# Project 5

Logan Bolton

2025-03-30

*Acknowledgement:* This code was created through the repurposing of code found in the lecture notes and through collaboration with Gemini 2.5 Pro. AI tools were very helpful for me while fixing errors and determining the correct syntax to plot graphs.

```r
library('igraph')
```

```
##
## Attaching package: 'igraph'

## The following objects are masked from 'package:stats':
##
##     decompose, spectrum

## The following object is masked from 'package:base':
##
##     union
```

```r
library(poweRlaw)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:igraph':
##
##     as_data_frame, groups, union

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(igraph)
library(ggplot2) # For plotting
```

## 1) Analysis of Random Graphs G(n, p)

Fix an integer n >= 10,000. For various values of the edge probability p such that pn is a constant, use computational experiments (plots and calculations) to verify the following theoretical properties of G(n, p):

### a) Mean degree c

- Verify that the mean degree c = p(n - 1).

- Plot c as a function of p.

```r
n <- 10000 # Number of vertices (n >= 10,000)

num_p_values <- 30 # Number of different p values to test
p_values <- seq(1/n, 10/n, length.out = num_p_values)

observed_mean_degrees <- numeric(length(p_values))
theoretical_mean_degrees <- numeric(length(p_values))

cat("Running simulation for n =", n, "...\n")
```

```
## Running simulation for n = 10000 ...
```

```r
for (i in 1:length(p_values)) {
  p <- p_values[i]

  # Generate a G(n, p) random graph
  # Use directed=FALSE and loops=FALSE for the standard G(n,p) model
  g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

  # Calculate the observed mean degree
  # Mean degree = Sum of degrees / n = 2 * number_of_edges / n
  if (gorder(g) > 0) { # Check if the graph has vertices
      degrees <- degree(g)
      observed_mean_degrees[i] <- mean(degrees)
  } else {
      observed_mean_degrees[i] <- 0 # Mean degree is 0 for an empty graph
  }

  theoretical_mean_degrees[i] <- p * (n - 1)
}
cat("Simulation finished.\n\n")
```

```
## Simulation finished.
```

```r
# --- Verification ---
cat("--- Verification Summary ---\n")
```

```
## --- Verification Summary ---
```

```r
differences <- observed_mean_degrees - theoretical_mean_degrees
relative_differences <- differences / theoretical_mean_degrees
# Handle cases where theoretical mean degree might be 0 (though not for p>0)
relative_differences[is.nan(relative_differences)] <- 0
relative_differences[is.infinite(relative_differences)] <- NA # Should not happen here

cat("Range of p values:", range(p_values), "\n")
```

```
## Range of p values: 1e-04 0.001
```

```r
cat("Range of theoretical mean degrees (c):", range(theoretical_mean_degrees), "\n")
```

```
## Range of theoretical mean degrees (c): 0.9999 9.999
```

```r
cat("Range of observed mean degrees:", range(observed_mean_degrees), "\n")
```

```
## Range of observed mean degrees: 1.0058 9.9862
```

```r
cat("Mean absolute difference:", mean(abs(differences)), "\n")
```

```
## Mean absolute difference: 0.0299008
```

```r
cat("Max absolute difference:", max(abs(differences)), "\n")
```

```
## Max absolute difference: 0.08385862
```

```r
cat("Mean absolute relative difference (%):", mean(abs(relative_differences), na.rm = TRUE) * 100, "%\n
```

```
## Mean absolute relative difference (%): 0.6620163 %
```

```r
cat("Max absolute relative difference (%):", max(abs(relative_differences), na.rm = TRUE) * 100, "%\n")
```

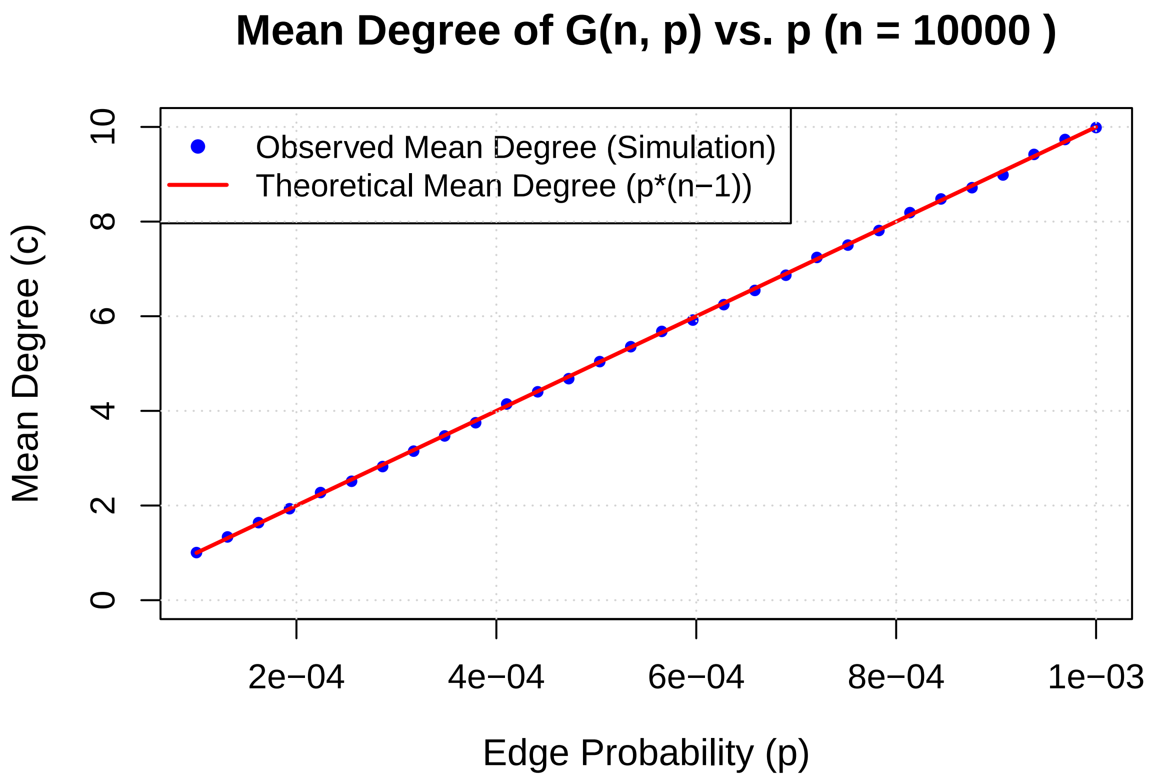
```
## Max absolute relative difference (%): 1.983356 %
```

```r
# Check if observed values are close to theoretical ones (e.g., within 5%)
# Note: Due to randomness, a single run might occasionally exceed a tight tolerance.
tolerance <- 0.05
close_enough <- abs(relative_differences) < tolerance
cat(sprintf("Percentage of simulations where observed c is within %.1f%% of theoretical c: %.2f%%\n",
            tolerance * 100, mean(close_enough, na.rm=TRUE) * 100))
```

```
## Percentage of simulations where observed c is within 5.0% of theoretical c: 100.00%
```

```r
# --- Plotting ---
# Set plot parameters for better readability
par(mar = c(5, 5, 4, 2) + 0.1) # Adjust margins

plot(p_values, observed_mean_degrees,
     type = "p", # Points
     pch = 16,   # Solid circles
     cex = 0.8,  # Smaller points
     col = "blue",
     xlab = "Edge Probability (p)",
     ylab = "Mean Degree (c)",
     main = paste("Mean Degree of G(n, p) vs. p (n =", format(n, scientific = FALSE), ")"),
     ylim = range(c(0, observed_mean_degrees, theoretical_mean_degrees)), # Ensure y-axis starts near 0
     cex.lab = 1.2, # Axis label size
     cex.axis = 1.1, # Axis tick size
     cex.main = 1.3) # Title size

# Add the theoretical line
lines(p_values, theoretical_mean_degrees,
      type = "l", # Line
      col = "red",
      lwd = 2) # Line width

# Add a legend
legend("topleft",
       legend = c("Observed Mean Degree (Simulation)", "Theoretical Mean Degree (p*(n-1))"),
       col = c("blue", "red"),
       pch = c(16, NA), # Point symbol for observed, none for line
       lty = c(NA, 1),  # Line type: none for observed, solid for theoretical
       lwd = c(NA, 2),  # Line width: none for observed, 2 for theoretical
       bg = "white")
```

```r
grid()
```



### b) Degree Distribution pk

Show that pk follows a Poisson distribution pk = e^(-c) × c^k/k! by plotting the empirical degree distribution (histogram) and overlay the theoretical Poisson curve.

```r
n <- 10000
target_c <- 5.0

if (n > 1) {
  p <- target_c / (n - 1)
} else {
  p <- 0 # Avoid division by zero if n=1
}

cat(sprintf("Parameters: n = %d, Target Mean Degree c = %.2f, p = %f\n", n, target_c, p))
```

```
## Parameters: n = 10000, Target Mean Degree c = 5.00, p = 0.000500
```

```r
# --- Generate G(n, p) Graph ---
g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)
cat("Graph generated.\n")
```

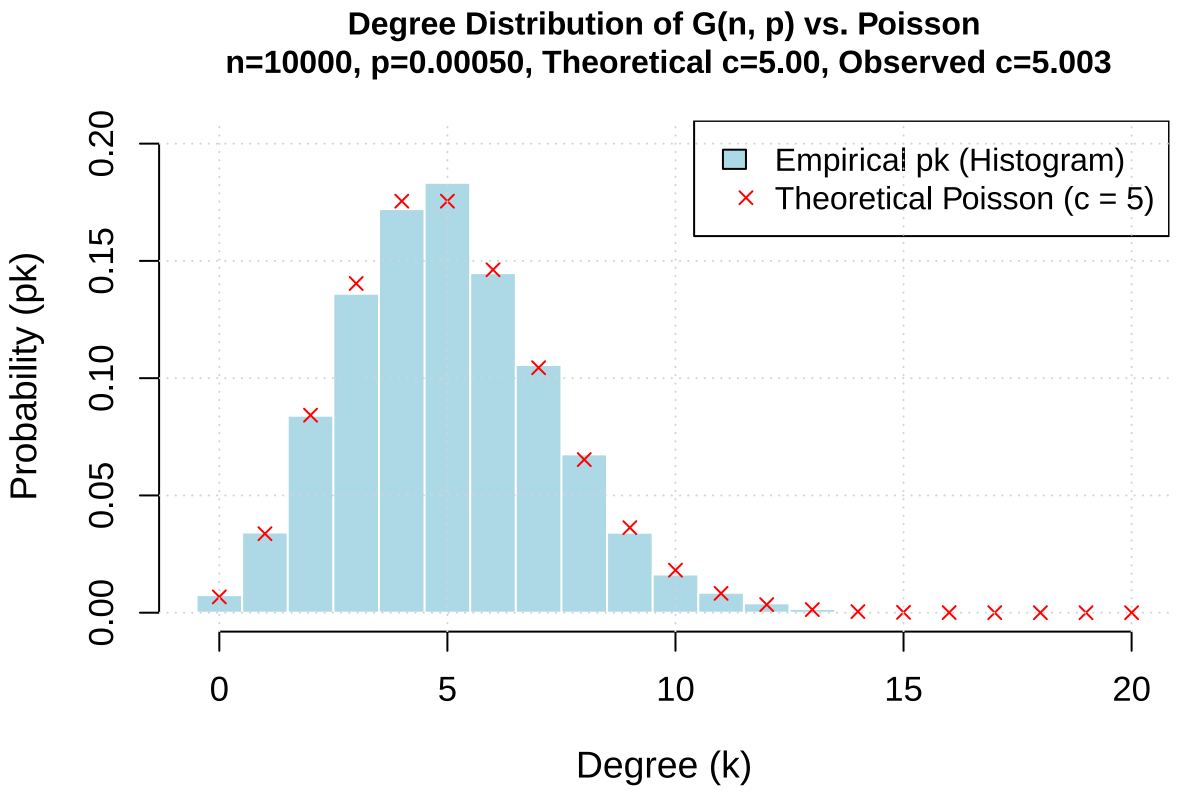
```
## Graph generated.
```

```r
# --- Calculate Empirical Degree Distribution ---
if (gorder(g) > 0) {
    degrees <- degree(g)
    observed_c <- mean(degrees) # Actual mean degree in this instance
    max_degree_observed <- if (length(degrees) > 0) max(degrees) else 0
} else {
    degrees <- numeric(0)
    observed_c <- 0
    max_degree_observed <- 0
    warning("Graph has no vertices.")
}

cat(sprintf("Observed Mean Degree in this sample: %.4f\n", observed_c))
```

```
## Observed Mean Degree in this sample: 5.0028
```

```r
k_range <- 0:(max_degree_observed + 2) # Extend range a bit

# Calculate Poisson probabilities P(X=k) = exp(-c) * c^k / k!
pk_theoretical <- dpois(k_range, lambda = target_c)

if (length(degrees) > 0) {
    # Define breaks to center integers: 0, 1, 2,... become bins [-0.5, 0.5), [0.5, 1.5), [1.5, 2.5) ...
    hist_breaks <- seq(-0.5, max(degrees) + 0.5, by = 1)
    hist_data <- hist(degrees, breaks = hist_breaks, plot = FALSE)
    # Get the maximum density/probability from histogram and theoretical
    ylim_max <- max(c(hist_data$density, pk_theoretical), na.rm = TRUE) * 1.1 # Add 10% margin
    xlim_max <- max(k_range)
} else {
    # Default limits for empty graph case
    hist_data <- list(density = numeric(0), breaks = c(-0.5, 0.5)) # Dummy data
    ylim_max <- max(c(0.1, pk_theoretical), na.rm = TRUE) * 1.1 # Ensure theoretical might still show
    xlim_max <- max(c(10, k_range)) # Sensible default x-limit
}


# Create the base histogram plot
hist(degrees,
     breaks = hist_data$breaks, # Use pre-calculated breaks
     probability = TRUE,       # Y-axis is probability density (equals prob. since width=1)
     col = "lightblue",        # Color for histogram bars
     border = "white",         # Color for bar borders
     xlab = "Degree (k)",
     ylab = "Probability (pk)",
     main = paste("Degree Distribution of G(n, p) vs. Poisson\n",
                  sprintf("n=%d, p=%.5f, Theoretical c=%.2f, Observed c=%.3f",
                          n, p, target_c, observed_c)),
     xlim = c(min(hist_data$breaks), xlim_max), # Use breaks min for xlim start
     ylim = c(0, ylim_max),
     cex.lab = 1.2,
     cex.axis = 1.1,
     cex.main = 1.0) # Smaller main title if long

points(k_range, pk_theoretical,
       pch = 4,    # Crosses symbol
       col = "red",
       cex = 0.9)
```

```r
legend("topright",
       legend = c("Empirical pk (Histogram)", paste0("Theoretical Poisson (c = ", target_c, ")")),
       fill = c("lightblue", NA),  # Color fill for histogram bar in legend
       border = c("black", NA),    # Border for histogram bar in legend
       pch = c(NA, 4),             # Symbol for theoretical points
       col = c(NA, "red"),         # Color for theoretical points
       pt.cex = c(NA, 0.9),        # Size for theoretical points symbol in legend
       bg="white")

grid()
```



