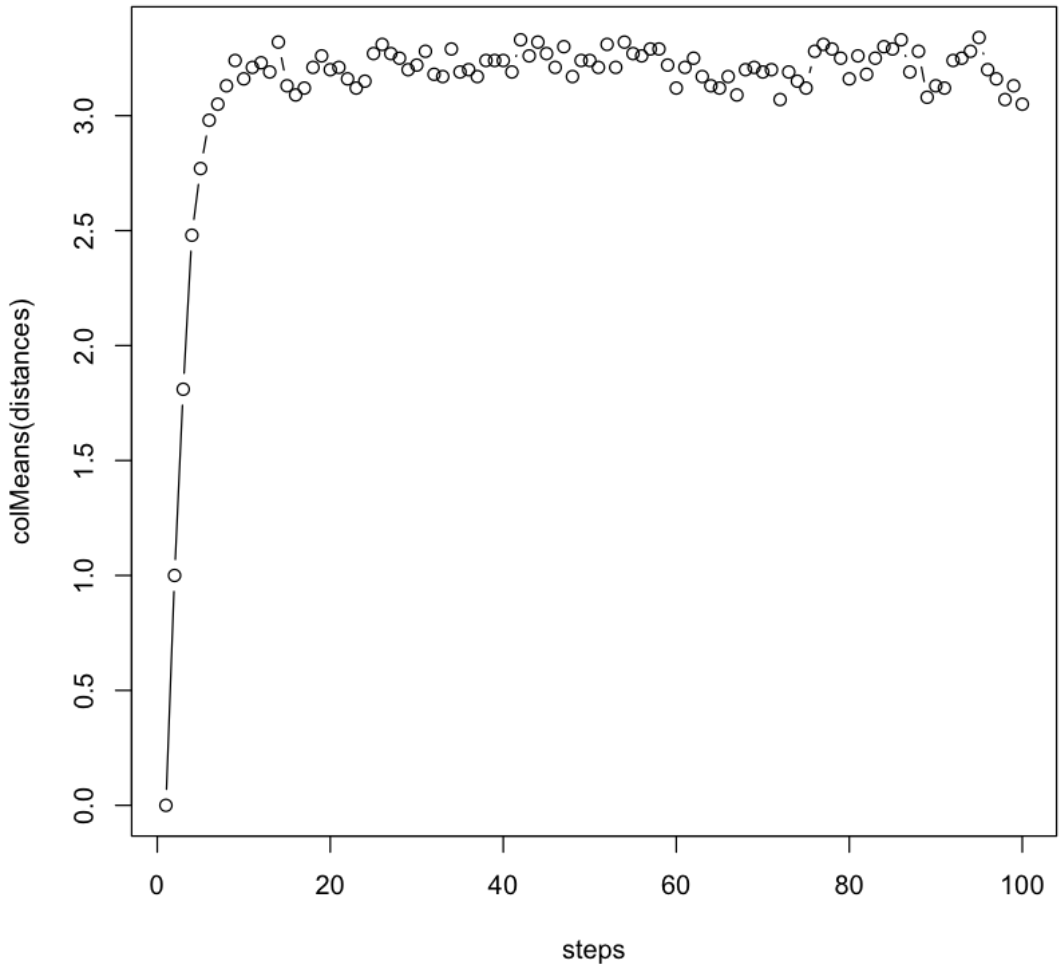
[7]:
```
# Random walk function from discussion notebook
random_walk = function (g, num_steps, start_node, transition_matrix = NULL){
    # Save distances at each step in a vector
    stepwise_distances = vector(mode="numeric",length=num_steps)
    if(is.null(transition_matrix))
        transition_matrix = create_transition_matrix(g)

    v = start_node
    for(i in 1:num_steps){
        stepwise_distances[i] = shortest.paths(g, start_node, v)[1][1]
        #fprintf('Step %d: %d\n', i, v)  # COMMENT THIS
        PMF = transition_matrix[v, ]
        v = sample(1:vcount(g), 1, prob = PMF)
    }

    return (list(stepwise_distances, v))
}
```

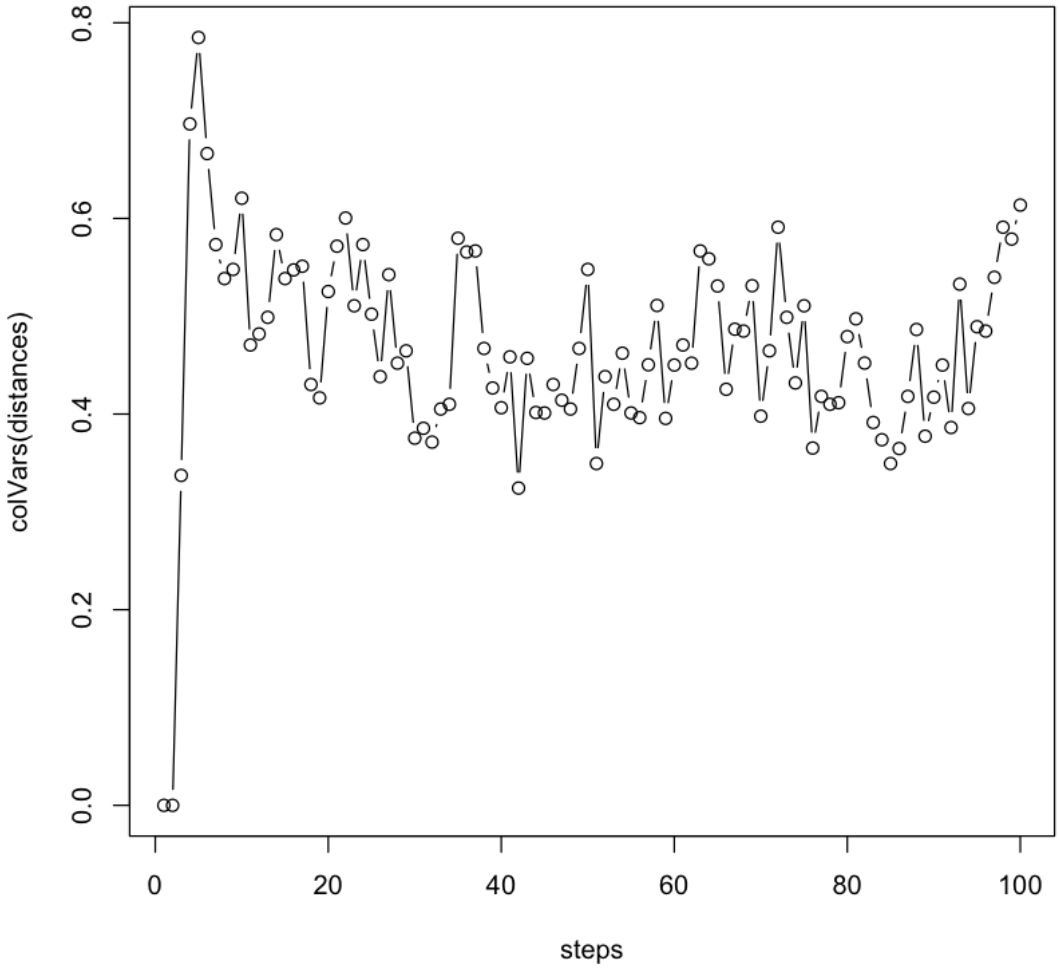
[8]:
```
# Get GCC so the transition matrix function works
g_components <- clusters(g)
    # which is the largest component
ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
gcc <- induced.subgraph(g, which(g_components$membership == ix))
```

[9]:
```
# Try a random walk num_trials times
num_trials = 100
num_steps = 100

# Initialize an array of distances
distances = matrix(NA, nrow=num_steps, ncol=num_trials)
# Initialize vector of end nodes
end_nodes = vector(mode="numeric",length=num_trials)


# Run the random walks
for(i in 1:num_trials){
    # Randomly pick a start node
    start_node <- sample(1:vcount(gcc), 1)
    list[stepwise_distances, v] = random_walk(gcc, num_steps, start_node)
    # Add stepwise_distances as column i in the matrix
    distances[i,] = stepwise_distances
    end_nodes[i] = v

}
```

[13]:
```
# Plot steps and mean distances
print(dim(distances))
steps = seq(1, num_steps, 1)
plot(steps,colMeans(distances),type="b")
plot(steps,colVars(distances),type="b")
```

```
[1] 100 100
```

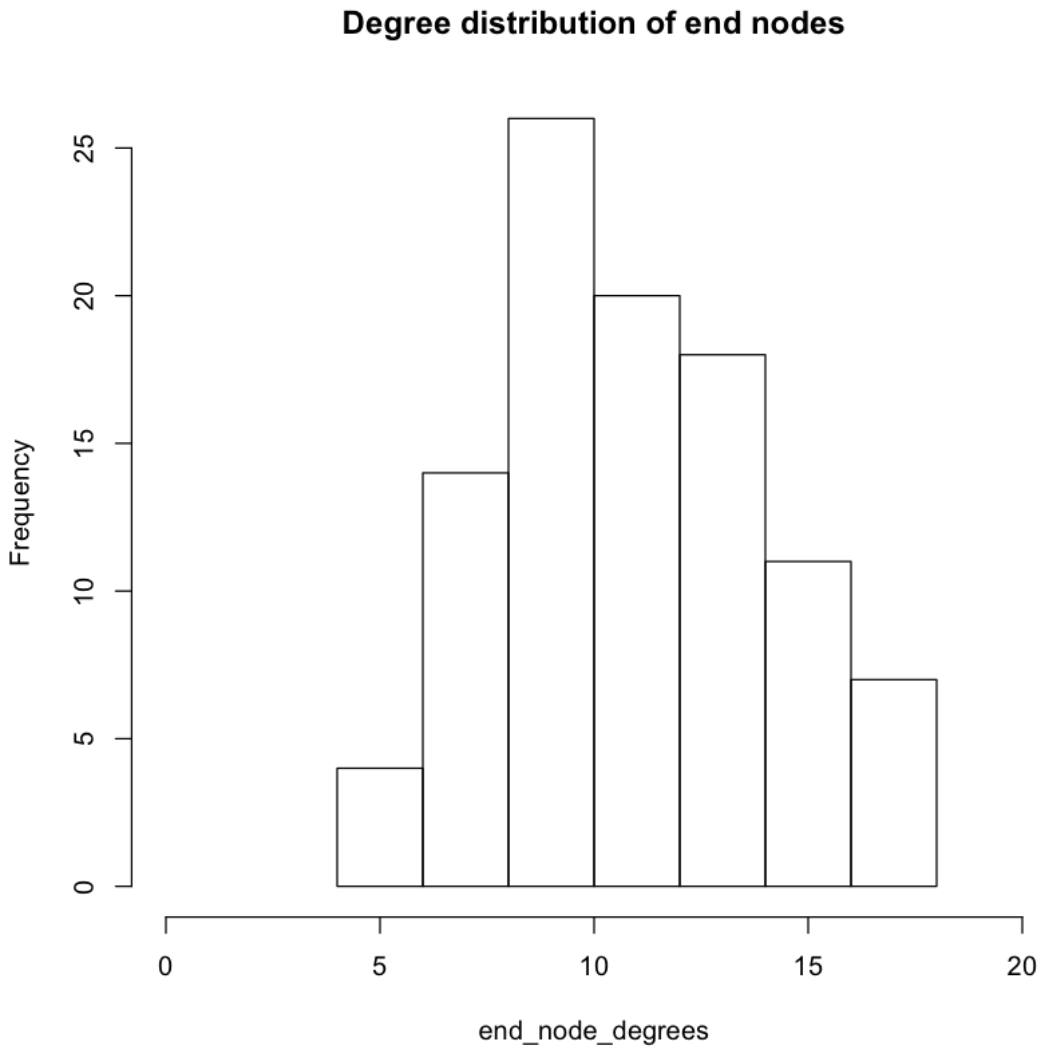

colMeans(distances)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
20
40
60
80
100
steps

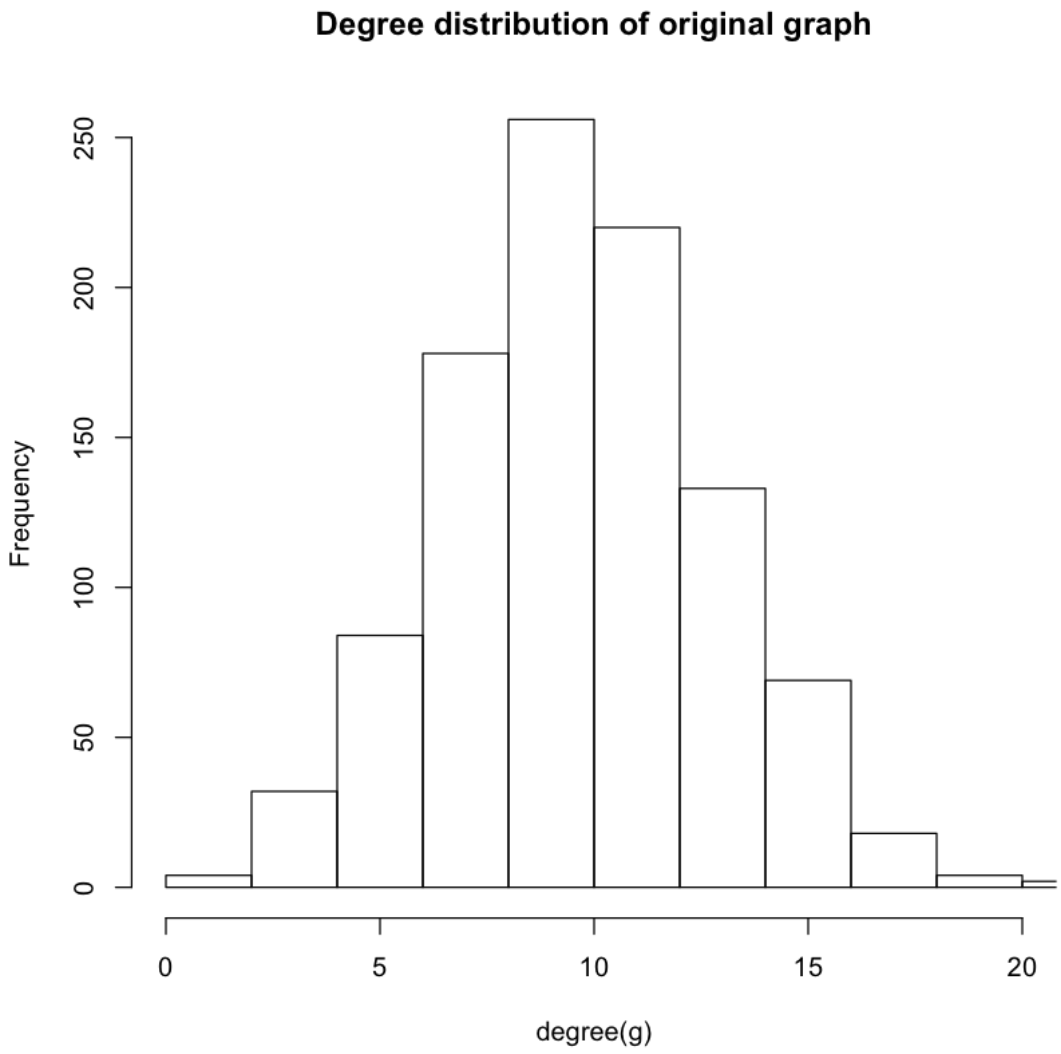


(c) Measure the degree distribution of the nodes reached at the end of the random walk. How does it compare to the degree distribution of graph?

