In [12]:
```
system('apt-get install libglpk-dev libgmp-dev libxml2-dev', intern=T)
```

'Reading package lists...' · 'Building dependency tree...' · 'Reading state information...' ·
'libgmp-dev is already the newest version (2:6.2.1+dfsg-3ubuntu1).' ·
'libglpk-dev is already the newest version (5.0-1).' ·
'libxml2-dev is already the newest version (2.9.13+dfsg-1ubuntu0.4).' ·
'0 upgraded, 0 newly installed, 0 to remove and 45 not upgraded.'

In [13]:
```
install.packages("igraph")
install.packages("resample")
install.packages("textTinyR")
install.packages("Matrix")
install.packages("pracma")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

In [ ]:
```
install.packages("igraph", type="binary")
```

In [14]:
```
library(igraph)
library(resample)
library(textTinyR)
library(Matrix)
library(pracma)
```

1.Generating Random Networks

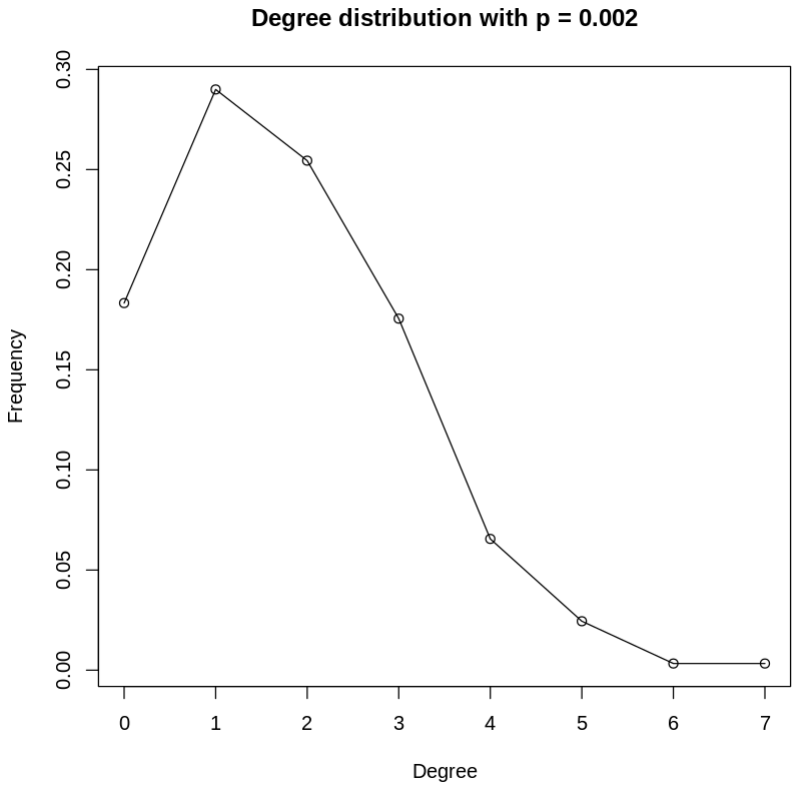
1. Create random networks using Erd¨os-R´enyi (ER) model

(a)

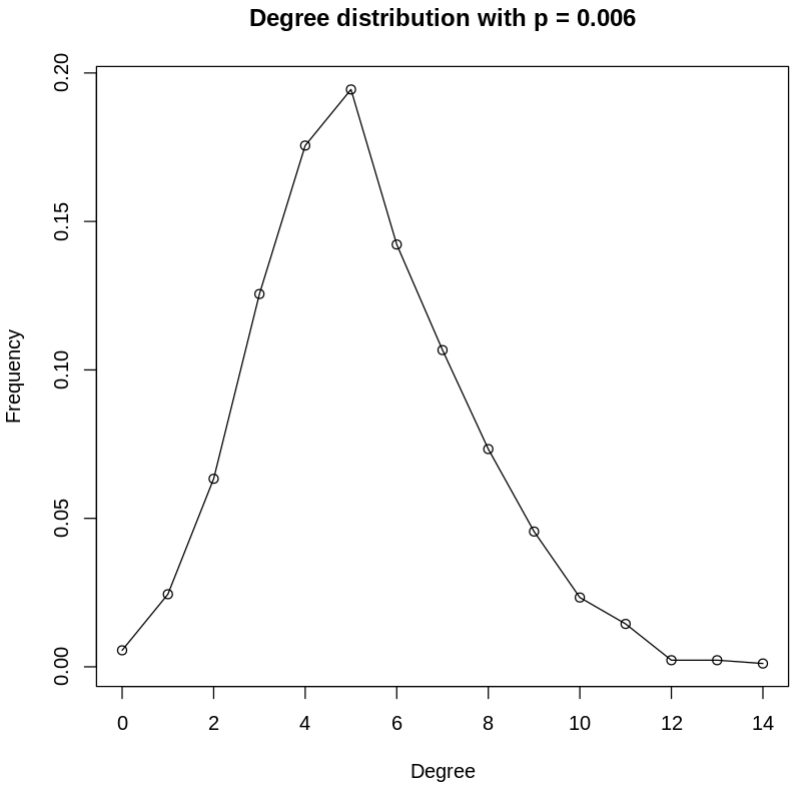
In [7]:
```
n <- 900 #num of nodes
proba <- c(0.002, 0.006, 0.012, 0.045, 0.1)

for (p in proba) {
g <- sample_gnp(n, p, directed=FALSE)
plot(seq_along(degree_distribution(g)) - 1, degree_distribution(g),
main=sprintf("Degree distribution with p = %.3f", p),
```

```
    xlab="Degree", ylab="Frequency")
lines(seq_along(degree.distribution(g)) - 1, degree.distribution(g), col='blacl
fprintf("Actual: (p=%.3f) mean=%.3f variance=%.3f\n", p,
        mean(degree(g)), var(degree(g)))
fprintf("theoretical: mean=%.3f variance=%.3f", (n - 1) * p,
        (n - 1) * p * (1 - p))
}
```

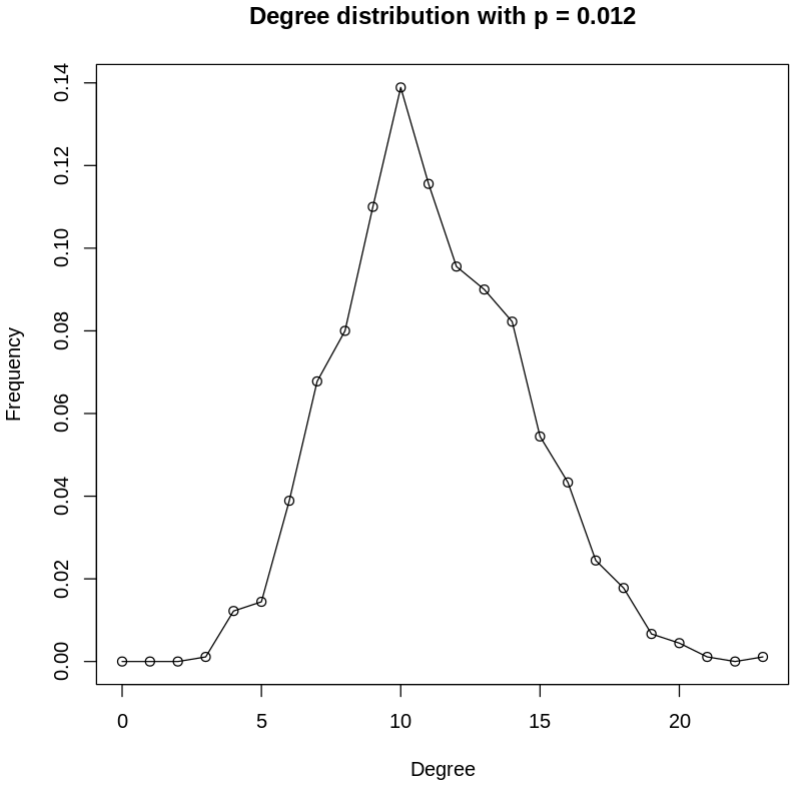
```
Actual: (p=0.002) mean=1.753 variance=1.759
theoretical: mean=1.798 variance=1.794
```



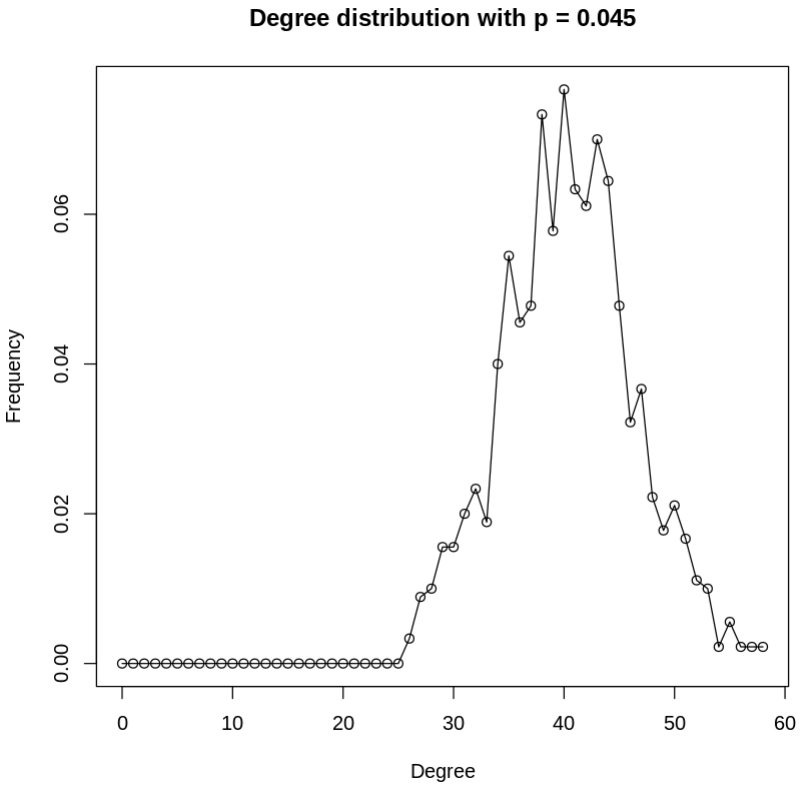

```
Actual: (p=0.006) mean=5.262 variance=5.117
theoretical: mean=5.394 variance=5.362
```



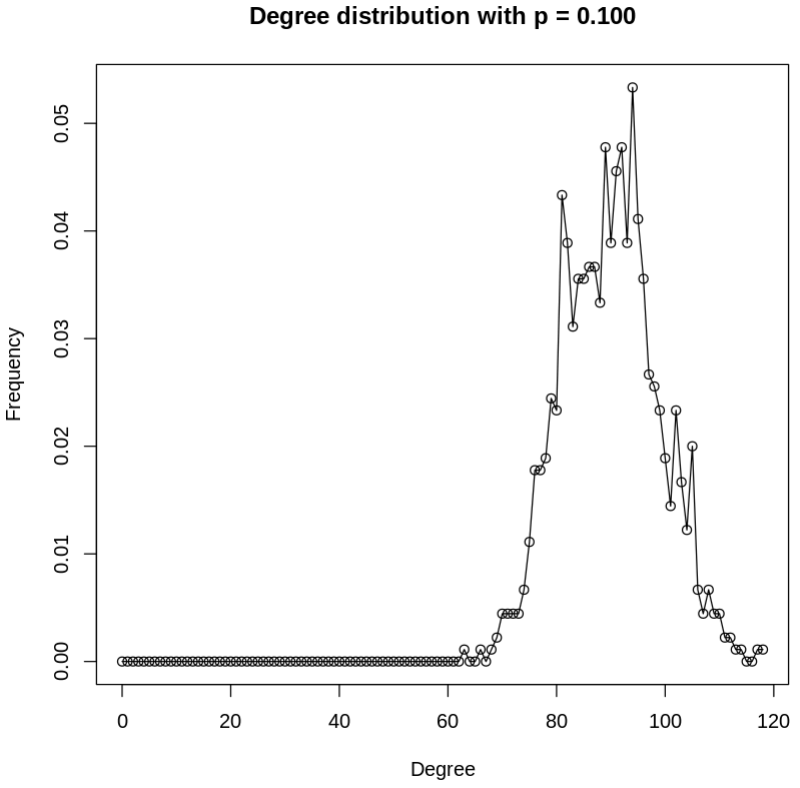

```
Actual: (p=0.012) mean=11.100 variance=10.506
theoretical: mean=10.788 variance=10.659
```




```
Actual: (p=0.045) mean=40.416 variance=35.266
theoretical: mean=40.455 variance=38.635
```




```
Actual: (p=0.100) mean=89.871 variance=77.427
theoretical: mean=89.900 variance=80.910
```



(a) We've observed binomial distributions. That's because in ER model, the decision to retain an edge for each vertex is made randomly, with a probability p amongst the remaining n-1 vertices. Therefore, the degree of a vertice should follow binomial distribution with mean (n-1)p and variance (n-1)p(1-p). As we can see from the above results, the experimental

mean and variance values are close to the theoretical values, which indicates that the distributions should be binomial.

(b)

In [17]:
```
n <- 900
trial_num <- 1000

for (p in proba) {
    gcc_is_found <- FALSE
    n_connected <- 0
    gcc_diameter <- NULL

    for (i in 1:trial_num) {
        g <- sample_gnp(n,p, directed=FALSE)
        if (is.connected(g)) {
            n_connected <- n_connected + 1
        }
        else if (!is.connected(g) & !gcc_is_found) {
            g_components <- clusters(g)
            max_size <- which.max(g_components$csize)
            gcc <- induced_subgraph(g, which(g_components$membership == max_si
            gcc_diameter <- diameter(gcc, directed=FALSE)
            gcc_is_found <- TRUE
        }
    }
    fprintf("Probability the network is connected (p = %0.3f):%.3f",p,n_connec

    if (is.null(gcc_diameter)) {
        fprintf("All generated networks are connected.\n")
    }
    else {
        fprintf("Diameter of GCC for non-connected network: %d.\n", gcc_diamet
    }
}
```

```
Probability the generated network is connected (p = 0.002) : 0.000.
Diameter of GCC for non-connected network: 26.
Probability the generated network is connected (p = 0.006) : 0.017.
Diameter of GCC for non-connected network: 9.
Probability the generated network is connected (p = 0.012) : 0.974.
Diameter of GCC for non-connected network: 5.
Probability the generated network is connected (p = 0.045) : 1.000.
All generated networks are connected.
Probability the generated network is connected (p = 0.100) : 1.000.
All generated networks are connected.
```

(b) No, not all ER networks are connected. The probabilities that a generated network is connected are 0，0.017，0.974，1.0，1.0. The diameters of GCCs are 26,9,5.

(c)

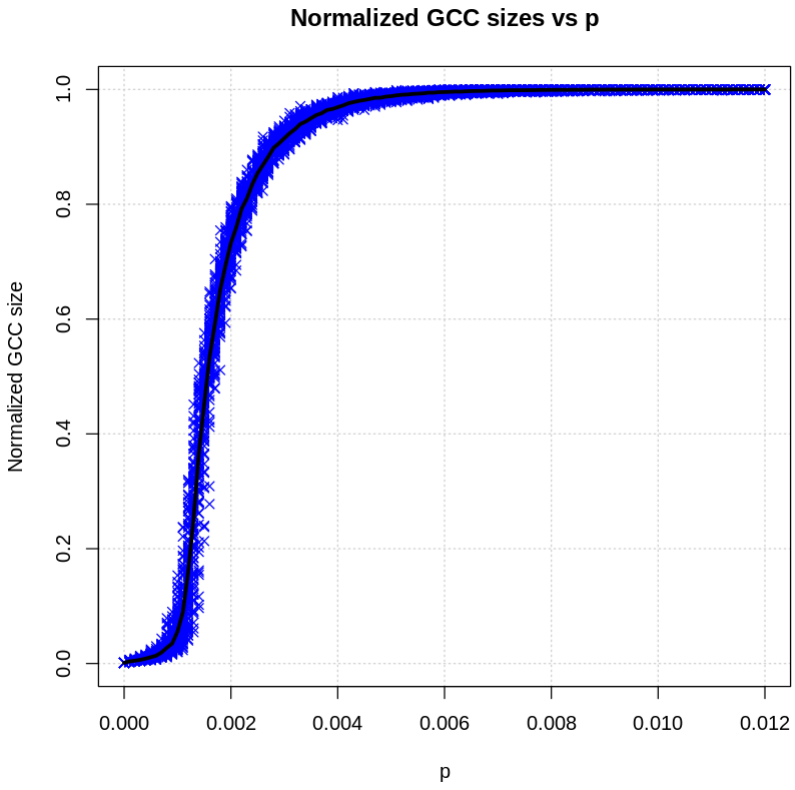
In [24]:
```
n <- 900
p_max <- 0.012
proba <- seq(0, p_max, 0.0001)
trial_num <- 100
```

```
gcc_sizes <- matrix(data=0.0, nrow=length(proba), ncol=trial_num)
avg_gcc_sizes <- matrix(data=0.0, nrow=length(proba), ncol=1)
gcc_thresh <- 0.1
```

In [26]:

```
for (i in 1:length(proba)){
    for (j in 1:trial_num){
        g <- sample_gnp(n, proba[i], directed=FALSE)
        g.components <- clusters(g)
        gcc_sizes[i, j] <- max(g.components$csize) / n
    }
    avg_gcc_sizes[i] <- mean(gcc_sizes[i,])
}


plot(
    rep(proba[1], trial_num),
    gcc_sizes[1, 1:trial_num],
    xlim=c(0, p_max),
    ylim=c(0, 1),
    xlab="p",
    main="Normalized GCC sizes vs p",
    ylab="Normalized GCC size",
    grid(),
    pch=4,
    col="blue",
)

for (j in 2:length(proba)) {
    points(rep(proba[j], trial_num), gcc_sizes[j, 1:trial_num], pch=4, col="blu
}

lines(proba, avg_gcc_sizes, lwd=3)

fprintf("P value where a giant connected component starts to emerge: %.4f\n",
        proba[min(which(avg_gcc_sizes > gcc_thresh))])
fprintf("P value where the giant connected component takes up over 99 percent
        proba[mean(which(avg_gcc_sizes>0.99))])
```

```
P value where a giant connected component starts to emerge: 0.0012
P value where the giant connected component takes up over 99 percent of the no
des: 0.0086
```



i. The emergence criterion is defined as the normalized GCC size equals to 0.1. From the above results, we can see that it starts to emerge when p=0.0012, which is quite close to the theoretical value (1/900=0.0011).

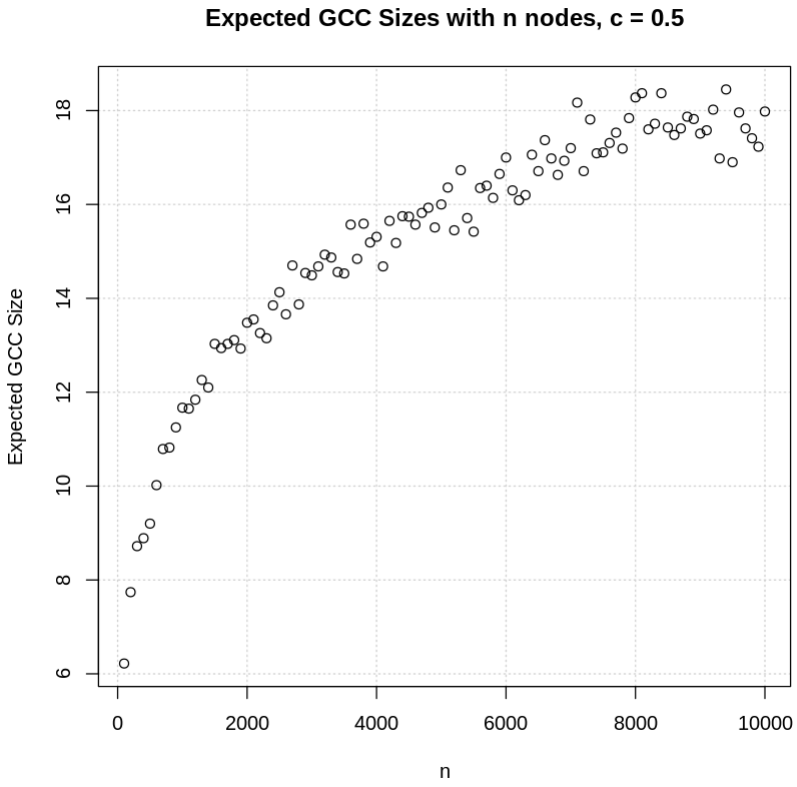
ii. The estimated value of p empirically is 0.0086.

(d)

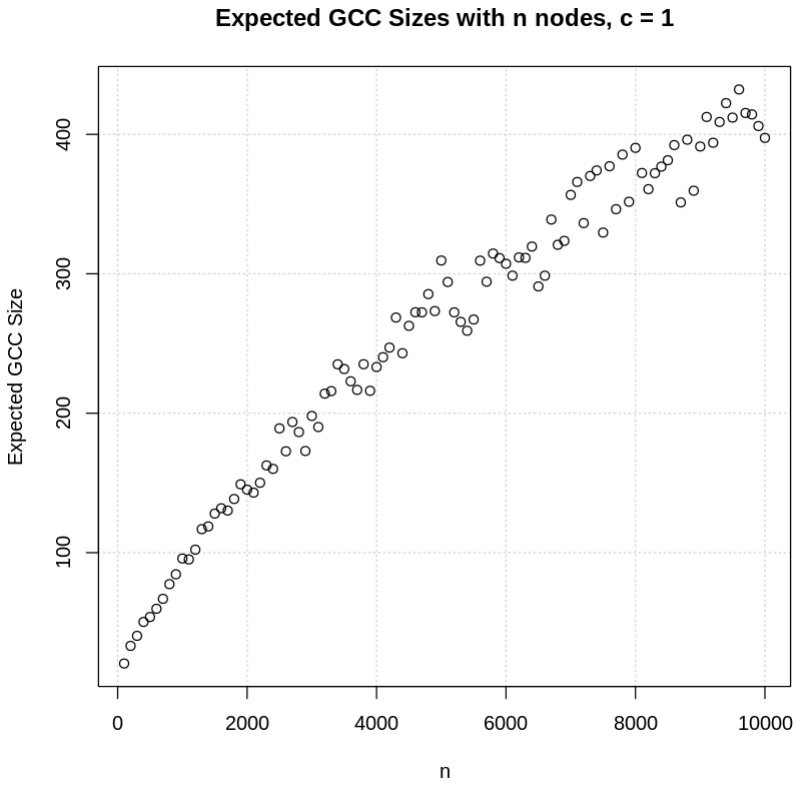
In [27]:

```
#i

n <- seq(100, 10000, 100)
get_expected_gcc_sizes <- function(n, c, trial_num=100, directed=FALSE) {
    gcc_sizes <- matrix(data=0.0, nrow=length(n), ncol=trial_num)
    avg_gcc_sizes = matrix(data=0.0, nrow=length(n), ncol=1)

    for (i in 1:length(n)){
        for (j in 1:trial_num){
            g <- sample_gnp(n[i], c / n[i], directed=directed)
            g.components <- clusters(g)
            gcc_sizes[i, j] <- max(g.components$csize)
        }
        avg_gcc_sizes[i] <- mean(gcc_sizes[i,])
    }

    return (avg_gcc_sizes)
}


avg_gcc_sizes <- get_expected_gcc_sizes(n, c=0.5)
plot(
```

```
    n,
    avg_gcc_sizes,
    main="Expected GCC Sizes with n nodes, c = 0.5",
    xlab="n",
    ylab="Expected GCC Size",
    grid(),
)
```

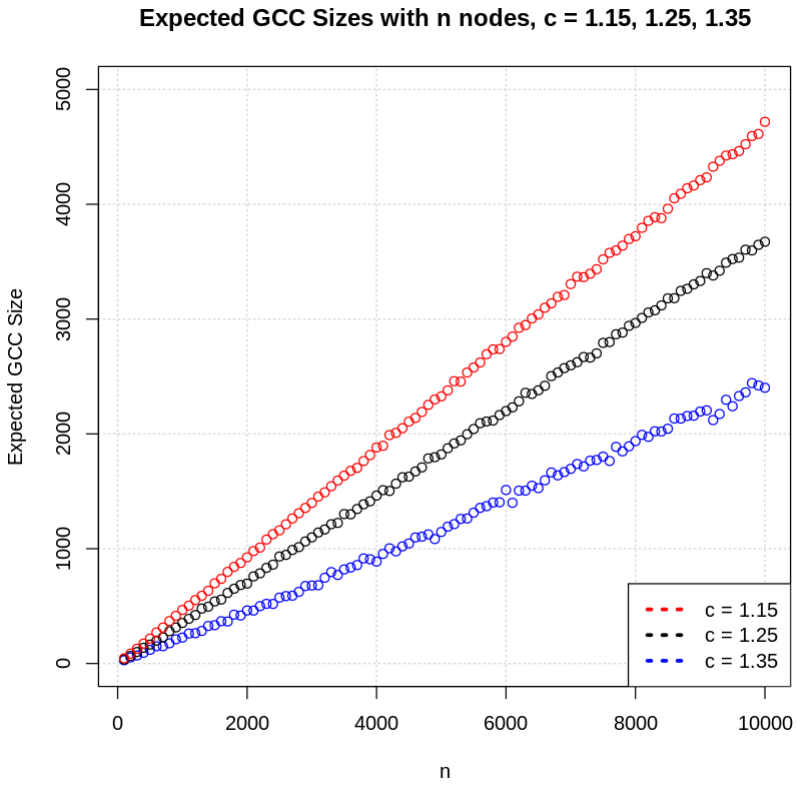


In [28]:
```
#ii

avg_gcc_sizes <- get_expected_gcc_sizes(n, c=1)
plot(
    n,
    avg_gcc_sizes,
    main="Expected GCC Sizes with n nodes, c = 1",
    xlab="n",
    ylab="Expected GCC Size",
    grid(),
)
```



In [30]:

```
#iii
avg_gcc_sizes <- get_expected_gcc_sizes(n, c=1.15)
plot(
    n,
    avg_gcc_sizes,
    main="Expected GCC Sizes with n nodes, c = 1.15, 1.25, 1.35",
    xlab="n",
    ylab="Expected GCC Size",
    grid(),
    col="blue",
    ylim=c(0, 5000),
)

avg_gcc_sizes <- get_expected_gcc_sizes(n, c=1.25)
points(n, avg_gcc_sizes, col="black")

avg_gcc_sizes <- get_expected_gcc_sizes(n, c=1.35)
points(n, avg_gcc_sizes, col="red")

legend('bottomright', legend=c("c = 1.15", "c = 1.25", "c = 1.35"),
       lty=c(3, 3, 3), lwd=c(3,3,3), col = c('red', 'black', 'blue'))
```



i. From the generated graph of (i), we can observe a monotonically increasing trend whose shape is like a logarithmic curve. The relationship between n and expected GCC size is logarithmic and as n increases, expected GCC size also increases.

ii. From graph(ii), we can see that the relationship between n and expected GCC size is still monotonically increase, but the degree of curve is smaller(a straighter line), more like an exponential relationship with an exponent which is close to 1.

iii. From graph(iii), we can see that the curves are linear,n and expected GCC size are linearly positively related.

iv. See i-iii above. n and expected GCC size are positively correlated, as c increases the slope increases. Higher num of nodes means large GCC size.

2. Create networks using preferential attachment model

(a)

In [31]:
```
check_preferential_connectness <- function(m, n, trial_num=100) {
    connected_count = 0
    for (i in 1:trial_num) {
        g <- sample_pa(n, m=m, directed=FALSE)
        if (is.connected(g)) {
            connected_count <- connected_count + 1
```

```
        }
    }
    fprintf("Percent of connected network with m = %d and n = %d: %.3f\n",
            m, n, connected_count / trial_num)
}


check_preferential_connectness(m=1, n=1050)
```

```
Percent of connected network with m = 1 and n = 1050: 1.000
```

(a) As the calculated percent is 1, we can conclude that such networks are always connected

(b)

In [32]:
```
measure_scores <- function(m, n) {
    g <- sample_pa(n, m=m, directed=FALSE)
    modularity_score <- modularity(cluster_fast_greedy(g))
    assortativity_score <- assortativity_degree(g, directed=FALSE) #compute deg
    fprintf("m = %d, n = %d: ", m, n)
    fprintf("modularity = %.3f; assortativity = %.3f\n",
            modularity_score, assortativity_score)
}


measure_scores(m=1, n=1050)
```

```
m = 1, n = 1050: modularity = 0.934; assortativity = -0.071
```

Assortativity is defined as the tendency of vertices to attach to vertices with similar properties in a graph. It measures the level of homophyly of the graph. We choose to compute the degree assortativity. The modularity of the network is 0.934 and the degree assortativity is -0.071.

(c)

In [33]:
```
measure_scores(m=1, n=10500)
```

```
m = 1, n = 10500: modularity = 0.979; assortativity = -0.034
```

The larger network with 10500 nodes got a higher modularity, which is 0.979. That may be because as nodes number increases, the sparsity of inter-community increases which lead to an increase in the modularity of the network.

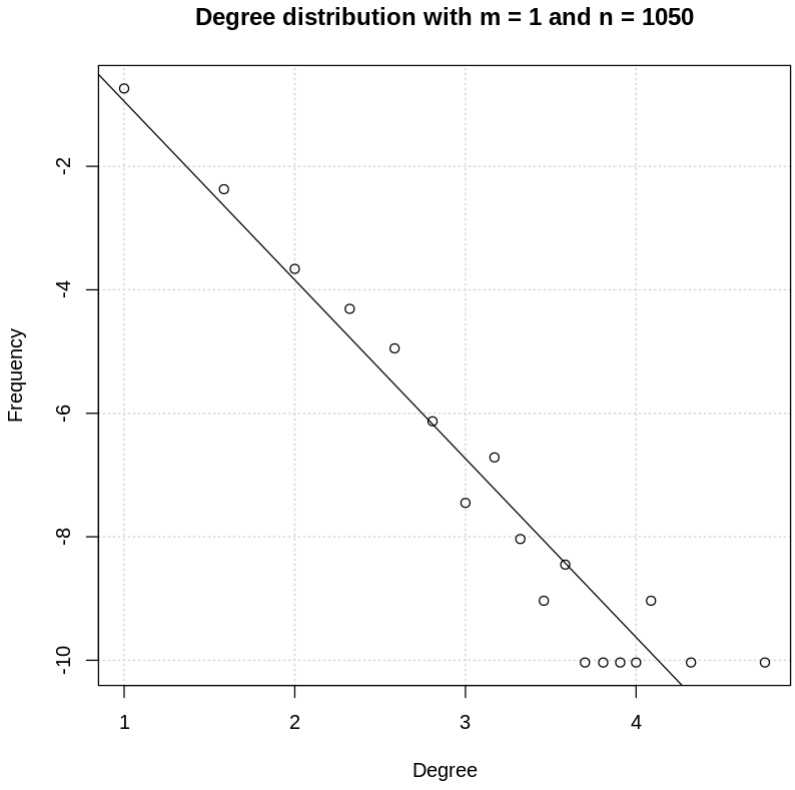
(d)

In [45]:
```
plot_degree_distribution <- function(m,n, g=NULL) {
    if (is.null(g)) {
        g <- sample_pa(n, m=m, directed=FALSE)
    }
    deg_dist <- degree.distribution(g)
    non_zero_idxs <- which(deg_dist != 0, arr.ind=TRUE)
    x <- log2(c(1: length(deg_dist)))[non_zero_idxs] #log-scale
```

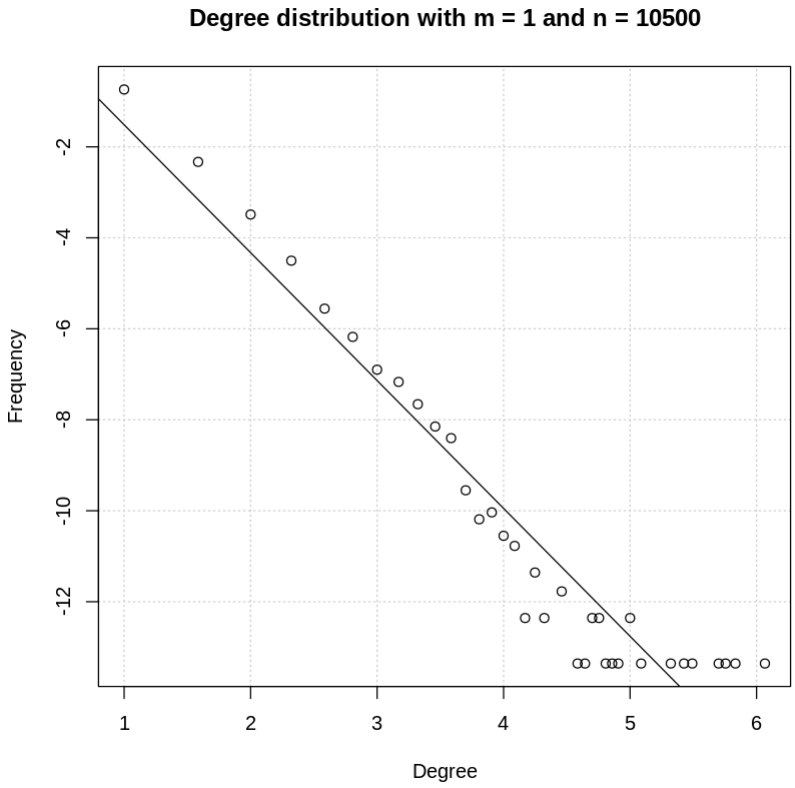
```
    y <- log2(deg_dist)[non_zero_idxs]
    plot(x,y,main=sprintf("Degree distribution with m = %d and n = %d",m, n),
        xlab="Degree",
        ylab="Frequency",
        grid()
    )
    linear_model <- lm(y ~ x) #construct a linear model
    abline(linear_model)
    fprintf("Slope with m = %d and n = %d: %.3f",m,n, coef(linear_model)[2])
}
```

In [35]:
```
plot_degree_distribution(m=1,n=1050)
plot_degree_distribution(m=1,n=10500)
```

```
Slope with m = 1 and n = 1050: -2.894
```




```
Slope with m = 1 and n = 10500: -2.811
```



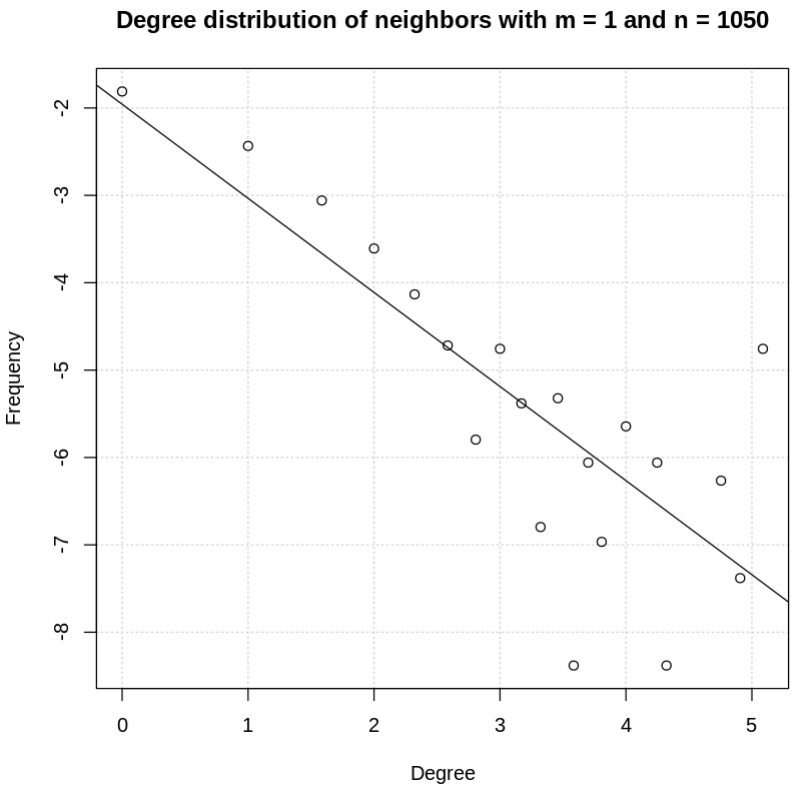
The estimated slope is -2.894 for n=1050 and -2.811 for 10500

(e)

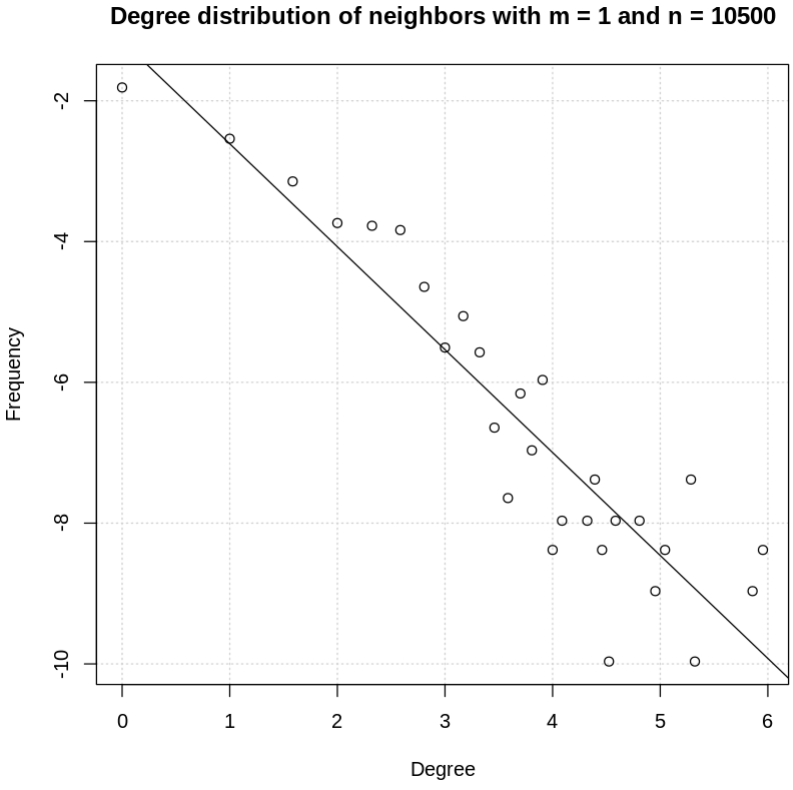
In [46]:
```
plot_neighbor_degree_distribution <- function(m,n, n_sample=1000) {
    g <- sample_pa(n,m=m,directed=FALSE)
    degs <- matrix(data=0.0, nrow=n_sample, ncol=1)

    for (i in 1: n_sample) {
        sample_node <- sample(n, 1) #sample a node
        neighbor <- neighbors(g, sample_node)
        sample_neighbor <- sample(neighbor, 1) #pick a neighbor
        degs[i, 1] <- degree(g, sample_neighbor)
    }

    deg_dist <- table(degs)
    x <- log2(as.numeric(names(deg_dist)))
    y <- log2(as.vector(deg_dist) / n_sample)
    plot(x,y,main=sprintf("Degree distribution of neighbors with m = %d and n =
        xlab="Degree",
        ylab="Frequency",
        grid()
    )
    linear_model <- lm(y ~ x)
    abline(linear_model)
    fprintf("Slope with m = %d and n = %d: %.3f",m,n, coef(linear_model)[2])
}
```

In [37]:
```
plot_neighbor_degree_distribution(m=1,n=1050) #network in 2(a)
plot_neighbor_degree_distribution(m=1,n=10500) #network in 2(c)
```