**c) Clustering Coefficients**

Verify that both the local and global clustering coefficients of G(n, p) are equal to p.

```r
n <- 20000 # Number of vertices (n >= 10,000)

# Define a range of p values
# For clustering coefficient = p, the p values themselves are the theoretical values
# Use the same range as before for consistency
num_p_values <- 200
p_values <- seq(1/n, 10/n, length.out = num_p_values)
# Alternative: A direct small range like seq(0.0001, 0.001, length.out = 30)

# --- Storage for Results ---
avg_local_cc_observed <- numeric(length(p_values))
global_cc_observed <- numeric(length(p_values))

for (i in 1:length(p_values)) {
  p <- p_values[i]

  # Generate a G(n, p) random graph
  g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

  # Calculate Average Local Clustering Coefficient
  # transitivity(..., type="local") gives NaN for nodes with degree < 2
  local_ccs <- transitivity(g, type = "local")
  # Average over nodes where it's defined (degree >= 2)
  avg_local_cc_observed[i] <- mean(local_ccs, na.rm = TRUE)
  # Handle cases where no node has degree >= 2 (results in NaN mean)
  if (is.nan(avg_local_cc_observed[i])) {
      avg_local_cc_observed[i] <- 0 # Assign 0 if undefined
  }


  # Calculate Global Clustering Coefficient (Transitivity)
  # This calculates 3 * triangles / connected_triples
  global_cc_observed[i] <- transitivity(g, type = "global")
   # Handle cases where global transitivity is NaN (no connected triples)
  if (is.nan(global_cc_observed[i])) {
       global_cc_observed[i] <- 0 # Assign 0 if undefined
  }

}

# --- Verification ---
cat("--- Verification Summary ---\n")
```

```
## --- Verification Summary ---
```

```r
# Compare Average Local CC to p
diff_local <- avg_local_cc_observed - p_values
rel_diff_local <- diff_local / p_values
rel_diff_local[p_values == 0] <- 0 # Handle p=0 case if included
rel_diff_local[is.infinite(rel_diff_local)] <- NA

cat("Average Local Clustering Coefficient vs. p:\n")
```

```
## Average Local Clustering Coefficient vs. p:
```

```r
cat("  Mean absolute difference:", mean(abs(diff_local)), "\n")
```

```
##   Mean absolute difference: 5.787089e-05
```

```r
cat("  Max absolute difference:", max(abs(diff_local)), "\n")
```

```
##   Max absolute difference: 0.0003298324
```

```r
cat("  Mean abs relative difference (%):", mean(abs(rel_diff_local), na.rm = TRUE) * 100, "%\n")
```

```
##   Mean abs relative difference (%): 34.47211 %
```

```r
cat("  Max abs relative difference (%):", max(abs(rel_diff_local), na.rm = TRUE) * 100, "%\n")
```

```
##   Max abs relative difference (%): 449.7632 %
```

```r
# Compare Global CC to p
diff_global <- global_cc_observed - p_values
rel_diff_global <- diff_global / p_values
rel_diff_global[p_values == 0] <- 0 # Handle p=0 case if included
rel_diff_global[is.infinite(rel_diff_global)] <- NA

cat("\nGlobal Clustering Coefficient vs. p:\n")
```

```
##
## Global Clustering Coefficient vs. p:
```

```r
cat("  Mean absolute difference:", mean(abs(diff_global)), "\n")
```

```
##   Mean absolute difference: 4.838871e-05
```

```r
cat("  Max absolute difference:", max(abs(diff_global)), "\n")
```

```
##   Max absolute difference: 0.0002108358
```

```r
cat("  Mean abs relative difference (%):", mean(abs(rel_diff_global), na.rm = TRUE) * 100, "%\n")
```

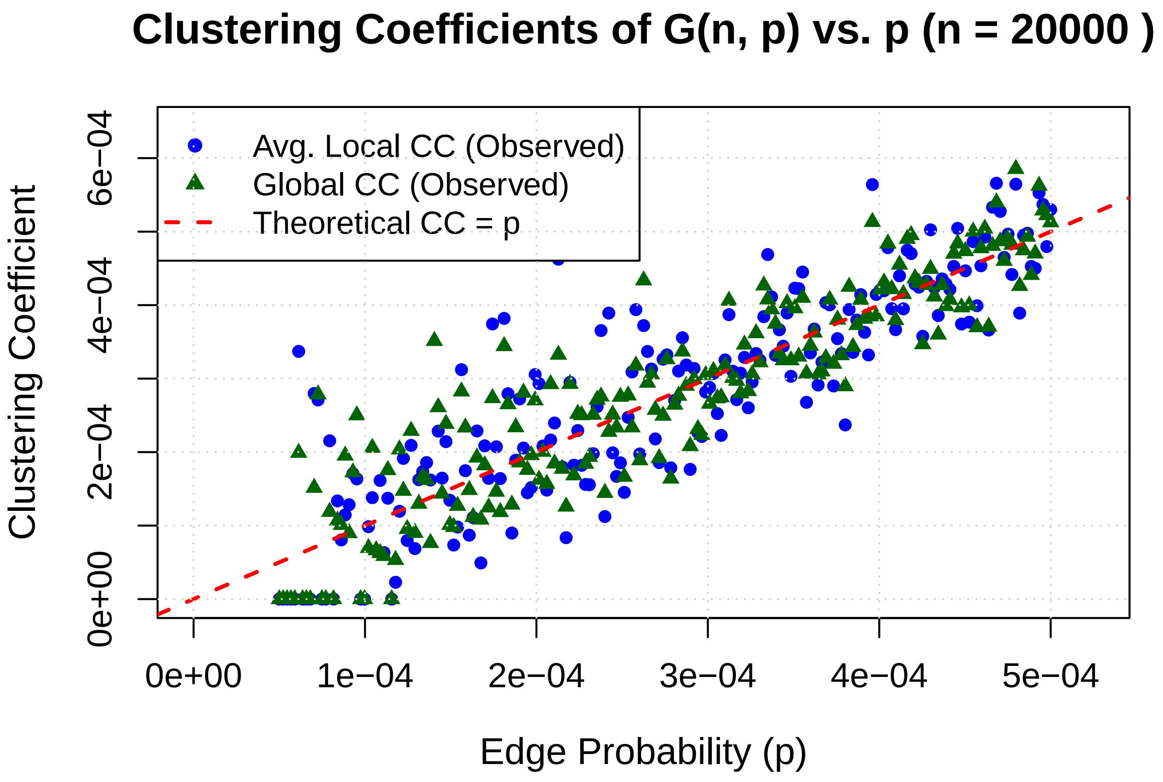
```
##   Mean abs relative difference (%): 29.32024 %
```

```r
cat("  Max abs relative difference (%):", max(abs(rel_diff_global), na.rm = TRUE) * 100, "%\n")
```

```
##   Max abs relative difference (%): 283.6642 %
```

```r
# --- Plotting ---
par(mar = c(5, 5, 4, 2) + 0.1) # Adjust margins

# Determine plot range
y_max <- max(c(0, p_values, avg_local_cc_observed, global_cc_observed), na.rm = TRUE) * 1.1
x_max <- max(p_values) * 1.05

plot(p_values, avg_local_cc_observed,
     pch = 16,    # Solid circles
     cex=0.9,
     col = "blue",
     xlab = "Edge Probability (p)",
     ylab = "Clustering Coefficient",
     main = paste("Clustering Coefficients of G(n, p) vs. p (n =", format(n, scientific = FALSE), ")"),
     xlim = c(0, x_max),
     ylim = c(0, y_max),
     cex.lab = 1.2,
     cex.axis = 1.1,
     cex.main = 1.3)

# Add points for the global clustering coefficient
points(p_values, global_cc_observed,
       pch = 17,    # Triangles
       cex=0.9,
       col = "darkgreen")

# Add the theoretical line CC = p (which is y = x on this plot)
abline(a = 0, b = 1, col = "red", lwd = 2, lty = 2) # y = 0 + 1*x

# Add a legend
```

```
legend("topleft",
       legend = c("Avg. Local CC (Observed)", "Global CC (Observed)", "Theoretical CC = p"),
       col = c("blue", "darkgreen", "red"),
       pch = c(16, 17, NA), # Point symbols
       lty = c(NA, NA, 2),  # Line types (dashed for theoretical)
       lwd = c(NA, NA, 2),  # Line widths
       bg = "white")

# Add grid lines
grid()
```



**d) Giant Component Threshold**

Confirm that the threshold probability for the emergence of a giant component is 1/(n-1).

```
n <- 10000 # Number of vertices (n >= 10,000)
num_simulations_per_p <- 10 # Number of graphs to average over for each p value
num_alpha_points <- 40      # Number of points (alpha values) to plot

# Calculate the theoretical threshold probability
if (n > 1) {
  p_c <- 1 / (n - 1)
} else {
  p_c <- 1 # Or handle n=1 case appropriately
}

# Define a range of multipliers 'alpha' for p, centered around the threshold alpha=1
# We want p = alpha * p_c
alpha_values <- seq(0.1, 3.0, length.out = num_alpha_points)
```

```r
p_values <- alpha_values * p_c

# --- Storage for Results ---
# Store the average relative size of the largest component for each alpha/p
avg_relative_largest_comp_size <- numeric(length(alpha_values))

# --- Computational Experiment ---
cat(sprintf("Running simulations for n=%d. Theoretical threshold p_c ~= %.6f (alpha=1)\n", n, p_c))
```