[23]:
```
# Convert vertex to degrees
end_node_degrees = vector(mode="numeric",length=num_trials)
for(i in 1:num_trials){
    node = end_nodes[i]
    end_node_degrees[i] = degree(gcc)[node]
}
# Plot distribution of end_node_degrees with 9 bins
hist(end_node_degrees, 9, xlim=c(0,20),main="Degree distribution of end nodes")

# Plot distribution of end_node_degrees with 9 bins
```

```
hist(degree(g), 9, xlim=c(0,20),main="Degree distribution of original graph")
```



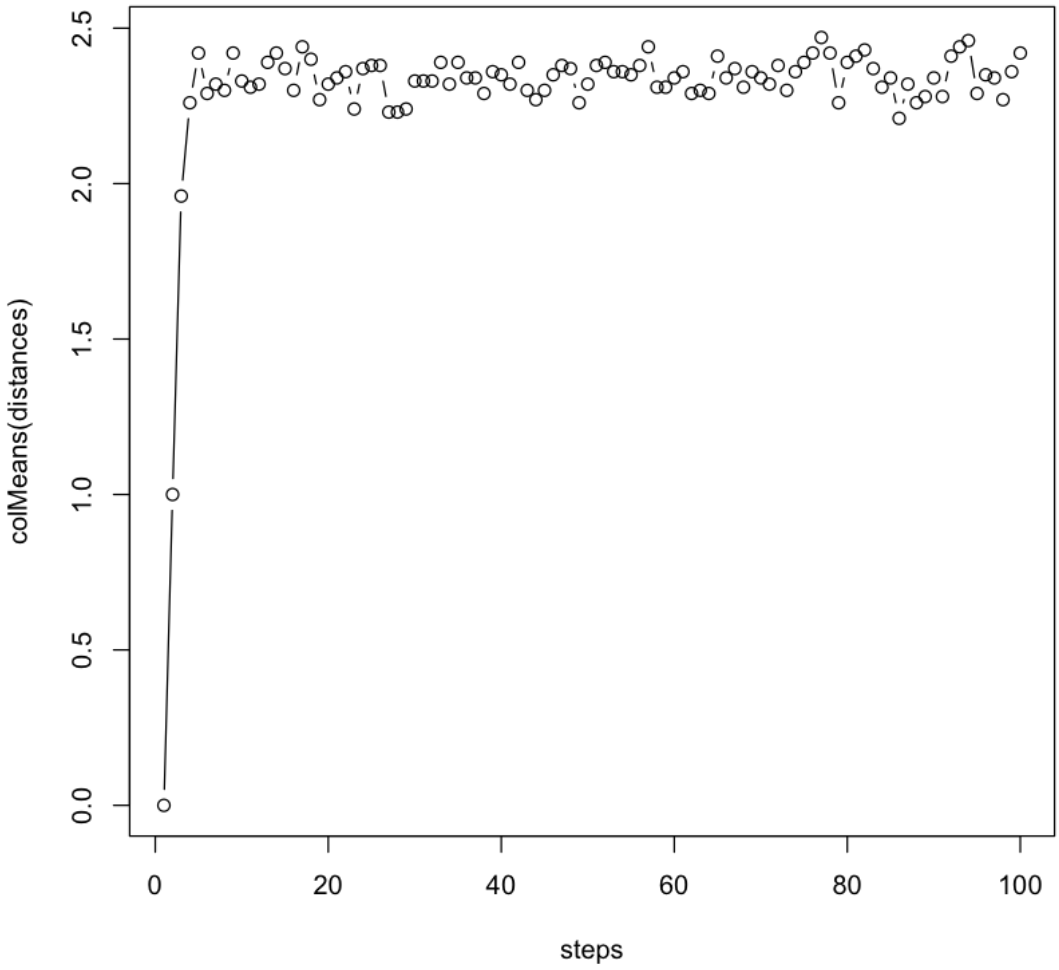


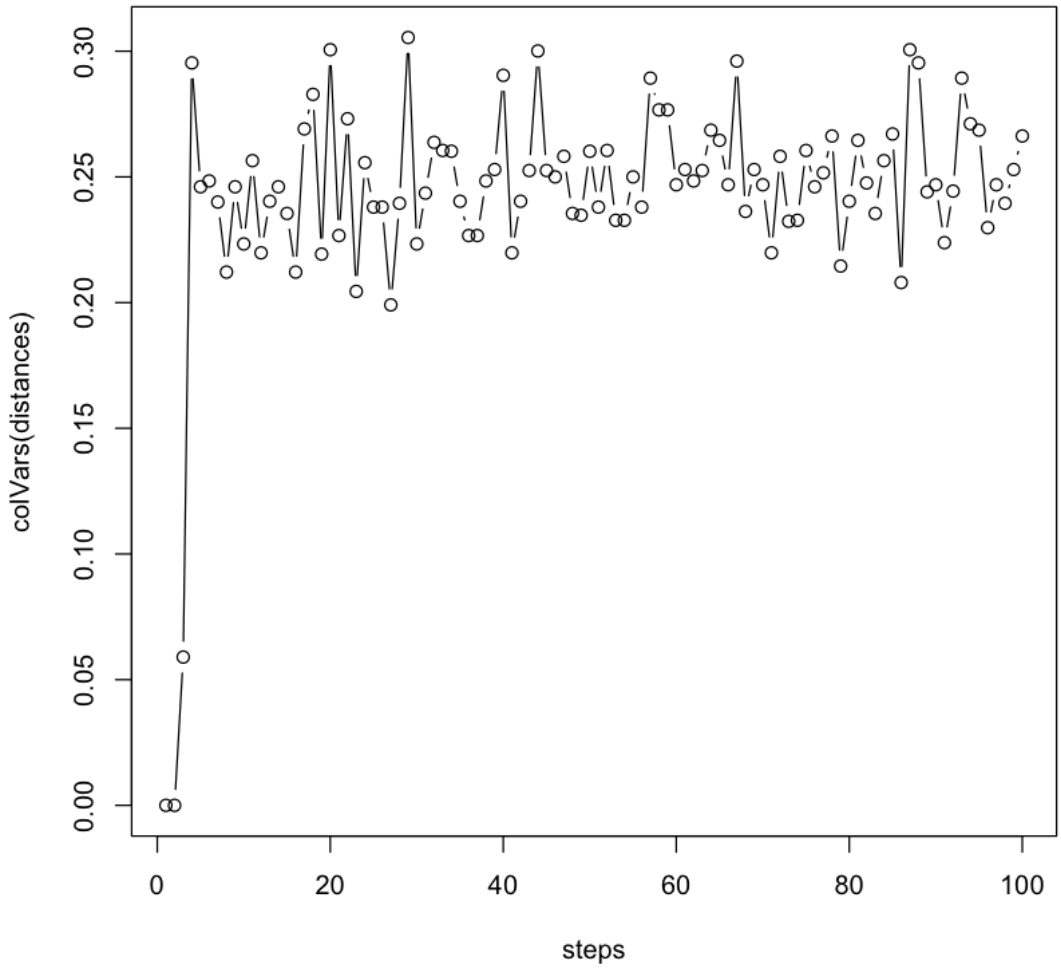
(d) Repeat 1(b) for undirected random networks with 10000 nodes. Compare the results and explain qualitatively. Does the diameter of the network play a role?

[24]:
```
g2 = sample_gnp(n=10000, p=0.01, directed=F)
# Get GCC so the transition matrix function works
g_components <- clusters(g2)
    # which is the largest component
ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
gcc2 <- induced.subgraph(g2, which(g_components$membership == ix))
# Try a random walk num_trials times
num_trials = 100
num_steps = 100
```

```
# Initialize an array of distances
distances = matrix(NA, nrow=num_steps, ncol=num_trials)
# Initialize vector of end nodes
end_nodes = vector(mode="numeric",length=num_trials)


# Run the random walks
for(i in 1:num_trials){
    # Randomly pick a start node
    start_node <- sample(1:vcount(gcc2), 1)
    list[stepwise_distances, v] = random_walk(gcc2, num_steps, start_node)
    # Add stepwise_distances as column i in the matrix
    distances[i,] = stepwise_distances
    end_nodes[i] = v

}
```

[25]:
```
plot(steps,colMeans(distances),type="b")
plot(steps,colVars(distances),type="b")
```


colMeans(distances)
0.0
0.5
1.0
1.5
2.0
2.5
0
20
40
60
80
100
steps



[11]:
```
a = rep(1, 3)
b = c(1, 2, 3)
c = rep(2, 3)

d = colSums(rbind(a,b,c))
print(d)
```

```
[1] 4 5 6
```

[12]:
```
e = d/2
print(e)
```

```
[1] 2.0 2.5 3.0
```

[ ]:

2_2

April 15, 2021

[1]:
```
library('igraph')
library('Matrix')
library('pracma')
```

```
Warning message:
"package 'igraph' was built under R version 3.6.3"
Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union

Warning message:
"package 'Matrix' was built under R version 3.6.3"Warning message:
"package 'pracma' was built under R version 3.6.3"
Attaching package: 'pracma'

The following objects are masked from 'package:Matrix':

    expm, lu, tril, triu
```

[2]:
```
install.packages("gsubfn")
install.packages("resample")
library('gsubfn')
library('resample')
```

```
package 'gsubfn' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\tamee\AppData\Local\Temp\Rtmpae1cgk\downloaded_packages
package 'resample' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
```

```
        C:\Users\tamee\AppData\Local\Temp\Rtmpae1cgk\downloaded_packages
```

```
Warning message:
"package 'gsubfn' was built under R version 3.6.3"Loading required package:
proto
Warning message:
"package 'proto' was built under R version 3.6.3"
Attaching package: 'resample'

The following object is masked from 'package:gsubfn':

    cat0
```

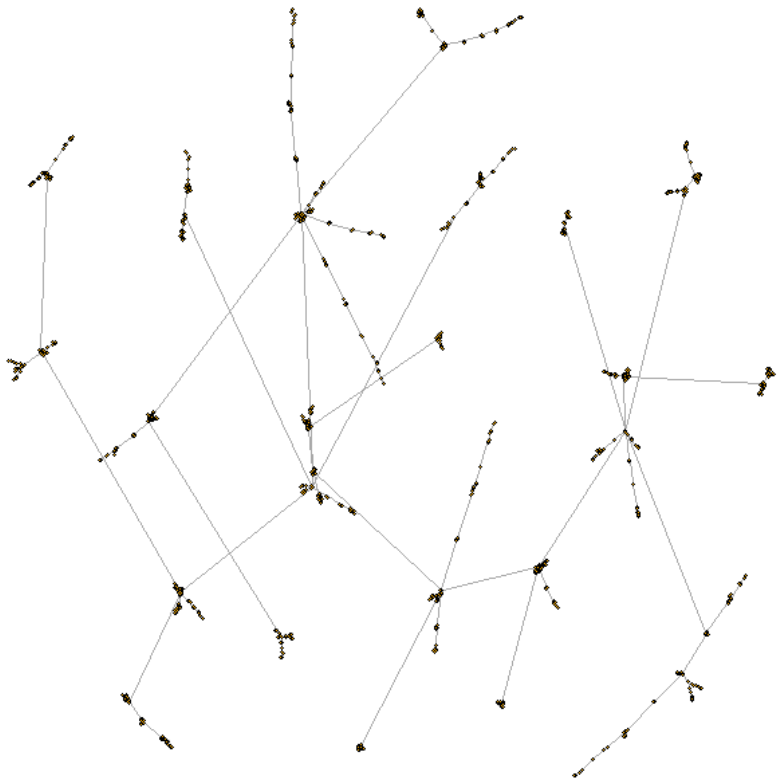
[ ]:

[ ]:

(a) Generate an undirected preferential attachment network with 1000 nodes, where each new node attaches to m = 1 old nodes.

[16]:
```r
# Create graph:
g <- sample_pa(1000, m=1, directed=FALSE)

# Plot graph
plot(g, vertex.size=1, vertex.label=NA)
png(file="plots/2_2_a_graph.png", width=600, height=450)

plot(g, vertex.size=1, vertex.label=NA)

dev.off()
```

png: 2



(b) Let a random walker start from a randomly selected node (no teleportation). We use t to denote the number of steps that the walker has taken. Measure the average distance (defined as the shortest path length) hs(t)i of the walker from his starting point at step t. Also, measure the variance 2

(c) = h(s(t) − hs(t)i) 2 i of this distance. Plot hs(t)i v.s. t and 2

(d) v.s. t. Here, the average h · i is over random choices of the starting nodes.

[17]:
```
# Transition matrix from discussion notebook
create_transition_matrix = function (g){

    # WARNING: make sure your graph is connected (you might input GCC of your␣
 ↪graph)
```

```
    vs = V(g)
    n = vcount(g)
    adj = as_adjacency_matrix(g)
    adj[diag(rowSums(adj) == 0)] = 1  # handle if the user is using the␣
↪function for networks with isolated nodes by creating self-edges
    z = matrix(rowSums(adj, , 1))

    transition_matrix = adj / repmat(z, 1, n)  # normalize to get probabilities

    return(transition_matrix)
}
```

[18]:
```
# Random walk function from discussion notebook
random_walk = function (g, num_steps, start_node, transition_matrix = NULL){
    # Save distances at each step in a vector
    stepwise_distances = vector(mode="numeric",length=num_steps)
    if(is.null(transition_matrix))
        transition_matrix = create_transition_matrix(g)

    v = start_node
    for(i in 1:num_steps){
        stepwise_distances[i] = shortest.paths(g, start_node, v)[1][1]
        #fprintf('Step %d: %d\n', i, v)  # COMMENT THIS
        PMF = transition_matrix[v, ]
        v = sample(1:vcount(g), 1, prob = PMF)
    }

    return (list(stepwise_distances, v))
}
```

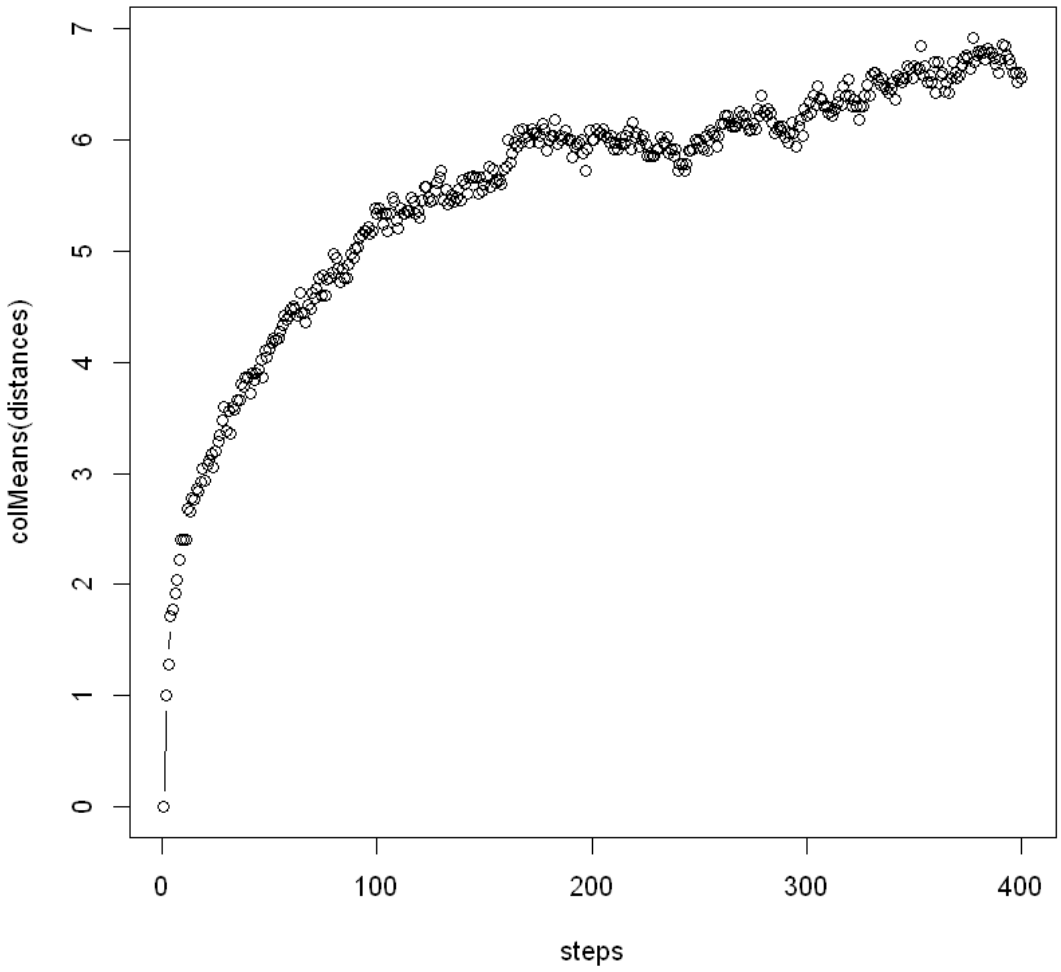
[19]:
```
# Get GCC so the transition matrix function works
g_components <- clusters(g)
    # which is the largest component
ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
gcc <- induced.subgraph(g, which(g_components$membership == ix))
```

[23]:
```
# Try a random walk num_trials times
num_trials = 100
num_steps = 400

# Initialize an array of distances
distances = matrix(NA, nrow=num_trials, ncol=num_steps)
# Initialize vector of end nodes
end_nodes = vector(mode="numeric",length=num_trials)
```

```
# Run the random walks
for(i in 1:num_trials){
    # Randomly pick a start node
    start_node <- sample(1:vcount(gcc), 1)
    list[stepwise_distances, v] = random_walk(gcc, num_steps, start_node)
    # Add stepwise_distances as column i in the matrix
    distances[i,] = stepwise_distances
    end_nodes[i] = v

}
```