```
Slope with m = 1 and n = 1050: −1.077
```




```
Slope with m = 1 and n = 10500: −1.462
```

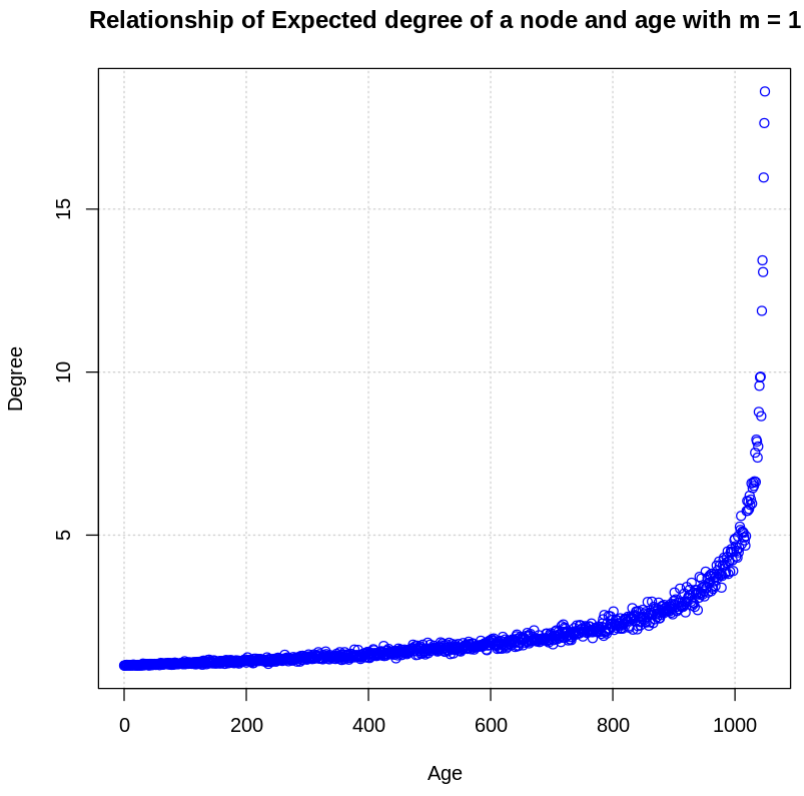


For the 2(a) graph and 2(c) graph, the distributions are both almost linear in log-scale. The slopes are -1.077 and -1.462 respectively, which are all larger than the slopes from the node degree distribution (-2.894 for n=1050 and -2.811 for 10500)

(f)

In [47]:
```
plot_age_degree_relation <- function(m=1, n=1050, trial_num=100) {
ages = matrix(data=0.0, nrow=n, ncol=1)
    for (i in 1: trial_num) {
        g <- sample_pa(n, m=1, directed=FALSE)
        ages <- ages + degree(g)
    }
    ages <- ages / trial_num
    plot(
        seq(n-1, 0, -1),
        ages,
        main=sprintf("Relationship of Expected degree of node and age(m = 1)"),
        xlab="Age",
        ylab="Degree",
        col="blue",
        grid()
    )

}
```

In [48]:
```
plot_age_degree_relation(m=1, n=1050)
```



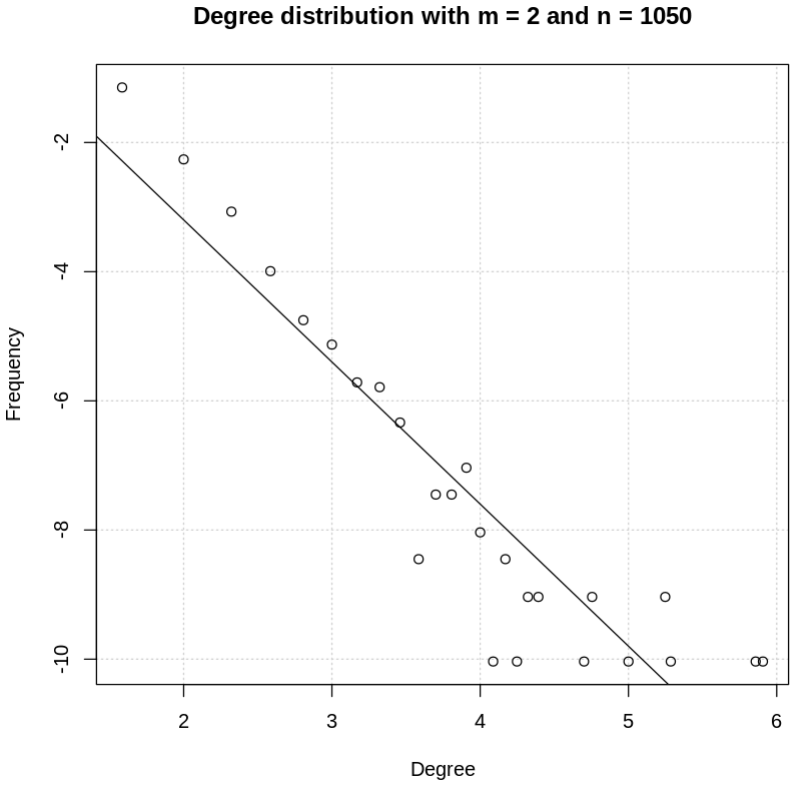
The relationship between Age and Degree is as above. As age increases, degree firstly increases slowly then abruptly as Age approaches 1000.

(g)

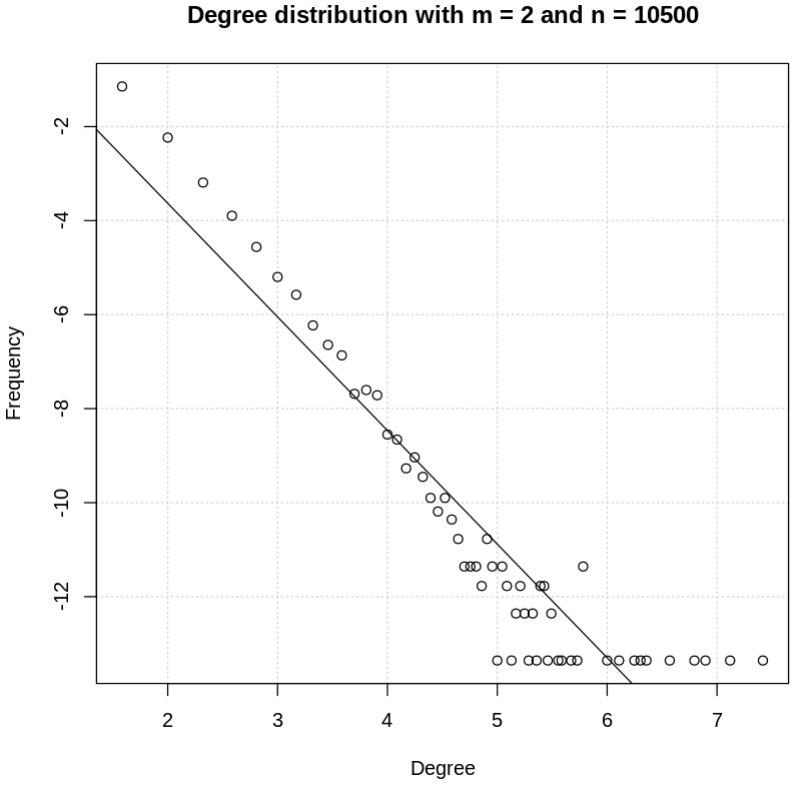
In [49]:
```
pipeline <- function(m) {
    check_preferential_connectness(m=m, n=1050)
    measure_scores(m=m, n=1050)
    measure_scores(m=m, n=10500)
    plot_degree_distribution(m=m, n=1050)
    plot_degree_distribution(m=m, n=10500)
    plot_neighbor_degree_distribution(m=m, n=1050)
    plot_neighbor_degree_distribution(m=m, n=10500)
    plot_age_degree_relation(m=m)
}
```

In [50]:
```
pipeline(m=2)
pipeline(m=6)
```

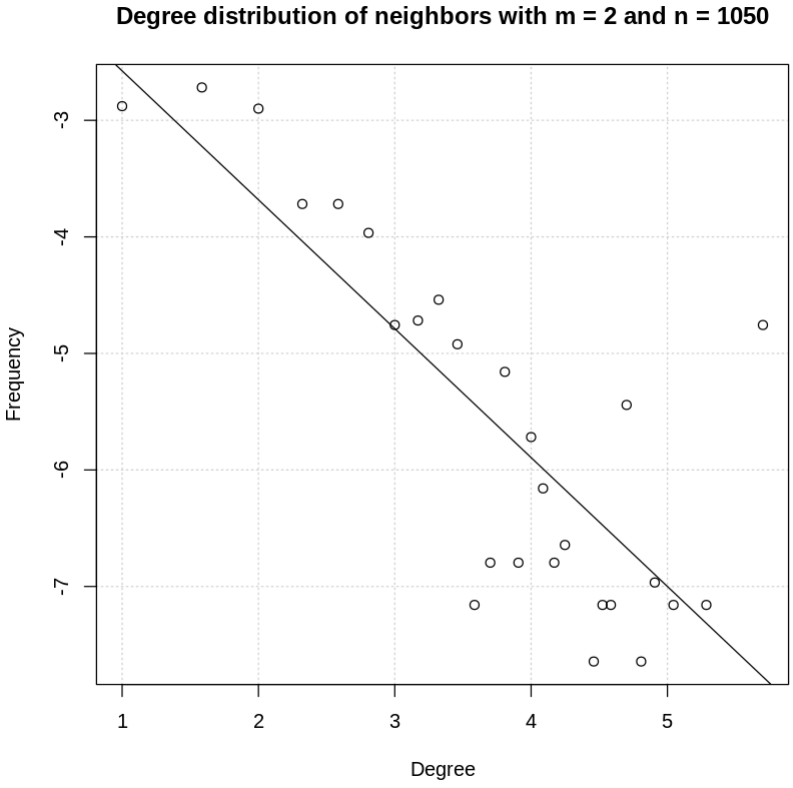
```
Percent of connected network with m = 2 and n = 1050: 1.000
m = 2, n = 1050: modularity = 0.524; assortativity = -0.057
m = 2, n = 10500: modularity = 0.527; assortativity = -0.010
Slope with m = 2 and n = 1050: -2.200
```



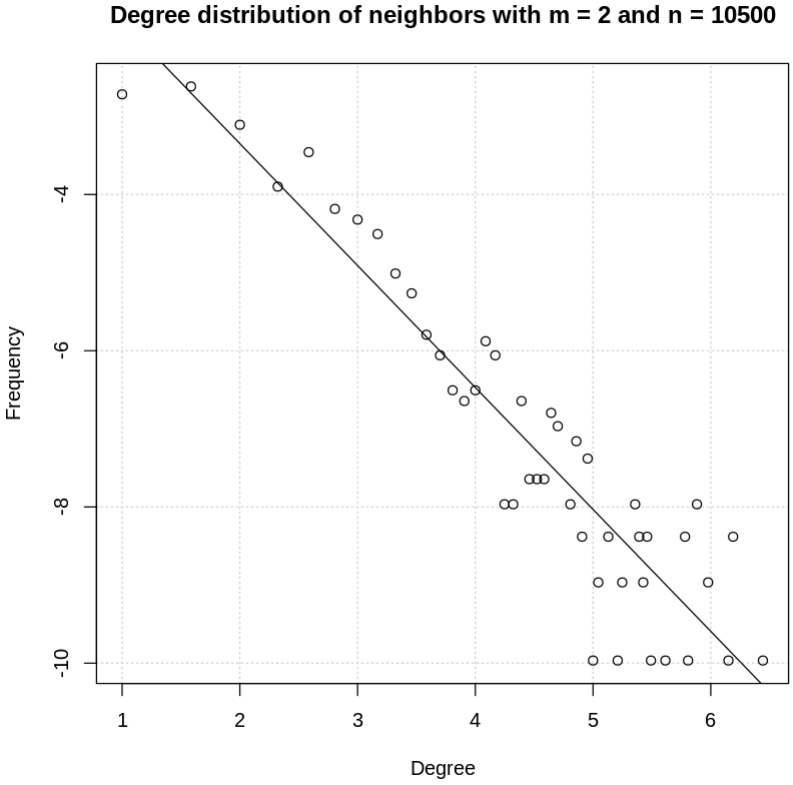

```
Slope with m = 2 and n = 10500: -2.417
```



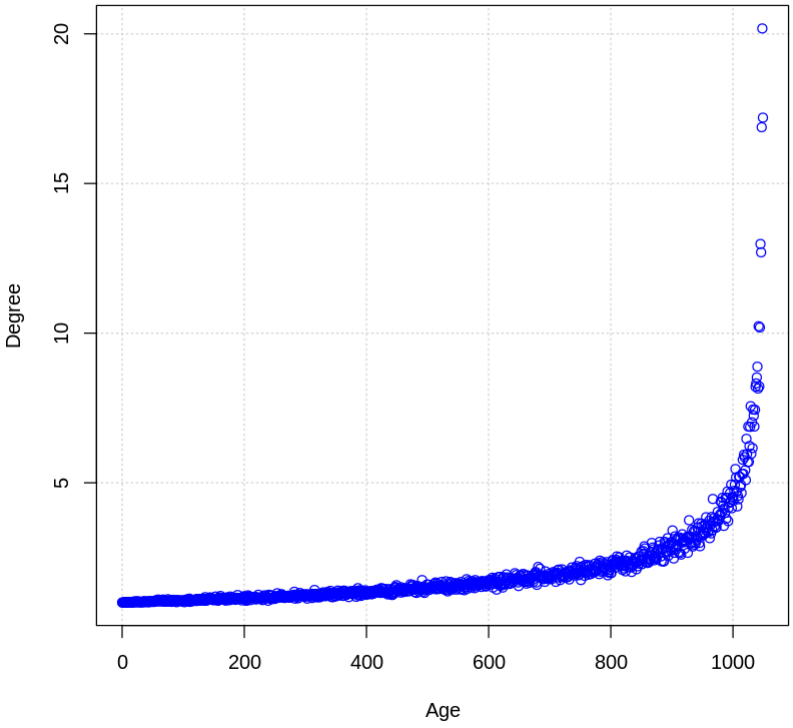
Slope with m = 2 and n = 1050: −1.106



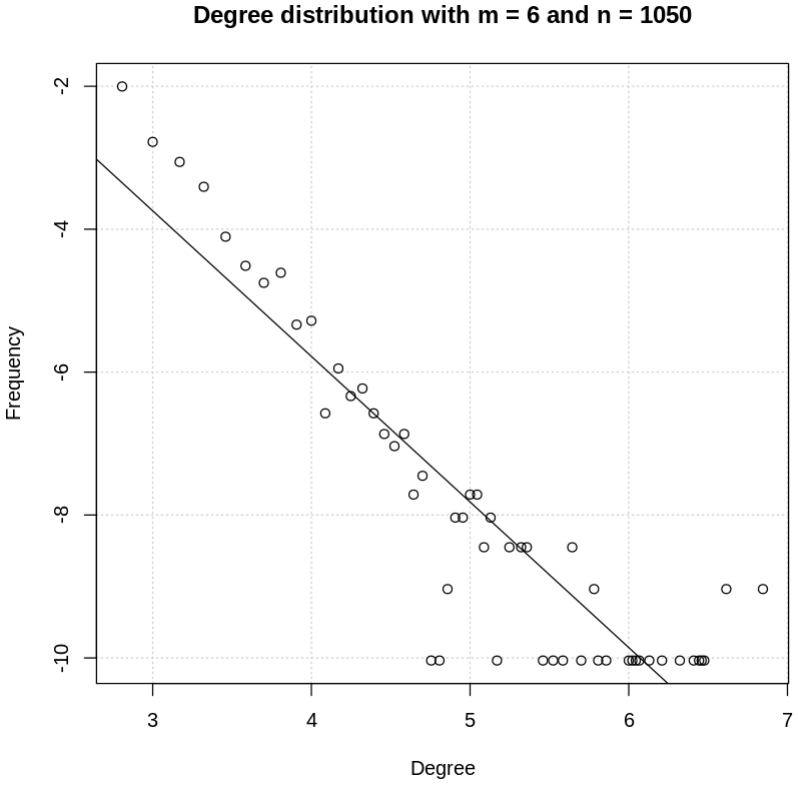
Slope with m = 2 and n = 10500: −1.561



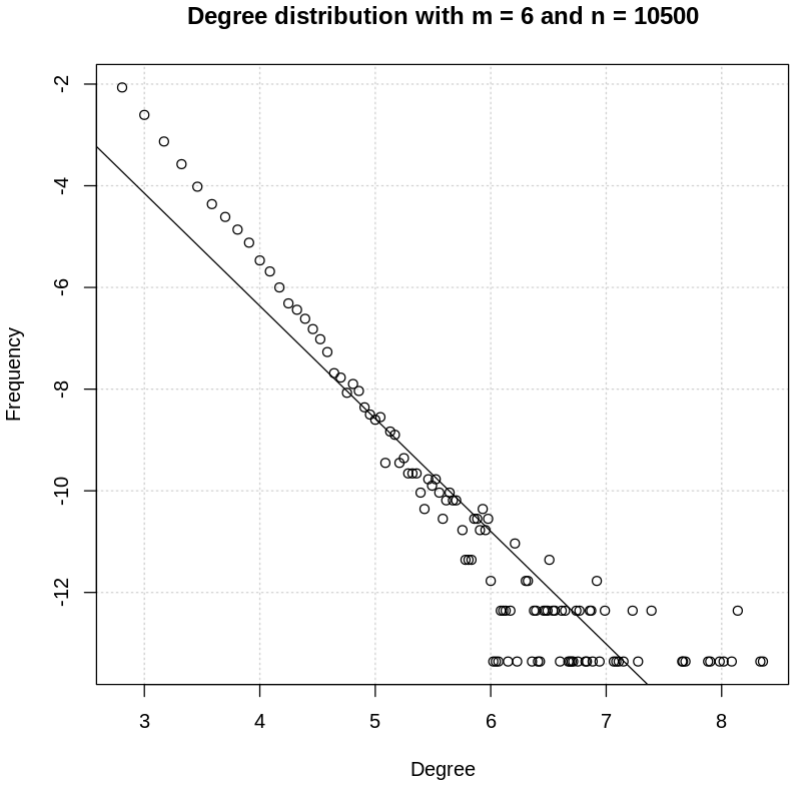

```
Percent of connected network with m = 6 and n = 1050: 1.000
m = 6, n = 1050: modularity = 0.247; assortativity = −0.027
m = 6, n = 10500: modularity = 0.245; assortativity = −0.003
```



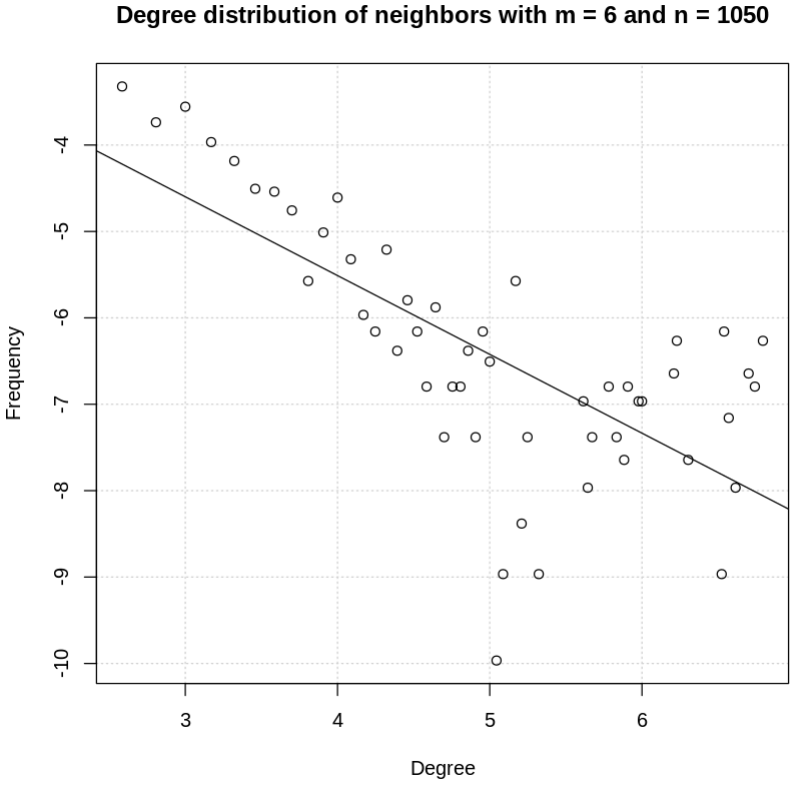