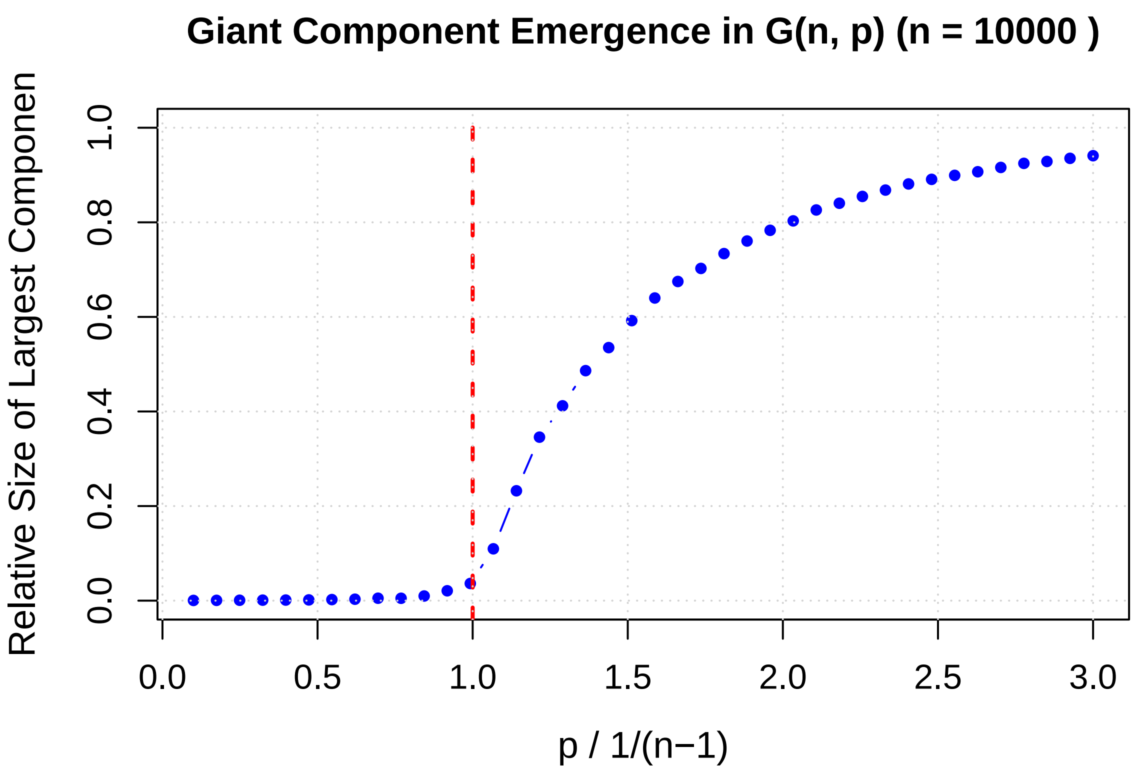
```
## Running simulations for n=10000. Theoretical threshold p_c ~= 0.000100 (alpha=1)
```

```r
for (i in 1:length(alpha_values)) {
  p <- p_values[i]
  alpha <- alpha_values[i]
  current_run_relative_sizes <- numeric(num_simulations_per_p)

  # Run multiple simulations for the current p value
  for (j in 1:num_simulations_per_p) {
    # Generate a G(n, p) random graph
    g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

    largest_comp_size <- 0 # Default size
    if (gorder(g) > 0) { # Check if graph is not empty
        comps <- components(g)
        # Check if components were found and csize is not NULL/empty
        if (!is.null(comps$csize) && length(comps$csize) > 0) {
            largest_comp_size <- max(comps$csize)
        } else {
             # If graph has nodes but components() doesn't return sizes
             # (unlikely for igraph), assume isolated nodes.
             # Or if the graph is truly empty (handled by gorder(g)>0 check).
             # If n>0 but no edges, largest component is size 1.
             if (gorder(g) > 0 && gsize(g) == 0) {
                 largest_comp_size <- 1
             } else {
                 largest_comp_size <- 0 # Should not happen normally
             }
        }
    }

    # Calculate relative size for this run
    current_run_relative_sizes[j] <- largest_comp_size / n

  } # End inner loop (simulations for one p)

  # Calculate the average relative size for this p value
  avg_relative_largest_comp_size[i] <- mean(current_run_relative_sizes, na.rm = TRUE)

} # End outer loop (over alpha/p values)
cat("Simulation finished.\n\n")
```

```
## Simulation finished.
```

```r
# --- Plotting ---
par(mar = c(5, 5, 4, 2) + 0.1) # Adjust margins
```

```
plot(alpha_values, avg_relative_largest_comp_size,
     type = "b", # Plot both points and lines
     pch = 16,
     cex = 0.8,
     col = "blue",
     xlab = expression("p / 1/(n-1)"), # Label using alpha = p/pc
     ylab = "Relative Size of Largest Componen",
     main = paste("Giant Component Emergence in G(n, p) (n =", format(n, scientific = FALSE), ")"),
     ylim = c(0, 1.0), # Relative size is between 0 and 1
     cex.lab = 1.2,
     cex.axis = 1.1,
     cex.main = 1.2)

# Add a vertical line at the theoretical threshold (alpha = 1)
abline(v = 1, col = "red", lwd = 2, lty = 2) # lty=2 for dashed line

grid()
```



**e) Fraction S of Vertices in the Giant Component**

Show that the fraction S of vertices in the giant component satisfies: 1 - S = e^(-cS) by comparing the empirical value of S with the theoretical prediction (using numerical methods if needed).

```
solve_S_equation <- function(c_val) {
  if (c_val <= 1) {
    return(0) # No giant component expected for c <= 1
  }

  # Define the function whose root we want to find: f(S) = 1 - S - exp(-c*S)
  f <- function(S, c_param) {
    1 - S - exp(-c_param * S)
  }

  # Find the root in the interval (epsilon, 1].
  # We use a small epsilon > 0 because S=0 is always a root,
  # and uniroot needs endpoints with different signs.
  # For c>1, f(epsilon) > 0 and f(1) < 0.
  epsilon <- 1e-9
  result <- tryCatch({
    uniroot(f, interval = c(epsilon, 1), c_param = c_val)
  }, error = function(e) {
    warning(paste("Could not find root for c =", c_val, ":", e$message))
    return(list(root = NA)) # Return NA if uniroot fails
  })

  return(result$root)
}

# --- Parameters ---
n <- 10000 # Number of vertices (n >= 10,000)
num_simulations_per_c <- 10 # Average over multiple runs for smoother results
num_c_points <- 30         # Number of different c values to test

# Choose a range of mean degrees 'c' > 1 (where a giant component exists)
# We'll vary c directly, then calculate p
min_c <- 1.1 # Start slightly above the threshold
max_c <- 10.0
c_values <- seq(min_c, max_c, length.out = num_c_points)
p_values <- c_values / (n - 1) # Calculate corresponding p values

# --- Storage for Results ---
observed_S_avg <- numeric(length(c_values))
theoretical_S <- numeric(length(c_values))

# --- Computational Experiment ---
cat(sprintf("Running simulations for n=%d...\n", n))
```

```
## Running simulations for n=10000...
```

```
for (i in 1:length(c_values)) {
  c_current <- c_values[i]
  p <- p_values[i]
  current_run_S_values <- numeric(num_simulations_per_c)

  # Calculate theoretical S *once* for this c
  theoretical_S[i] <- solve_S_equation(c_current)

  # Run multiple simulations for the current p value
  for (j in 1:num_simulations_per_c) {
    # Generate a G(n, p) random graph
    g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

    largest_comp_size <- 0 # Default size
```

```r
    if (gorder(g) > 0) {
        comps <- components(g)
        if (!is.null(comps$csize) && length(comps$csize) > 0) {
            largest_comp_size <- max(comps$csize)
        } else if (gorder(g) > 0 && gsize(g) == 0) {
             largest_comp_size <- 1
        }
    }

    # Calculate observed relative size S for this run
    current_run_S_values[j] <- largest_comp_size / n

  } # End inner loop (simulations for one c)

  # Calculate the average observed S for this c value
  observed_S_avg[i] <- mean(current_run_S_values, na.rm = TRUE)

} # End outer loop (over c values)


# --- Verification ---
cat("--- Verification Summary ---\n")
```

```
## --- Verification Summary ---
```

```r
# Calculate differences (ignore potential NAs from failed root finding)
valid_indices <- !is.na(theoretical_S)
diff_S <- observed_S_avg[valid_indices] - theoretical_S[valid_indices]
rel_diff_S <- diff_S / theoretical_S[valid_indices]
# Handle division by zero if theoretical_S is exactly 0 (shouldn't happen for c>1)
rel_diff_S[theoretical_S[valid_indices] == 0] <- 0
rel_diff_S[is.infinite(rel_diff_S)] <- NA

cat("Comparison of Observed S vs Theoretical S from 1-S=exp(-cS):\n")
```

```
## Comparison of Observed S vs Theoretical S from 1-S=exp(-cS):
```

```r
cat("  Mean absolute difference:", mean(abs(diff_S), na.rm=TRUE), "\n")
```

```
##   Mean absolute difference: 0.0005916424
```

```r
cat("  Max absolute difference:", max(abs(diff_S), na.rm=TRUE), "\n")
```

```
##   Max absolute difference: 0.00422475
```

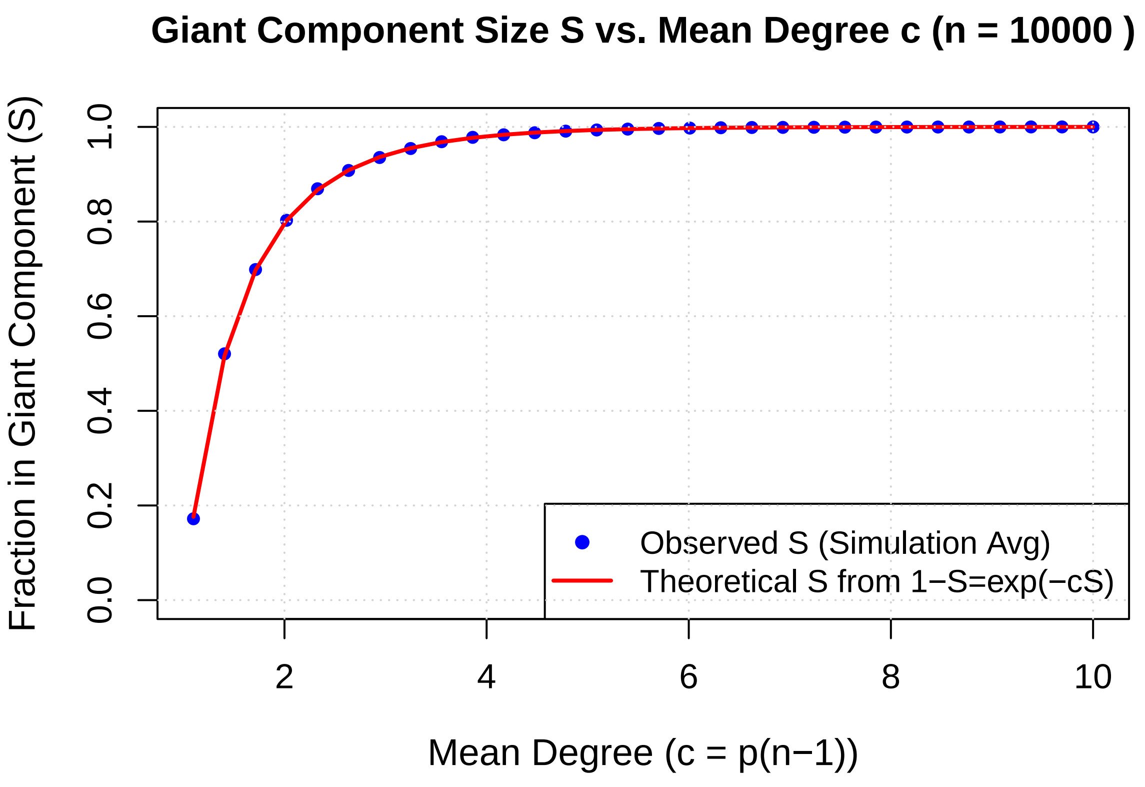
```r
cat("  Mean abs relative difference (%):", mean(abs(rel_diff_S), na.rm = TRUE) * 100, "%\n")
```

```
##   Mean abs relative difference (%): 0.1415903 %
```

```r
cat("  Max abs relative difference (%):", max(abs(rel_diff_S), na.rm = TRUE) * 100, "%\n")
```

```
##   Max abs relative difference (%): 2.39859 %
```

```r
# --- Plotting ---
par(mar = c(5, 5, 4, 2) + 0.1) # Adjust margins

plot(c_values, observed_S_avg,
     type = "p", # Points for observed
     pch = 16,    # Solid circles
     cex = 0.9,
     col = "blue",
     xlab = "Mean Degree (c = p(n-1))",
     ylab = "Fraction in Giant Component (S)",
     main = paste("Giant Component Size S vs. Mean Degree c (n =", format(n, scientific = FALSE), ")"),
     ylim = c(0, 1.0),
     xlim = range(c_values),
     cex.lab = 1.2,
     cex.axis = 1.1,
     cex.main = 1.2)

# Add the theoretical curve
lines(c_values[valid_indices], theoretical_S[valid_indices],
      type = "l", # Line for theoretical
      col = "red",
      lwd = 2)

# Add a legend
legend("bottomright", # Position legend appropriately
       legend = c("Observed S (Simulation Avg)", "Theoretical S from 1-S=exp(-cS)"),
       col = c("blue", "red"),
       pch = c(16, NA), # Point symbol for observed, none for line
       lty = c(NA, 1),  # Line type: none for observed, solid for theoretical
       lwd = c(NA, 2),  # Line width
       bg = "white")

# Add grid lines
grid()
```