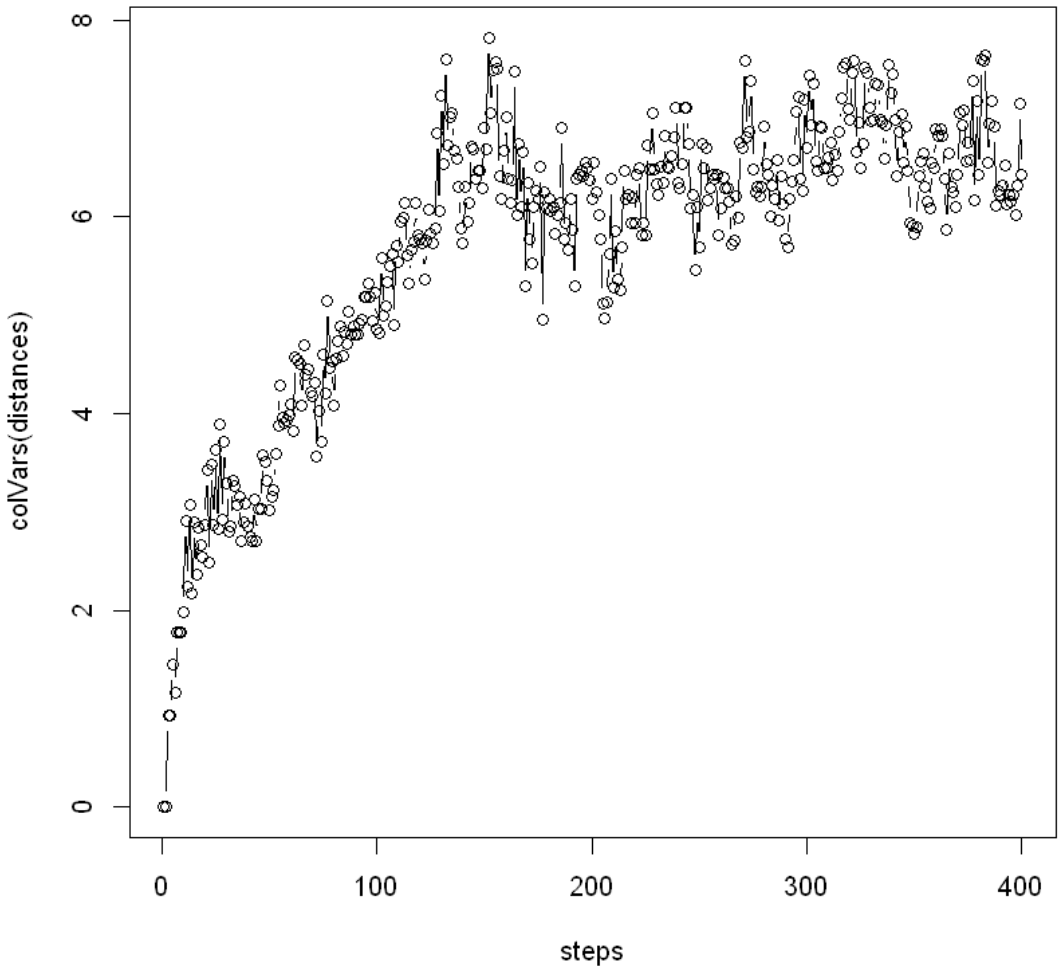
[24]:
```
# Plot steps and mean distances
print(dim(distances))
steps = seq(1, num_steps, 1)
plot(steps,colMeans(distances),type="b")
plot(steps,colVars(distances),type="b")
png(file="plots/2_2_b_mean.png", width=600, height=450)

plot(steps,colMeans(distances),type="b")

dev.off()
png(file="plots/2_2_b_var.png", width=600, height=450)

plot(steps,colVars(distances),type="b")

dev.off()
```

```
[1] 100 400
```



png: 2

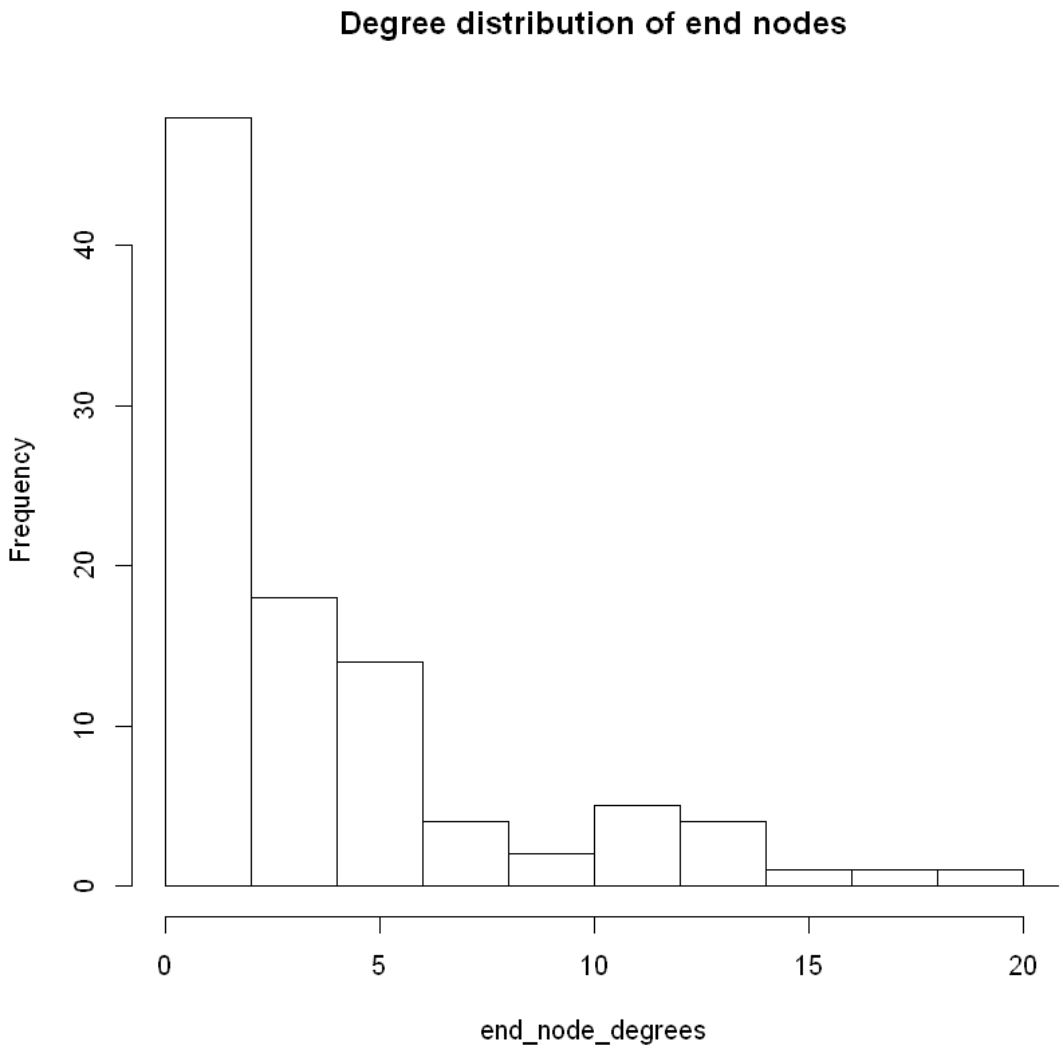
png: 2



(c) Measure the degree distribution of the nodes reached at the end of the random walk. How does it compare to the degree distribution of graph?

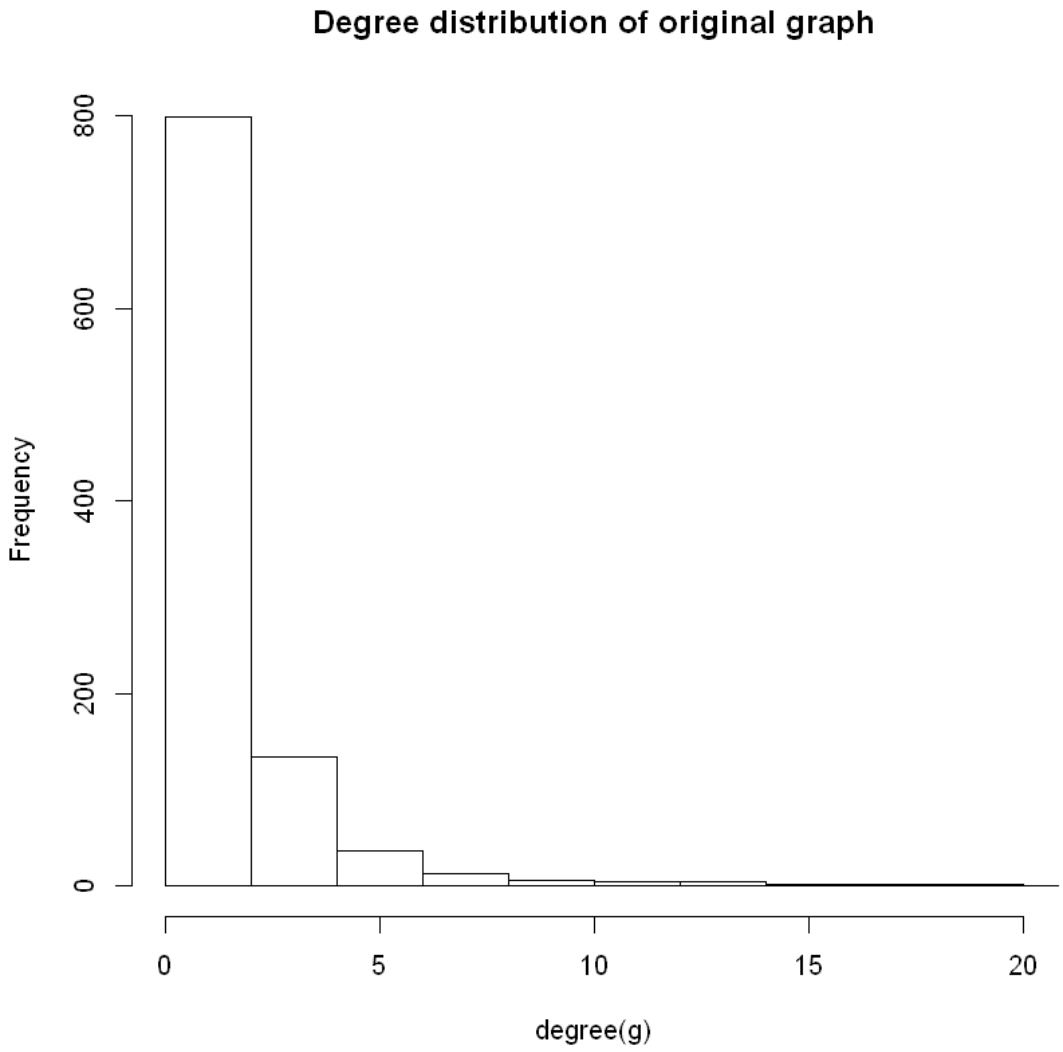
[25]:
```
# Convert vertex to degrees
end_node_degrees = vector(mode="numeric",length=num_trials)
for(i in 1:num_trials){
    node = end_nodes[i]
    end_node_degrees[i] = degree(gcc)[node]
}
# Plot distribution of end_node_degrees with 9 bins
hist(end_node_degrees, 9, xlim=c(0,20),main="Degree distribution of end nodes")

# Plot distribution of end_node_degrees with 9 bins
```

```
hist(degree(g), 9, xlim=c(0,20),main="Degree distribution of original graph")
png(file="plots/2_2_c_end_nodes.png", width=600, height=450)

hist(end_node_degrees, 9, xlim=c(0,20),main="Degree distribution of end nodes")

dev.off()
png(file="plots/2_2_c_og.png", width=600, height=450)

hist(degree(g), 9, xlim=c(0,20),main="Degree distribution of original graph")

dev.off()
```



png: 2

png: 2

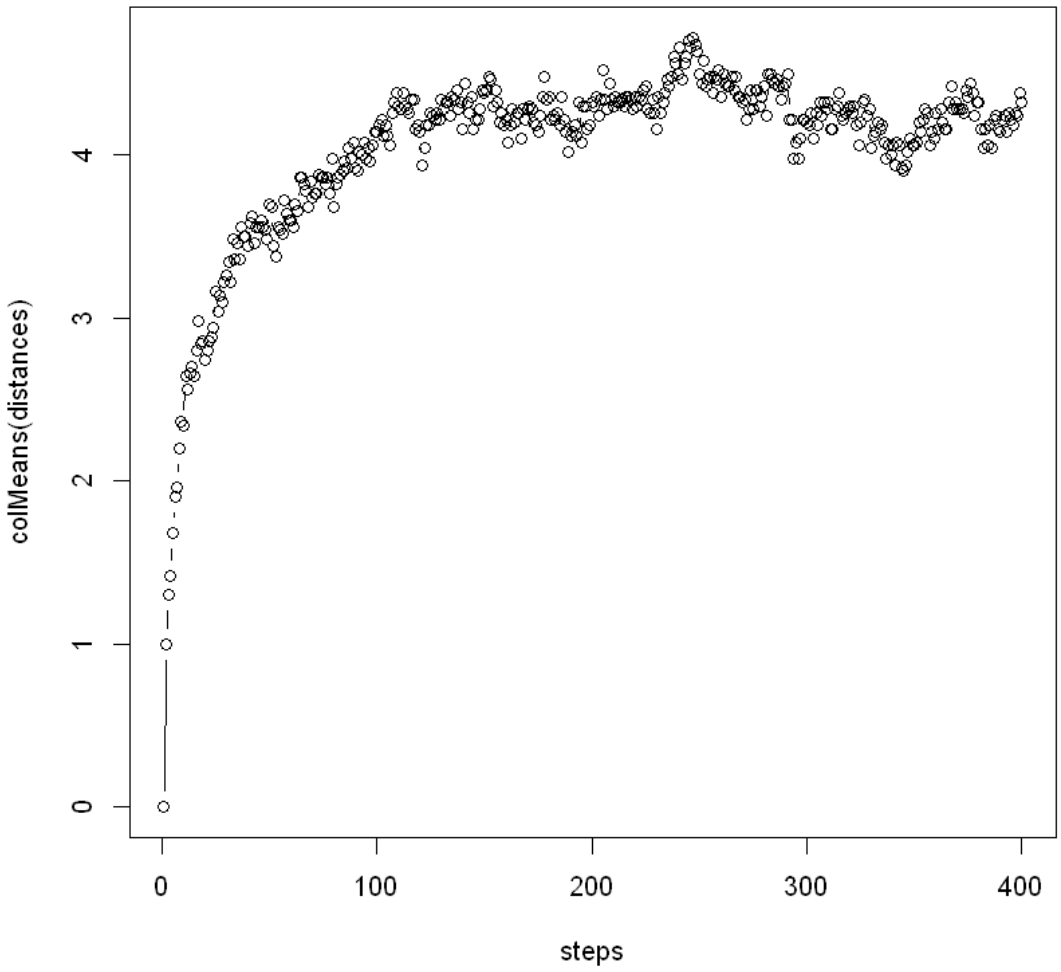


(d) Repeat 2(b) for preferential attachment networks with 100 and 10000 nodes, and m = 1. Compare the results and explain qualitatively. Does the diameter of the network play a role?

[11]:
```
#n=100
g3 <- sample_pa(100, m=1,directed=FALSE)
# Get GCC so the transition matrix function works
g_components <- clusters(g3)
    # which is the largest component
ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
gcc3 <- induced.subgraph(g3, which(g_components$membership == ix))
# Try a random walk num_trials times
```

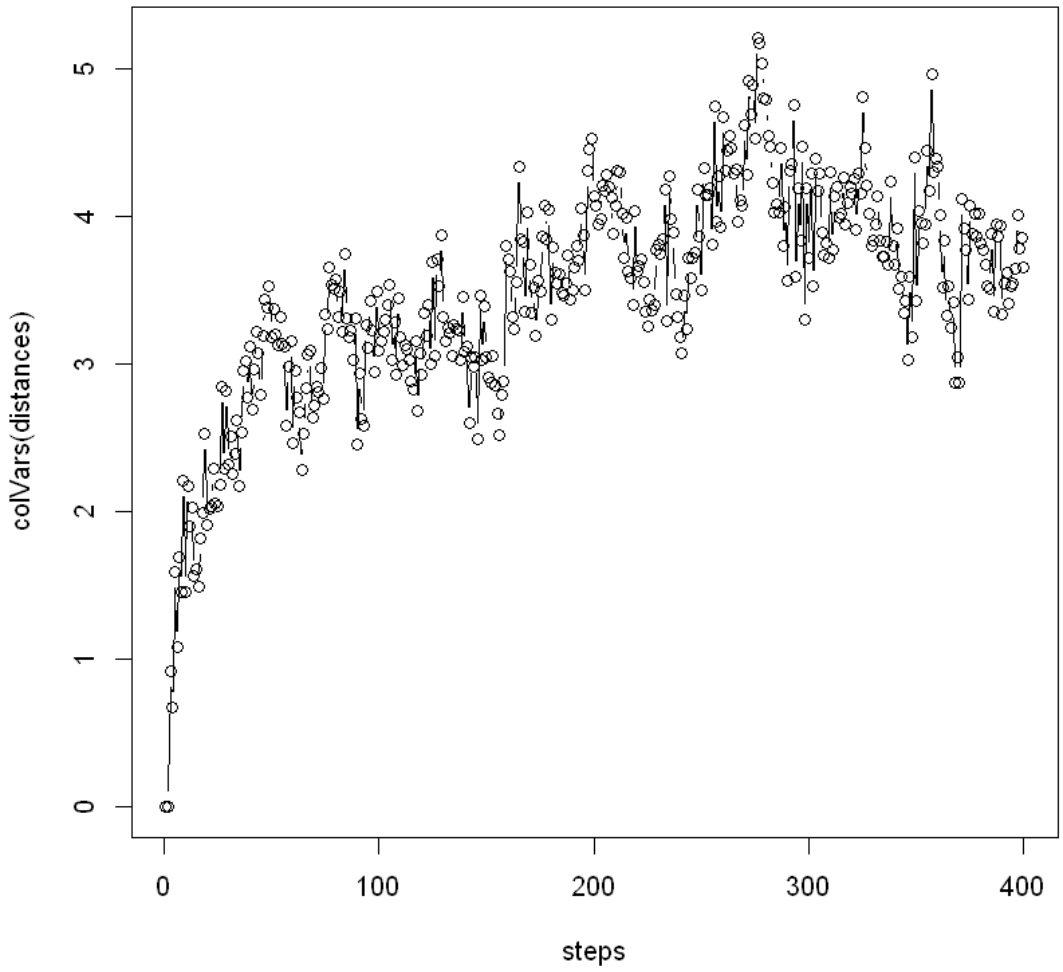
```
num_trials = 100
num_steps = 400

# Initialize an array of distances
distances = matrix(NA, nrow=num_trials, ncol=num_steps)
# Initialize vector of end nodes
end_nodes = vector(mode="numeric",length=num_trials)


# Run the random walks
for(i in 1:num_trials){
    # Randomly pick a start node
    start_node <- sample(1:vcount(gcc3), 1)
    list[stepwise_distances, v] = random_walk(gcc3, num_steps, start_node)
    # Add stepwise_distances as column i in the matrix
    distances[i,] = stepwise_distances
    end_nodes[i] = v

}
```