```
Slope with m = 6 and n = 1050: −2.037
```



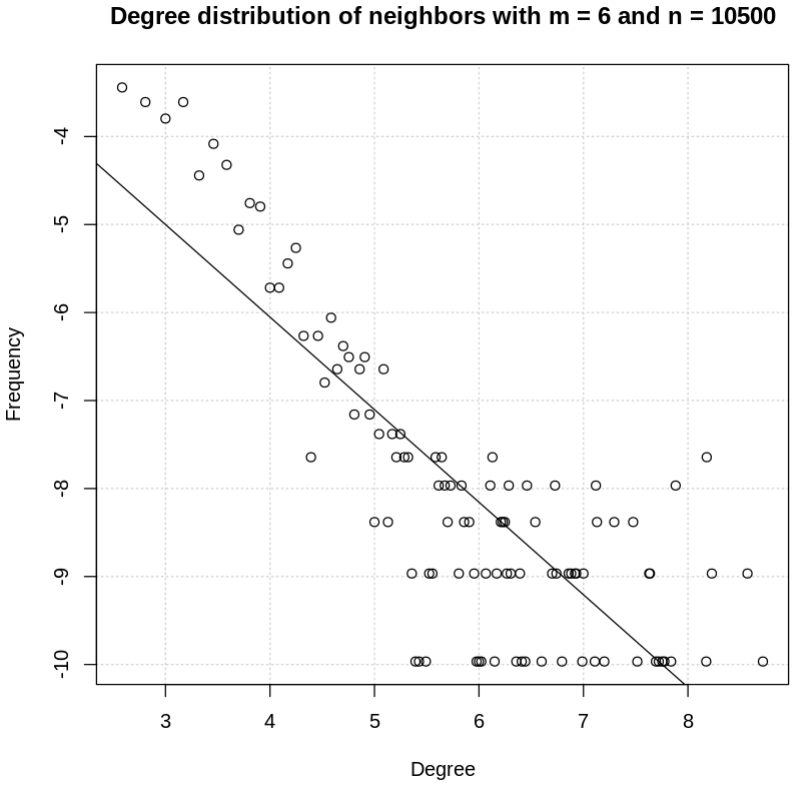
Slope with m = 6 and n = 10500: −2.217

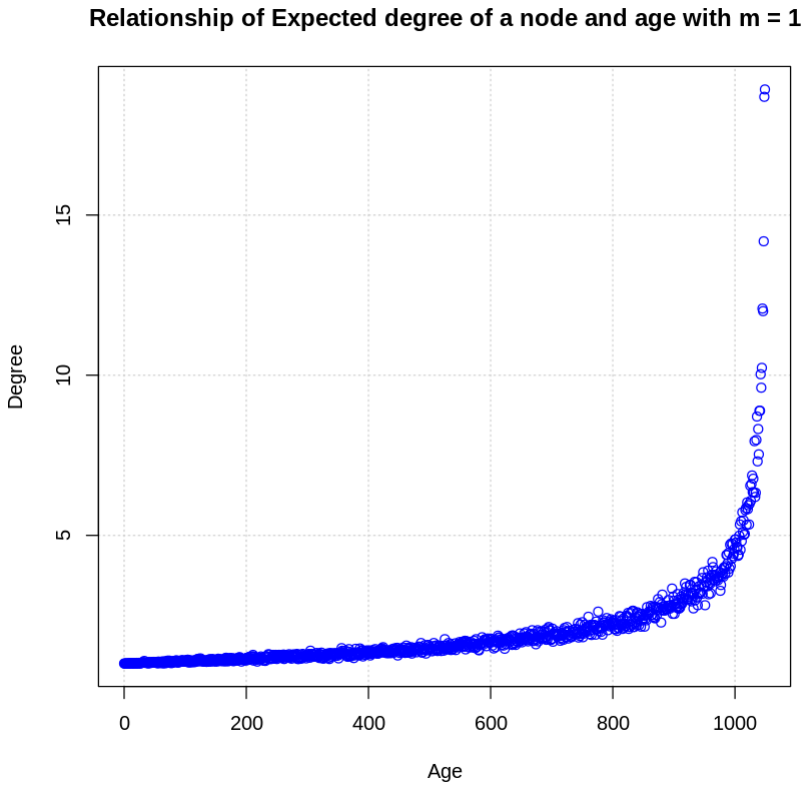


Slope with m = 6 and n = 1050: −0.912




```
Slope with m = 6 and n = 10500: −1.052
```





(a) Both of m=2 and m=6 networks are connected.

(b) m = 2, n = 1050: modularity = 0.524; assortativity = -0.057

m = 6, n = 1050: modularity = 0.247; assortativity = -0.027

Modularity decreases as m increases and Assortativity increases as m increases.

(c) m = 2, n = 10500: modularity = 0.527; assortativity = -0.010

m = 6, n = 10500: modularity = 0.245; assortativity = -0.003

Modularity decreases as m increases and Assortativity increases as m increases.

(d) node degree distribution

Slope with m = 2 and n = 1050: -2.200

Slope with m = 2 and n = 10500: -2.417

Slope with m = 6 and n = 1050: -2.037

Slope with m = 6 and n = 10500: -2.217

The slopes slightly increases as m increases.

(e) neighbor degree distribution

Slope with m = 2 and n = 1050: -1.106

Slope with m = 2 and n = 10500: -1.561

Slope with m = 6 and n = 1050: -0.912

Slope with m = 6 and n = 10500: -1.052

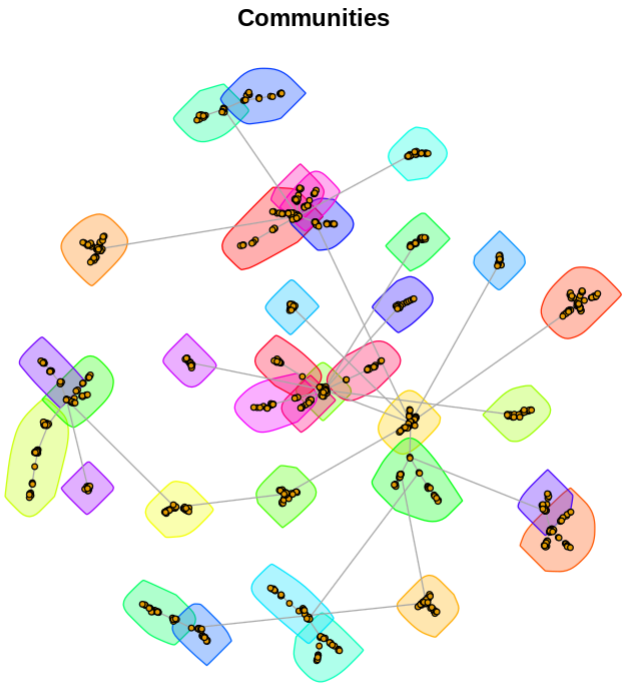
The slopes increases as m increases. The value of the slopes of neighbor degree distribution are larger than those of the node degree distribution.

(f) The trends are quite similar with different values of m. Both of m=2 and m=6 show that degree increases as m increases.

(h)

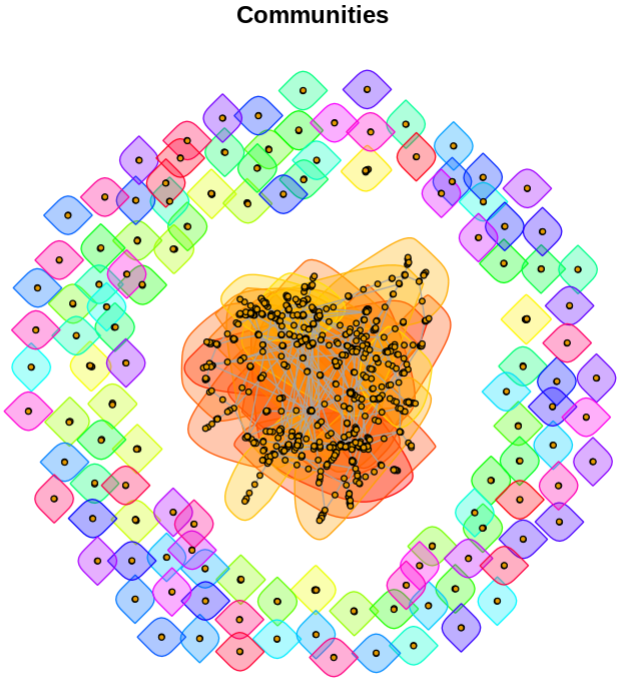
In [51]:
```
plot_communities <- function(g) {
    communities <- cluster_fast_greedy(g)
    plot(g, mark.groups=groups(communities), vertex.size=2, vertex.label="",
         main="Communities")
    modularity_score <- modularity(communities)
    fprintf("Modularity: %.3f", modularity_score)
}

g <- sample_pa(1050, m=1, directed=FALSE)
plot_communities(g)
```

```
Modularity: 0.933
```



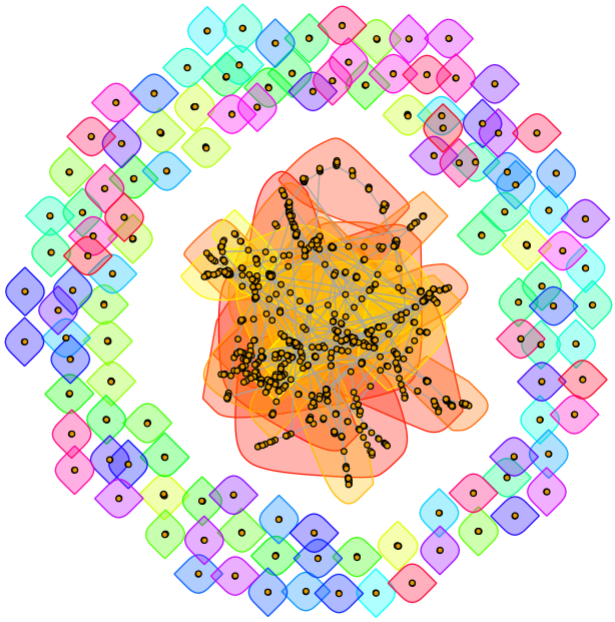
In [52]:
```
g_new1 <- sample_degseq(degree(g), method="simple.no.multiple")
plot_communities(g_new1)
```

```
Modularity: 0.852
```



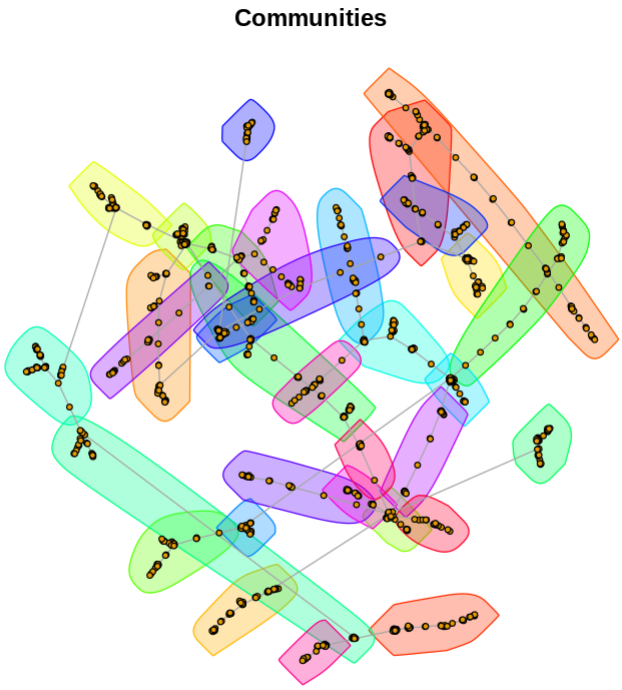
In [53]:
```
g_new2 <- sample_degseq(degree(g), method="simple.no.multiple.uniform")
plot_communities(g_new2)
```

```
Modularity: 0.846
```



In [54]:
```
g_new3 <- sample_degseq(degree(g), method="vl")
plot_communities(g_new3)
```

```
Modularity: 0.936
```



For the degseg, we tried the three methods:"vl", "simple.no.multiple", "simple.no.multiple.uniform". Based on the above results, for sub-matching with methods "simple.no.multiple" and "simple.no.multiple.uniform", they generated networks with a large number of unconnected vertices. However, for networks generated using preferential attachment model or "vl" method, they have more local high-connected regions and sparse connections among individual communities.

Also, from modularity scores we can see that preferential attachment model and "vl" method generate higher modularity than the other two sub-matching methods, which is corresponding to the network patterns we observe.

In [1]:
```
install.packages("igraph")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

In [2]:
```
install.packages("pracma")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

In [3]:
```
install.packages("matrixStats")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

In [4]:
```
install.packages("resample")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

In [5]:
```
install.packages("matrix")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Warning message:
“package ‘matrix’ is not available for this version of R

A version of this package for your version of R might be available elsewhere,
see the ideas at
https://cran.r-project.org/doc/manuals/r-patched/R-admin.html#Installing-packages”
Warning message:
“Perhaps you meant ‘Matrix’ ?”
```

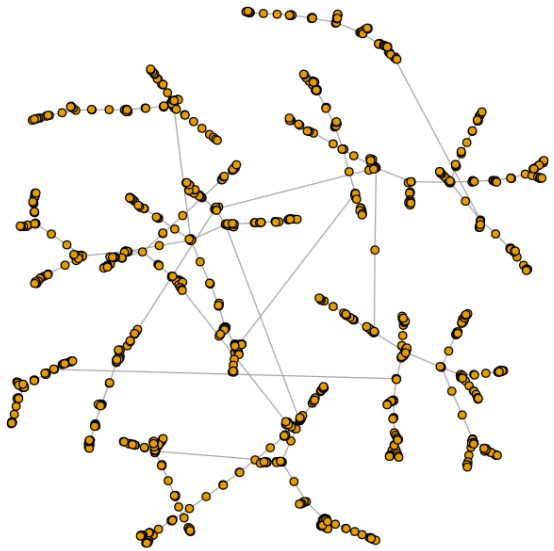
In [6]:
```
library('igraph')
library('Matrix')
library('pracma')
library('matrixStats')
library("resample")
```

```
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



Attaching package: ‘resample’


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

    colVars
```

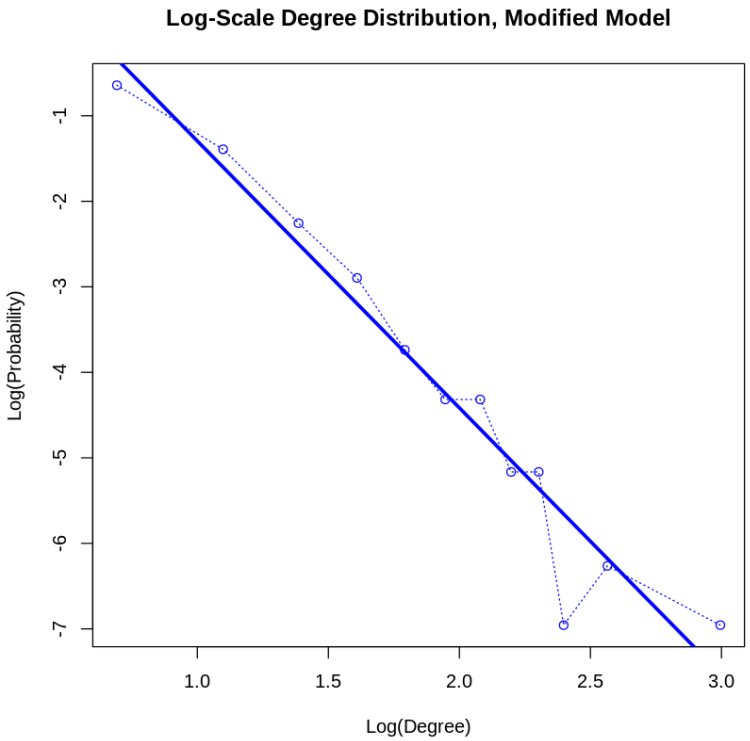
Question 1.3(a)

In [ ]:
```
# Setup for modified preferential attachment model graph creation
nodes_count <- 1050
model_params <- list(directed=FALSE, m=1, pa.exp=1, aging.exp=-1, deg.coef=1, age.coef=1, zero.deg.appeal=1, zero.age.appeal=0

# Creating and plotting the graph
mod_graph <- do.call(sample_pa_age, c(list(n=nodes_count), model_params))
plot(mod_graph, vertex.size=3, vertex.label=NA, main="Modified Preferential Attachment Model, Nodes = 1050")

# Calculating and plotting degree distribution
deg_dist <- degree.distribution(mod_graph)
indices <- which(deg_dist > 0)
log_degrees <- log(seq_along(deg_dist))[indices]
log_frequencies <- log(deg_dist)[indices]

plot(log_degrees, log_frequencies, main="Log-Scale Degree Distribution, Modified Model", xlab="Log(Degree)", ylab="Log(Probabi
lines(log_degrees, log_frequencies, lty=3, col="blue")

# Performing and displaying linear regression analysis
regression_result <- lm(log_frequencies ~ log_degrees)
cat("Linear regression results for the modified model:\n")
print(coef(regression_result))
abline(regression_result, col="blue", lwd=3)
```

Modified Preferential Attachment Model, Nodes = 1050

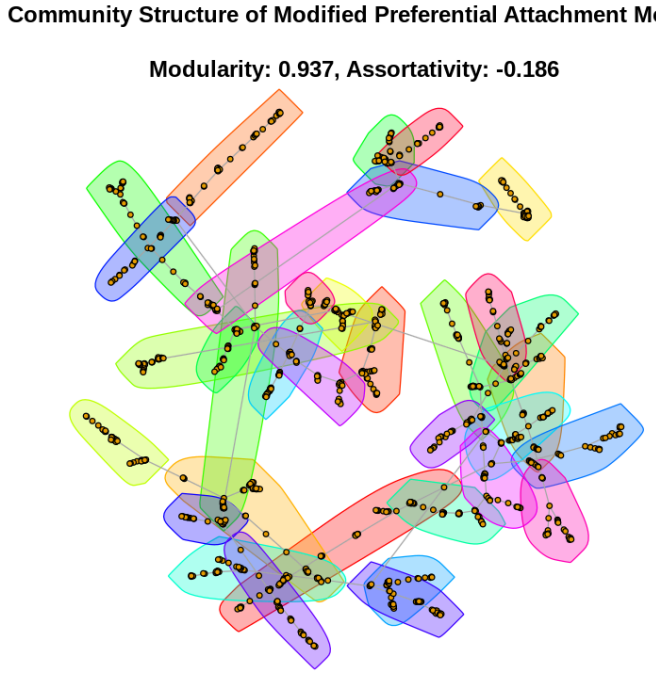


```
Linear regression results for the modified model:
(Intercept) log_degrees
   1.825836   -3.120841
```



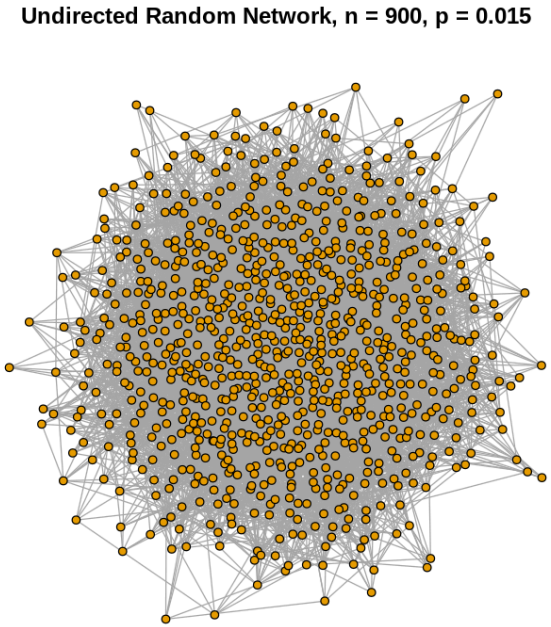
Question 1.3(b)

In [ ]:
```
# Community structure
community_structure <- fastgreedy.community(mod_graph)
modularity_value <- modularity(community_structure)
degree_assortativity <- assortativity_degree(mod_graph, directed=FALSE)
plot(mod_graph, mark.groups=groups(community_structure), vertex.size=2, vertex.label=NA,
     main=sprintf("Community Structure of Modified Preferential Attachment Model
                  \nModularity: %.3f, Assortativity: %.3f", modularity_value, degree_assortativity))
```



Question 2.1(a)

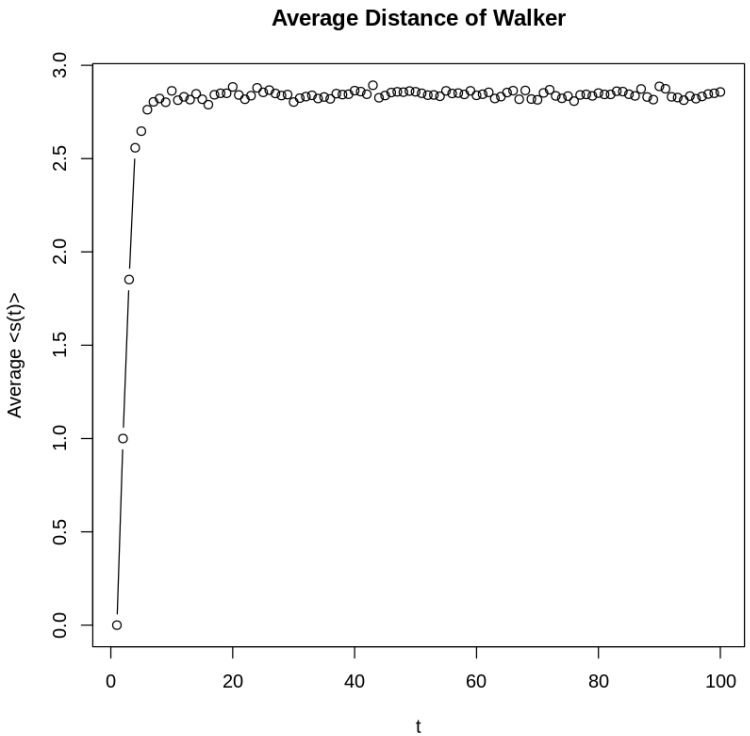
In [ ]:
```
g <- random.graph.game(n=900, p=0.015, directed=FALSE)
plot(g, vertex.size=3, vertex.label=NA,
     main="Undirected Random Network, n = 900, p = 0.015")
```

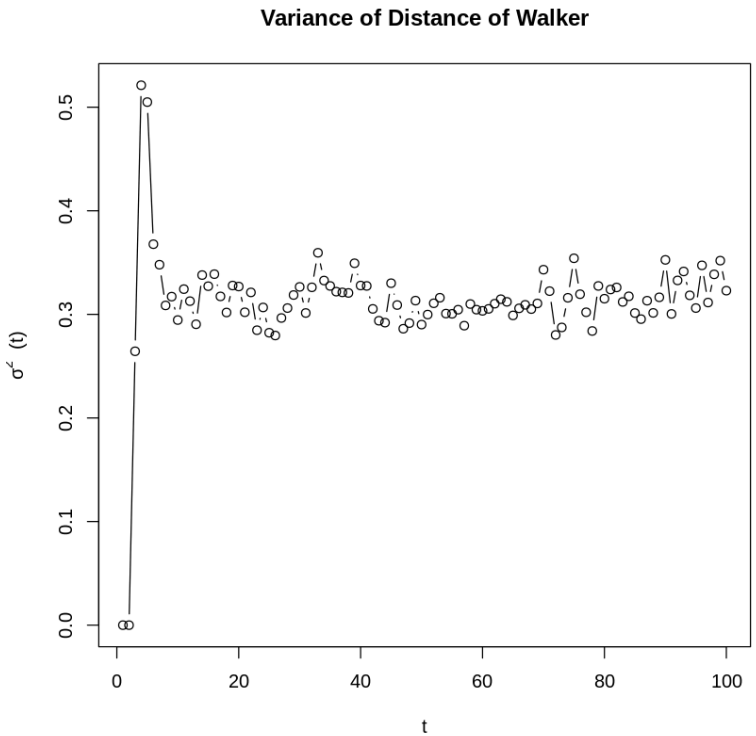


Question 2.1(b)

In [ ]:
```
extract_largest_component <- function(network_graph){
    if(is_connected(network_graph)){
            main_component <- network_graph
        }
        else{
            g_components <- components(network_graph)
            gcc_idx <- which.max(g_components$csize)
            main_component <- induced_subgraph(network_graph, which(g_components$membership == gcc_idx))
        }

    return(main_component)
}
```

```
perform_walk_analysis <- function(g, n, steps_limit=100, simulation_runs=1000){
    shortest_lens <- matrix(NA, nrow=simulation_runs, ncol=steps_limit) # Initialize with NA
    for(i in 1:simulation_runs){
        gcc <- extract_largest_component(g)
        v_start <- sample(V(gcc), 1)
        vs <- random_walk(gcc, v_start, steps_limit) # Ensure this returns up to max_steps vertices