### f) Small Components

- Verify that small components are trees.
- Show that the average size of small components is: R = 2/(2 - c + cS)

```r
solve_S_equation <- function(c_val) {
  if (c_val <= 1) { return(0) }
  f <- function(S, c_param) { 1 - S - exp(-c_param * S) }
  epsilon <- 1e-9
  result <- tryCatch({
    uniroot(f, interval = c(epsilon, 1), c_param = c_val, tol = 1e-9)
  }, error = function(e) {
    # warning(paste("Could not find root for c =", c_val, ":", e$message))
    return(list(root = NA))
  })
  # Check if root finding actually succeeded
  if(is.na(result$root) || !is.numeric(result$root)){
      # Fallback or alternative method could be added here if needed
      # For now, just return NA to indicate failure
      return(NA)
  }
  # Ensure root is within valid bounds (e.g., due to tolerance issues)
  root_val <- result$root
  if (root_val < 0 || root_val > 1) {
      # warning(paste("Root out of bounds (0,1] for c =", c_val))
      return(NA) # Consider it invalid
  }
  return(root_val)
}


# --- Parameters ---
n <- 10000
num_simulations_per_c <- 10 # Average results for stability
num_c_points <- 30

# Range of mean degrees 'c' > 1
min_c <- 1.1
max_c <- 6.0 # Reduce max_c slightly, as S approaches 1, denominator in R -> 2-c+c = 2
c_values <- seq(min_c, max_c, length.out = num_c_points)
p_values <- c_values / (n - 1)

# --- Storage for Results ---
observed_avg_R <- numeric(length(c_values))          # Avg size of small components
observed_frac_trees <- numeric(length(c_values))     # Fraction of small comps that are trees
theoretical_R <- numeric(length(c_values))

# --- Computational Experiment ---
cat(sprintf("Running simulations for n=%d...\n", n))
```

```
## Running simulations for n=10000...
```

```r
for (i in 1:length(c_values)) {
  c_current <- c_values[i]
  p <- p_values[i]
```

```r
# Store results for the simulations for this c
current_run_R_values <- numeric(num_simulations_per_c)
current_run_tree_fractions <- numeric(num_simulations_per_c)
valid_runs_for_R <- 0
valid_runs_for_trees <- 0

# --- Calculate Theoretical R ---
theoretical_S_val <- solve_S_equation(c_current)
if (!is.na(theoretical_S_val) && (2 - c_current + c_current * theoretical_S_val) != 0) {
    theoretical_R[i] <- 2 / (2 - c_current + c_current * theoretical_S_val)
} else {
    theoretical_R[i] <- NA # Undefined if S calculation failed or denominator is zero
}

# --- Inner Loop: Multiple simulations per c ---
for (j in 1:num_simulations_per_c) {
  g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

  if (gorder(g) == 0) next # Skip empty graph

  comps <- components(g)
  comp_sizes <- comps$csize
  num_components <- length(comp_sizes)

  if (num_components == 0) next # Skip if no components found

  # Identify Giant Component (GC) and small components
  idx_gc <- which.max(comp_sizes)
  size_gc <- comp_sizes[idx_gc]

  small_comp_indices <- setdiff(1:num_components, idx_gc)
  num_small_components <- length(small_comp_indices)

  # --- Analysis of Small Components ---
  if (num_small_components > 0) {
    small_comp_sizes <- comp_sizes[small_comp_indices]

    # 1. Calculate Average Size R for this run
    observed_R_this_run <- mean(small_comp_sizes)
    current_run_R_values[j] <- observed_R_this_run
    valid_runs_for_R <- valid_runs_for_R + 1

    # 2. Check how many small components are trees
    num_trees_among_small <- 0
    for (k_idx in small_comp_indices) {
      # Get vertices in this small component
      verts_in_comp_k <- which(comps$membership == k_idx)
      size_k <- length(verts_in_comp_k) # Should equal comp_sizes[k_idx]

      if (size_k == 1) {
        # Single node component is trivially a tree (0 edges = 1 - 1)
        num_trees_among_small <- num_trees_among_small + 1
      } else {
          # Create induced subgraph for this component
          sub_g <- induced_subgraph(g, vids = verts_in_comp_k, impl = "auto")
          num_edges_in_comp = gsize(sub_g)
          # Check tree condition: |E| = |V| - 1
          if (num_edges_in_comp == size_k - 1) {
            num_trees_among_small <- num_trees_among_small + 1
          }
        }
      } # End loop over small components

      # Calculate fraction of small components that are trees for this run
      frac_trees_this_run <- num_trees_among_small / num_small_components
      current_run_tree_fractions[j] <- frac_trees_this_run
      valid_runs_for_trees <- valid_runs_for_trees + 1

    } else {
      # No small components (graph might be connected or empty)
      # Assign NA or decide how to handle this. Let's use NA for averaging.
      current_run_R_values[j] <- NA
      current_run_tree_fractions[j] <- NA # Or 1.0 if vacuously true? Let's use NA.
    }

  } # End inner loop (simulations for one c)

  # --- Aggregate results for this c ---
  observed_avg_R[i] <- mean(current_run_R_values, na.rm = TRUE)
  observed_frac_trees[i] <- mean(current_run_tree_fractions, na.rm = TRUE)
   # If all runs resulted in NA (e.g., always connected), the mean will be NaN. Handle this.
   if(is.nan(observed_avg_R[i])) observed_avg_R[i] <- NA
   if(is.nan(observed_frac_trees[i])) observed_frac_trees[i] <- NA


} # End outer loop (over c values)
cat("Simulation finished.\n\n")
```

```
## Simulation finished.
```

```r
cat("--- Verification Summary ---\n")
```

```
## --- Verification Summary ---
```

```r
cat("Fraction of Small Components that are Trees:\n")
```

```
## Fraction of Small Components that are Trees:
```

```r
# Check if any results were obtained
valid_tree_indices <- !is.na(observed_frac_trees)
if(any(valid_tree_indices)) {
    cat(sprintf("  Observed Mean Fraction: %.5f\n", mean(observed_frac_trees, na.rm=TRUE)))
    cat(sprintf("  Observed Min Fraction: %.5f\n", min(observed_frac_trees, na.rm=TRUE)))
    cat(sprintf("  %% of c values where avg fraction was > 0.999: %.1f%%\n",
                100 * mean(observed_frac_trees[valid_tree_indices] > 0.999)))
} else {
    cat("  No valid tree fraction data obtained.\n")
}
```

```
##   Observed Mean Fraction: 0.99999
##   Observed Min Fraction: 0.99989
##   % of c values where avg fraction was > 0.999: 100.0%
```

```
cat("\nAverage Size (R) of Small Components:\n")
```

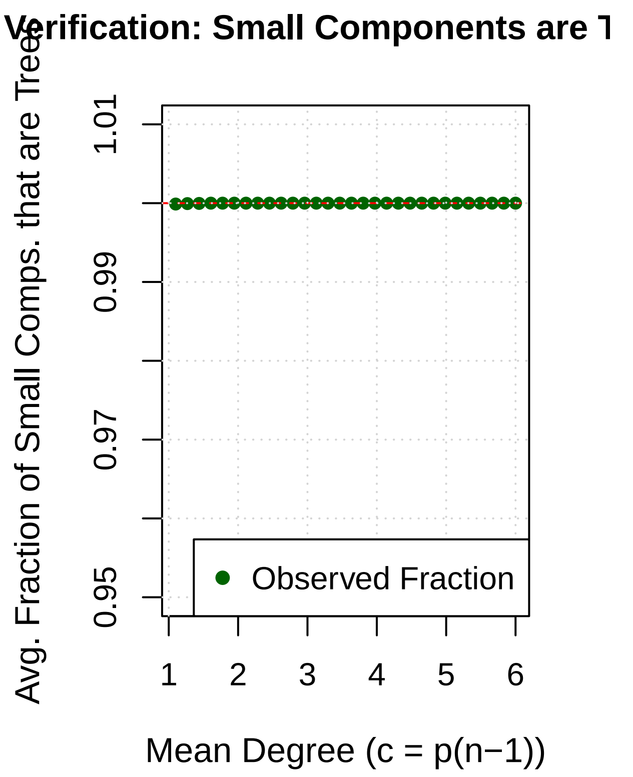
```
##
## Average Size (R) of Small Components:
```

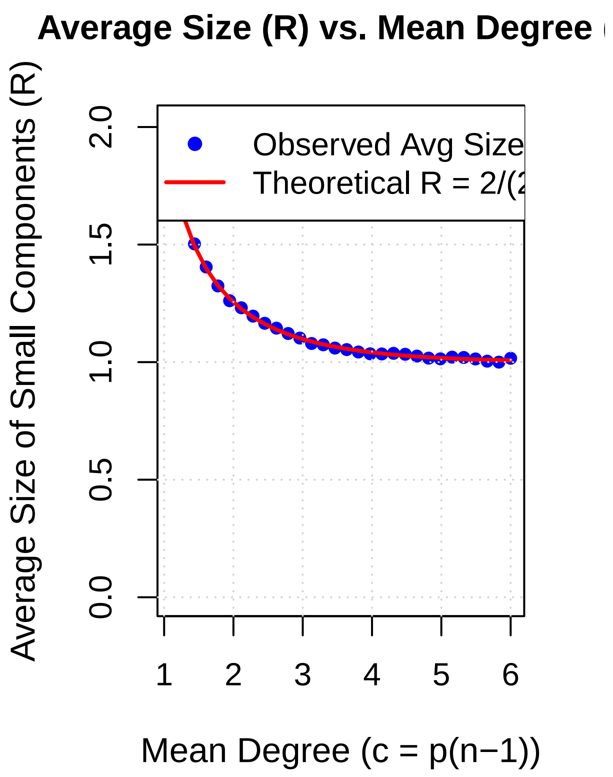
```
# Compare Observed R vs Theoretical R (only where theoretical R is defined)
valid_R_indices <- !is.na(observed_avg_R) & !is.na(theoretical_R)
if (any(valid_R_indices)) {
    diff_R <- observed_avg_R[valid_R_indices] - theoretical_R[valid_R_indices]
    rel_diff_R <- diff_R / theoretical_R[valid_R_indices]
    rel_diff_R[is.infinite(rel_diff_R)] <- NA # Avoid Inf if theoretical R is near 0

    cat("  Comparison vs Theoretical R = 2 / (2 - c + cS):\n")
    cat(sprintf("    Mean absolute difference: %.4f\n", mean(abs(diff_R), na.rm=TRUE)))
    cat(sprintf("    Max absolute difference: %.4f\n", max(abs(diff_R), na.rm=TRUE)))
    cat(sprintf("    Mean abs relative difference (%%): %.2f%%\n", 100 * mean(abs(rel_diff_R), na.rm=TRU
    cat(sprintf("    Max abs relative difference (%%): %.2f%%\n", 100 * max(abs(rel_diff_R), na.rm=TRUE
} else {
     cat("  No valid data for comparing observed and theoretical R.\n")
}
```

```
##   Comparison vs Theoretical R = 2 / (2 - c + cS):
##     Mean absolute difference: 0.0054
##     Max absolute difference: 0.0197
##     Mean abs relative difference (%): 0.46%
##     Max abs relative difference (%): 1.08%
```

```
# --- Plotting ---

# Plot 1: Fraction of Small Components that are Trees
par(mfrow = c(1, 2), mar = c(5, 4.5, 4, 2) + 0.1) # Arrange plots side-by-side

plot(c_values[valid_tree_indices], observed_frac_trees[valid_tree_indices],
     type = "b", pch = 16, col = "darkgreen", cex = 0.9,
     xlab = "Mean Degree (c = p(n-1))",
     ylab = "Avg. Fraction of Small Comps. that are Trees",
     main = "Verification: Small Components are Trees",
     ylim = c(min(0.95, min(observed_frac_trees, na.rm=T)), 1.01), # Zoom in near 1.0
     cex.lab = 1.1, cex.axis = 1.0, cex.main = 1.1)
abline(h = 1, col = "red", lty = 2) # Line at y=1.0
grid()
legend("bottomright", legend="Observed Fraction", col="darkgreen", pch=16, bg="white")


# Plot 2: Average Size of Small Components (R)
plot(c_values[valid_R_indices], observed_avg_R[valid_R_indices],
     type = "p", pch = 16, col = "blue", cex = 0.9,
     xlab = "Mean Degree (c = p(n-1))",
     ylab = "Average Size of Small Components (R)",
     main = "Average Size (R) vs. Mean Degree (c)",
     ylim = c(0, max(observed_avg_R[valid_R_indices], theoretical_R[valid_R_indices], na.rm = TRUE) * 1
     cex.lab = 1.1, cex.axis = 1.0, cex.main = 1.1)
```

```
# Add theoretical curve R = 2 / (2 - c + cS)
lines(c_values[valid_R_indices], theoretical_R[valid_R_indices],
      col = "red", lwd = 2)

grid()
legend("topleft",
       legend = c("Observed Avg Size (R)", "Theoretical R = 2/(2-c+cS)"),
       col = c("blue", "red"),
       pch = c(16, NA), lty = c(NA, 1), lwd = c(NA, 2), bg = "white")
```





**g) Fraction of Vertices in Small Components**

Verify that the fraction of vertices in small components follows (e^(-sc)*(sc)^(s-1))/s!

```
solve_S_equation <- function(c_val) {
  if (c_val <= 1) {
    return(0)
  }
  f <- function(S, c_param) { 1 - S - exp(-c_param * S) }
  epsilon <- 1e-9
  result <- tryCatch({
    uniroot(f, interval = c(epsilon, 1), c_param = c_val, tol = 1e-9)
  }, error = function(e) {
    warning(paste("Could not find root for S, c =", c_val, ":", e$message))
    return(list(root = NA))
  })
  if(is.na(result$root) || !is.numeric(result$root)){ return(NA) }
  root_val <- result$root
  if (root_val < 0 || root_val > 1) { return(NA) }
```

```r
  return(root_val)
}

# --- Parameters ---
n <- 10000
c_fixed <- 1.5 # Mean degree > 1
p <- c_fixed / (n - 1)
num_simulations <- 100 # Increase for smoother empirical curve

cat(sprintf("Parameters: n=%d, c=%.2f, p=%.6f\n", n, c_fixed, p))
```

```
## Parameters: n=10000, c=1.50, p=0.000150
```

```r
cat(sprintf("Running %d simulations...\n", num_simulations))
```

```
## Running 100 simulations...
```

```r
# --- Storage for all small component sizes ---
all_small_comp_sizes <- list()
total_small_components_collected <- 0

# --- Simulation Loop ---
for (sim in 1:num_simulations) {
  g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

  if (gorder(g) == 0) next

  comps <- components(g)
  comp_sizes <- comps$csize
  num_components <- length(comp_sizes)

  if (num_components <= 1) next # Need at least one small component

  # Identify Giant Component (GC)
  idx_gc <- which.max(comp_sizes)

  # Get sizes of small components
  small_comp_indices <- setdiff(1:num_components, idx_gc)
  if (length(small_comp_indices) > 0) {
      current_small_sizes <- comp_sizes[small_comp_indices]
      all_small_comp_sizes[[length(all_small_comp_sizes) + 1]] <- current_small_sizes
      total_small_components_collected <- total_small_components_collected + length(current_small_sizes
  }

  if (sim %% 20 == 0 || sim == num_simulations) cat(sprintf("  Simulation %d completed.\n", sim))

} # End simulation loop
```

```
##   Simulation 20 completed.
##   Simulation 40 completed.
##   Simulation 60 completed.
##   Simulation 80 completed.
##   Simulation 100 completed.
```

```r
cat(sprintf("Finished simulations. Collected %d small components.\n", total_small_components_collected)
```