[12]:
```
steps = seq(1, num_steps, 1)

plot(steps,colMeans(distances),type="b")
plot(steps,colVars(distances),type="b")

png(file="plots/2_2_d_100m.png", width=600, height=450)

plot(steps,colMeans(distances),type="b")

dev.off()
png(file="plots/2_2_d_100v.png", width=600, height=450)

plot(steps,colVars(distances),type="b")

dev.off()
```



png: 2

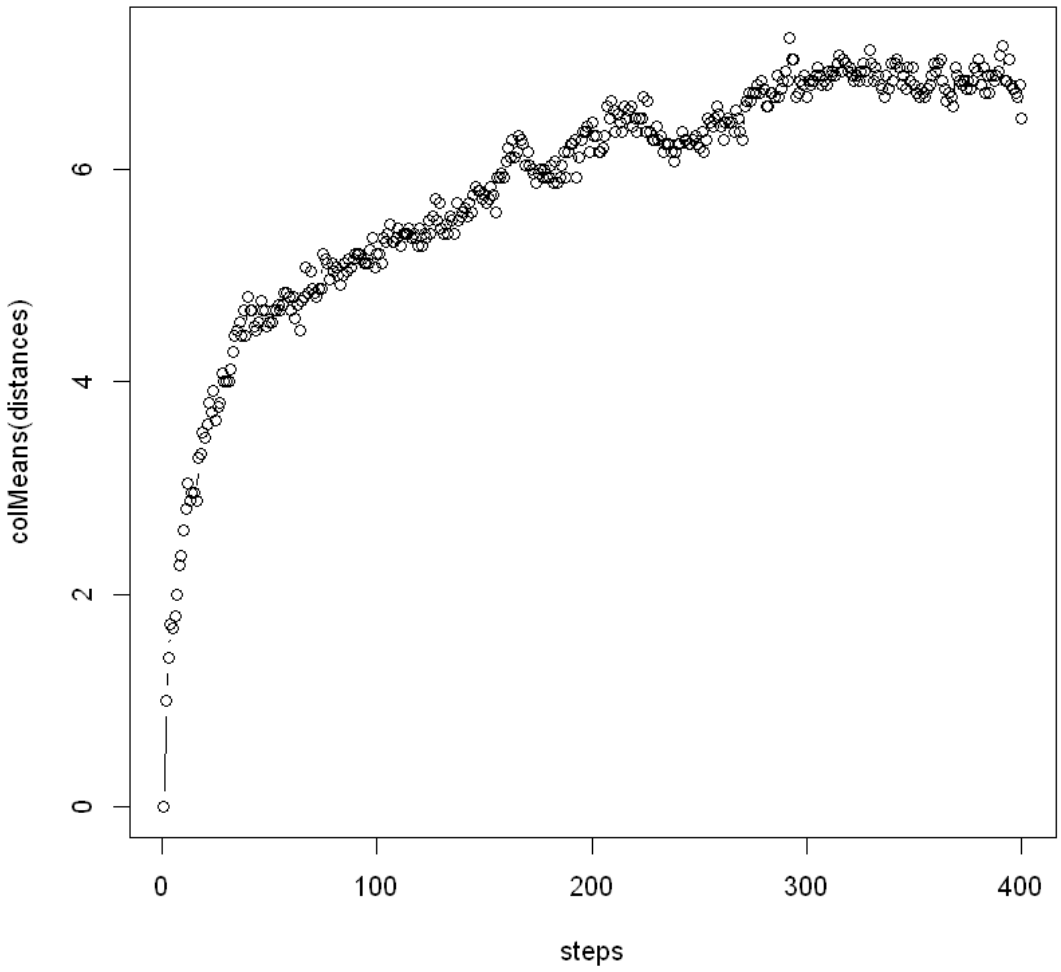
png: 2



[14]:
```
#n=10000
g2 <- sample_pa(10000, m=1,, directed=FALSE)
# Get GCC so the transition matrix function works
g_components <- clusters(g2)
    # which is the largest component
ix <- which.max(g_components$csize) # like np.argmax(...)
    # get the subgraph correspondent to just the giant component
gcc2 <- induced.subgraph(g2, which(g_components$membership == ix))
# Try a random walk num_trials times
num_trials = 50
num_steps = 400

# Initialize an array of distances
```

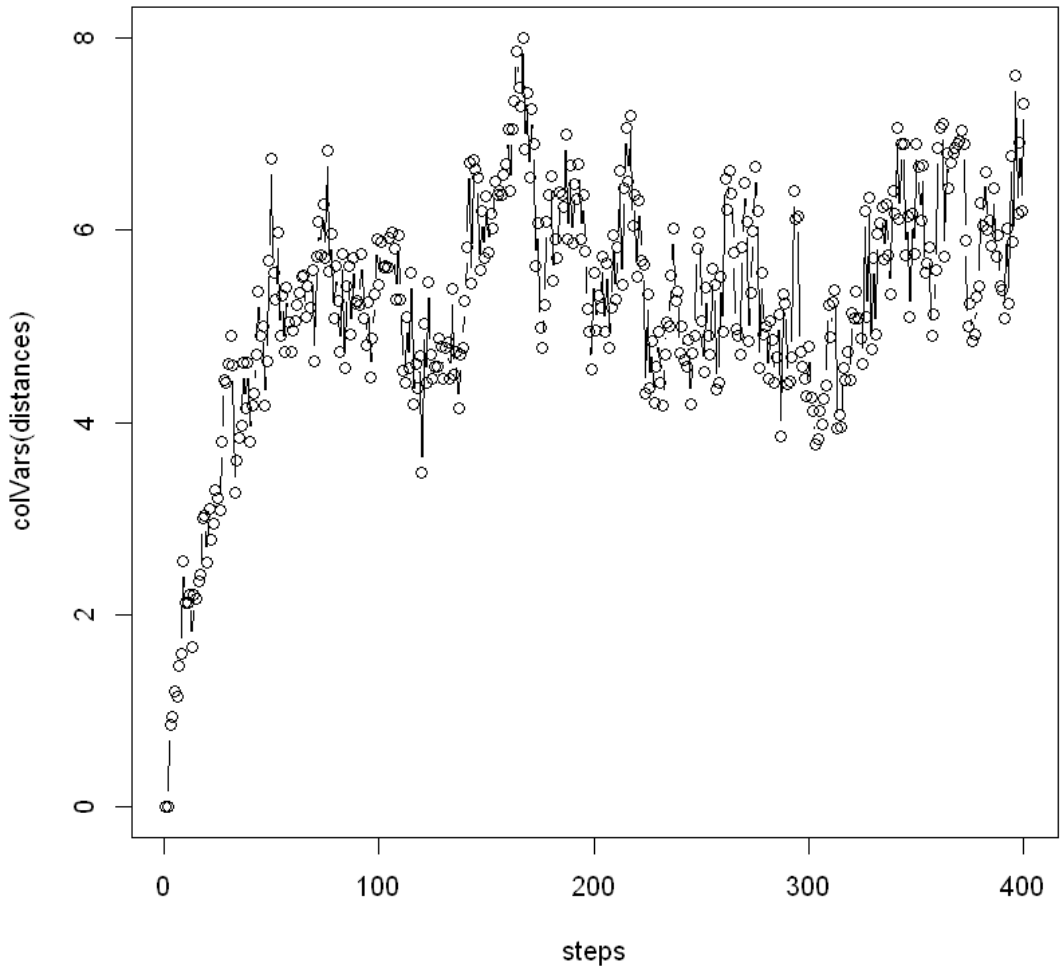
```
distances = matrix(NA, ncol=num_steps, nrow=num_trials)
# Initialize vector of end nodes
end_nodes = vector(mode="numeric",length=num_trials)


# Run the random walks
for(i in 1:num_trials){
    # Randomly pick a start node
    start_node <- sample(1:vcount(gcc2), 1)
    list[stepwise_distances, v] = random_walk(gcc2, num_steps, start_node)
    # Add stepwise_distances as column i in the matrix
    distances[i,] = stepwise_distances
    end_nodes[i] = v

}
```

[15]:
```
plot(steps,colMeans(distances),type="b")
plot(steps,colVars(distances),type="b")

png(file="plots/2_2_d_10000m.png", width=600, height=450)

plot(steps,colMeans(distances),type="b")

dev.off()
png(file="plots/2_2_d_10000v.png", width=600, height=450)

plot(steps,colVars(distances),type="b")

dev.off()
```



png: 2

png: 2



[16]:
```
print("Diameters: n=100,1000,10000")
print(diameter(g3))
print(diameter(g))
print(diameter(g2))
```

```
[1] "Diameters: n=100,1000,10000"
[1] 13
[1] 32
[1] 56
```

Looks like diameter has minimal impact to variances, and negligible impact to means.

[ ]:

2_3_2_4

April 15, 2021

1 Import libs

[1]:
```
library('igraph')
library('Matrix')
library('pracma')
```

```
Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union


Attaching package: 'pracma'

The following objects are masked from 'package:Matrix':

    expm, lu, tril, triu
```

2 P 2.3 - PageRank

a) We are going to create a directed random network with 1000 nodes, using the preferential attachment model. Note that in a directed preferential attachment network, the out-degree of every node is m, while the in-degrees follow a power law distribution. One problem of performing random walk in such a network is that, the very first node will have no outbounding edges, and be a "black hole" which a random walker can never "escape" from. To address that, let's generate another 1000-node random network with preferential attachment model, and merge the two networks by adding the edges of the second graph to the first graph with a shuffling of the indices of the nodes. Create such a network using $m = 4$. Measure the probability that the walker visits each node. Is this probability related to the degree of the nodes?

[2]:
```
create_graph <- function(num_nodes=1000, custom_m=4) {
    # function that creates graphs for questions 2.3 and 2.4

    # create 2 random directed networks using PA model
    g_1dir <- sample_pa(num_nodes, m=custom_m, directed=TRUE)
    g_2dir <- sample_pa(num_nodes, m=custom_m, directed=TRUE)

    # create a vector of indices of length num_nodes
    idx <- seq(from = 1, to = num_nodes, by = 1)
    # permute the indices randomly
    idx <- sample(idx)
    # if permutation did not help
    while (idx[1] == 1) {
        # permute the indices randomly
        idx <- sample(idx)
    }
    # permute vertices of the second graph according to the indices vector
    g_2dir <- permute(g_2dir, idx)
    # get edges as a !proper! list
    g_2dir_edgelist <- c(t(as_edgelist(g_2dir)))

    # add new edges to the graph
    g_1dir_new <- add_edges(g_1dir, g_2dir_edgelist)

    return(g_1dir_new)
}
```

[25]:
```
calculate_probabilities <- function(graph, trials, steps, alpha,␣
↪teleportation_probs, deg_mode="in") {
    # the function calculates probabilities of visiting through random walk for␣
↪each node
    # input variables:
    # graph:                    graph for which we will calculate probabilities
    # trials:                   the number of times we will init a completely new
    # steps:                    number of steps to perform at each iteration
    # alpha:                    probability of teleportation
    # teleportation_probs:      probabilities to teleport to each node

    # returns:
    # probs_vs_degree:          a vector showing average prbability of visiting␣
↪for a node with corresponding degree

    # calculate graph size
    num_nodes <- gorder(graph)
    # init array of node names
    node_names <- seq(from = 1, to = num_nodes, by = 1)
    # init array of visitis to each node
```

```
    visits <- c(zeros(1, num_nodes))

    # iterate "trials" times
    for (trial in 1:trials) {
        # choose starting point
        current_node <- sample(node_names, 1)
        # add visits to the array
        visits[current_node] <- visits[current_node] + 1
        # iterate "steps times"
        for (step in 1:steps) {
            # choose whether to teleport
            if (runif(1) > alpha) {
                # locate node's neighbours
                nbs <- neighbors(graph, current_node, mode = c("out"))
                # perform odrinary step
                current_node <- sample(nbs, 1)
            } else {
                # perform teleportation
                current_node <- sample(node_names, 1, prob =␣
↪teleportation_probs)
            }
            # add visits to the array
            visits[current_node] <- visits[current_node] + 1
        }
    }

    # get probability of visiting each node
    visits <- visits / (trials * (1 + steps))
    # init array of probabilities as a function of degree
    probs_vs_degree <- c(zeros(1, num_nodes))
    # now for each degree we need to find all nodes with that degree and␣
↪calculate average
    # probability of visiting said node; for that, init node counter
    deg_vs_nodes_counter <- c(zeros(1, num_nodes))
    # iterate through all nodes to find their degree and visit probability
    for (i in 1:num_nodes) {
        probs_vs_degree[degree(graph, v=i, mode=deg_mode)] <-␣
↪probs_vs_degree[degree(graph, v=i, mode=deg_mode)] + visits[i]
        deg_vs_nodes_counter[degree(graph, v=i, mode=deg_mode)] <-␣
↪deg_vs_nodes_counter[degree(graph, v=i, mode=deg_mode)] + 1
    }
    # now if there is at least one node with specified degree average␣
↪probability over the number of nodes
    for (i in 1:num_nodes) {
        if (deg_vs_nodes_counter[i] != 0) {
            probs_vs_degree[i] <- probs_vs_degree[i] / deg_vs_nodes_counter[i]
        }
```

```
    }

    # return the result
    return(probs_vs_degree)
}
```

[4]:
```
generate_mask <- function(vector_w_zeros) {
    # function that generates mask, which removes zero values from the vector

    # find which indices we need to remove
    to_remove <- which(vector_w_zeros == 0)
    # initialize mask
    mask <- c(ones(1, length(vector_w_zeros)))
    # mark elements we need to remove in the mask
    for (i in 1:length(vector_w_zeros)) {
        if (is.element(i, to_remove)) {
            mask[i] <- 0
        }
    }
    # convert mask to boolean
    mask <- mask > 0.5

    return(mask)
}
```

Let's gennerate one example graph to see how different algorithms affect degree distribution in its particular case

[53]:
```
example_graph <- create_graph()
```

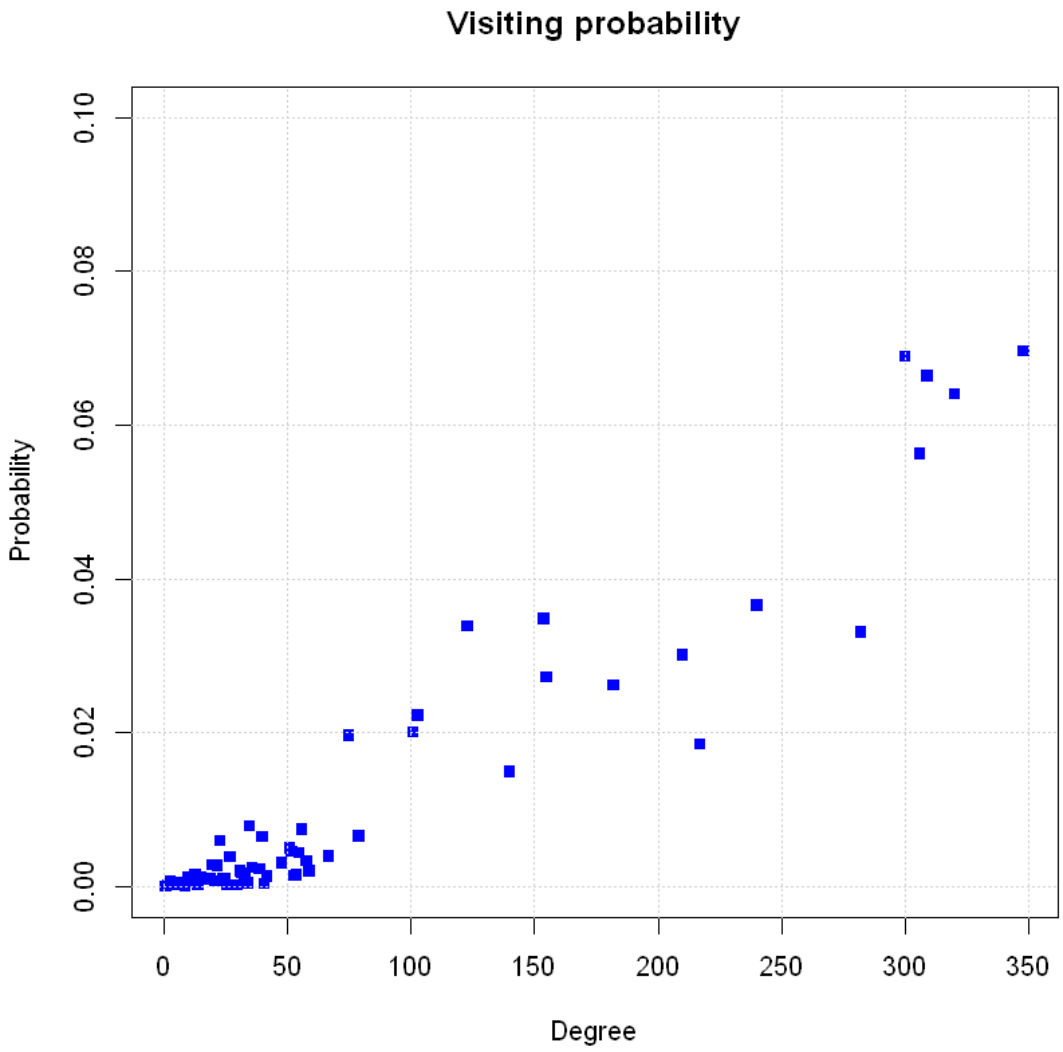
Let's see how random walk performs on that particular graph

[54]:
```
# set number of trials for each graph
num_trails <- 50
# set number of steps
num_steps <- 500
# set teleportation probability
alpha <- 0
# calculate probabilities of visit
visiting_probs_default <- calculate_probabilities(example_graph, num_trails,␣
 ↪num_steps, alpha, c())
```

[61]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(visiting_probs_default)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
```

```
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

grid()