        # Loop through each step up to the length of vs or steps_limit
        for(step in 1:min(length(vs), steps_limit)){
            # Calculate shortest path length from the starting node to the current node
            path_length <- shortest.paths(gcc, v_start, vs[step])
            shortest_lens[i, step] <- path_length
        }
    }

    avg_distances <- colMeans(shortest_lens, na.rm = TRUE) # Ignore NA values for uncompleted steps
    distance_variance <- apply(shortest_lens, 2, var, na.rm = TRUE) # Calculate variance for each step, ignoring NA

    # Plotting
    plot(1:steps_limit, avg_distances, type="b", xlab="t", ylab="Average <s(t)>", main="Average Distance of Walker")
    plot(1:steps_limit, distance_variance, type="b", xlab="t", ylab=expression(sigma^2~"(t)"), main="Variance of Distance of W
}

perform_walk_analysis(g, 900)
```

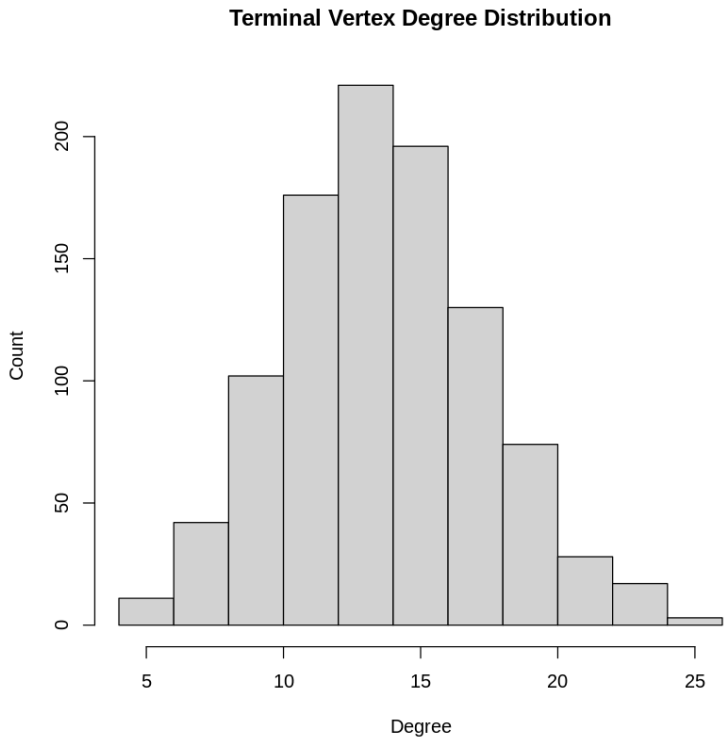

Average Distance of Walker
Average <s(t)>
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
20
40
60
80
100
t

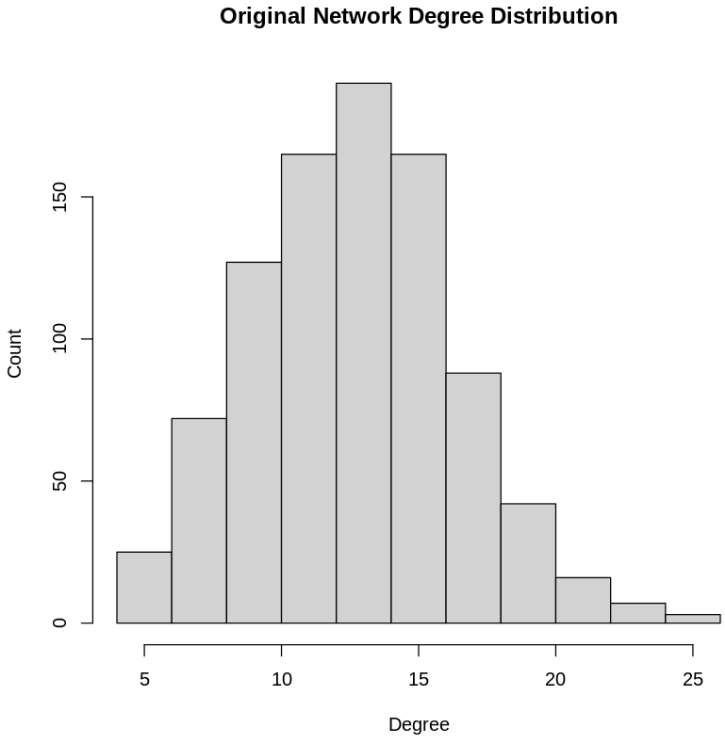


Question 2.1(c)

In [ ]:
```
extract_largest_component <- function(network_graph){
    if(is_connected(network_graph)){
            main_component <- network_graph
        }
        else{
            g_components <- components(network_graph)
            gcc_idx <- which.max(g_components$csize)
            main_component <- induced_subgraph(network_graph, which(g_components$membership == gcc_idx))
        }

    return(main_component)
}
```

```
fetch_terminal_degrees <- function(network, steps_limit=100, trial_count=1000){
    terminal_degrees_matrix <- matrix(0, nrow=trial_count, ncol=1)
    for(trial in 1:trial_count){
        main_component <- extract_largest_component(network)

        starting_vertex <- sample(V(main_component), 1)
        path_vertices <- random_walk(main_component, starting_vertex, steps_limit)
        terminal_degrees_matrix[trial, 1] <- degree(main_component, path_vertices[length(path_vertices)])
    }

    return (terminal_degrees_matrix)
}

final_vertex_degrees <- fetch_terminal_degrees(g)

hist(final_vertex_degrees, main="Terminal Vertex Degree Distribution", xlab="Degree", ylab="Count")
hist(degree(g), main="Original Network Degree Distribution", xlab="Degree", ylab="Count")
```


Terminal Vertex Degree Distribution
Count
0
50
100
150
200
5
10
15
20
25
Degree



Answer: The degree distributions of the nodes encountered at the conclusion of the random walk closely mirror the degree distribution observed across the entire graph, both adhering to a binomial distribution pattern.

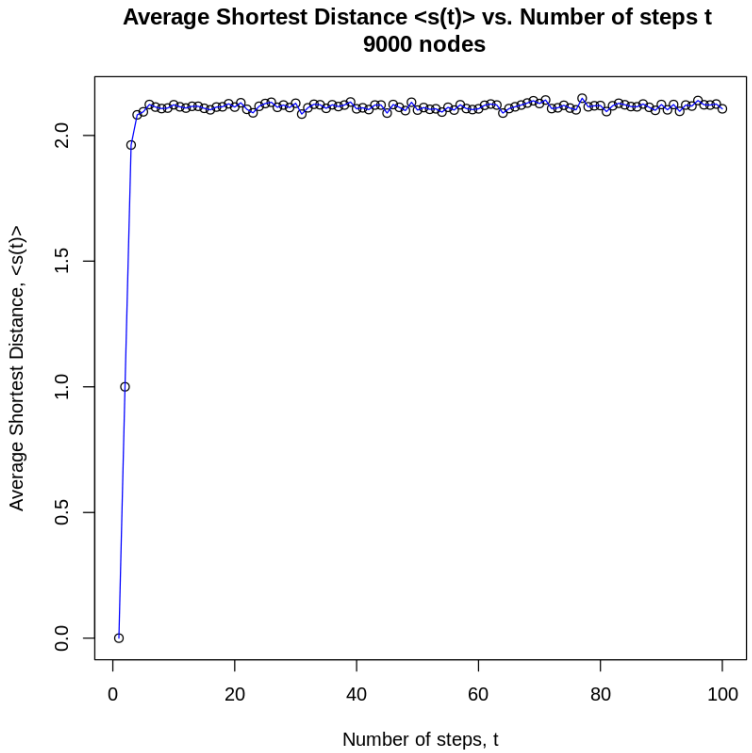
Question 2.1(d)

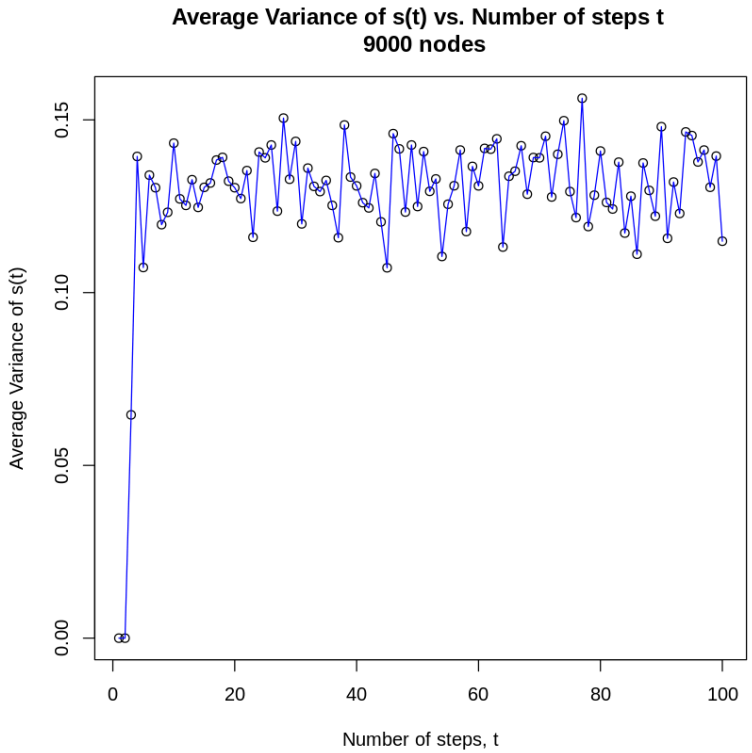
Answer: When examining the plots for the average shortest distance and the average variance, it's noticeable that as the network size increases, both the average shortest distance and variance from the random walk tend to stabilize more quickly, reaching lower values after fewer steps. This observation aligns with the intuition that a larger network, characterized by greater complexity and connectivity, allows for shorter paths between any two given nodes, thereby reducing variance. Additionally, the network's diameter serves as a key indicator of increasing connectivity with network expansion. In the experiments conducted here, larger networks exhibited a smaller diameter (3) compared to smaller networks (5), indicating that the maximum separation between any

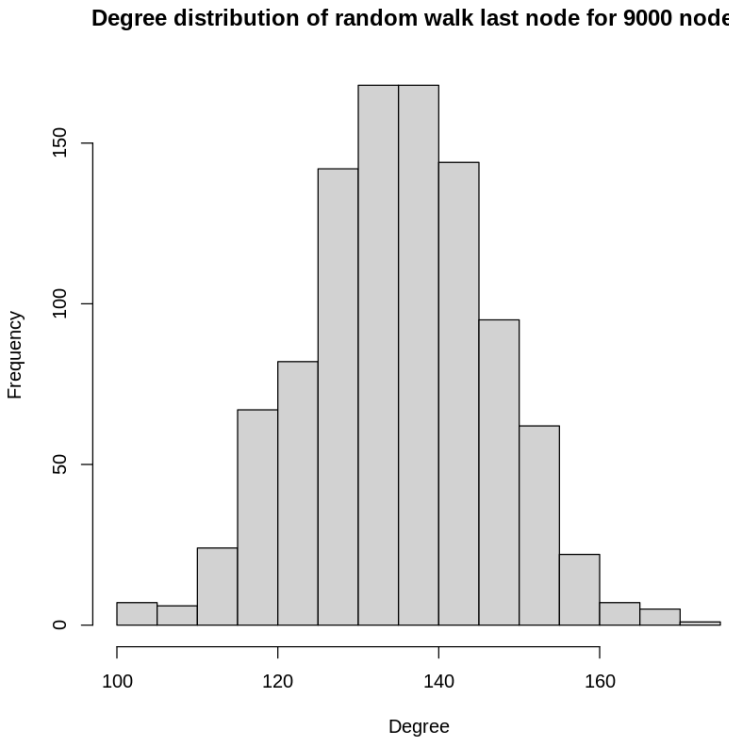
two nodes in the network decreases as the network grows. This reduction in diameter underscores the enhanced interconnectedness within larger networks.

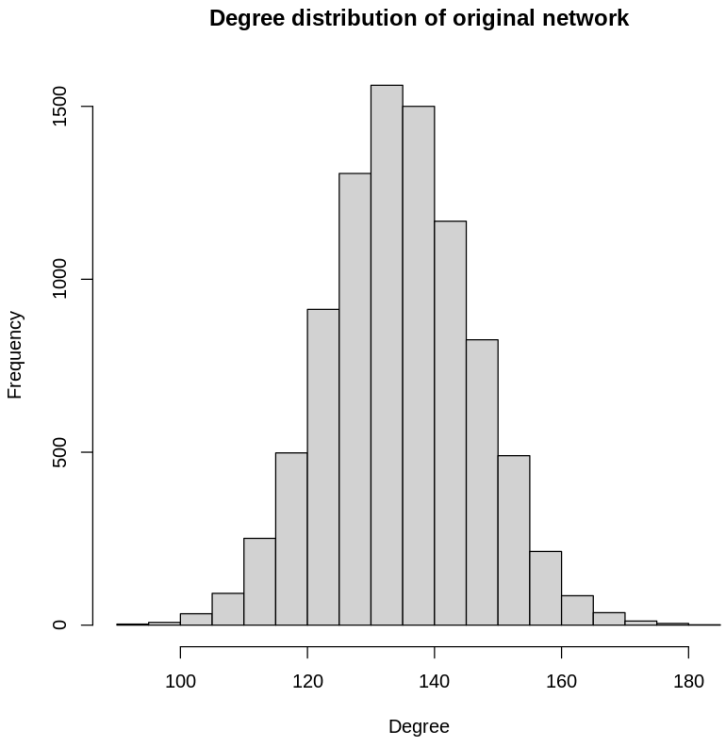
In [ ]:
```
distance_measurements <- matrix(data=0.0, nrow=1000, ncol=100)
terminal_degrees <- matrix(data=0.0, nrow=1000, ncol=1)

for (iteration in 1:1000) {
  network <- random.graph.game(n=9000, p=0.015, directed=FALSE)
  main_component <- extract_largest_component(network) # Correct function name already used

  starting_point <- sample(V(main_component), 1)
  path_traversed <- random_walk(main_component, starting_point, 100) # Confirm this function returns 100 steps

  # Filling distance measurements based on walked path
  for (movement in 1:100) {
    if (movement <= length(path_traversed)) {
      distance_measurements[iteration, movement] <- shortest.paths(main_component, starting_point, path_traversed[movement])
    } else {
      distance_measurements[iteration, movement] <- NA
    }
  }

  terminal_degrees[iteration, 1] <- degree(main_component, path_traversed[length(path_traversed)])
}
```

In [ ]:
```
plot(seq(100), colMeans(distance_measurements),
     main=sprintf("Average Shortest Distance <s(t)> vs. Number of steps t \n %d nodes", n),
     xlab = "Number of steps, t", ylab = "Average Shortest Distance, <s(t)>")
lines(seq(100), colMeans(distance_measurements), col="blue")

plot(seq(100), colVars(distance_measurements),
     main=sprintf("Average Variance of s(t) vs. Number of steps t \n %d nodes", n),
     xlab = "Number of steps, t", ylab = "Average Variance of s(t)")
lines(seq(100), colVars(distance_measurements), col="blue")