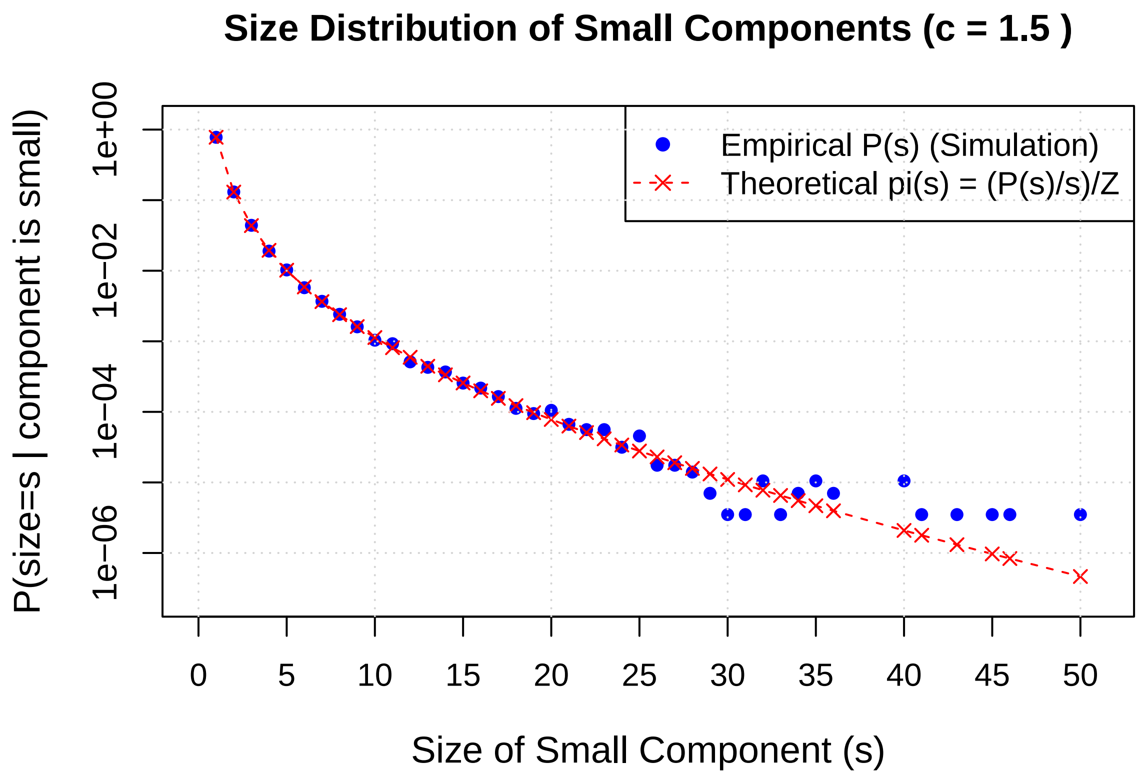
```
## Finished simulations. Collected 285616 small components.
```

```r
# --- Process Collected Data ---
if (total_small_components_collected > 0) {
  # Combine all sizes into one vector
  all_sizes_vector <- unlist(all_small_comp_sizes)

  # Calculate empirical frequency distribution
  # P_emp(s) = (Number of small components of size s) / (Total number of small components)
  size_counts <- table(all_sizes_vector)
  s_values_observed <- as.numeric(names(size_counts))
  pk_empirical <- as.numeric(size_counts) / total_small_components_collected

  # Limit max s for plotting if needed (distribution decays quickly)
  max_s_plot <- min(max(s_values_observed), 50) # Adjust as needed
  # Ensure we only work with s values >= 1 for theoretical part
  s_values_plot_indices <- which(s_values_observed >= 1 & s_values_observed <= max_s_plot)
  s_values_plot <- s_values_observed[s_values_plot_indices]
  pk_empirical_plot <- pk_empirical[s_values_plot_indices]


  # --- Calculate Theoretical Distribution P(s) (Unnormalized mu(s)) ---
  # P(s) = Prob(random vertex is in finite component of size s)
  # Using P(s) = [ (c*s)^(s-1) / s! ] * exp(-c*s)
  pk_theoretical_unnormalized_mu <- numeric(length(s_values_plot))

  for (idx in 1:length(s_values_plot)) {
      s <- s_values_plot[idx]
      # s=1 case
      if (s == 1) {
          log_pk <- -c_fixed
      } else { # s > 1 case
          # Use logs for numerical stability: log(P) = (s-1)*log(c*s) - lgamma(s+1) - c*s
          cs_term = c_fixed * s
          # Check for potential numerical issues or invalid inputs
          if (cs_term <= 0 || !is.finite(cs_term)) {
              log_pk <- -Inf
          } else {
              term1 <- (s - 1) * log(cs_term)
              term2 <- lgamma(s + 1) # log(s!)
              term3 <- c_fixed * s
              # Ensure terms are finite before subtraction
              if (is.finite(term1) && is.finite(term2) && is.finite(term3)) {
                   log_pk <- term1 - term2 - term3
              } else {
                  log_pk <- -Inf # Assign -Inf if any term calculation failed
              }
          }
      }
      # Check for NaN/Inf before exponentiating
      if (is.finite(log_pk)) {
            pk_theoretical_unnormalized_mu[idx] <- exp(log_pk)
```

```
    } else {
          pk_theoretical_unnormalized_mu[idx] <- 0 # Assign 0 if calculation failed
    }
}

# --- Calculate Terms for pi(s) Distribution ---
# pi(s) is proportional to P(s) / s
# This represents the expected number of components of size s (up to a factor n)
terms_pi_s <- pk_theoretical_unnormalized_mu / s_values_plot
# Handle potential division by zero for s=0 (should not be in s_values_plot now)
# Handle potential NaN if pk_theoretical_unnormalized_mu was 0
terms_pi_s[!is.finite(terms_pi_s)] <- 0

# --- Calculate Normalization Constant Z for pi(s) ---
# Z = Sum_{k=1}^\infty (P(k) / k) approximated over the plotted range
# This represents the expected total number of small components (up to a factor n)
normalization_constant_Z <- sum(terms_pi_s, na.rm = TRUE)
cat(sprintf("Normalization Constant Z = Sum(P(k)/k) approx = %.5f\n", normalization_constant_Z))

# --- FINAL Correctly Normalized Theoretical Distribution pi(s) ---
# pi(s) = P(size=s | component is small) = (P(s) / s) / Z
 if (normalization_constant_Z > 1e-12) { # Avoid division by zero
      pk_theoretical_normalized_pi <- terms_pi_s / normalization_constant_Z
 } else {
     pk_theoretical_normalized_pi <- terms_pi_s * 0 # Set to zero if Z is effectively zero
     warning("Normalization constant Z = Sum(P(k)/k) is close to zero.")
 }

# --- Plotting ---
par(mar = c(5, 5, 4, 2) + 0.1) # Adjust margins

# Determine plot range carefully for log scale
# Filter out zero probabilities before taking min for log scale
y_min_emp <- min(pk_empirical_plot[pk_empirical_plot > 0], na.rm = TRUE)
y_min_the <- min(pk_theoretical_normalized_pi[pk_theoretical_normalized_pi > 0], na.rm = TRUE)
y_min_plot <- min(y_min_emp, y_min_the, na.rm = TRUE) * 0.5 # Add buffer below min
if (!is.finite(y_min_plot) || y_min_plot <= 0) y_min_plot <- 1e-8 # Fallback if all are zero or NA

y_max_plot <- max(c(pk_empirical_plot, pk_theoretical_normalized_pi), na.rm = TRUE) * 1.5 # Buffer ab

plot(s_values_plot, pk_empirical_plot,
     type = "p", # Points
     pch = 16,   # Solid circles
     col = "blue",
     cex = 0.9,
     log = "y",  # Log scale for y-axis
     xlab = "Size of Small Component (s)",
     ylab = "P(size=s | component is small)", # Correct label interpretation
     main = paste("Size Distribution of Small Components (c =", c_fixed, ")"),
     ylim = c(y_min_plot, y_max_plot),
     xlim = c(0, max_s_plot + 1),
     cex.lab = 1.2, cex.axis = 1.1, cex.main = 1.2,
     xaxt = "n") # Suppress default x-axis to draw custom ticks
```

```
  axis(1, at = seq(0, max_s_plot, by = 5)) # Custom x-axis ticks

  # Overlay the NEW theoretical probabilities pi(s)
  points(s_values_plot, pk_theoretical_normalized_pi,
         type = "p", # Points
         pch = 4,    # Crosses
         col = "red",
         cex = 0.9)
  lines(s_values_plot, pk_theoretical_normalized_pi, # Add lines for theoretical
         type = "l",
         col = "red",
         lty = 2) # Dashed line


  # Add a legend
  legend("topright",
         legend = c("Empirical P(s) (Simulation)", "Theoretical pi(s) = (P(s)/s)/Z"), # Updated legend
         col = c("blue", "red"),
         pch = c(16, 4), # Point symbols
         lty = c(NA, 2), # Line type for theoretical
         bg = "white")

  # Add grid lines (corrected)
  grid()

} else {
  cat("No small components were collected during the simulations.\n")
  cat("Try increasing num_simulations or using a 'c' value closer to 1.\n")
}
```

```
## Normalization Constant Z = Sum(P(k)/k) approx = 0.28665
```



**h) Diameter**

Show that the diameter of G(n, p) follows: diameter = A + ln(n)/ln(c) where A is a constant.

```r
target_c <- 5.0

n_values <- c(10000, 15000, 20000, 25000, 30000) # Linear scale

# Number of replicates for each n to average out randomness
num_replicates <- 5 # Increase if results are too noisy and time permits

cat(sprintf("Target Average Degree (c): %.2f\n", target_c))
```

```
## Target Average Degree (c): 5.00
```

```r
cat("N values to test:", paste(n_values, collapse=", "), "\n")
```

```
## N values to test: 10000, 15000, 20000, 25000, 30000
```

```r
cat("Replicates per n:", num_replicates, "\n")
```

```
## Replicates per n: 5
```

```r
# --- Storage for Results ---
results <- data.frame(
  n = integer(),
  log_n = numeric(),
  p = numeric(),
  avg_diameter = numeric(),
  sd_diameter = numeric() # Standard deviation of diameter across replicates
)
```

```r
# --- Computational Experiment ---
cat("Running simulations...\n")
```

```
## Running simulations...
```

```r
start_time <- Sys.time()

for (n in n_values) {
  if (n <= 1) next # Skip n=1

  # Calculate p for this n to keep c constant
  p <- target_c / (n - 1)

  # Ensure p is valid
  if (p < 0 || p > 1) {
    cat(sprintf("Skipping n=%d, invalid p=%.5f\n", n, p))
    next
  }

  cat(sprintf("Processing n = %d (p = %.6f)...\n", n, p))
  diameters_for_n <- numeric(num_replicates)

  for (i in 1:num_replicates) {
    g <- sample_gnp(n = n, p = p, directed = FALSE, loops = FALSE)

    # Check connectivity and find largest component
    comp_info <- components(g, mode = "weak") # "strong" or "weak" is same for undirected

    if (comp_info$no == 0) {
        # Empty graph case (shouldn't happen for n>0)
        diameters_for_n[i] <- NA # Or 0? Or handle as error?
        warning(paste("Empty graph generated for n=", n))
    } else if (comp_info$no == 1) {
      # Graph is connected
      diameters_for_n[i] <- diameter(g, unconnected = FALSE) # Use FALSE since known connected
    } else {
      # Graph is disconnected, find diameter of the largest component
      largest_comp_id <- which.max(comp_info$csize)
      nodes_in_largest <- which(comp_info$membership == largest_comp_id)

      # Create subgraph of the largest component
      sg <- induced_subgraph(g, vids = nodes_in_largest)

      if (gorder(sg) > 1) {
          diameters_for_n[i] <- diameter(sg, unconnected = FALSE)
      } else {
          # Largest component has only 1 node (or 0 somehow)
          diameters_for_n[i] <- 0
      }
    }
    # Optional: progress within replicates
    # cat(sprintf("  Rep %d/%d, diameter=%d\n", i, num_replicates, diameters_for_n[i]))
  }
  # Store average and standard deviation
  valid_diameters <- diameters_for_n[!is.na(diameters_for_n)]
  if (length(valid_diameters) > 0) {
      results <- rbind(results, data.frame(
        n = n,
        log_n = log(n), # Natural logarithm
        p = p,
        avg_diameter = mean(valid_diameters),
        sd_diameter = sd(valid_diameters)
      ))
  } else {
      cat(sprintf("Warning: No valid diameters calculated for n=%d\n", n))
  }

}
```

```
## Processing n = 10000 (p = 0.000500)...
## Processing n = 15000 (p = 0.000333)...
## Processing n = 20000 (p = 0.000250)...
## Processing n = 25000 (p = 0.000200)...
## Processing n = 30000 (p = 0.000167)...
```

```
end_time <- Sys.time()
cat("Simulations finished. Time taken:", format(end_time - start_time), "\n\n")
```

```
## Simulations finished. Time taken: 5.440332 mins
```

```
# --- Analysis ---
print("--- Simulation Results ---")
```

```
## [1] "--- Simulation Results ---"
```