# save plot
png(file="plots/2_3_a-1.png", width=600, height=450)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

grid()

dev.off()
```

png: 2



Note: when I tried to run for 5000 iteration, the results were about the same, and the highest difference in probability was less than 0.01. For the sake of saving time, we will run for 500 iterations from now on.

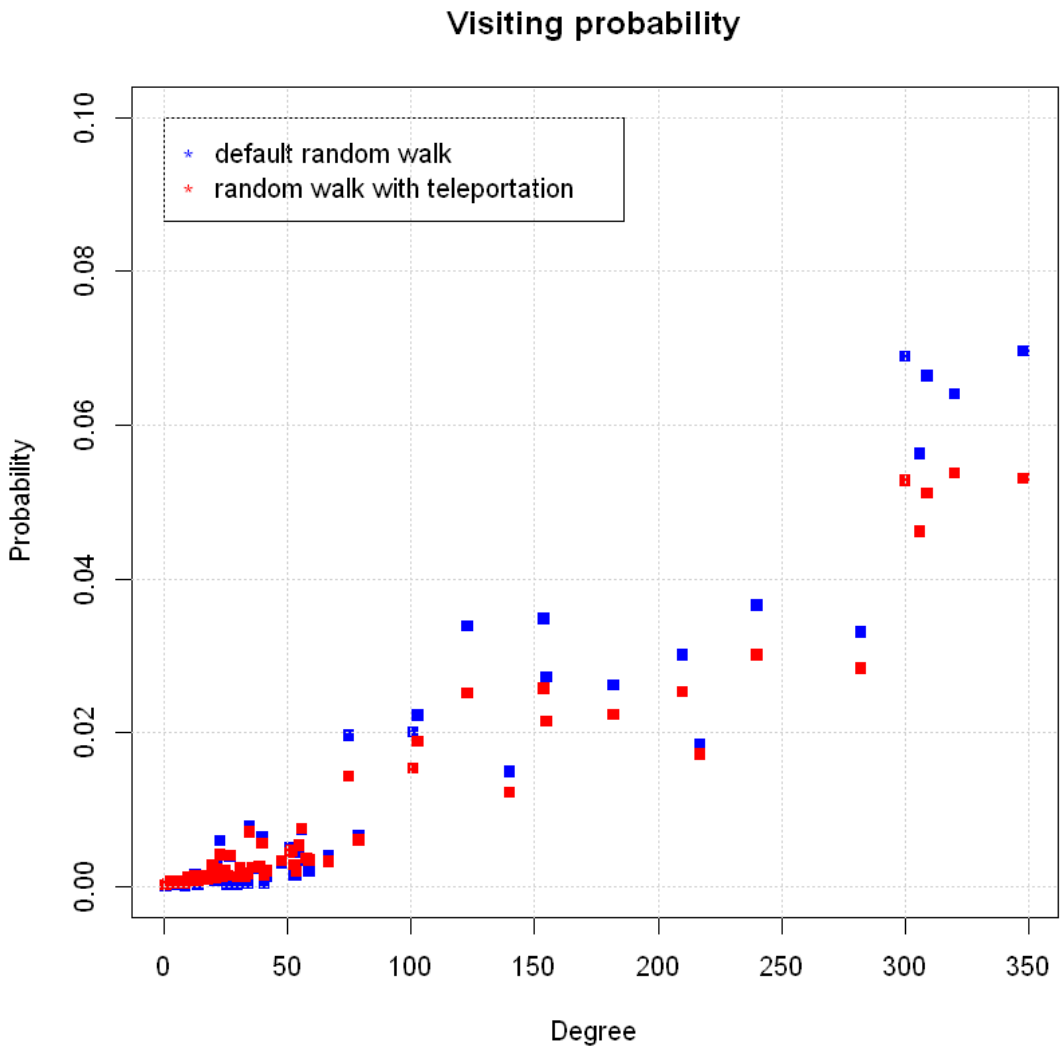
b) In all previous questions, we didn't have any teleportation. Now, we use a teleportation probability of $\alpha = 0.15$. By performing random walks on the network created in 3(a), measure the probability that the walker visits each node. Is this probability related to the degree of the node?

[56]:
```
# set number of trials for each graph
num_trails <- 50
# set number of steps
num_steps <- 500
# set teleportation probability
alpha <- 0.15
```

```
# create a set of unifrom probabilities
uni <- c(ones(1, 1000))
# calculate probabilities of visit
visiting_probs_teleport <- calculate_probabilities(example_graph, num_trails,␣
↪num_steps, alpha, uni)
```

[62]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(visiting_probs_teleport)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".", cex=7)
legend(0, 0.1, legend=c("default random walk", "random walk with␣
↪teleportation"),
       col=c("blue", "red"), pch="*")
grid()

# save plot
png(file="plots/2_3_b-1.png", width=600, height=450)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=4,
     ylim=c(0, 0.1))

points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".", cex=4)
legend(0, 0.1, legend=c("default random walk", "random walk with␣
↪teleportation"),
       col=c("blue", "red"), pch="*")
grid()

dev.off()
```

png: 2



From the plot above, we can see that adding teleportation decreses visiting probability of the nodes with the high probability. This is because instead looping around these center node, the walker will sometimes teleport to the outskirts of the graph, so the number of visits to the nodes with very small degree also grows.

3 P 2.4 - Personalized PageRank - previous version

a) Suppose you have your own notion of importance. Your interest in a node is proportional to the node's PageRank, because you totally rely upon Google to decide which website to visit (assume that these nodes represent websites). Again, use random walk on network generated in question 3 to simulate this personalized PageRank. Here the teleportation probability to each node is proportional to its PageRank (as opposed to the regular PageRank, where at teleportation, the chance of visiting all nodes are the same and equal to $\frac{1}{N}$. Again, let the teleportation probability

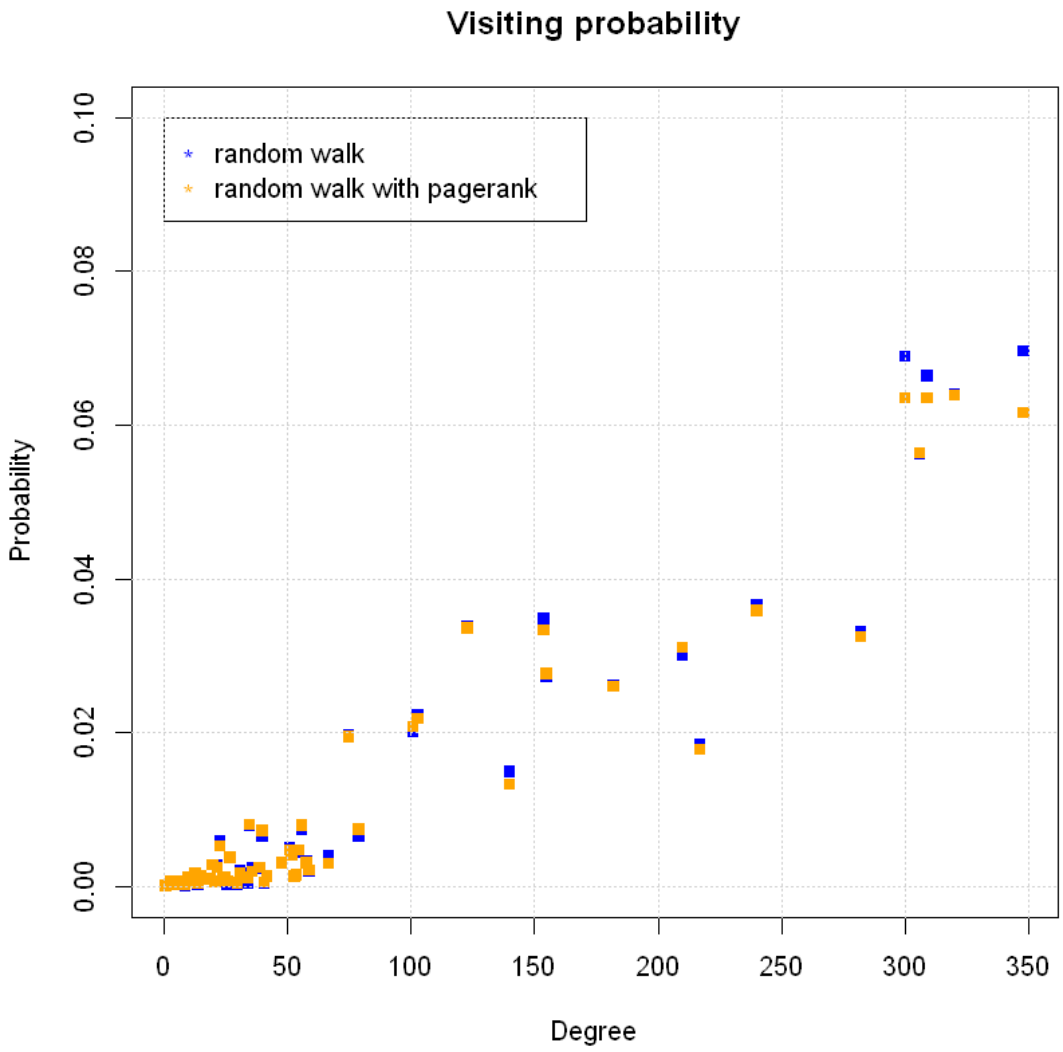
be equal to $\alpha = 0.15$. Compare the results with 3a).

[58]:
```r
# set number of trials for each graph
num_trails <- 50
# set number of steps
num_steps <- 500
# set teleportation probability
alpha <- 0.15

# find pagerank (with teleportation)
pr <- page_rank(example_graph)
# unpack pagerank
pr <- as.numeric(unlist(pr[1][seq(from = 1, to = 1000, by = 1)]))
# normilize the scores, for them to become probabilities
pr <- pr / sum(pr)

# calculate probabilities for current graph
visiting_probs_pagerank <- calculate_probabilities(example_graph, num_trails,␣
↪num_steps, alpha, pr)
```

[78]:
```r
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(visiting_probs_teleport)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

# points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".",␣
↪cex=4)
points(degrees[mask], visiting_probs_pagerank[mask], col="orange", pch=".",␣
↪cex=7)

legend(0, 0.1, legend=c("random walk", "random walk with pagerank"),
       col=c("blue", "orange"), pch="*")
grid()

# save plot
png(file="plots/2_4_a-1.png", width=600, height=450)

plot(degrees[mask], visiting_probs_default[mask],
     main="Visiting probability",
```

```
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

# points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".",␣
 ↪cex=4)
points(degrees[mask], visiting_probs_pagerank[mask], col="orange", pch=".",␣
 ↪cex=7)

legend(0, 0.1, legend=c("random walk", "random walk with pagerank"),
       col=c("blue", "orange"), pch="*")
grid()

dev.off()
```

png: 2



We can see that teleportation in accordance with pagerank scores (note: pagerank is computed for teleportation probability of $\alpha = 0.15$, because computation for $\alpha = 0$ isn't possibble for most of the graphs) strongly increases visiting probbability of center node and several nodes adjusted to it, while decreasing probability of all other ones. This is because now the central node has the highest chance to being teleported to, so even if a walker randomly entered outskirts it gets returned to the center. This negatively affects visiting chance of all other nodes.

b) Find two nodes in the network with median PageRanks. Repeat part 4(a) if teleportations land only on those two nodes (with probabilities 1/2, 1/2). How are the PageRank values affected?

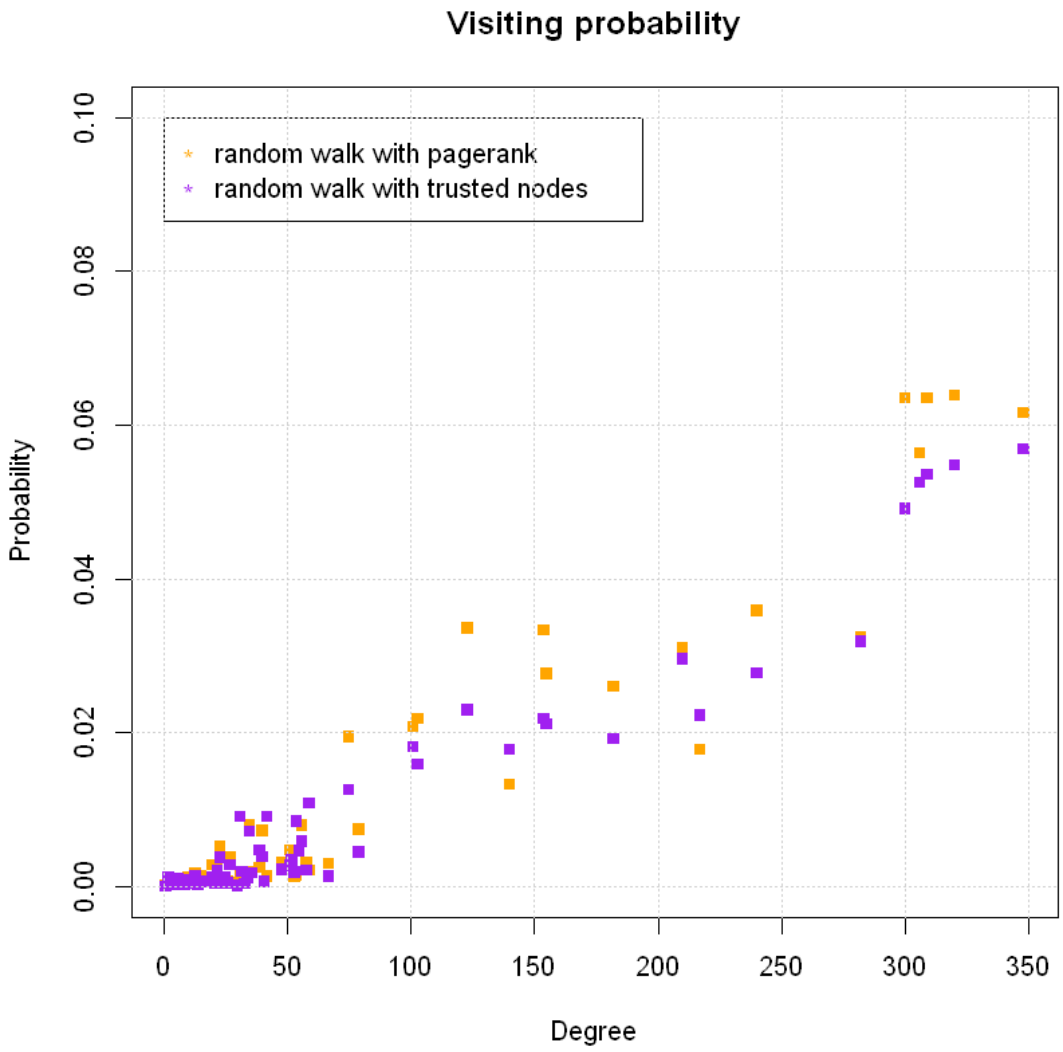
[59]:
```
# set number of trials for each graph
num_trails <- 50
# set number of steps
num_steps <- 500
```

```
# set teleportation probability
alpha <- 0.15

# find pagerank (with teleportation)
pr <- page_rank(example_graph)
# unpack pagerank
pr <- as.numeric(unlist(pr[1][seq(from = 1, to = 1000, by = 1)]))
# sort pagerank scores
sorted_idxs <- order(pr)
# init array for personalized pagerank
personalized_pr <- c(zeros(1, 1000))
# set median probabilities to 1/2
personalized_pr[sorted_idxs[500]] = 1/2
personalized_pr[sorted_idxs[501]] = 1/2

# calculate probabilities for current graph
visiting_probs_personalized_pagerank <- calculate_probabilities(example_graph,␣
↪num_trails, num_steps, alpha, personalized_pr)
```

[64]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(visiting_probs_teleport)

plot(degrees[mask], visiting_probs_pagerank[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="orange",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

# points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".",␣
↪cex=4)
points(degrees[mask], visiting_probs_personalized_pagerank[mask], col="purple",␣
↪pch=".", cex=7)

legend(0, 0.1, legend=c("random walk with pagerank", "random walk with trusted␣
↪nodes"),
       col=c("orange", "purple"), pch="*")
grid()

# save plot
png(file="plots/2_4_b-1.png", width=600, height=450)

plot(degrees[mask], visiting_probs_pagerank[mask],
     main="Visiting probability",
```

```
     xlab="Degree",
     ylab="Probability",
     col="orange",
     pch=".",
     cex=7,
     ylim=c(0, 0.1))

# points(degrees[mask], visiting_probs_teleport[mask], col="red", pch=".",␣
 ↪cex=4)
points(degrees[mask], visiting_probs_personalized_pagerank[mask], col="purple",␣
 ↪pch=".", cex=7)

legend(0, 0.1, legend=c("random walk with pagerank", "random walk with trusted␣
 ↪nodes"),
       col=c("orange", "purple"), pch="*")
grid()

dev.off()
```

png: 2



Teleportation to trusted nodes punishes the nodes with highest visiting probabilities (as algorithm does not work in their favor anymore), while increasing probabilities of visit for nodes in the middle. This is because trusted nodes had probabilities of visiting somewhere in the middle, similar to nodes connected to them. The nodes with small degrees obviously have low chance of being visited.

Now let's generate $\sim$ 1000 (change after test) random graphs and see if that trend persists on average

[65]:
```
# set number of graphs
num_graphs <- 3000
# set number of trials for each graph
num_trails <- 2
# set number of steps
num_steps <- 500
```

```
print("starting")
flush.console()

# init visiting probabilities as a function of degree for all methods
avg_visiting_probs_default <- zeros(1, 1000)
avg_visiting_probs_teleport <- zeros(1, 1000)
avg_visiting_probs_pagerank <- zeros(1, 1000)
avg_visiting_probs_personalized_pagerank <- zeros(1, 1000)
# init counter to find out how often nodes with specified degree were present␣
 ↪in the graphs
degree_counter <- zeros(1, 1000)