hist(terminal_degrees, xlab='Degree', ylab='Frequency',
     main="Degree distribution of random walk last node for 9000 nodes")
hist(degree(g), xlab='Degree', ylab='Frequency',
     main="Degree distribution of original network")
```


Average Shortest Distance <s(t)> vs. Number of steps t
9000 nodes
Average Shortest Distance, <s(t)>
0.0
0.5
1.0
1.5
2.0
0
20
40
60
80
100
Number of steps, t

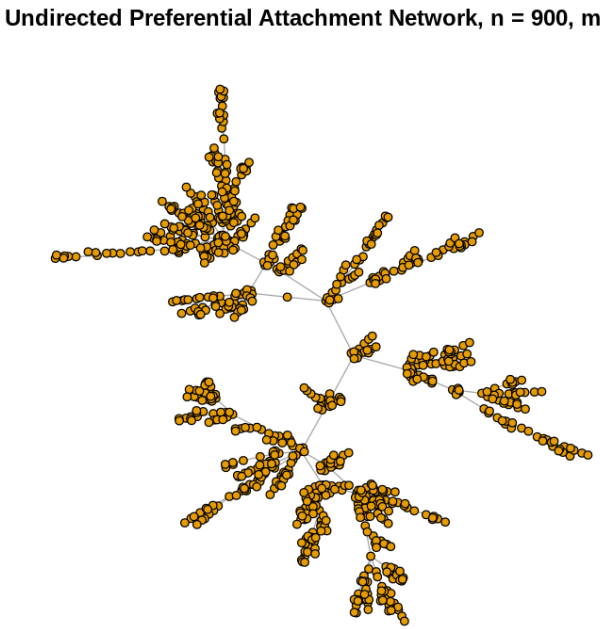

Average Variance of s(t) vs. Number of steps t
9000 nodes
Average Variance of s(t)
0.00
0.05
0.10
0.15
0
20
40
60
80
100
Number of steps, t


Degree distribution of random walk last node for 9000 nodes
Frequency
0
50
100
150
100
120
140
160
Degree



Question 2.2(a)

In [ ]:
```
g <- sample_pa(900, m=1, directed=F)
plot(g, vertex.size=3, vertex.label=NA,
     main="Undirected Preferential Attachment Network, n = 900, m = 1")
```



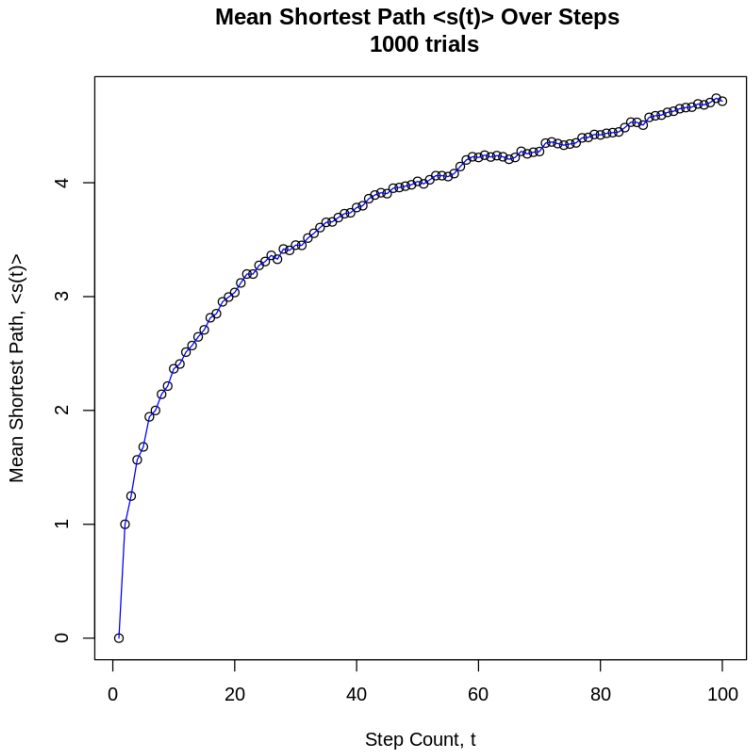
Question 2.2(b)

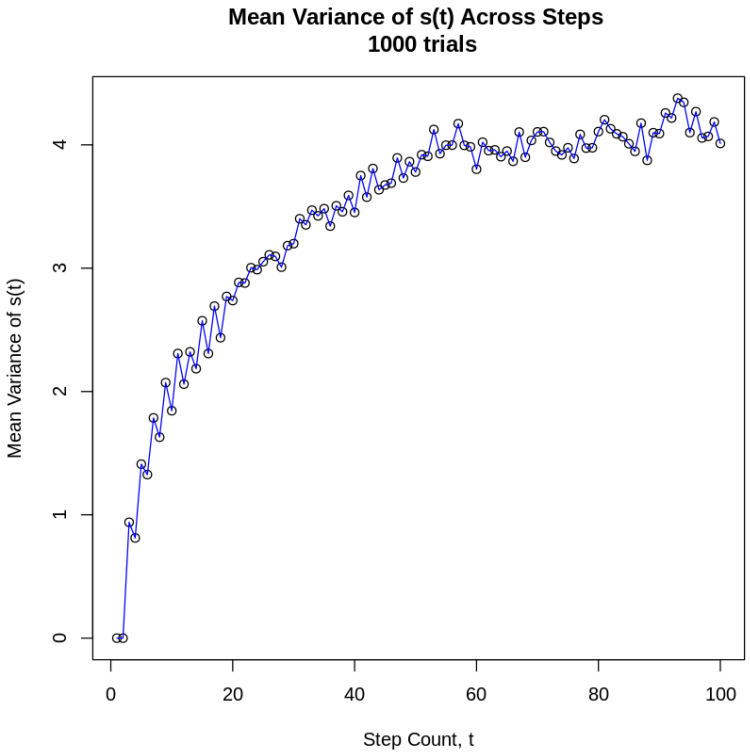
In [ ]:
```
node_count <- 900
edge_factor <- 1
trial_runs <- 1000
max_walks <- 100
distance_matrix <- matrix(data=0.0, nrow=trial_runs, ncol=max_walks)
terminal_deg_matrix <- matrix(data=0.0, nrow=trial_runs, ncol=1)

for (run in 1:trial_runs) {
  # Reconstructing the network and extracting its largest component as before
  network <- barabasi.game(node_count, m=edge_factor, directed=FALSE)
  largest_comp <- if (is.connected(network)) network else {
    components_info <- clusters(network)
```

```
    biggest_cluster_index <- which.max(components_info$csize)
    induced_subgraph(network, which(components_info$membership == biggest_cluster_index))
  }

  start_vertex <- sample(V(largest_comp), 1)
  path_taken <- random_walk(largest_comp, start_vertex, max_walks) # Assume this returns a vector of visited vertices

  # Assuming path_taken is ordered, calculate distances for each step
  for (step in 1:length(path_taken)) {
    if (step <= max_walks) {
      # Ensure path_taken[step] correctly references a single vertex for each step
      distance_matrix[run, step] <- shortest.paths(largest_comp, start_vertex, path_taken[step])
    }
  }

  terminal_vertex <- path_taken[length(path_taken)]
  terminal_deg_matrix[run, 1] <- degree(largest_comp, terminal_vertex)
}
```

In [ ]:

```
plot(1:max_walks, colMeans(distance_matrix),
     main=sprintf("Mean Shortest Path <s(t)> Over Steps \n %d trials",
                  trial_runs),
     xlab = "Step Count, t", ylab = "Mean Shortest Path, <s(t)>")
lines(1:max_walks, colMeans(distance_matrix), col="blue")

plot(1:max_walks, colVars(distance_matrix),
     main=sprintf("Mean Variance of s(t) Across Steps \n %d trials",
                  trial_runs),
     xlab = "Step Count, t", ylab = "Mean Variance of s(t)")
lines(1:max_walks, colVars(distance_matrix), col="blue")
```

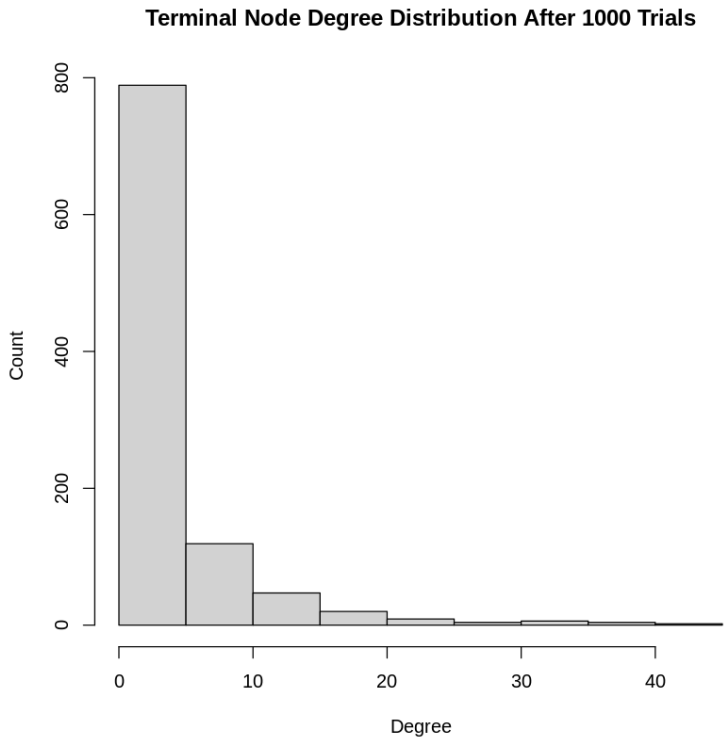

Mean Shortest Path <s(t)> Over Steps
1000 trials
Mean Shortest Path, <s(t)>
Step Count, t
0
1
2
3
4
0
20
40
60
80
100

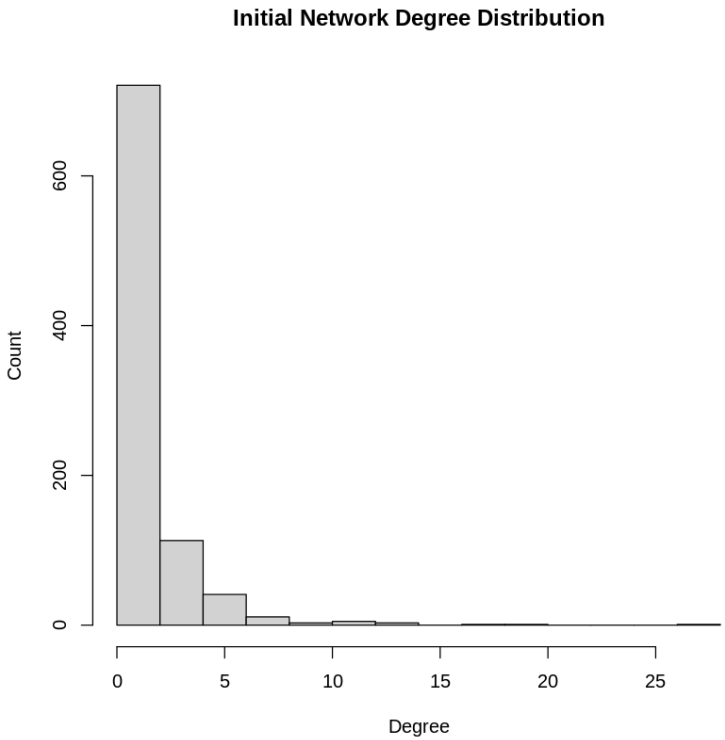


Question 2.2(c)

Answer: The degree distribution of the nodes encountered at the conclusion of the random walk closely resembles the graph's overall degree distribution, both adhering to a power-law distribution.

In [ ]:
```
hist(terminal_deg_matrix, xlab='Degree', ylab='Count',
     main="Terminal Node Degree Distribution After 1000 Trials")
hist(degree(g), xlab='Degree', ylab='Count',
     main="Initial Network Degree Distribution")
```


Terminal Node Degree Distribution After 1000 Trials
Count
0
200
400
600
800
0
10
20
30
40
Degree



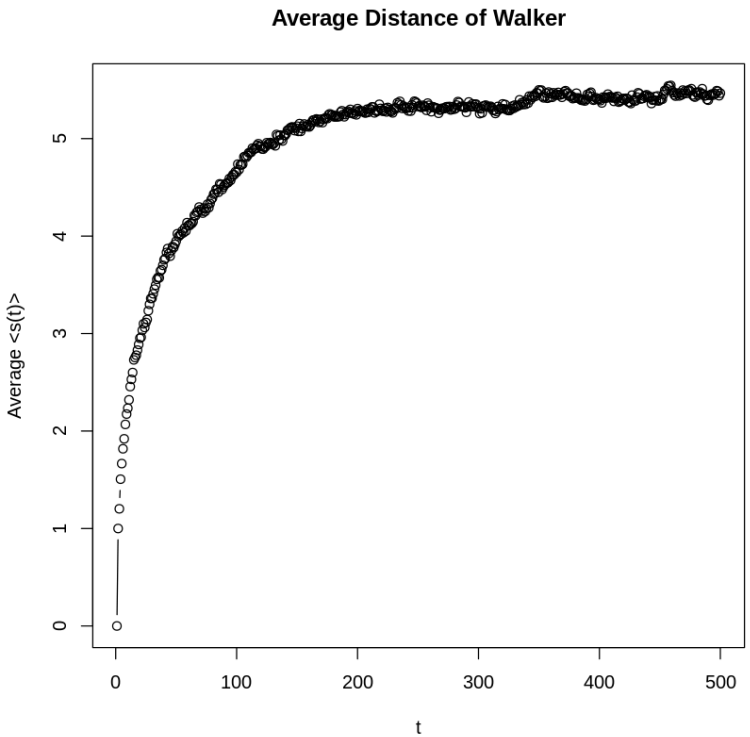
Question 2.2(d)

Answer:When examining outcomes from networks of 90 and 9000 nodes, it becomes apparent that both the average shortest distance and average variance incrementally increase with more steps taken. In the context of larger networks, these metrics settle at values exceeding those of smaller networks. This phenomenon is aligned with the characteristics observed in undirected preferential attachment networks, as illustrated in 2(a), which display pronounced intra-community connections contrasted by weaker links between communities. Consequently, the extent of random walks and the scale of the network amplify the likelihood of concluding the walk at a node situated further away, thus extending the average shortest distance. This assertion is further supported by the diameters observed in the networks: a diameter of 29 for the 9000-node network versus a mere 14 for the 90-node network, indicating a direct relationship between network size and the breadth of node-to-node separation.

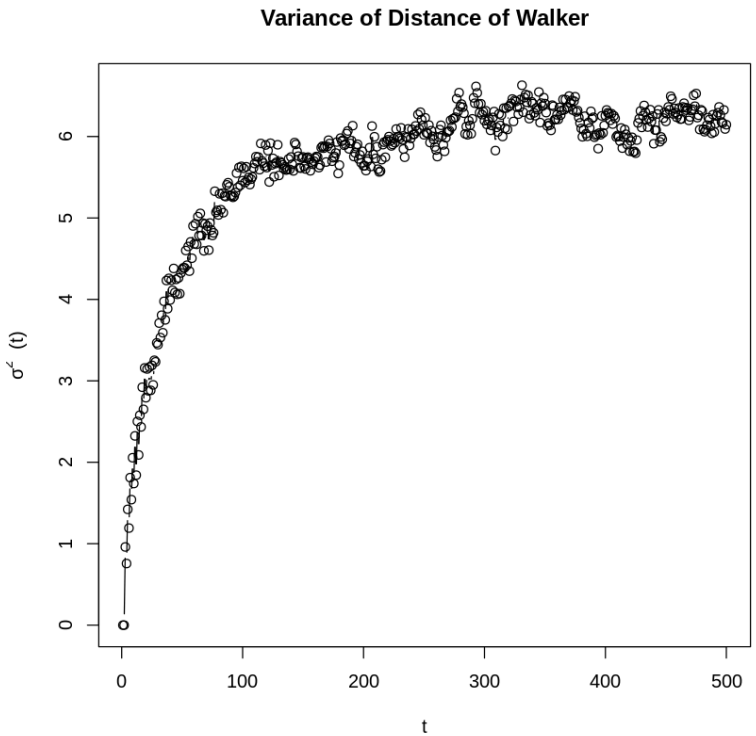
In [7]:
```
extract_largest_component <- function(network_graph){
    if(is_connected(network_graph)){
            main_component <- network_graph
        }
        else{
            g_components <- components(network_graph)
            gcc_idx <- which.max(g_components$csize)
            main_component <- induced_subgraph(network_graph, which(g_components$membership == gcc_idx))
        }

    return(main_component)
}
perform_walk_analysis <- function(g, n, steps_limit=100, simulation_runs=1000){
    shortest_lens <- matrix(NA, nrow=simulation_runs, ncol=steps_limit) # Initialize with NA
    for(i in 1:simulation_runs){
        gcc <- extract_largest_component(g)
        v_start <- sample(V(gcc), 1)
        vs <- random_walk(gcc, v_start, steps_limit) # Ensure this returns up to max_steps vertices

        # Loop through each step up to the length of vs or steps_limit
        for(step in 1:min(length(vs), steps_limit)){
            # Calculate shortest path length from the starting node to the current node
            path_length <- distances(gcc, v_start, vs[step])
            shortest_lens[i, step] <- path_length
        }
    }

    avg_distances <- colMeans(shortest_lens, na.rm = TRUE) # Ignore NA values for uncompleted steps
    distance_variance <- apply(shortest_lens, 2, var, na.rm = TRUE) # Calculate variance for each step, ignoring NA

    # Plotting
    plot(1:steps_limit, avg_distances, type="b", xlab="t", ylab="Average <s(t)>", main="Average Distance of Walker")
    plot(1:steps_limit, distance_variance, type="b", xlab="t", ylab=expression(sigma^2~"(t)"), main="Variance of Distance of W
}

node_sizes <- c(90, 9000)
step_limits <- c(500, 5000)
for(index in seq_along(node_sizes)){
    network_graph <- sample_pa(n=node_sizes[index], m=1, directed=FALSE)
    cat(sprintf("n=%d, ", node_sizes[index]))
```

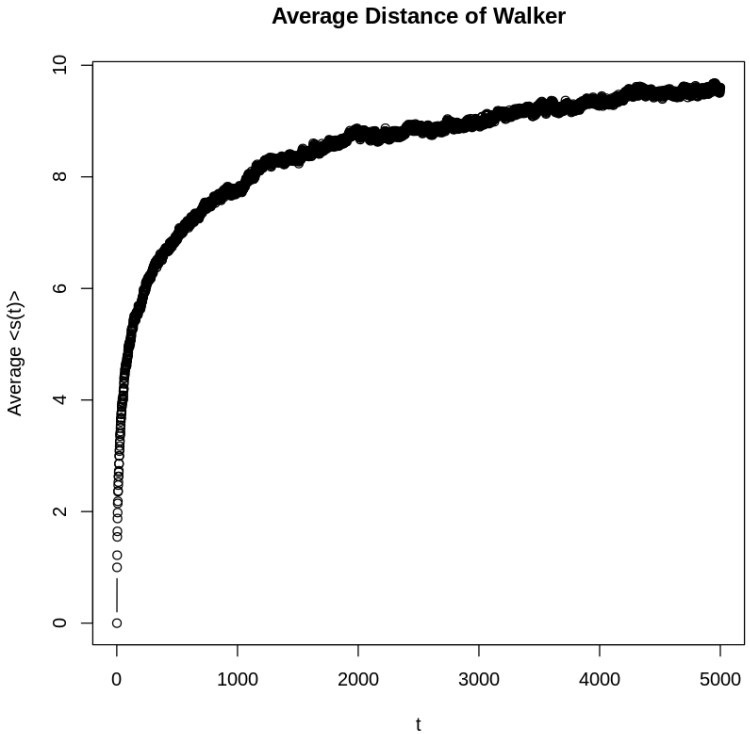
```
    cat(sprintf("diameter: %d\n", diameter(network_graph)))
    perform_walk_analysis(network_graph, node_sizes[index], steps_limit=step_limits[index])
}
```

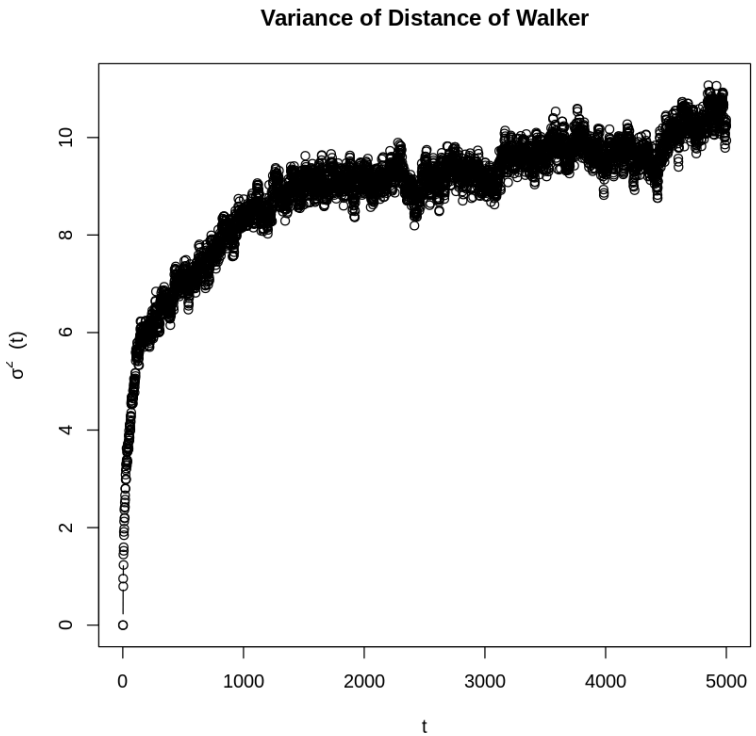
```
n=90, diameter: 14
```




```
n=9000, diameter: 29
```


Variance of Distance of Walker
σ^2 (t)
0
1
2
3
4
5
6
0
100
200
300
400
500
t


Average Distance of Walker
Average <s(t)>
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
t


Variance of Distance of Walker
σ² (t)
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
t

In [ ]:

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 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 900 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 900-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. For example, 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?

Ans:

The Pearson Correlation Coefficient is 0.821770, indicating a moderate to strong positive correlation between node degree and visitation probability. The positive correlation coefficient confirms that nodes with higher degrees are more likely to be visited in the random walks.