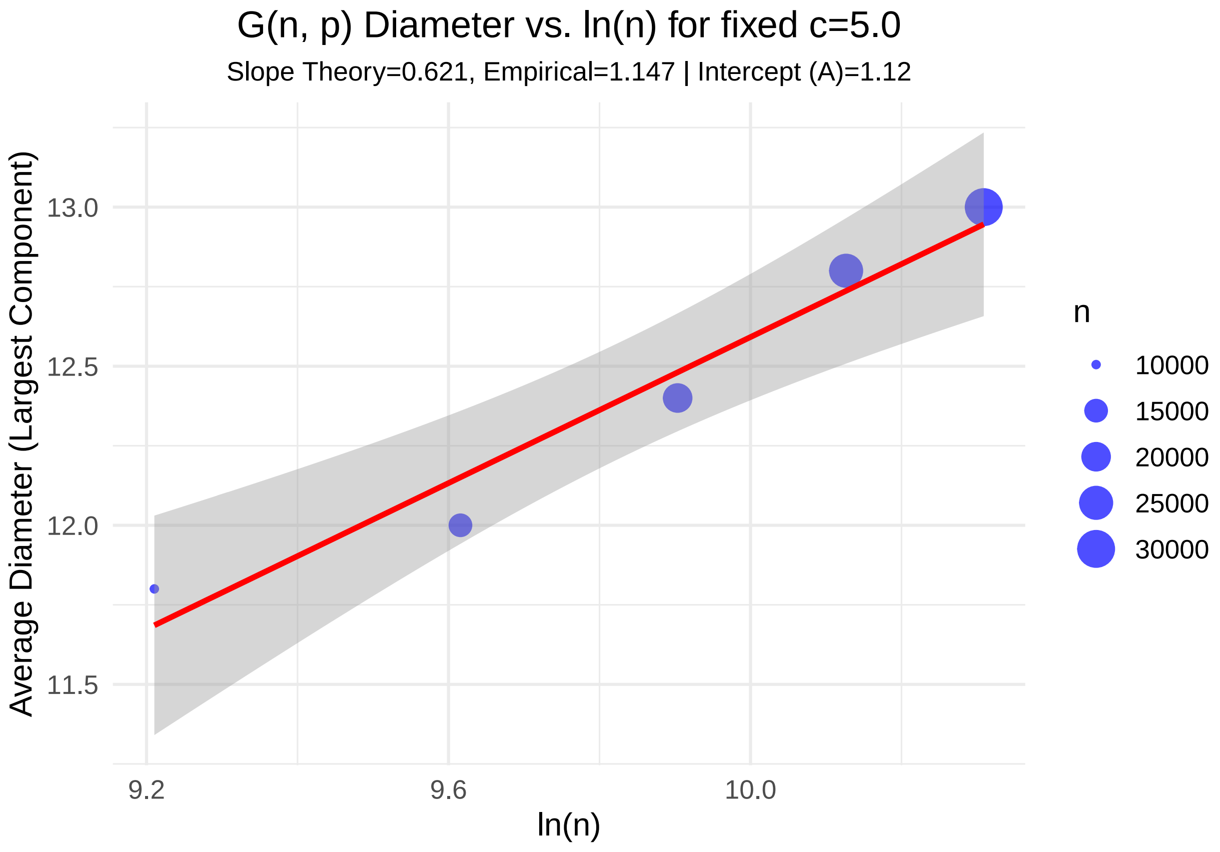
```
print(results)
```

```
##       n     log_n            p avg_diameter sd_diameter
## 1 10000  9.210340 0.0005000500         11.8   0.8366600
## 2 15000  9.615805 0.0003333556         12.0   0.0000000
## 3 20000  9.903488 0.0002500125         12.4   0.5477226
## 4 25000 10.126631 0.0002000080         12.8   0.8366600
## 5 30000 10.308953 0.0001666722         13.0   0.0000000
```

```
if (nrow(results) < 2) {
  cat("Not enough data points for regression analysis.\n")
} else {
  # Calculate theoretical slope
  theoretical_slope <- 1 / log(target_c)
  cat(sprintf("\nTheoretical slope (1 / ln(c)) = 1 / ln(%.2f) = %.4f\n",
              target_c, theoretical_slope))

  # Perform linear regression: avg_diameter ~ log(n)
  model <- lm(avg_diameter ~ log_n, data = results)

  cat("\n--- Linear Regression: avg_diameter ~ log(n) ---\n")
  print(summary(model))

  # Extract coefficients
  intercept_A <- coef(model)[1]
```

```r
  empirical_slope_B <- coef(model)[2]

  cat(sprintf("\nEstimated Intercept (A): %.4f\n", intercept_A))
  cat(sprintf("Estimated Slope (B): %.4f\n", empirical_slope_B))

  # Compare empirical slope with theoretical slope
  cat(sprintf("\nComparison:\n"))
  cat(sprintf("  Theoretical Slope = %.4f\n", theoretical_slope))
  cat(sprintf("  Empirical Slope   = %.4f\n", empirical_slope_B))
  cat(sprintf("  Relative Difference = %.2f %%\n",
              100 * abs(empirical_slope_B - theoretical_slope) / theoretical_slope))

  # --- Plotting ---
  plot_title <- sprintf("G(n, p) Diameter vs. ln(n) for fixed c=%.1f", target_c)
  plot_subtitle <- sprintf("Slope Theory=%.3f, Empirical=%.3f | Intercept (A)=%.2f",
                           theoretical_slope, empirical_slope_B, intercept_A)

  gg <- ggplot(results, aes(x = log_n, y = avg_diameter)) +
    geom_point(aes(size = n), color = "blue", alpha = 0.7) + # Size points by n
    geom_smooth(method = "lm", se = TRUE, color = "red", formula = y ~ x) + # Add regression line + CI
    # Optional: Add error bars if sd_diameter is meaningful
    # geom_errorbar(aes(ymin = avg_diameter - sd_diameter, ymax = avg_diameter + sd_diameter),
    #               width = 0.05, alpha = 0.5) +
    labs(
      title = plot_title,
      subtitle = plot_subtitle,
      x = "ln(n)",
      y = "Average Diameter (Largest Component)",
      size = "n" # Legend title for size
    ) +
    theme_minimal(base_size = 12) +
    theme(plot.title = element_text(hjust = 0.5),
          plot.subtitle = element_text(hjust = 0.5, size=10))

  print(gg)
}
```

```
##
## Theoretical slope (1 / ln(c)) = 1 / ln(5.00) = 0.6213
##
## --- Linear Regression: avg_diameter ~ log(n) ---
##
## Call:
## lm(formula = avg_diameter ~ log_n, data = results)
##
## Residuals:
##        1        2        3        4        5
##  0.11446 -0.15075 -0.08082  0.06315  0.05396
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.1180     1.4536   0.769  0.49782
## log_n         1.1474     0.1477   7.768  0.00444 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1281 on 3 degrees of freedom
## Multiple R-squared:  0.9526, Adjusted R-squared:  0.9368
## F-statistic: 60.34 on 1 and 3 DF,  p-value: 0.004439
##
##
## Estimated Intercept (A): 1.1180
## Estimated Slope (B): 1.1474
##
## Comparison:
##   Theoretical Slope = 0.6213
##   Empirical Slope   = 1.1474
##   Relative Difference = 84.66 %
```



## 2) Empirical Analysis of Real-World Network

Analyze the ca-GrQc dataset from https://snap.stanford.edu/data/ca-GrQc.html in the Stanford Large Network Dataset Collection.

```r
txt_file <- "/Users/log/Github/Spring2025Classes/social_networks/project6/ca-GrQc.txt"

edges <- read.table(txt_file, skip = 4, header = FALSE)
colnames(edges) <- c("FromNodeId", "ToNodeId")
g <- graph_from_data_frame(edges, directed = FALSE)
g <- simplify(g, remove.multiple = TRUE, remove.loops = TRUE)
```

```r
print(paste("Number of nodes:", vcount(g)))
```

```
## [1] "Number of nodes: 5242"
```

```r
print(paste("Number of edges:", ecount(g)))
```

```
## [1] "Number of edges: 14484"
```

## Optimize this

### a) Construct a graph G based on the data set

Analyze some basic network properties of G including order, size, density, connectivity (if G is not connected, find the number of components of G and the fraction of vertices in the largest component), and clustering coefficient.

```r
num_nodes <- vcount(g)
print(paste("Order (Number of Nodes):", num_nodes))
```

```
## [1] "Order (Number of Nodes): 5242"
```

```r
# 2. Size (Number of Edges)
num_edges <- ecount(g)
print(paste("Size (Number of Edges):", num_edges))
```

```
## [1] "Size (Number of Edges): 14484"
```

```r
# 3. Density
# Density = E / E_max, where E is the number of edges, E_max is the max possible edges.
# For an undirected graph without loops, E_max = N * (N - 1) / 2
graph_density <- edge_density(g)
print(paste("Density:", sprintf("%.6f", graph_density))) # Format for readability
```

```
## [1] "Density: 0.001054"
```

```r
# 4. Connectivity
is_graph_connected <- is_connected(g)
print(paste("Is the graph connected?", is_graph_connected))
```

```
## [1] "Is the graph connected? FALSE"
```

```r
if (!is_graph_connected) {
  components_info <- components(g)
  num_components <- components_info$no
  print(paste("Number of connected components:", num_components))

  # Size of the largest component
  lcc_size <- max(components_info$csize)
  print(paste("Size of the largest connected component (LCC):", lcc_size))

  # Fraction of vertices in the largest component
  fraction_in_lcc <- lcc_size / num_nodes
  print(paste("Fraction of vertices in LCC:", sprintf("%.4f", fraction_in_lcc)))
} else {
  print("The graph is connected, consisting of a single component.")
  # If connected, LCC size is total nodes, fraction is 1.
  lcc_size <- num_nodes
  fraction_in_lcc <- 1.0
  print(paste("Size of the largest connected component (LCC):", lcc_size))
  print(paste("Fraction of vertices in LCC:", sprintf("%.4f", fraction_in_lcc)))
}
```

```
## [1] "Number of connected components: 355"
## [1] "Size of the largest connected component (LCC): 4158"
## [1] "Fraction of vertices in LCC: 0.7932"
```

```
# 5. Clustering Coefficient (Transitivity)
# igraph provides two main types:
# - Global clustering coefficient (transitivity): ratio of triangles to connected triples
# - Average clustering coefficient: average of the local clustering coefficient for each node

global_cc <- transitivity(g, type = "global")
average_cc <- transitivity(g, type = "average")

print(paste("Global Clustering Coefficient (Transitivity):", sprintf("%.4f", global_cc)))
```

```
## [1] "Global Clustering Coefficient (Transitivity): 0.6298"
```

```
print(paste("Average Clustering Coefficient:", sprintf("%.4f", average_cc)))
```

```
## [1] "Average Clustering Coefficient: 0.6865"
```

```
# Store results for potential later use
basic_properties <- list(
  order = num_nodes,
  size = num_edges,
  density = graph_density,
  is_connected = is_graph_connected,
  num_components = if (!is_graph_connected) components_info$no else 1,
  lcc_size = lcc_size,
  fraction_in_lcc = fraction_in_lcc,
  global_clustering_coeff = global_cc,
  average_clustering_coeff = average_cc
)
```