# time the code
ptm <- proc.time()

for (h in 1:num_graphs) {

    # create graph
    temp_g <- create_graph()

    # create a set of unifrom probabilities
    uni <- c(ones(1, 1000)) / 1000
    # find pagerank (with teleportation)
    pr <- page_rank(temp_g)
    # unpack pagerank
    pr <- as.numeric(unlist(pr[1][seq(from = 1, to = 1000, by = 1)]))
    # normilize the scores, for them to become probabilities
    pr <- pr / sum(pr)
    # sort pagerank scores
    sorted_idxs <- order(pr)
    # init array for personalized pagerank
    personalized_pr <- c(zeros(1, 1000))
    # set median probabilities to 1/2
    personalized_pr[sorted_idxs[500]] = 1/2
    personalized_pr[sorted_idxs[501]] = 1/2

    # calculate probabilities for current graph
    avg_visiting_probs_default <- avg_visiting_probs_default +␣
 ↪calculate_probabilities(temp_g, num_trails, num_steps, 0, c())
    avg_visiting_probs_teleport <- avg_visiting_probs_teleport +␣
 ↪calculate_probabilities(temp_g, num_trails, num_steps, 0.15, uni)
    avg_visiting_probs_pagerank <- avg_visiting_probs_pagerank +␣
 ↪calculate_probabilities(temp_g, num_trails, num_steps, 0.15, pr)
```

```
    avg_visiting_probs_personalized_pagerank <-␣
↪avg_visiting_probs_personalized_pagerank + calculate_probabilities(temp_g,␣
↪num_trails, num_steps, 0.15, personalized_pr)

    # check which degrees were present in the graph
    degs <- degree_distribution(temp_g, mode="in") * 1000
    for (i in 1:length(degs)) {
        if (degs[i] > 0) {
            degree_counter[i] <- degree_counter[i] + 1
        }
    }


    if (h %% 25 == 0) {
        # do not forget to flush console to be able to print inside the loop
        print(c("calculating for graph", h))
        flush.console()

        # print out time
        print(proc.time() - ptm)
        flush.console()


        # time the code
        ptm <- proc.time()
    }


}


# normilize probs
for (i in 1:1000) {
    if (degree_counter[i] > 0) {
        avg_visiting_probs_default[i] <- avg_visiting_probs_default[i] /␣
↪degree_counter[i]
        avg_visiting_probs_teleport[i] <- avg_visiting_probs_teleport[i] /␣
↪degree_counter[i]
        avg_visiting_probs_pagerank[i] <- avg_visiting_probs_pagerank[i] /␣
↪degree_counter[i]
        avg_visiting_probs_personalized_pagerank[i] <-␣
↪avg_visiting_probs_personalized_pagerank[i] / degree_counter[i]
    }
}
```

```
[1] "starting"
[1] "calculating for graph" "25"
```

```
   user  system elapsed
 178.26    3.38  181.90
[1] "calculating for graph" "50"
   user  system elapsed
 172.79    3.30  176.35
[1] "calculating for graph" "75"
   user  system elapsed
 183.18    3.70  187.14
[1] "calculating for graph" "100"
   user  system elapsed
 198.92    3.28  202.78
[1] "calculating for graph" "125"
   user  system elapsed
 223.66    3.53  227.84
[1] "calculating for graph" "150"
   user  system elapsed
 196.31    3.17  200.00
[1] "calculating for graph" "175"
   user  system elapsed
 184.91    3.36  188.58
[1] "calculating for graph" "200"
   user  system elapsed
 191.27    3.69  195.25
[1] "calculating for graph" "225"
   user  system elapsed
 188.93    3.44  193.42
[1] "calculating for graph" "250"
   user  system elapsed
 176.74    3.31  180.66
[1] "calculating for graph" "275"
   user  system elapsed
 184.70    3.28  188.83
[1] "calculating for graph" "300"
   user  system elapsed
 169.72    3.30  173.20
[1] "calculating for graph" "325"
   user  system elapsed
 191.87    3.26  195.37
[1] "calculating for graph" "350"
   user  system elapsed
 167.25    2.83  170.39
[1] "calculating for graph" "375"
   user  system elapsed
 157.64    2.99  160.71
[1] "calculating for graph" "400"
   user  system elapsed
 172.96    3.26  176.36
[1] "calculating for graph" "425"
```

```
   user  system elapsed
 171.57    3.25  174.96
[1] "calculating for graph" "450"
   user  system elapsed
 147.61    3.16  150.90
[1] "calculating for graph" "475"
   user  system elapsed
 140.79    2.56  143.42
[1] "calculating for graph" "500"
   user  system elapsed
 143.42    2.97  146.57
[1] "calculating for graph" "525"
   user  system elapsed
 150.68    3.08  153.79
[1] "calculating for graph" "550"
   user  system elapsed
 146.86    2.50  149.52
[1] "calculating for graph" "575"
   user  system elapsed
 144.07    2.61  146.87
[1] "calculating for graph" "600"
   user  system elapsed
 142.68    2.84  145.52
[1] "calculating for graph" "625"
   user  system elapsed
 144.28    3.44  147.92
[1] "calculating for graph" "650"
   user  system elapsed
 144.34    2.64  147.05
[1] "calculating for graph" "675"
   user  system elapsed
 138.33    2.86  141.34
[1] "calculating for graph" "700"
   user  system elapsed
 135.94    2.83  138.89
[1] "calculating for graph" "725"
   user  system elapsed
 132.26    2.73  135.03
[1] "calculating for graph" "750"
   user  system elapsed
 134.14    2.49  136.68
[1] "calculating for graph" "775"
   user  system elapsed
 133.35    2.78  136.16
[1] "calculating for graph" "800"
   user  system elapsed
 132.39    2.90  135.36
[1] "calculating for graph" "825"
```

```
   user  system elapsed
 133.08    2.69  135.82
[1] "calculating for graph" "850"
   user  system elapsed
 131.40    2.83  134.25
[1] "calculating for graph" "875"
   user  system elapsed
 132.16    2.84  135.05
[1] "calculating for graph" "900"
   user  system elapsed
 132.34    2.71  135.06
[1] "calculating for graph" "925"
   user  system elapsed
 131.96    2.76  134.75
[1] "calculating for graph" "950"
   user  system elapsed
 132.59    2.91  135.54
[1] "calculating for graph" "975"
   user  system elapsed
 131.78    2.69  134.46
[1] "calculating for graph" "1000"
   user  system elapsed
 132.58    2.75  135.36
[1] "calculating for graph" "1025"
   user  system elapsed
 132.30    2.65  134.97
[1] "calculating for graph" "1050"
   user  system elapsed
 132.42    2.86  135.30
[1] "calculating for graph" "1075"
   user  system elapsed
 132.36    2.88  135.23
[1] "calculating for graph" "1100"
   user  system elapsed
 132.59    3.20  135.86
[1] "calculating for graph" "1125"
   user  system elapsed
 131.36    2.94  134.32
[1] "calculating for graph" "1150"
   user  system elapsed
 132.28    2.81  135.12
[1] "calculating for graph" "1175"
   user  system elapsed
 132.03    2.80  134.89
[1] "calculating for graph" "1200"
   user  system elapsed
 132.00    2.82  134.88
[1] "calculating for graph" "1225"
```

```
   user  system elapsed
 132.30    2.71  135.03
[1] "calculating for graph" "1250"
   user  system elapsed
 133.25    2.68  136.01
[1] "calculating for graph" "1275"
   user  system elapsed
 132.37    2.83  135.21
[1] "calculating for graph" "1300"
   user  system elapsed
 131.96    3.07  135.03
[1] "calculating for graph" "1325"
   user  system elapsed
 132.29    2.85  135.17
[1] "calculating for graph" "1350"
   user  system elapsed
 132.57    2.65  135.22
[1] "calculating for graph" "1375"
   user  system elapsed
 132.81    2.93  135.79
[1] "calculating for graph" "1400"
   user  system elapsed
 132.91    2.64  135.63
[1] "calculating for graph" "1425"
   user  system elapsed
 131.85    3.30  135.20
[1] "calculating for graph" "1450"
   user  system elapsed
 132.08    2.59  134.69
[1] "calculating for graph" "1475"
   user  system elapsed
 133.07    2.94  136.00
[1] "calculating for graph" "1500"
   user  system elapsed
 132.43    2.94  135.39
[1] "calculating for graph" "1525"
   user  system elapsed
 132.19    2.94  135.17
[1] "calculating for graph" "1550"
   user  system elapsed
 132.70    3.03  135.86
[1] "calculating for graph" "1575"
   user  system elapsed
 132.28    2.59  134.91
[1] "calculating for graph" "1600"
   user  system elapsed
 132.61    2.89  135.58
[1] "calculating for graph" "1625"
```

```
   user  system elapsed
 131.66    2.86  134.51
[1] "calculating for graph" "1650"
   user  system elapsed
 133.05    3.02  136.08
[1] "calculating for graph" "1675"
   user  system elapsed
 132.04    2.89  135.00
[1] "calculating for graph" "1700"
   user  system elapsed
 133.04    3.08  136.16
[1] "calculating for graph" "1725"
   user  system elapsed
 132.06    2.75  134.82
[1] "calculating for graph" "1750"
   user  system elapsed
 133.72    2.46  136.21
[1] "calculating for graph" "1775"
   user  system elapsed
 132.37    3.15  135.56
[1] "calculating for graph" "1800"
   user  system elapsed
 132.20    2.67  134.92
[1] "calculating for graph" "1825"
   user  system elapsed
 132.05    2.89  134.96
[1] "calculating for graph" "1850"
   user  system elapsed
 133.03    2.67  135.76
[1] "calculating for graph" "1875"
   user  system elapsed
 132.92    2.45  135.42
[1] "calculating for graph" "1900"
   user  system elapsed
 132.21    3.03  135.27
[1] "calculating for graph" "1925"
   user  system elapsed
 133.42    2.74  136.19
[1] "calculating for graph" "1950"
   user  system elapsed
 131.56    2.86  134.42
[1] "calculating for graph" "1975"
   user  system elapsed
 133.52    2.51  136.03
[1] "calculating for graph" "2000"
   user  system elapsed
 133.08    2.68  135.80
[1] "calculating for graph" "2025"
```

```
   user  system elapsed
 131.57    2.57  134.17
[1] "calculating for graph" "2050"
   user  system elapsed
 132.51    2.91  135.47
[1] "calculating for graph" "2075"
   user  system elapsed
 131.99    2.45  134.45
[1] "calculating for graph" "2100"
   user  system elapsed
 133.73    2.41  136.16
[1] "calculating for graph" "2125"
   user  system elapsed
 132.57    2.64  135.32
[1] "calculating for graph" "2150"
   user  system elapsed
 133.66    2.27  135.93
[1] "calculating for graph" "2175"
   user  system elapsed
 132.09    2.73  134.85
[1] "calculating for graph" "2200"
   user  system elapsed
 132.71    2.77  135.50
[1] "calculating for graph" "2225"
   user  system elapsed
 132.59    2.76  135.36
[1] "calculating for graph" "2250"
   user  system elapsed
 131.57    3.08  134.67
[1] "calculating for graph" "2275"
   user  system elapsed
 132.35    2.75  135.10
[1] "calculating for graph" "2300"
   user  system elapsed
 132.22    2.91  135.18
[1] "calculating for graph" "2325"
   user  system elapsed
 132.89    2.95  135.88
[1] "calculating for graph" "2350"
   user  system elapsed
 132.45    2.76  135.23
[1] "calculating for graph" "2375"
   user  system elapsed
 132.88    2.93  135.82
[1] "calculating for graph" "2400"
   user  system elapsed
 132.56    2.89  135.45
[1] "calculating for graph" "2425"
```

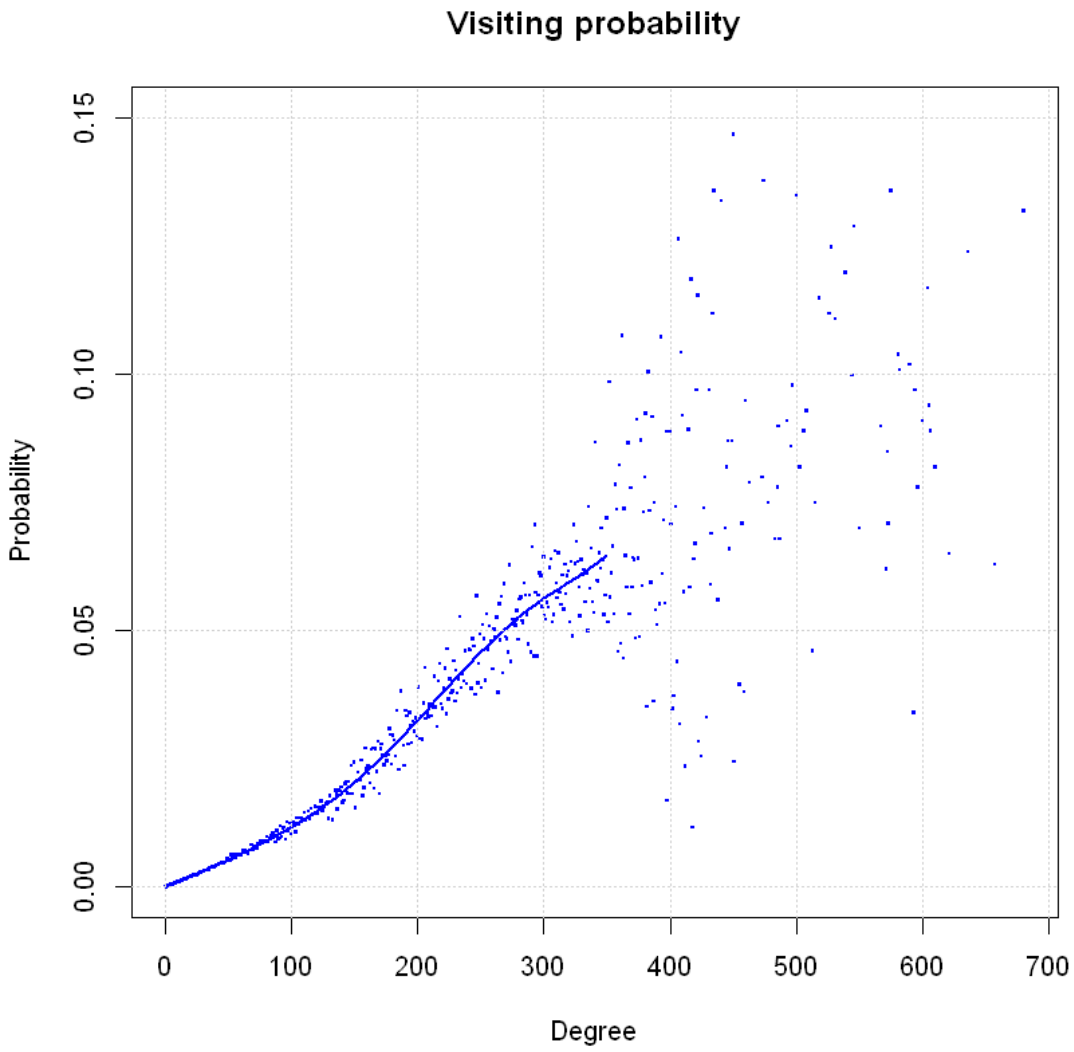
```
   user  system elapsed
 133.11    2.65  135.78
[1] "calculating for graph" "2450"
   user  system elapsed
 131.64    3.09  134.77
[1] "calculating for graph" "2475"
   user  system elapsed
 132.16    3.05  135.23
[1] "calculating for graph" "2500"
   user  system elapsed
 131.89    3.31  135.24
[1] "calculating for graph" "2525"
   user  system elapsed
 132.04    3.14  135.22
[1] "calculating for graph" "2550"
   user  system elapsed
 132.17    2.99  135.18
[1] "calculating for graph" "2575"
   user  system elapsed
 132.49    2.93  135.42
[1] "calculating for graph" "2600"
   user  system elapsed
 132.17    3.18  135.35
[1] "calculating for graph" "2625"
   user  system elapsed
 131.55    3.15  134.70
[1] "calculating for graph" "2650"
   user  system elapsed
 133.01    3.02  136.05
[1] "calculating for graph" "2675"
   user  system elapsed
 132.43    2.95  135.37
[1] "calculating for graph" "2700"
   user  system elapsed
 131.76    2.97  134.75
[1] "calculating for graph" "2725"
   user  system elapsed
 132.42    2.75  135.19
[1] "calculating for graph" "2750"
   user  system elapsed
 132.24    2.64  134.89
[1] "calculating for graph" "2775"
   user  system elapsed
 132.78    2.55  135.33
[1] "calculating for graph" "2800"
   user  system elapsed
 132.61    2.87  135.52
[1] "calculating for graph" "2825"
```

```
   user  system elapsed
 132.31    3.17  135.53
[1] "calculating for graph" "2850"
   user  system elapsed
 132.14    2.89  135.03
[1] "calculating for graph" "2875"
   user  system elapsed
 133.49    2.83  136.34
[1] "calculating for graph" "2900"
   user  system elapsed
 131.26    3.08  134.36
[1] "calculating for graph" "2925"
   user  system elapsed
 132.99    2.91  135.91
[1] "calculating for graph" "2950"
   user  system elapsed
 132.28    3.01  135.29
[1] "calculating for graph" "2975"
   user  system elapsed
 131.57    2.97  134.57
[1] "calculating for graph" "3000"
   user  system elapsed
 132.11    3.27  135.40
```

[66]:
```r
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(avg_visiting_probs_default)

plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.15))