```
install.packages("igraph")
install.packages("resample")
install.packages("textTinyR")
install.packages("Matrix")
install.packages("pracma")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

also installing the dependencies ‘Rcpp’, ‘RcppArmadillo’, ‘BH’


Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

```
library(igraph)
library(resample)
library(textTinyR)
library(Matrix)
library(pracma)
```

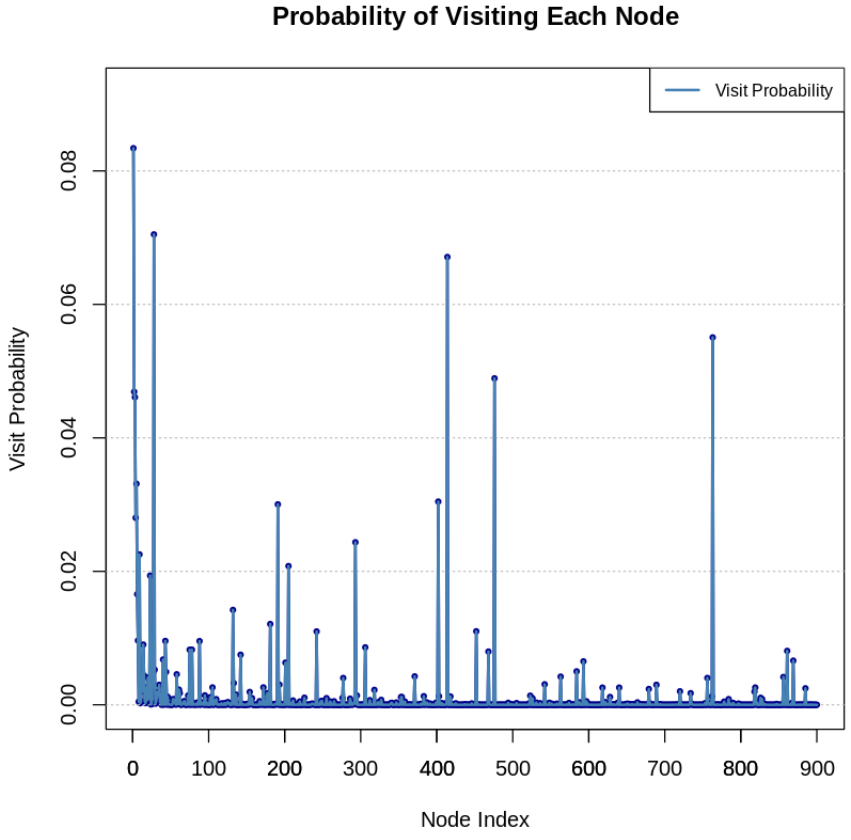
```
Attaching package: ‘igraph’

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

    decompose, spectrum

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

    union

Loading required package: Matrix

Attaching package: ‘pracma’

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

    expm, lu, tril, triu
```

```
# Function to create a transition matrix for a graph
create_transition_matrix <- function(g) {
  adj_matrix <- as_adjacency_matrix(g)
  # Set diagonal to 1 for nodes with no outbound edges to ensure no dead ends
  adj_matrix[diag(rowSums(adj_matrix) == 0)] <- 1
  # Normalize rows to sum to 1
  transition_matrix <- adj_matrix / rowSums(adj_matrix)
  return(transition_matrix)
}

# Function to perform a random walk on a graph
random_walk <- function(g, num_steps, start_node, transition_matrix = NULL) {
  if (is.null(transition_matrix)) {
    transition_matrix <- create_transition_matrix(g)
  }
```

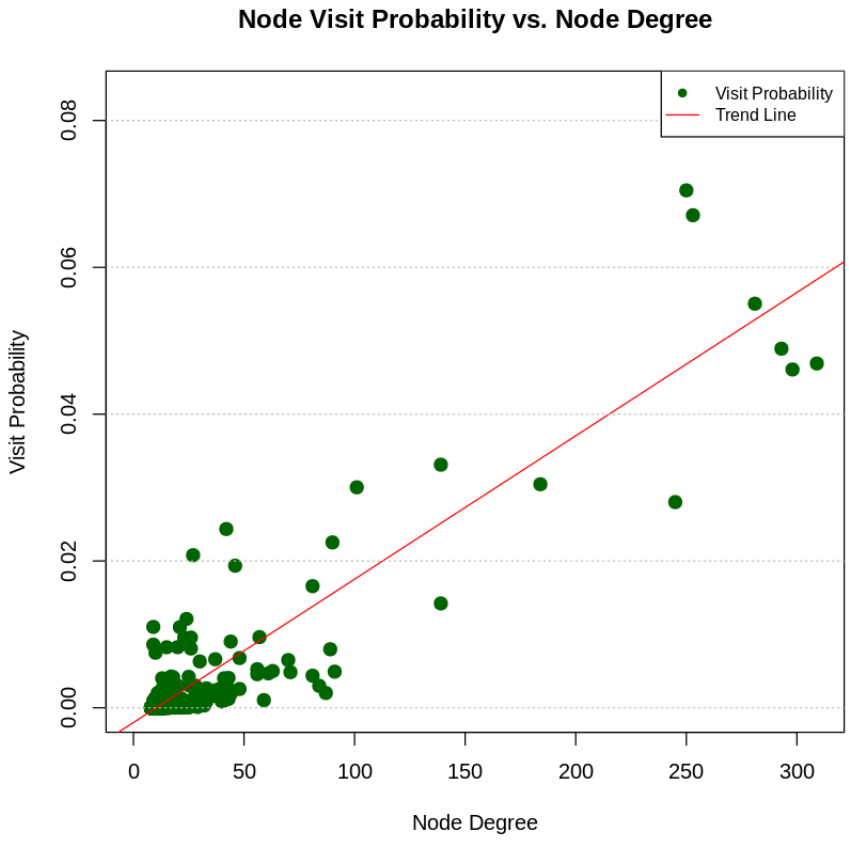
```
  walked_nodes <- numeric(num_steps)
  current_node <- start_node

  for (i in 1:num_steps) {
    next_node <- sample(1:vcount(g), 1, prob = transition_matrix[current_node, ])
    walked_nodes[i] <- next_node
    current_node <- next_node
  }

  return(walked_nodes)
}

# Constants for simulation
num_nodes <- 900
out_degree <- 4
num_iterations <- 1000
num_steps <- 1000

# Initialize vector to store visit counts
visited_nodes <- numeric(num_nodes)

# Create and merge two networks
network1 <- barabasi.game(num_nodes, m = out_degree, directed = TRUE)
network2 <- barabasi.game(num_nodes, m = out_degree, directed = TRUE)
shuffled_network2 <- permute(network2, sample(num_nodes))
merged_network <- add_edges(network1, c(t(as_edgelist(shuffled_network2))))

# Start random walk simulations
for (i in 1:num_iterations) {
  if (is.connected(merged_network)) {
    start_node <- sample(num_nodes, 1)
    walked_nodes <- random_walk(merged_network, num_steps, start_node)
  } else {
    # Handle non-connected network by focusing on the largest connected component
    components <- components(merged_network)
    largest_component <- which.max(components$csize)
    original_ids <- which(components$membership == largest_component)
    gcc <- induced_subgraph(merged_network, original_ids)
    start_node <- sample(vcount(gcc), 1)
    walked_nodes <- random_walk(gcc, num_steps, start_node)
    # Remap node indices to original
    walked_nodes <- original_ids[walked_nodes]
  }

  # Update visit count for each node
  for (node in walked_nodes) {
    visited_nodes[node] <- visited_nodes[node] + 1
  }
}
```

```
# Calculate probability of visiting each node
node_probabilities <- visited_nodes / (num_steps * num_iterations)
```

```
# Plot the probability distribution of visiting each node
plot(seq_len(num_nodes), node_probabilities, type = 'o', col = 'darkblue',
     main = 'Probability of Visiting Each Node',
     xlab = 'Node Index',
     ylab = 'Visit Probability',
     pch = 20, cex = 0.6, lwd = 2, ylim = c(0, max(node_probabilities) * 1.1))

# Add a grid to improve readability
grid(nx = NA, ny = NULL, col = "gray", lty = "dotted")

# Enhance the line connecting the points
lines(seq_len(num_nodes), node_probabilities, col = 'steelblue', lwd = 2)

# Add more detailed axis labels if needed
axis(side = 1, at = seq(0, num_nodes, by = 100), las = 1)
#axis(side = 2, las = 1)

# Add a legend
legend("topright", legend = "Visit Probability",
        col = "steelblue", lty = 1, lwd = 2, cex = 0.8)
```



```
# Get degrees of nodes in the merged network
# 'all' sums in-degrees and out-degrees
node_degrees <- degree(merged_network, mode = "all")

# Plot node visit probabilities against node degrees
plot(node_degrees, node_probabilities, xlab = 'Node Degree', ylab = 'Visit Probability',
     main = 'Node Visit Probability vs. Node Degree', pch = 19, col = 'darkgreen',
     cex = 1.2, lwd = 2, xlim = c(0, max(node_degrees)))

# Adding a best fit line to visualize trends
abline(lm(node_probabilities ~ node_degrees), col = "red")

# Enhance plot readability
grid(nx = NA, ny = NULL, col = "gray", lty = "dotted")
legend("topright",
        legend = c("Visit Probability", "Trend Line"),
        col = c("darkgreen", "red"),
        pch = c(19, NA),
        lty = c(NA, 1),
        cex = 0.8)
```



```
# Print the Pearson correlation coefficient and the slope and intercept of the linear regressi
print(sprintf("Pearson correlation coefficient: %f", cor(degree(g_merged), node_prob)))
#summary(lm(node_prob ~ degree(g_merged)))  # Detailed regression analysis
```

```
[1] "Pearson correlation coefficient: 0.821770"
```

(b)

In all previous questions, we didn't have any teleportation. Now, we use a teleportation probability of α = 0.2 (teleport out of a node with prob=0.2 instead of going to its neighbor). By performing random walks on the network created in 3(a), measure the probability that the walker visits each node. How is this probability related to the degree of the node and α ?

Ans:

Incorporating a teleportation probability into the random walk allows the walker to either move to a neighboring node with a probability of 1−α or teleport randomly to any node with a probability of α. This adjustment introduces greater randomness into the visit frequencies across the network.

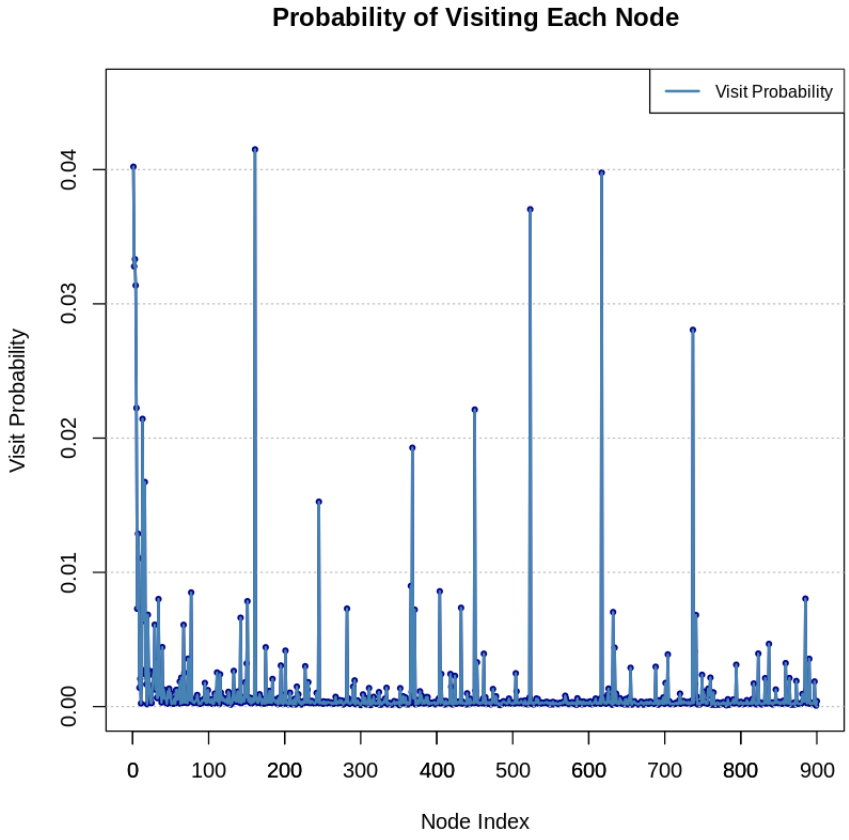
The Pearson correlation coefficient is 0.954783, indicating a strong but slightly reduced correlation between node degree and visit probability compared to non-teleportation scenarios. The linear regression yielded a slope of 0.0001257, showing that higher-degree nodes are still more likely to be visited, though the difference is less pronounced due to teleportation. This suggests that teleportation modestly levels the playing field, increasing the visit chances of lower-degree nodes.

In conclusion, teleportation reduces disparities in visit probabilities across nodes, promoting a more even distribution of node visits in the network, which aligns with the principles underlying algorithms like PageRank.

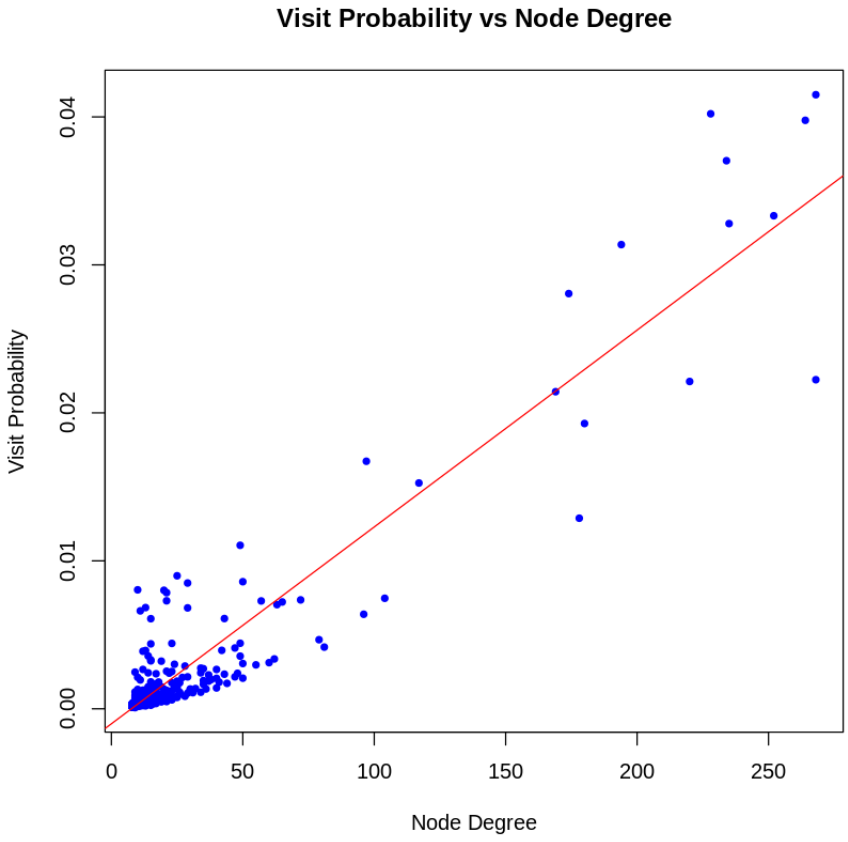
```
if (!requireNamespace("igraph", quietly = TRUE)) {
    install.packages("igraph")
}
library(igraph)

# Function to create a transition matrix for a graph
create_transition_matrix <- function(g) {
    adj_matrix <- as_adjacency_matrix(g, sparse = FALSE)
    # Ensure no dead ends by setting self-loops where necessary
    adj_matrix[diag(rowSums(adj_matrix) == 0)] <- 1
    # Normalize the adjacency matrix to get transition probabilities
    transition_matrix <- adj_matrix / rowSums(adj_matrix)
    return(transition_matrix)
}

# Function to perform a random walk on a graph
random_walk <- function(g, num_steps, start_node, alpha = 0) {
    n <- vcount(g)
    walked_nodes <- numeric(num_steps)
    current_node <- start_node
    transition_matrix <- create_transition_matrix(g)

    for (i in 1:num_steps) {
        if (runif(1) < alpha) {
            # Teleport to a random node
            current_node <- sample(n, 1)
        } else {
            # Continue walking to next node based on transition probabilities
            current_node <- sample(1:n, 1, prob = transition_matrix[current_node, ])
        }
        walked_nodes[i] <- current_node
    }

    return(walked_nodes)
}
```

```
# Constants for simulation
num_nodes <- 900
out_degree <- 4
num_iterations <- 100
num_steps <- 1000
teleportation_prob <- 0.2  # Teleportation probability

# Initialize vector to store visit counts
visited_nodes <- numeric(num_nodes)

# Create and merge two networks
network1 <- barabasi.game(num_nodes, m = out_degree, directed = TRUE)
network2 <- barabasi.game(num_nodes, m = out_degree, directed = TRUE)
shuffled_network2 <- permute(network2, sample(num_nodes))
merged_network <- add_edges(network1, c(t(as_edgelist(shuffled_network2))))

# Ensure the network is connected
if (!is.connected(merged_network)) {
    components_info <- components(merged_network)
    largest_component <- which.max(components_info$csize)
    merged_network <- induced_subgraph(merged_network,
                    which(components_info$membership == largest_component))
}

# Random walk simulations with teleportation
for (i in 1:num_iterations) {
    start_node <- sample(num_nodes, 1)
    walked_nodes <- random_walk(merged_network, num_steps, start_node, teleportation_prob)
    for (node in walked_nodes) {
        visited_nodes[node] <- visited_nodes[node] + 1
    }
}

# Calculate the probability of visiting each node
node_probabilities <- visited_nodes / (num_steps * num_iterations)
```

```
# Plot the probability distribution of visiting each node
plot(seq_len(num_nodes), node_probabilities, type = 'o', col = 'darkblue',
     main = 'Probability of Visiting Each Node', xlab = 'Node Index',
     ylab = 'Visit Probability', pch = 20, cex = 0.6,
     lwd = 2, ylim = c(0, max(node_probabilities) * 1.1))

# Add a grid to improve readability
grid(nx = NA, ny = NULL, col = "gray", lty = "dotted")

# Enhance the line connecting the points
lines(seq_len(num_nodes), node_probabilities, col = 'steelblue', lwd = 2)

axis(side = 1, at = seq(0, num_nodes, by = 100), las = 1)
#axis(side = 2, las = 1)

#Add a legend
legend("topright", legend = "Visit Probability",
        col = "steelblue", lty = 1, lwd = 2, cex = 0.8)
```



```
# Plot visit probability as a function of node degree
plot(degree(merged_network),
    node_probabilities,
    main = "Visit Probability vs Node Degree",
    xlab = "Node Degree", ylab = "Visit Probability", pch = 20, col = 'blue')
abline(lm(node_probabilities ~ degree(merged_network)),
       col = "red")
```



```
# Statistical analysis
node_degrees <- degree(merged_network)
correlation <- cor(node_degrees, node_probabilities)
regression_model <- lm(node_probabilities ~ node_degrees)

# Print Pearson correlation coefficient and linear regression details
cat(sprintf("Pearson correlation coefficient: %.4f\n", correlation))
cat("Linear regression (slope and intercept):\n")
print(coef(regression_model))
```

```
Pearson correlation coefficient: 0.9415
Linear regression (slope and intercept):
  (Intercept)  node_degrees
-0.0010113697  0.0001330246
```

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 flexibility 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 propor- tional 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 N1). Again, let the teleportation probability be equal to α = 0.2. Compare the results with 3(a).

Ans:

This experiment extends the approach used in question 3(b) by modifying how the random walker teleports during the simulation. Unlike in the standard PageRank model where each node has an equal chance of being chosen during a teleport, here the probability of teleporting to a node is weighted by its PageRank. This means nodes with higher PageRanks are more likely to be chosen as teleportation destinations.

Procedure: We compute the PageRank of each node using the page_rank function. The computed PageRanks then influence the teleportation probabilities, making it more likely to teleport to nodes with higher PageRanks.