### b) Generate a configuration model G*

Generate a configuration model G* that has the same degree sequence as that of G's.

```
g_undir <- as.undirected(g, mode = "collapse")
```

```
## Warning: `as.undirected()` was deprecated in igraph 2.1.0.
## i Please use `as_undirected()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
deg_seq_original <- degree(g_undir)
print(paste("Successfully extracted degree sequence from the original graph. Number of degrees:", lengt
```

```
## [1] "Successfully extracted degree sequence from the original graph. Number of degrees: 5242"
```

```
# Check if the sum of degrees is even (necessary condition for graph construction)
if (sum(deg_seq_original) %% 2 != 0) {
  stop("Sum of degrees is odd. Cannot construct a graph with this degree sequence. This is unexpected f
} else {
    print("Sum of degrees is even, proceeding to generate G*.")
}
```

```
## [1] "Sum of degrees is even, proceeding to generate G*."
```

```
# 2. Generate the configuration model graph G* using sample_degseq
# We use the degree sequence from the original graph 'g_undir'.
# The method 'fast.heur.simple' is recommended by igraph warnings over 'simple.no.multiple'.
# It attempts to create a simple graph (no loops, no multiple edges) matching the degree sequence.
# Note: The *pure* configuration model allows loops and multi-edges (methods 'simple' or 'vl'),
# but often the goal is to compare against a simple graph, hence this method choice.
print("Generating Configuration Model graph G*...")
```

```
## [1] "Generating Configuration Model graph G*..."
```

```
G_star <- sample_degseq(deg_seq_original, method = "fast.heur.simple")
print("Configuration Model graph G* generated successfully.")
```

```
## [1] "Configuration Model graph G* generated successfully."
```

```
# 3. Optional Verification: Check if G* has the correct degree sequence
deg_seq_G_star <- degree(G_star)
if (length(deg_seq_original) == length(deg_seq_G_star) && all(sort(deg_seq_original) == sort(deg_seq_G_s
  print("Verification successful: G* has the same degree sequence as G.")
} else {
  warning("Verification potentially failed: The degree sequence of G* does not perfectly match G. This r
  # You could add more detailed comparison here if needed
  # print(summary(deg_seq_original))
  # print(summary(deg_seq_G_star))
}
```

```
## [1] "Verification successful: G* has the same degree sequence as G."
```

```
deg_seq_G_star <- degree(G_star)
print("Extracted degree sequence from the Configuration Model graph G*.")
```

```
## [1] "Extracted degree sequence from the Configuration Model graph G*."
```

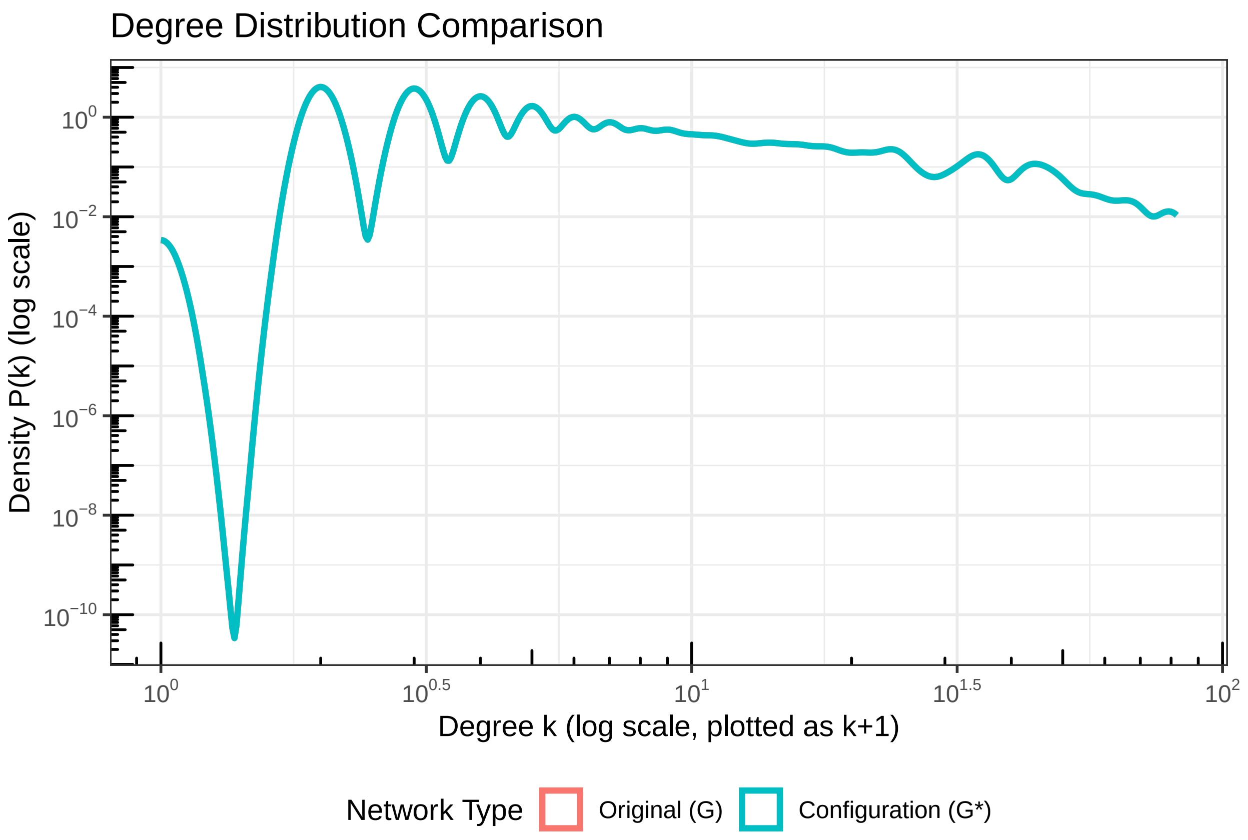
```
# Create a data frame suitable for ggplot
# Combine degree sequences and add a factor to identify the network
deg_df <- data.frame(
  Degree = c(deg_seq_original, deg_seq_G_star),
  Network = factor(
              rep(c("Original (G)", "Configuration (G*)"),
                  times = c(length(deg_seq_original), length(deg_seq_G_star))),
              levels = c("Original (G)", "Configuration (G*)") # Control plotting order
            )
)

print("Created combined data frame for plotting.")
```

```
## [1] "Created combined data frame for plotting."
```

```
head(deg_df) # Show the first few rows
```

```
##   Degree      Network
## 1      8 Original (G)
## 2     13 Original (G)
## 3      29 Original (G)
## 4      20 Original (G)
## 5       4 Original (G)
## 6      25 Original (G)
```

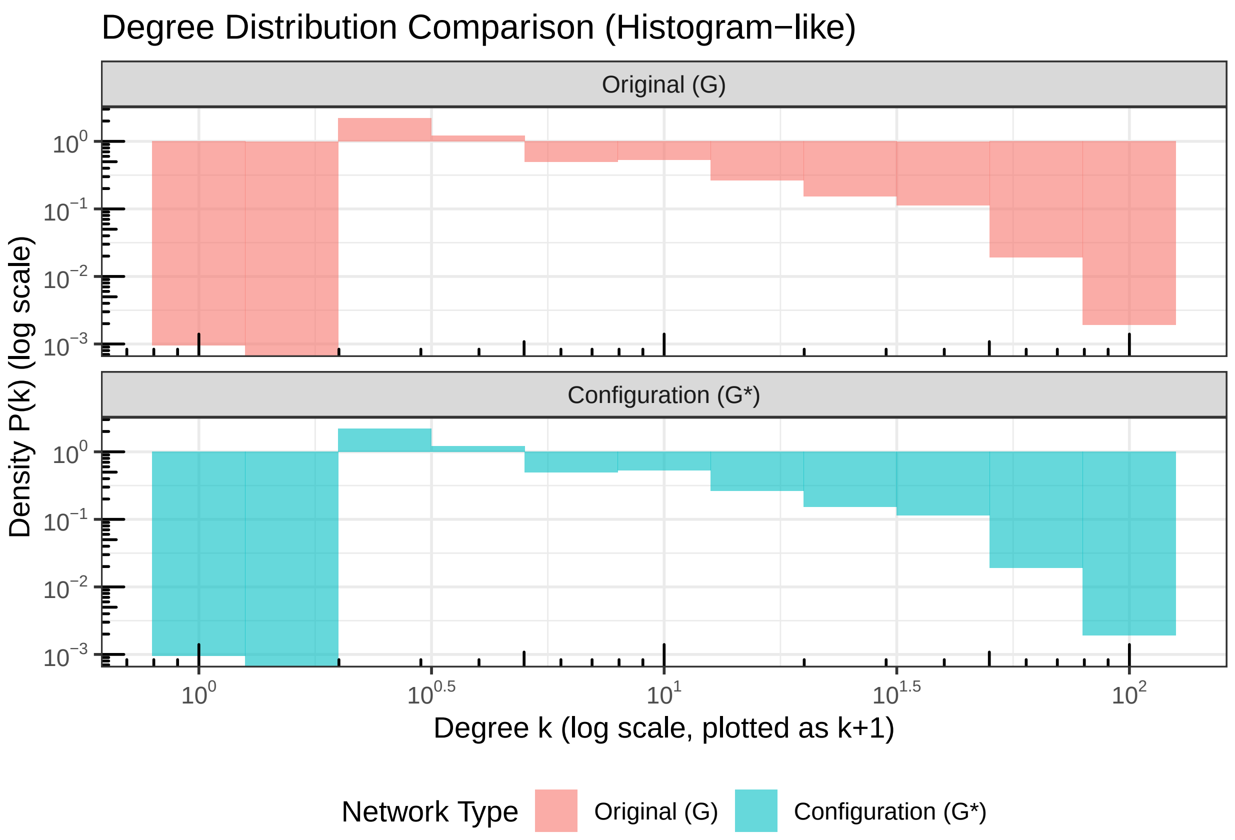
```r
# --- Plotting the Degree Distributions ---

# Using density plots on a log-log scale is common for network degrees

plot_deg_dist <- ggplot(deg_df, aes(x = Degree + 1, color = Network)) + # Add 1 to Degree to avoid log(
  geom_density(alpha = 0.7, linewidth = 1.1, adjust=0.5) + # Use geom_density for smoothed distribution
  scale_x_log10(breaks = scales::trans_breaks("log10", function(x) 10^x), # Log scale for x-axis
                labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  scale_y_log10(breaks = scales::trans_breaks("log10", function(x) 10^x), # Log scale for y-axis
                labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  labs(
    title = "Degree Distribution Comparison",
    x = "Degree k (log scale, plotted as k+1)",
    y = "Density P(k) (log scale)",
    color = "Network Type" # Legend title
  ) +
  theme_bw() + # A clean theme
  theme(legend.position = "bottom") +
  annotation_logticks() # Add log tick marks

# Print the plot
print(plot_deg_dist)
```



```r
# --- Alternative: Histogram-like plot using stat_bin ---
# This gives a feel closer to the binned plots in the original example, but automated by ggplot
# Note: You might need to adjust binwidth or bins for a good look

plot_deg_hist <- ggplot(deg_df, aes(x = Degree + 1, fill = Network)) + # Add 1 to Degree to avoid log(0
  # Use stat_bin to create histogram bars, position="identity" overlays them
  # Use ..density.. on y-axis to get probability density
  stat_bin(aes(y = ..density..), binwidth = 0.2, position = "identity", alpha = 0.6) +
  scale_x_log10(breaks = scales::trans_breaks("log10", function(x) 10^x),
                labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  scale_y_log10(breaks = scales::trans_breaks("log10", function(x) 10^x),
                labels = scales::trans_format("log10", scales::math_format(10^.x)),
                limits = c(NA, NA)) + # Adjust y-limits if needed, NA keeps defaults
  labs(
    title = "Degree Distribution Comparison (Histogram-like)",
    x = "Degree k (log scale, plotted as k+1)",
    y = "Density P(k) (log scale)",
    fill = "Network Type"
  ) +
  theme_bw() +
  theme(legend.position = "bottom") +
  annotation_logticks() +
  facet_wrap(~Network, ncol=1) # Separate panels for clarity

print(plot_deg_hist)
```

```
## Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(density)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
## Warning in scale_y_log10(breaks = scales::trans_breaks("log10", function(x)
## 10^x), : log-10 transformation introduced infinite values.
```