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.8)

line1 <- predict(lo1)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)

grid()

# save plot
png(file="plots/2_3_a-2.png", width=600, height=450)
```

```r
plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.15))

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.8)

line1 <- predict(lo1)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)

grid()

dev.off()
```

png: 2

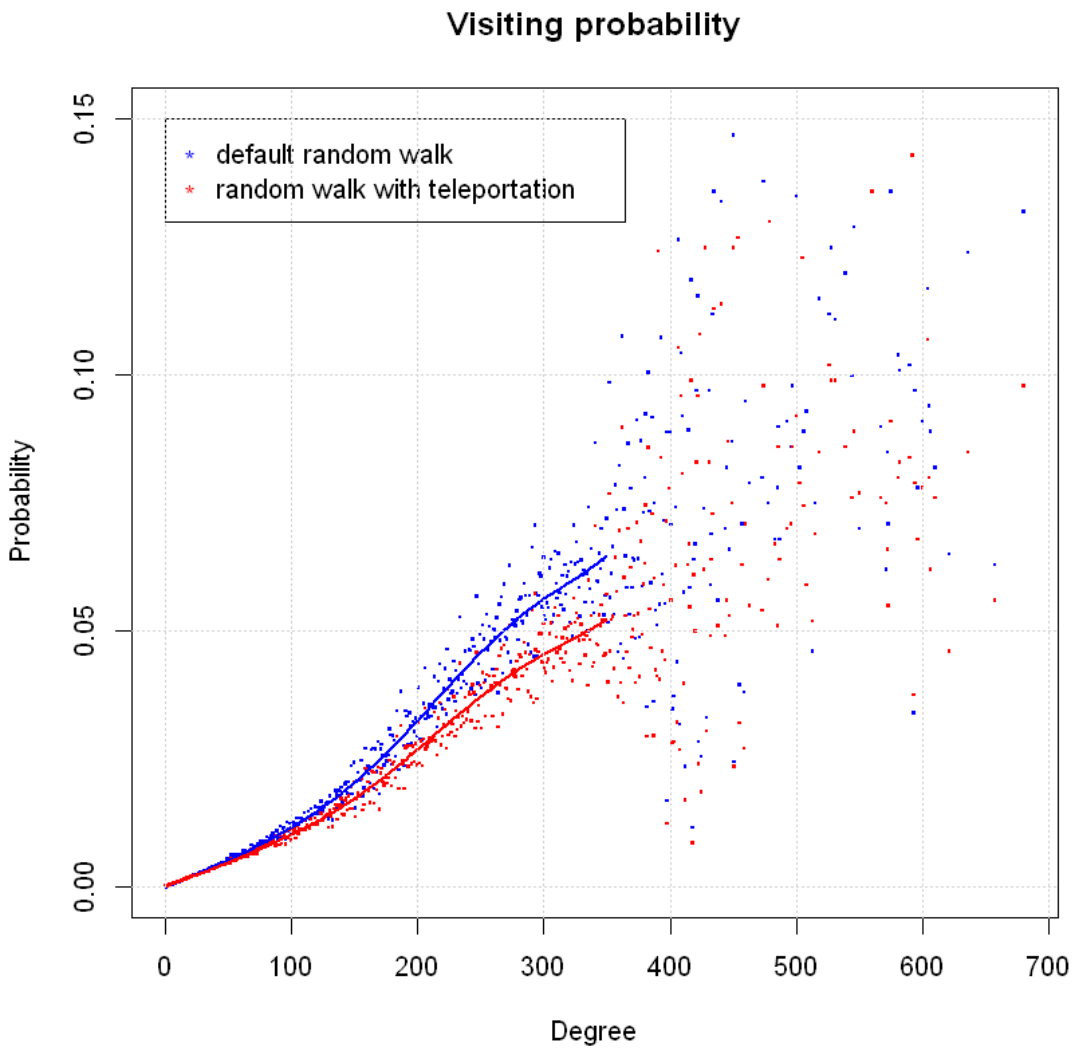


[67]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(avg_visiting_probs_default)

plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.15))
```

```
points(degrees[mask], avg_visiting_probs_teleport[mask], col="red", pch=".",␣
 ↪cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.8)
lo2 <- smooth.spline(degrees[mask], avg_visiting_probs_teleport[mask], spar=0.8)

line1 <- predict(lo1)
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)
lines(predict(lo2)$y[0:350], col='red', lwd=2)

legend(0, 0.15, legend=c("default random walk", "random walk with␣
 ↪teleportation"),
       col=c("blue", "red"), pch="*")

grid()

# save plot
png(file="plots/2_3_b-2.png", width=600, height=450)

plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.15))

points(degrees[mask], avg_visiting_probs_teleport[mask], col="red", pch=".",␣
 ↪cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.8)
lo2 <- smooth.spline(degrees[mask], avg_visiting_probs_teleport[mask], spar=0.8)

line1 <- predict(lo1)
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)
lines(predict(lo2)$y[0:350], col='red', lwd=2)

legend(0, 0.15, legend=c("default random walk", "random walk with␣
 ↪teleportation"),
       col=c("blue", "red"), pch="*")

grid()
```

```
dev.off()
```

png: 2

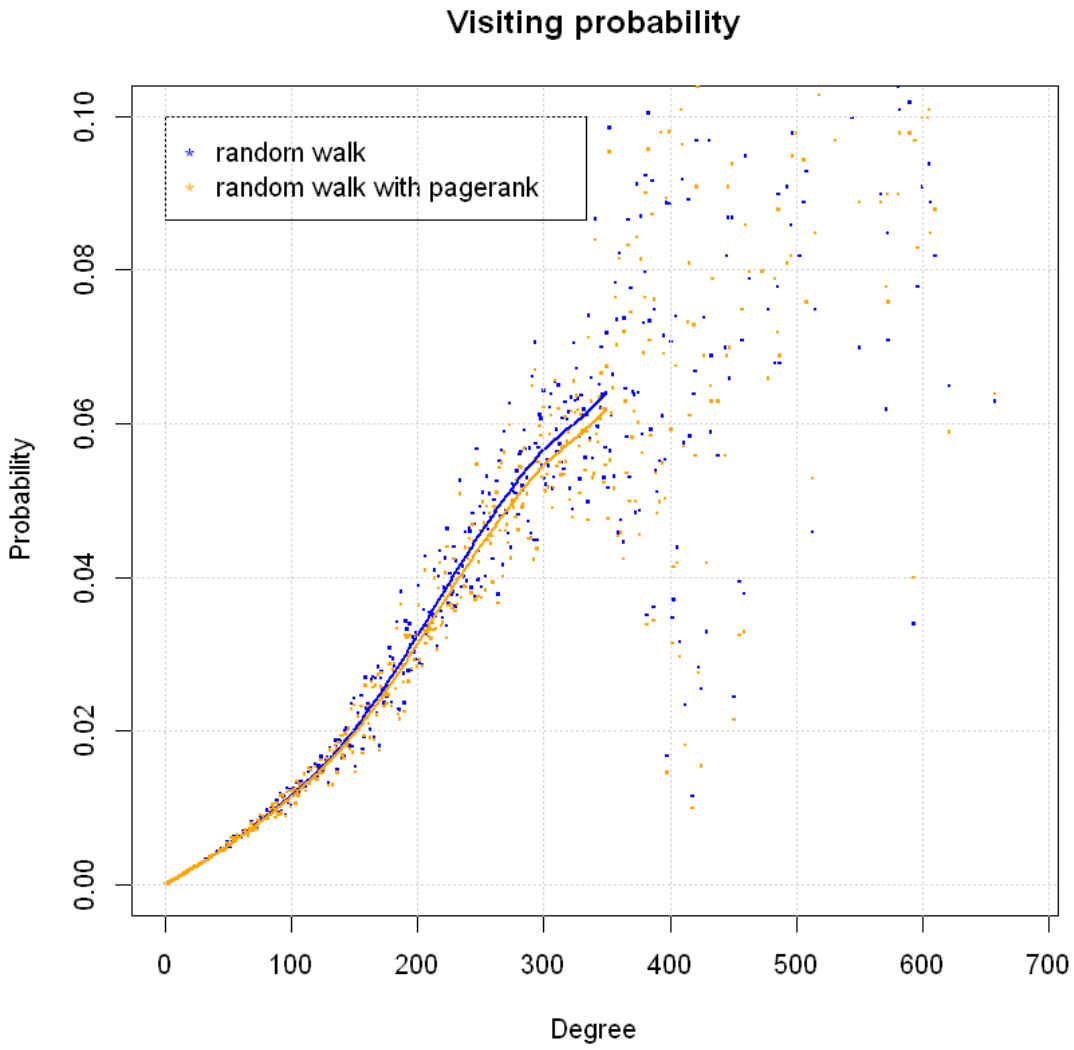


[79]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(avg_visiting_probs_default)

plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
```

```
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.1))

points(degrees[mask], avg_visiting_probs_pagerank[mask], col="orange", pch=".",␣
 ↪cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.7)
lo2 <- smooth.spline(degrees[mask], avg_visiting_probs_pagerank[mask], spar=0.7)

line1 <- predict(lo1)
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)
lines(predict(lo2)$y[0:350], col='orange', lwd=2)

legend(0, 0.1, legend=c("random walk", "random walk with pagerank"),
       col=c("blue", "orange"), pch="*")

grid()

# save plot
png(file="plots/2_4_a-2.png", width=600, height=450)

plot(degrees[mask], avg_visiting_probs_default[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="blue",
     pch=".",
     cex=2,
     ylim=c(0, 0.1))

points(degrees[mask], avg_visiting_probs_pagerank[mask], col="orange", pch=".",␣
 ↪cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_default[mask], spar=0.7)
lo2 <- smooth.spline(degrees[mask], avg_visiting_probs_pagerank[mask], spar=0.7)

line1 <- predict(lo1)
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='blue', lwd=2)
lines(predict(lo2)$y[0:350], col='orange', lwd=2)

legend(0, 0.1, legend=c("random walk", "random walk with pagerank"),
```

```
       col=c("blue", "orange"), pch="*")

grid()

dev.off()
```

png: 2

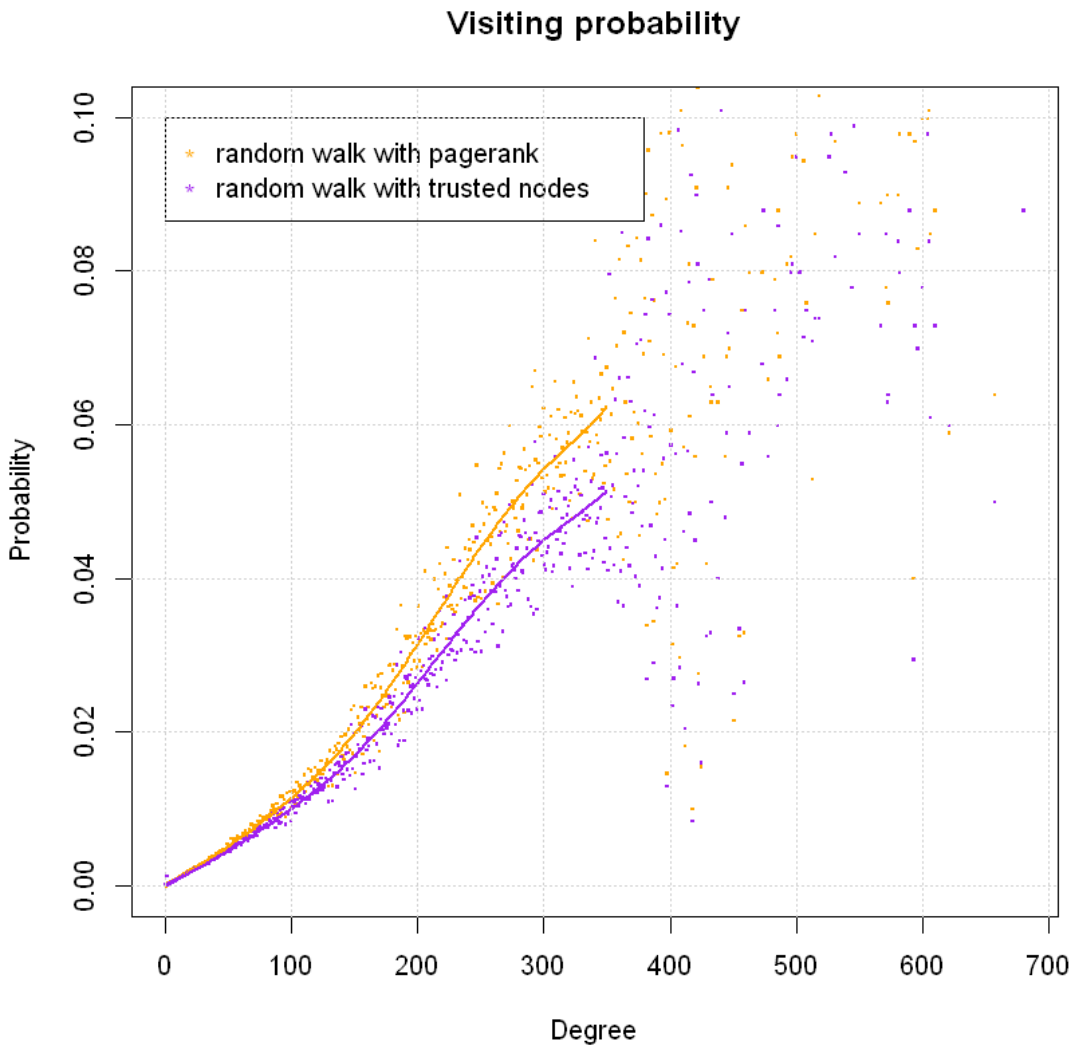


[82]:
```
# convert data to the form acceptable for plotting
degrees <- seq(from=1, to=1000, by=1)
mask <- generate_mask(avg_visiting_probs_default)

plot(degrees[mask], avg_visiting_probs_pagerank[mask],
```

```
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="orange",
     pch=".",
     cex=2,
     ylim=c(0, 0.1))

points(degrees[mask], avg_visiting_probs_personalized_pagerank[mask],␣
↪col="purple", pch=".", cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_pagerank[mask], spar=0.8)
lo2 <- smooth.spline(degrees[mask],␣
↪avg_visiting_probs_personalized_pagerank[mask], spar=0.8)

line1 <- predict(lo1)
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='orange', lwd=2)
lines(predict(lo2)$y[0:350], col='purple', lwd=2)

legend(0, 0.1, legend=c("random walk with pagerank", "random walk with trusted␣
↪nodes"),
       col=c("orange", "purple"), pch="*")

grid()

# save plot
png(file="plots/2_4_b-2.png", width=600, height=450)

plot(degrees[mask], avg_visiting_probs_pagerank[mask],
     main="Visiting probability",
     xlab="Degree",
     ylab="Probability",
     col="orange",
     pch=".",
     cex=2,
     ylim=c(0, 0.1))

points(degrees[mask], avg_visiting_probs_personalized_pagerank[mask],␣
↪col="purple", pch=".", cex=2)

lo1 <- smooth.spline(degrees[mask], avg_visiting_probs_pagerank[mask], spar=0.8)
lo2 <- smooth.spline(degrees[mask],␣
↪avg_visiting_probs_personalized_pagerank[mask], spar=0.8)

line1 <- predict(lo1)
```

```
line2 <- predict(lo2)

lines(predict(lo1)$y[0:350], col='orange', lwd=2)
lines(predict(lo2)$y[0:350], col='purple', lwd=2)

legend(0, 0.1, legend=c("random walk with pagerank", "random walk with trusted␣
 ↪nodes"),
       col=c("orange", "purple"), pch="*")

grid()

dev.off()
```

png: 2



4 Helper functions

[ ]:
```
plot(g_test, edge.arrow.size=.4, edge.curved=.1, vertex.size=1, vertex.label=NA)
```

[64]:
```
temp_sc <- page_rank(temp_g)
temp_sc <- as.numeric(unlist(temp_sc[1][seq(from = 1, to = 1000, by = 1)]))
print(temp_sc)
```