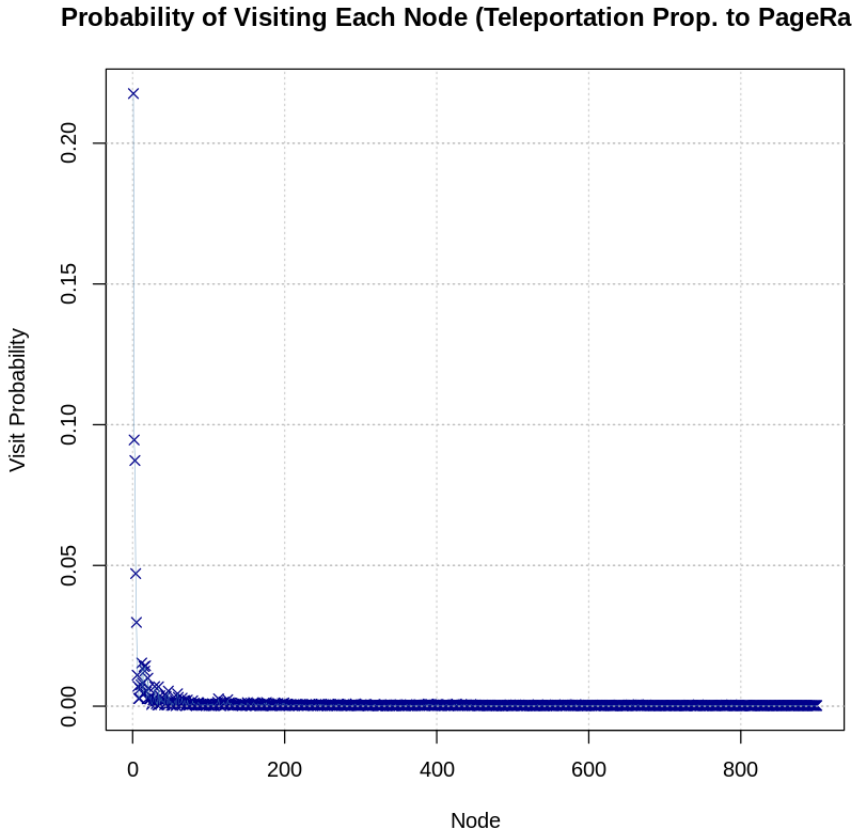
Results and Analysis: We measure how frequently each node is visited during the random walks and analyze this data in relation to the node's degree. The Pearson correlation coefficient for the relationship between node visit probability and node degree is 0.8659, indicating a strong positive correlation. The slope of the linear regression is 0.0004094. When compared with the results from question 3(b), the slope is higher, closely approaching the scenario in 3(a) where teleportation was uniform across all nodes.

Implications: The increased slope and higher visit probabilities for nodes with higher degrees suggest that when teleportations are biased towards higher PageRank nodes, the influence of these nodes on the network's dynamics intensifies. This setup makes the behavior more akin to that seen in question 3(a), where there was no teleportation bias, but due to the weighted teleportation, high PageRank nodes are still favored more than in a uniformly random setup.

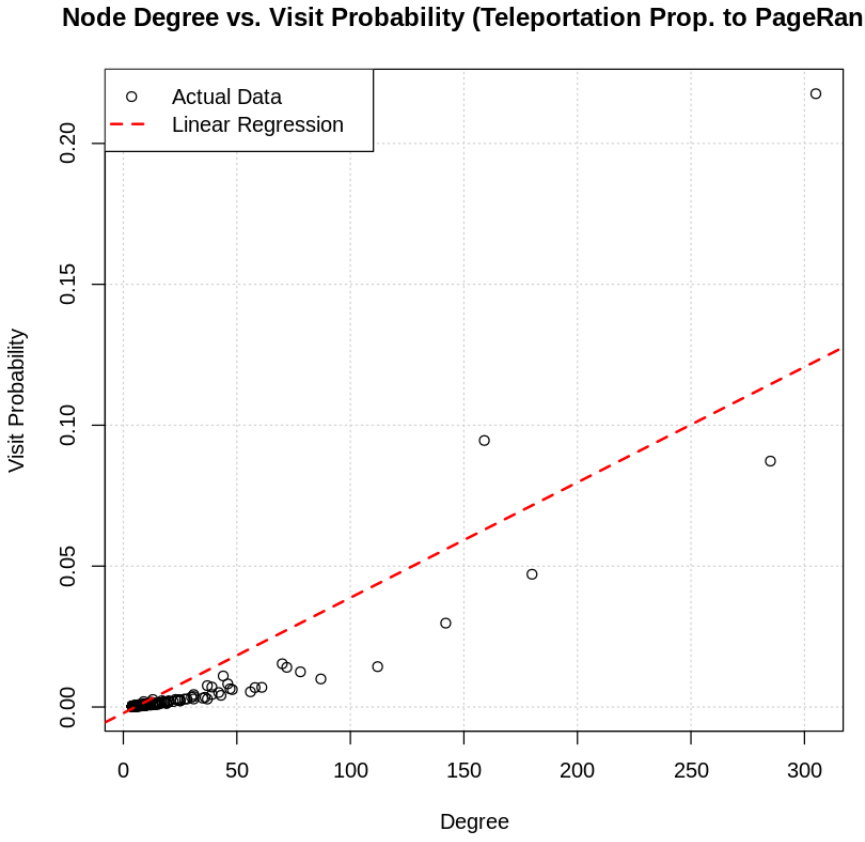
Conclusion: The alteration to teleportation probabilities results in a dynamic where nodes with higher degrees, typically also having higher PageRanks, receive more visits. This effect mimics a natural browsing behavior where users are more likely to visit more prominent or 'trusted' sites, illustrating the impact of personalizing the teleportation component in the PageRank algorithm.

```
# Load necessary packages
if (!requireNamespace("igraph", quietly = TRUE)) {
    install.packages("igraph")
}
library(igraph)

# Initialize variables for the simulation
num_steps <- 1000
num_trials <- 100
alpha_teleportation <- 0.2  # Teleportation probability
node_visit_count <- numeric(vcount(gf))  # Initialize visit count vector
stable_node_position <- ceiling(log(vcount(gf)))  # Calculate burn-in period

# Perform random walks and accumulate visits for each node
for (trial in 1:num_trials) {
  start_node <- sample(1:vcount(gf), 1)
  walked_nodes <- random_walk_custom(
    g = gf,
    num_steps = num_steps,
    start_node = start_node,
    alpha = alpha_teleportation,
    visit_mode = 'page_rank'
  )

  # Update visit counts excluding the burn-in period
  for (step in (stable_node_position + 1):length(walked_nodes)) {
    node_visit_count[walked_nodes[step]] <- node_visit_count[walked_nodes[step]] + 1
  }
}

# Calculate the visit probability for each node
node_probabilities <- node_visit_count / ((num_steps - stable_node_position) * num_trials)
node_indices <- seq_len(vcount(gf))  # Node indices for plotting
```

```
# Plot the visit probability for each node as a scatter plot
plot(node_indices, node_probabilities, pch = 4, col = 'darkblue',
     main = 'Probability of Visiting Each Node (Teleportation Prop. to PageRank)',
     xlab = 'Node', ylab = 'Visit Probability', grid())

# Add a grid to improve readability
grid(nx = NA, ny = NULL, col = "gray", lty = "dotted")

# Enhance the plot with a connecting line
lines(node_indices, node_probabilities, lwd = 0.3, col = 'steelblue')
```



```
# Plot visit probability as a function of node degree
plot(degree(gf), node_probabilities, xlab = 'Degree', ylab = 'Visit Probability',
     main = "Node Degree vs. Visit Probability (Teleportation Prop. to PageRank)", grid())

# Add linear regression line
regression_results <- lm(node_probabilities ~ degree(gf))
abline(regression_results, col = "red", lwd = 2, lty = 2)

# Add a legend to the plot
legend('topleft', legend = c("Actual Data", "Linear Regression"),
       lty = c(NA, 2), lwd = c(1, 2), pch = c(1, NA), col = c('black', 'red'))
```



```
# Output Pearson correlation coefficient and regression details
pearson_correlation <- cor(degree(gf), node_probabilities)
cat(sprintf("Pearson correlation coefficient: %.4f\n", pearson_correlation))
cat("Linear regression details (slope and intercept):\n")
print(summary(regression_results)$coefficients)
```

```
Pearson correlation coefficient: 0.8659
Linear regression details (slope and intercept):
                 Estimate   Std. Error   t value      Pr(>|t|)
(Intercept) -0.0021549889 1.578871e-04 -13.64892  1.110656e-38
degree(gf)   0.0004093997 7.892011e-06  51.87521 2.106245e-272
```

(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?

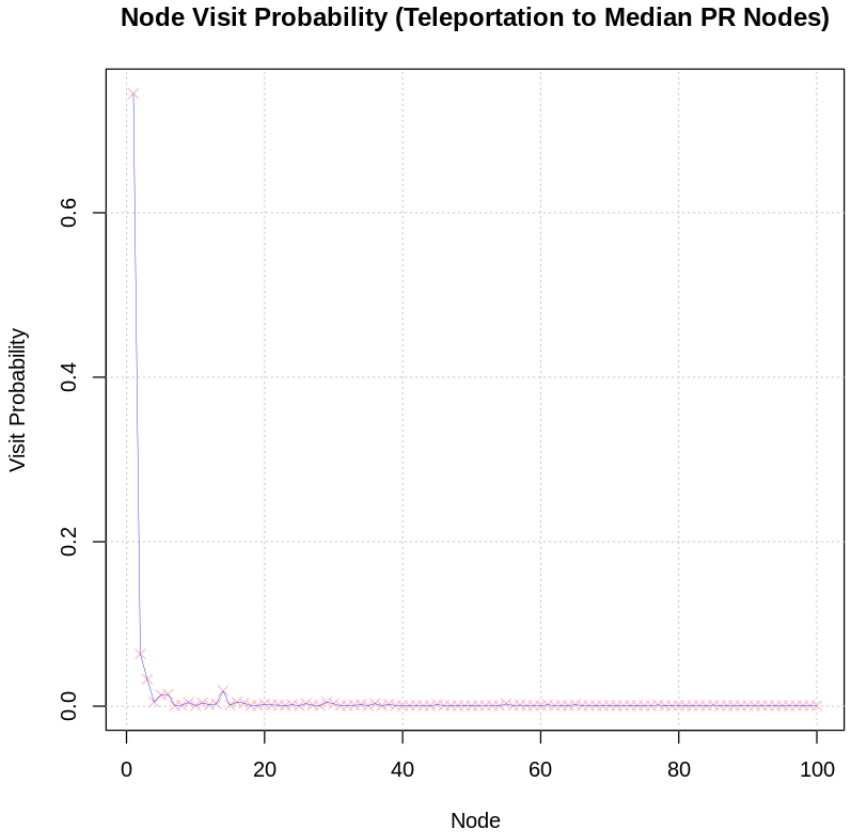
```
# Load necessary libraries
if (!requireNamespace("igraph", quietly = TRUE)) {
    install.packages("igraph")
}
library(igraph)

# Create a graph using the Barabási–Albert model
gf <- barabasi.game(n = 100, m = 2, directed = TRUE)
```

```
# Calculate PageRank of each node
page_ranks <- page_rank(gf)$vector
```

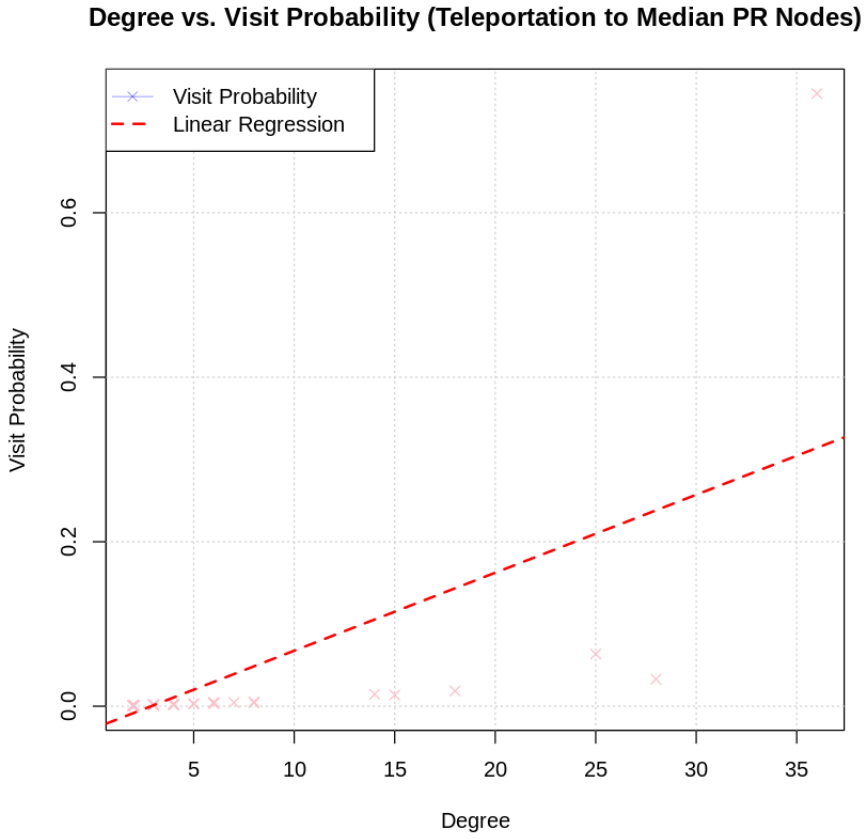
```
# Define the custom random walk function
random_walk_custom <- function(g, num_steps, start_node, alpha, teleportation_vector) {
    n <- vcount(g)
    walked_nodes <- numeric(num_steps)
    current_node <- start_node

    for (i in 1:num_steps) {
        if (runif(1) < alpha) {
            # Teleport to a random node based on the teleportation vector
            current_node <- sample(seq_len(n), 1, prob = teleportation_vector)
        } else {
            # Walk to one of the current node's neighbors
            neighbors_list <- neighbors(g, current_node, mode = "out")
            if (length(neighbors_list) > 0) {
                # Sample from the list of neighbors
                current_node <- sample(neighbors_list, 1)
            }
        }
        walked_nodes[i] <- current_node
    }

    return(walked_nodes)
}

# PageRank calculated
num_steps <- 1000
num_trials <- 100
alpha_teleportation <- 0.2  # Teleportation probability
# Normalized PageRank as teleportation probability
teleportation_vector <- page_rank(gf)$vector / sum(page_rank(gf)$vector)

# Prepare the node visit count array
node_visit_count <- numeric(vcount(gf))
```

```
num_steps <- 1000
num_trials <- 100
alpha_teleportation <- 0.2  # Teleportation probability

# Prepare the node visit count array
node_visit_count <- numeric(vcount(gf))

# Calculate the position of the stable node
stable_node <- ceiling(log(vcount(gf)))

# Perform the random walk simulation
for (trial in 1:num_trials) {
    start_node <- sample(vcount(gf), 1, prob = teleportation_vector)
    walked_nodes <- random_walk_custom(
        g = gf,
        num_steps = num_steps,
        start_node = start_node,
        alpha = alpha_teleportation,
        teleportation_vector = teleportation_vector
    )
    for (node in walked_nodes) {
        # Correctly increment the visit count
        node_visit_count[node] <- node_visit_count[node] + 1
    }
}
```

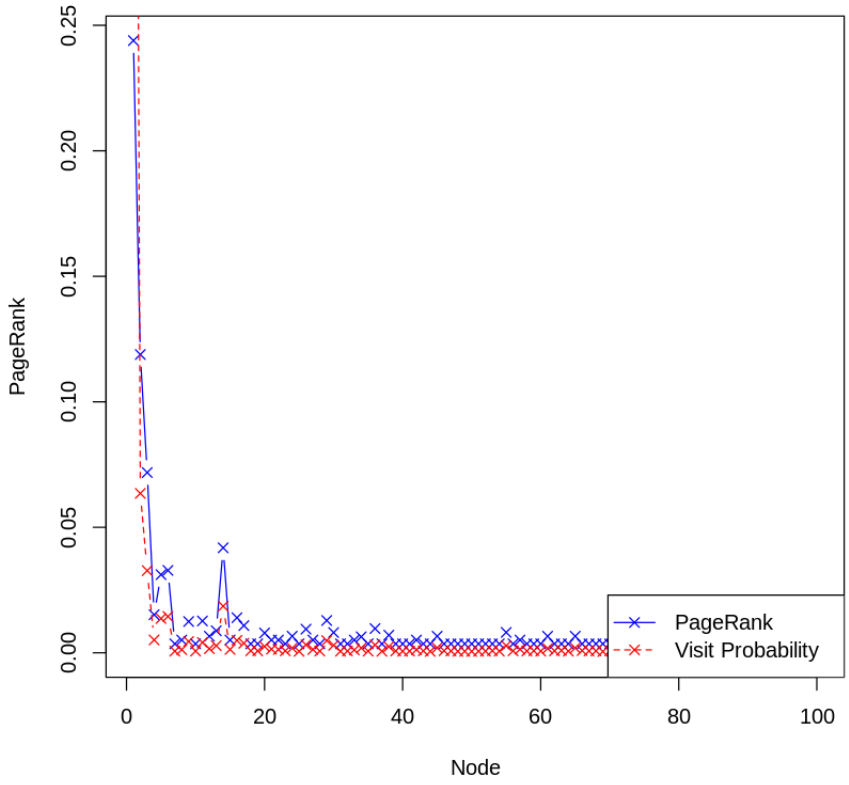
```
# Calculate visit probabilities
node_probabilities <- node_visit_count / ((num_steps - stable_node) * num_trials)
node_indices <- seq_len(vcount(gf))  # Node indices for plotting

# Plot the visit probabilities for each node
plot(node_indices, node_probabilities, pch = 4, col = 'pink',
     main = 'Node Visit Probability (Teleportation to Median PR Nodes)',
     xlab = 'Node', ylab = 'Visit Probability')
grid()
lines(node_indices, node_probabilities, lwd = 0.3, col = 'blue')
```



```
# Plot visit probability as a function of node degree
plot(degree(gf), node_probabilities, pch = 4, col = 'pink',
     main = 'Degree vs. Visit Probability (Teleportation to Median PR Nodes)',
     xlab = 'Degree', ylab = 'Visit Probability')
grid()
abline(lm(node_probabilities ~ degree(gf)), col = "red", lwd = 2, lty = 2)

# Add a legend
legend('topleft', legend = c("Visit Probability", "Linear Regression"),
       lty = c(1, 2), lwd = c(0.3, 2), pch = c(4, NA), col = c('blue', 'red'))
```



```
# Additional plot for visual comparison of PageRank and calculated node probabilities
pageranks <- page_rank(gf)$vector
plot(pageranks, col = "blue", type = 'b', pch = 4,
    ylim = c(0, max(pageranks)), xlab = 'Node', ylab = 'PageRank')
lines(node_probabilities, col = "red", type = 'b',
    lty = 2, pch = 4, ylab = 'Visit Probability')
legend("bottomright", legend = c("PageRank", "Visit Probability"),
    col = c("blue", "red"), lty = c(1, 2), pch = c(4, 4))
```




```
# Output the Pearson correlation coefficient and linear regression details
correlation <- cor(degree(gf), node_probabilities)
cat(sprintf("Pearson correlation coefficient: %.4f\n", correlation))
cat("Regression slope and intercept:\n")
print(summary(lm(node_probabilities ~ degree(gf)))$coefficients)
```

```
Pearson correlation coefficient: 0.6762
Regression slope and intercept:
                Estimate  Std. Error   t value     Pr(>|t|)
(Intercept) -0.027310264 0.006885754 -3.966198 1.390066e-04
degree(gf)   0.009482364 0.001043536  9.086766 1.160496e-14
```

(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?

Ans:

Yes, the PageRank equation can be adjusted to accommodate the self-reinforcement effect seen when users preferentially teleport to a set of trusted web pages, similar to a personalized PageRank approach.

Original PageRank Equation: $PR(u) = (1 - \alpha) * \Sigma[v \in B(u)] (PR(v) / L(v)) + \alpha * (1 / N)$
Where:

- PR(u) is the PageRank of node (u),
- α is the damping factor,
- B(u) contains nodes pointing to (u),
- L(v) is the number of outbound links from node (v),
- N is the total number of nodes.

Modified PageRank for Self-Reinforcement: To tailor PageRank for personalized browsing behaviors, we replace the uniform teleportation probability with a personalized probability vector (P(u)), where (P(u)) is higher for trusted web pages:

$PR(u) = (1 - \alpha) * \Sigma[v \in B(u)] (PR(v) / L(v)) + \alpha * P(u)$

Here, P(u) can be defined based on user preferences or historical data, emphasizing certain web pages over others.

Scenario with Trusted Nodes Set (T): If teleportation occurs only to a set (T) of trusted nodes, and nowhere else, the teleportation probabilities are adjusted as follows: [P(i) =

$$\begin{cases} \frac{1}{|T|} & \text{if } i \in T \\ 0 & \text{otherwise} \end{cases}$$

]

Where (|T|) is the number of nodes in the trusted set. This modification ensures that the PageRank algorithm prioritizes nodes within (T) by increasing their likelihood of receiving teleports, effectively modeling the self-reinforcement behavior seen in real-world web browsing.

Conclusion:

By integrating personalized teleportation probabilities into the PageRank algorithm, we can more accurately model real-world user interactions on the web. This adaptation makes PageRank more relevant for personalized search applications, where user preferences significantly influence the importance of web pages.