```
  [1] 0.0448673927 0.0338151149 0.0395835031 0.0222509698 0.0266277453
  [6] 0.0138748866 0.0100051961 0.0118842208 0.0056638683 0.0010856615
 [11] 0.0095178880 0.0183740780 0.0033183219 0.0005552176 0.0066008126
 [16] 0.0010499643 0.0028581962 0.0063947474 0.0049816732 0.0036018740
 [21] 0.0015308504 0.0012476829 0.0005663178 0.0018060536 0.0005651762
 [26] 0.0002252026 0.0057610021 0.0007346347 0.0002354309 0.0013535218
 [31] 0.0013095900 0.0037869041 0.0016891659 0.0043133704 0.0020292005
 [36] 0.0012919701 0.0015960421 0.0001839282 0.0029826662 0.0004097761
 [41] 0.0005658172 0.0011562396 0.0003846760 0.0002631910 0.0004938409
 [46] 0.0173059266 0.0131794853 0.0007896257 0.0013598222 0.0006222577
 [51] 0.0001659375 0.0005946478 0.0001500000 0.0007163122 0.0005664792
 [56] 0.0004585297 0.0030040124 0.0116500295 0.0008592047 0.0129965299
 [61] 0.0014096439 0.0006690134 0.0088191080 0.0008524934 0.0001978125
 [66] 0.0002597001 0.0022990033 0.0002724520 0.0185225247 0.0004464651
 [71] 0.0005557856 0.0004909141 0.0004416600 0.0016289899 0.0001993834
 [76] 0.0001962076 0.0008290106 0.0009949447 0.0002217390 0.0003446577
 [81] 0.0005611812 0.0011378092 0.0001693242 0.0002340144 0.0004995121
 [86] 0.0002459569 0.0002467154 0.0003900529 0.0001676309 0.0002224071
 [91] 0.0002884197 0.0001843441 0.0024174748 0.0004167413 0.0016632535
 [96] 0.0004057629 0.0070340156 0.0012383137 0.0001500000 0.0001500000
[101] 0.0016186701 0.0005358392 0.0001874948 0.0002404693 0.0001500000
[106] 0.0576727605 0.0002154434 0.0001500000 0.0003588400 0.0069828269
[111] 0.0046964815 0.0001659375 0.0007871025 0.0001500000 0.0280102783
[116] 0.0002156233 0.0004719215 0.0002011992 0.0005836524 0.0003202831
[121] 0.0004726865 0.0005835747 0.0002099505 0.0034364745 0.0032709247
[126] 0.0002890619 0.0105767918 0.0001500000 0.0032817869 0.0001989433
[131] 0.0006071661 0.0027722428 0.0010989595 0.0004421860 0.0009121397
[136] 0.0001659375 0.0003005366 0.0002931649 0.0002795230 0.0003091195
[141] 0.0001792167 0.0001659375 0.0001500000 0.0007714998 0.0001818750
[146] 0.0002030597 0.0002056866 0.0032687795 0.0005666541 0.0002706100
[151] 0.0005869947 0.0003106427 0.0001659375 0.0003349217 0.0002375874
[156] 0.0003541428 0.0017880451 0.0419903716 0.0004687538 0.0004921503
[161] 0.0038416654 0.0001500000 0.0002801144 0.0002170201 0.0008209366
[166] 0.0002742992 0.0005277067 0.0001854844 0.0005742444 0.0070556822
[171] 0.0001818750 0.0002144258 0.0001871541 0.0003333907 0.0001659375
[176] 0.0003829945 0.0001500000 0.0001818750 0.0002137500 0.0010163085
[181] 0.0003232385 0.0003178357 0.0004455387 0.0003947738 0.0002764017
```

```
[186] 0.0002705691 0.0002343458 0.0001833996 0.0003407205 0.0003733383
[191] 0.0002032524 0.0001676309 0.0002488890 0.0002126164 0.0006082956
[196] 0.0002030127 0.0001995059 0.0002489576 0.0002421235 0.0001676309
[201] 0.0002728292 0.0001500000 0.0001659375 0.0016493978 0.0001659375
[206] 0.0002060688 0.0004106056 0.0002154434 0.0002068167 0.0004237668
[211] 0.0010214923 0.0007934319 0.0001919447 0.0002716504 0.0003432301
[216] 0.0074775793 0.0002006642 0.0002816784 0.0008639914 0.0002551123
[221] 0.0002284384 0.0002172948 0.0003377166 0.0002350420 0.0007315230
[226] 0.0003475325 0.0025665815 0.0004189523 0.0003980021 0.0001500000
[231] 0.0001659375 0.0001728233 0.0002617247 0.0001659375 0.0002028926
[236] 0.0001818750 0.0002207268 0.0001500000 0.0002871160 0.0003461940
[241] 0.0002141153 0.0002330742 0.0001500000 0.0001839709 0.0002574156
[246] 0.0001997260 0.0004677662 0.0001500000 0.0004819729 0.0001758973
[251] 0.0002709417 0.0002143742 0.0330427116 0.0005577891 0.0002100456
[256] 0.0002351274 0.0001659375 0.0001659375 0.0003098679 0.0001770154
[261] 0.0002028926 0.0003268394 0.0001676309 0.0002181797 0.0002277919
[266] 0.0007937401 0.0003371317 0.0002693273 0.0001659375 0.0002424333
[271] 0.0004209338 0.0003297581 0.0001659375 0.0001659375 0.0001894268
[276] 0.0002045859 0.0002426900 0.0002447443 0.0002662893 0.0001500000
[281] 0.0001500000 0.0002016872 0.0001500000 0.0009452502 0.0001500000
[286] 0.0002768497 0.0001912426 0.0004335807 0.0005197816 0.0003307266
[291] 0.0002614125 0.0002030725 0.0001500000 0.0001500000 0.0001500000
[296] 0.0001500000 0.0001500000 0.0001676309 0.0016571631 0.0001676309
[301] 0.0002216283 0.0001696841 0.0001500000 0.0002148078 0.0003487400
[306] 0.0007047175 0.0001852617 0.0002363436 0.0002203982 0.0012649390
[311] 0.0033188520 0.0001852617 0.0001500000 0.0002210756 0.0001734733
[316] 0.0001500000 0.0001500000 0.0002630764 0.0001500000 0.0008038276
[321] 0.0002010772 0.0001659375 0.0001818750 0.0001500000 0.0002330049
[326] 0.0002971542 0.0059145070 0.0001500000 0.0002005503 0.0005819399
[331] 0.0001500000 0.0002544913 0.0043024482 0.0001500000 0.0002064783
[336] 0.0002099099 0.0002193674 0.0002218978 0.0008371788 0.0002503817
[341] 0.0002220665 0.0002456144 0.0001970411 0.0001500000 0.0001500000
[346] 0.0002409966 0.0001500000 0.0001500000 0.0001960679 0.0002257194
[351] 0.0002360957 0.0001500000 0.0001818750 0.0002137500 0.0006099507
[356] 0.0002082717 0.0001943729 0.0002144713 0.0001500000 0.0001500000
[361] 0.0001695041 0.0003857279 0.0004601332 0.0001767087 0.0012451847
[366] 0.0003755767 0.0001659375 0.0026194354 0.0003493850 0.0001500000
[371] 0.0001500000 0.0002013791 0.0001500000 0.0001659375 0.0003222645
[376] 0.0002613677 0.0002020016 0.0001704596 0.0001996858 0.0001871350
[381] 0.0007331429 0.0003139433 0.0001500000 0.0001500000 0.0006918621
[386] 0.0001659375 0.0004604365 0.0003300769 0.0002292561 0.0002118238
[391] 0.0003341706 0.0001500000 0.0001500000 0.0001500000 0.0005421207
[396] 0.0002157307 0.0002237016 0.0002515775 0.0001500000 0.0002154434
[401] 0.0001854633 0.0001659375 0.0001921118 0.0002749809 0.0002437390
[406] 0.0002255099 0.0002563690 0.0002421365 0.0001659375 0.0001500000
[411] 0.0001818750 0.0001881668 0.0002273754 0.0001961072 0.0001500000
[416] 0.0001500000 0.0001678348 0.0001500000 0.0001500000 0.0001818750
[421] 0.0001835684 0.0001707931 0.0047532946 0.0001835684 0.0001680098
```

```
[426] 0.0001835684 0.0004459435 0.0001500000 0.0001978125 0.0008807099
[431] 0.0001500000 0.0001764517 0.0001722941 0.0001830058 0.0001659375
[436] 0.0002138744 0.0011770925 0.0012785286 0.0001500000 0.0001695041
[441] 0.0001500000 0.0003533398 0.0001679907 0.0001500000 0.0001890907
[446] 0.0001500000 0.0001500000 0.0001500000 0.0001659375 0.0008826505
[451] 0.0005266854 0.0008036634 0.0001500000 0.0002353319 0.0001697100
[456] 0.0001695041 0.0001500000 0.0002443079 0.0001933220 0.0001697264
[461] 0.0003697879 0.0001500000 0.0002504612 0.0001500000 0.0001659375
[466] 0.0001500000 0.0002282732 0.0001500000 0.0002542626 0.0001835684
[471] 0.0046041632 0.0001500000 0.0001857273 0.0001500000 0.0001659375
[476] 0.0001899643 0.0001500000 0.0001736187 0.0001855854 0.0002102058
[481] 0.0001500000 0.0009270586 0.0014429949 0.0001500000 0.0001500000
[486] 0.0001676309 0.0001500000 0.0001500000 0.0001818750 0.0015791592
[491] 0.0001835684 0.0013878156 0.0001500000 0.0001500000 0.0001659375
[496] 0.0001659375 0.0001500000 0.0004343938 0.0002032524 0.0002011992
[501] 0.0006707428 0.0001500000 0.0001686039 0.0001659375 0.0002047659
[506] 0.0001659375 0.0001500000 0.0001659375 0.0001659375 0.0001886484
[511] 0.0006090698 0.0001995059 0.0001852617 0.0003658166 0.0002082252
[516] 0.0004721977 0.0002691622 0.0035452745 0.0001700349 0.0001500000
[521] 0.0002188301 0.0001500000 0.0001676309 0.0004696208 0.0001659375
[526] 0.0002935773 0.0001893681 0.0001741957 0.0001500000 0.0001500000
[531] 0.0001818750 0.0001659375 0.0001500000 0.0001500000 0.0001500000
[536] 0.0001659375 0.0001952280 0.0001500000 0.0001855056 0.0001676309
[541] 0.0001500000 0.0004153641 0.0001500000 0.0001835684 0.0001500000
[546] 0.0001500000 0.0001500000 0.0001500000 0.0001500000 0.0002628247
[551] 0.0014333702 0.0001659375 0.0001500000 0.0001659375 0.0001500000
[556] 0.0002028701 0.0001500000 0.0001500000 0.0001500000 0.0002209256
[561] 0.0001500000 0.0001818750 0.0001500000 0.0001500000 0.0001716497
[566] 0.0001500000 0.0003015718 0.0001500000 0.0001676309 0.0001659375
[571] 0.0001500000 0.0001659375 0.0001500000 0.0009518685 0.0002315608
[576] 0.0001659375 0.0001500000 0.0001818750 0.0001500000 0.0001500000
[581] 0.0001500000 0.0002066307 0.0004688199 0.0002431900 0.0001659375
[586] 0.0002497926 0.0001500000 0.0001500000 0.0001500000 0.0002642704
[591] 0.0038438440 0.0001500000 0.0001500000 0.0001659375 0.0009870618
[596] 0.0001835684 0.0001925608 0.0001835684 0.0001500000 0.0001979230
[601] 0.0210219765 0.0002307453 0.0001500000 0.0005047496 0.0001500000
[606] 0.0001500000 0.0030324494 0.0001500000 0.0002289286 0.0001659375
[611] 0.0001659375 0.0002350115 0.0001500000 0.0001715573 0.0001500000
[616] 0.0001818750 0.0042123448 0.0002689576 0.0001500000 0.0001856598
[621] 0.0001500000 0.0001500000 0.0001500000 0.0001689087 0.0001678570
[626] 0.0001910325 0.0062429382 0.0001839709 0.0026058035 0.0002129556
[631] 0.0001500000 0.0002361885 0.0001500000 0.0001659375 0.0001871350
[636] 0.0001500000 0.0001500000 0.0001659375 0.0001500000 0.0001500000
[641] 0.0003536570 0.0002222934 0.0006753108 0.0001500000 0.0001884438
[646] 0.0002209536 0.0001837483 0.0004164144 0.0001500000 0.0001856598
[651] 0.0001694951 0.0001500000 0.0001659375 0.0001500000 0.0001500000
[656] 0.0002110135 0.0001500000 0.0002292052 0.0001818750 0.0007128711
[661] 0.0001500000 0.0001659375 0.0001500000 0.0001937078 0.0001500000
```

```
[666] 0.0001676309 0.0001500000 0.0001659375 0.0001500000 0.0001835684
[671] 0.0001818750 0.0016333742 0.0001500000 0.0001500000 0.0002805760
[676] 0.0001500000 0.0001659375 0.0001500000 0.0002927402 0.0001676309
[681] 0.0001500000 0.0002037621 0.0001500000 0.0001862570 0.0001500000
[686] 0.0001676309 0.0058015645 0.0001830058 0.0004976009 0.0005069680
[691] 0.0001659375 0.0001500000 0.0001500000 0.0001500000 0.0003285492
[696] 0.0002505723 0.0078816870 0.0001500000 0.0001500000 0.0001659375
[701] 0.0001500000 0.0001500000 0.0002043246 0.0001500000 0.0121995875
[706] 0.0001856216 0.0001500000 0.0001500000 0.0009285500 0.0001500000
[711] 0.0001500000 0.0001818750 0.0001659375 0.0016112391 0.0001500000
[716] 0.0001500000 0.0001500000 0.0018071371 0.0002650525 0.0003101780
[721] 0.0001500000 0.0001500000 0.0001500000 0.0001676309 0.0001659375
[726] 0.0001500000 0.0001500000 0.0001659375 0.0001659375 0.0001500000
[731] 0.0001500000 0.0001695552 0.0002277243 0.0002472218 0.0001818750
[736] 0.0001659375 0.0001500000 0.0001500000 0.0001500000 0.0001834835
[741] 0.0006153394 0.0034043280 0.0003143495 0.0002917361 0.0001956997
[746] 0.0001500000 0.0001676309 0.0001693242 0.0001978125 0.0003293304
[751] 0.0002276500 0.0001779640 0.0001500000 0.0001701266 0.0002151181
[756] 0.0003163222 0.0001695041 0.0001500000 0.0001500000 0.0001500000
[761] 0.0001500000 0.0001500000 0.0006099560 0.0002277243 0.0001500000
[766] 0.0002118533 0.0002875937 0.0002030725 0.0001659375 0.0001500000
[771] 0.0001500000 0.0049323758 0.0002029911 0.0009715264 0.0002467152
[776] 0.0002112903 0.0001750951 0.0001659375 0.0001659375 0.0001500000
[781] 0.0002587988 0.0001659375 0.0001500000 0.0003257679 0.0001500000
[786] 0.0001676309 0.0002077712 0.0001500000 0.0001700349 0.0001659375
[791] 0.0001500000 0.0001500000 0.0002025260 0.0001680662 0.0001659375
[796] 0.0002905901 0.0006986336 0.0001500000 0.0001500000 0.0001500000
[801] 0.0001659375 0.0003416910 0.0001659375 0.0001835684 0.0002172659
[806] 0.0001500000 0.0001500000 0.0001500000 0.0001813055 0.0001659375
[811] 0.0001818750 0.0001500000 0.0002756750 0.0001818750 0.0001943624
[816] 0.0001500000 0.0002436618 0.0001659375 0.0001500000 0.0001500000
[821] 0.0001500000 0.0008392923 0.0004053737 0.0001659375 0.0001500000
[826] 0.0009711234 0.0001500000 0.0001676309 0.0001659375 0.0001676309
[831] 0.0004049831 0.0001500000 0.0001500000 0.0001500000 0.0013199268
[836] 0.0001500000 0.0001500000 0.0001659375 0.0015468188 0.0001500000
[841] 0.0001659375 0.0001500000 0.0001693242 0.0001500000 0.0001818750
[846] 0.0001659375 0.0001500000 0.0001500000 0.0002417991 0.0002074809
[851] 0.0001500000 0.0001659375 0.0001500000 0.0002364609 0.0001500000
[856] 0.0001659375 0.0001500000 0.0030828612 0.0001500000 0.0001500000
[861] 0.0001500000 0.0001659375 0.0007602610 0.0001500000 0.0001500000
[866] 0.0001500000 0.0001659375 0.0001500000 0.0001727827 0.0001500000
[871] 0.0015718663 0.0001659375 0.0001500000 0.0001500000 0.0001500000
[876] 0.0001682378 0.0001500000 0.0001500000 0.0001500000 0.0001818750
[881] 0.0002642403 0.0001500000 0.0002019328 0.0003024865 0.0001659375
[886] 0.0001500000 0.0003151606 0.0001500000 0.0009154082 0.0001500000
[891] 0.0001835684 0.0001676309 0.0001659375 0.0002069329 0.0001500000
[896] 0.0001864682 0.0002049649 0.0001500000 0.0001500000 0.0001500000
[901] 0.0003591538 0.0001500000 0.0001895635 0.0002212664 0.0001500000
```

```
[906] 0.0001659375 0.0001500000 0.0001659375 0.0001500000 0.0001500000
[911] 0.0001500000 0.0001995059 0.0001500000 0.0001500000 0.0001500000
[916] 0.0001500000 0.0001840488 0.0001835684 0.0001500000 0.0001500000
[921] 0.0001659375 0.0001500000 0.0002349475 0.0002161149 0.0001500000
[926] 0.0001500000 0.0001500000 0.0002513759 0.0001500000 0.0001500000
[931] 0.0001659375 0.0001500000 0.0002946399 0.0001837483 0.0001500000
[936] 0.0001500000 0.0002427976 0.0001500000 0.0001500000 0.0185995279
[941] 0.0001500000 0.0001659375 0.0003151392 0.0001676309 0.0001500000
[946] 0.0026748037 0.0001500000 0.0001500000 0.0001735598 0.0001500000
[951] 0.0001500000 0.0001500000 0.0001500000 0.0011576447 0.0001728563
[956] 0.0001500000 0.0001500000 0.0001500000 0.0001818750 0.0020674290
[961] 0.0002361893 0.0001500000 0.0002039989 0.0001500000 0.0001500000
[966] 0.0001500000 0.0001500000 0.0001500000 0.0001500000 0.0001885640
[971] 0.0001659375 0.0001500000 0.0001933220 0.0002304489 0.0001500000
[976] 0.0001500000 0.0001500000 0.0001678108 0.0001500000 0.0001500000
[981] 0.0001500000 0.0002363961 0.0002098269 0.0001500000 0.0001500000
[986] 0.0001500000 0.0001500000 0.0002891030 0.0001736308 0.0003094305
[991] 0.0001500000 0.0002137500 0.0001500000 0.0005821484 0.0001500000
[996] 0.0001500000 0.0001500000 0.0001659375 0.0001500000 0.0002381655
```