### c) Analyze model properties Analyze some basic network properties of G*.

```
num_nodes_star <- vcount(G_star)
print(paste("G* Order (Number of Nodes):", num_nodes_star))
```

```
## [1] "G* Order (Number of Nodes): 5242"
```

```
# 2. Size (Number of Edges)
# Should be the same as the original graph because the degree sequence sum is preserved
num_edges_star <- ecount(G_star)
print(paste("G* Size (Number of Edges):", num_edges_star))
```

```
## [1] "G* Size (Number of Edges): 14484"
```

```
# 3. Density
# Density = E / E_max, where E is the number of edges, E_max is the max possible edges.
# For an undirected graph without loops, E_max = N * (N - 1) / 2
# Should be the same as the original graph if N and E are the same.
graph_density_star <- edge_density(G_star)
print(paste("G* Density:", sprintf("%.6f", graph_density_star))) # Format for readability
```

```
## [1] "G* Density: 0.001054"
```

```
# 4. Connectivity
is_connected_star <- is_connected(G_star)
print(paste("Is G* connected?", is_connected_star))
```

```
## [1] "Is G* connected? FALSE"
```

```
if (!is_connected_star) {
  components_info_star <- components(G_star)
```

```r
  num_components_star <- components_info_star$no
  print(paste("G* Number of connected components:", num_components_star))

  # Size of the largest component in G*
  lcc_size_star <- max(components_info_star$csize)
  print(paste("G* Size of the largest connected component (LCC):", lcc_size_star))

  # Fraction of vertices in the largest component in G*
  fraction_in_lcc_star <- lcc_size_star / num_nodes_star
  print(paste("G* Fraction of vertices in LCC:", sprintf("%.4f", fraction_in_lcc_star)))
} else {
  print("G* is connected, consisting of a single component.")
  # If connected, LCC size is total nodes, fraction is 1.
  lcc_size_star <- num_nodes_star
  fraction_in_lcc_star <- 1.0
  print(paste("G* Size of the largest connected component (LCC):", lcc_size_star))
  print(paste("G* Fraction of vertices in LCC:", sprintf("%.4f", fraction_in_lcc_star)))
}
```

```
## [1] "G* Number of connected components: 23"
## [1] "G* Size of the largest connected component (LCC): 5198"
## [1] "G* Fraction of vertices in LCC: 0.9916"
```

```r
# 5. Clustering Coefficient (Transitivity)
# Global clustering coefficient (transitivity)
global_cc_star <- transitivity(G_star, type = "global")

# Average clustering coefficient
# Note: transitivity(type="average") might return NaN if degrees are low or graph is small.
# A more robust way is often to calculate local CC for each node and average non-NaN values.
local_cc_vector_star <- transitivity(G_star, type = "local")
average_cc_star <- mean(local_cc_vector_star, na.rm = TRUE) # Average excluding NaNs

print(paste("G* Global Clustering Coefficient (Transitivity):", sprintf("%.4f", global_cc_star)))
```

```
## [1] "G* Global Clustering Coefficient (Transitivity): 0.0087"
```

```r
print(paste("G* Average Clustering Coefficient:", sprintf("%.4f", average_cc_star)))
```

```
## [1] "G* Average Clustering Coefficient: 0.0091"
```

```r
# Store results for potential later use or comparison
basic_properties_star <- list(
  order = num_nodes_star,
  size = num_edges_star,
  density = graph_density_star,
  is_connected = is_connected_star,
  num_components = if (!is_connected_star) components_info_star$no else 1,
  lcc_size = lcc_size_star,
  fraction_in_lcc = fraction_in_lcc_star,
  global_clustering_coeff = global_cc_star,
  average_clustering_coeff = average_cc_star
)

print("--- Analysis of G* complete ---")
```

```
## [1] "--- Analysis of G* complete ---"
```

```
if (exists("basic_properties")) {
    print("--- Comparison G vs G* ---")
    print(paste("Order: G =", basic_properties$order))
    print(paste("Size: G =", basic_properties$size))
    print(paste("Density: G =", sprintf("%.6f", basic_properties$density)))
    print(paste("Connected: G =", basic_properties$is_connected))
    print(paste("# Components: G =", basic_properties$num_components))
    print(paste("LCC Fraction: G =", sprintf("%.4f", basic_properties$fraction_in_lcc)))
    print(paste("Global CC: G =", sprintf("%.4f", basic_properties$global_clustering_coeff)))
    print(paste("Average CC: G =", sprintf("%.4f", basic_properties$average_clustering_coeff)))
    print("---------------------------")
}
```

```
## [1] "--- Comparison G vs G* ---"
## [1] "Order: G = 5242"
## [1] "Size: G = 14484"
## [1] "Density: G = 0.001054"
## [1] "Connected: G = FALSE"
## [1] "# Components: G = 355"
## [1] "LCC Fraction: G = 0.7932"
## [1] "Global CC: G = 0.6298"
## [1] "Average CC: G = 0.6865"
## [1] "---------------------------"
```

### d) Compare networks

identify similarities and differences between G and G*.

```
if (exists("basic_properties")) {
    print("--- Comparison G vs G* ---")
    print(paste("Order: G =", basic_properties$order, ", G* =", basic_properties_star$order))
    print(paste("Size: G =", basic_properties$size, ", G* =", basic_properties_star$size))
    print(paste("Density: G =", sprintf("%.6f", basic_properties$density), ", G* =", sprintf("%.6f", bas
    print(paste("Connected: G =", basic_properties$is_connected, ", G* =", basic_properties_star$is_conn
    print(paste("# Components: G =", basic_properties$num_components, ", G* =", basic_properties_star$nu
    print(paste("LCC Fraction: G =", sprintf("%.4f", basic_properties$fraction_in_lcc), ", G* =", sprin
    print(paste("Global CC: G =", sprintf("%.4f", basic_properties$global_clustering_coeff), ", G* =",
    print(paste("Average CC: G =", sprintf("%.4f", basic_properties$average_clustering_coeff), ", G* ="
    print("---------------------------")
}
```

```
## [1] "--- Comparison G vs G* ---"
## [1] "Order: G = 5242 , G* = 5242"
## [1] "Size: G = 14484 , G* = 14484"
## [1] "Density: G = 0.001054 , G* = 0.001054"
## [1] "Connected: G = FALSE , G* = FALSE"
## [1] "# Components: G = 355 , G* = 23"
## [1] "LCC Fraction: G = 0.7932 , G* = 0.9916"
## [1] "Global CC: G = 0.6298 , G* = 0.0087"
## [1] "Average CC: G = 0.6865 , G* = 0.0091"
## [1] "---------------------------"
```

```
print("--- Calculating Assortativity ---")
```

```
## [1] "--- Calculating Assortativity ---"
```

```r
assort_g <- assortativity_degree(g_undir, directed = FALSE)
assort_g_star <- assortativity_degree(G_star, directed = FALSE)

print(paste("Assortativity (G):", sprintf("%.4f", assort_g)))
```

```
## [1] "Assortativity (G): 0.6593"
```

```r
print(paste("Assortativity (G*):", sprintf("%.4f", assort_g_star)))
```

```
## [1] "Assortativity (G*): 0.0025"
```

```r
print("Interpretation: Positive values indicate assortativity (high-degree nodes connect to high-degree
```

```
## [1] "Interpretation: Positive values indicate assortativity (high-degree nodes connect to high-degre
```

```r
print("---------------------------------")
```

```
## [1] "---------------------------------"
```

```r
# --- 2. Average Path Length ---
print("--- Calculating Average Path Length (may take time) ---")
```

```
## [1] "--- Calculating Average Path Length (may take time) ---"
```

```r
avg_path_g <- mean_distance(g_undir, directed = FALSE)
avg_path_g_star <- mean_distance(G_star, directed = FALSE)

print(paste("Average Path Length (G):", sprintf("%.4f", avg_path_g)))
```

```
## [1] "Average Path Length (G): 6.0485"
```

```r
print(paste("Average Path Length (G*):", sprintf("%.4f", avg_path_g_star)))
```

```
## [1] "Average Path Length (G*): 4.3345"
```

```r
# --- 3. Diameter ---
print("--- Calculating Diameter (may take time) ---")
```

```
## [1] "--- Calculating Diameter (may take time) ---"
```

```r
diam_g <- diameter(g_undir, directed = FALSE, unconnected = TRUE)
diam_g_star <- diameter(G_star, directed = FALSE, unconnected = TRUE)

print(paste("Diameter (G):", diam_g))
```

```
## [1] "Diameter (G): 17"
```

```r
print(paste("Diameter (G*):", diam_g_star))
```

```
## [1] "Diameter (G*): 10"
```

```r
print("(Note: For disconnected graphs, diameter is often calculated for the LCC or reported as Inf)")
```

```
## [1] "(Note: For disconnected graphs, diameter is often calculated for the LCC or reported as Inf)"
```

```r
print("---------------------------------")
```

```
## [1] "---------------------------------"
```

```r
# --- 4. Centrality Distributions ---
print("--- Calculating Centrality Distributions (may take time) ---")
```

```
## [1] "--- Calculating Centrality Distributions (may take time) ---"
```

```r
# a) Betweenness Centrality
print("Calculating Betweenness...")
```

```
## [1] "Calculating Betweenness..."
```

```r
btw_g <- betweenness(g_undir, directed = FALSE, normalized = TRUE)
btw_g_star <- betweenness(G_star, directed = FALSE, normalized = TRUE)

# b) Closeness Centrality
print("Calculating Closeness...")
```

```
## [1] "Calculating Closeness..."
```

```r
close_g <- closeness(g_undir, mode = "total", normalized = TRUE)
close_g_star <- closeness(G_star, mode = "total", normalized = TRUE)

# c) Eigenvector Centrality
print("Calculating Eigenvector...")
```

```
## [1] "Calculating Eigenvector..."
```

```r
eig_g_result <- eigen_centrality(g_undir, directed = FALSE, scale = TRUE)
```

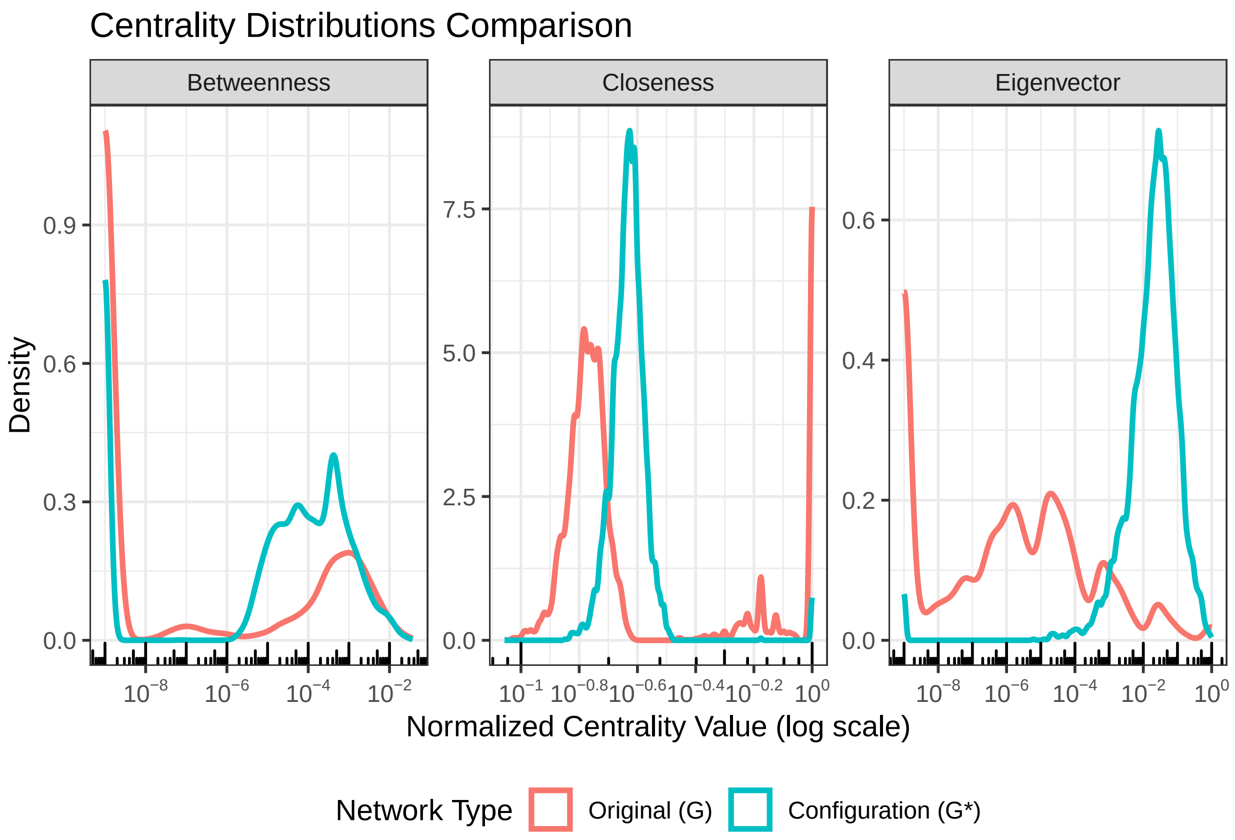
```
## Warning: The `scale` argument of `eigen_centrality()` is deprecated as of igraph 2.1.1.
## i eigen_centrality() will always behave as if scale=TRUE were used.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
eig_g_star_result <- eigen_centrality(G_star, directed = FALSE, scale = TRUE)
eig_g <- eig_g_result$vector
eig_g_star <- eig_g_star_result$vector


# --- Plotting Centrality Distributions ---
print("Plotting Centrality Distributions...")
```

```
## [1] "Plotting Centrality Distributions..."
```

```r
# Combine data for plotting
centrality_df <- data.frame(
  Value = c(btw_g, btw_g_star, close_g, close_g_star, eig_g, eig_g_star),
  Centrality = factor(rep(c("Betweenness", "Closeness", "Eigenvector"),
                         each = length(btw_g) + length(btw_g_star))),
  Network = factor(rep(rep(c("Original (G)", "Configuration (G*)"), each = length(btw_g)), 3),
                  levels = c("Original (G)", "Configuration (G*)"))
)

# Plotting (using density plots, add + small constant for log scale if needed)
plot_centrality <- ggplot(centrality_df, aes(x = Value + 1e-9, color = Network)) + # Add small value fo
  geom_density(alpha = 0.7, linewidth=1.0, adjust=0.5) +
  scale_x_log10(breaks = scales::trans_breaks("log10", function(x) 10^x),
               labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  facet_wrap(~ Centrality, scales = "free") +
  labs(
    title = "Centrality Distributions Comparison",
    x = "Normalized Centrality Value (log scale)",
    y = "Density",
    color = "Network Type"
  ) +
  theme_bw() +
  theme(legend.position = "bottom") +
  annotation_logticks(sides = "b")

print(plot_centrality)
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (`stat_density()`).
```



```
print("----------------------------------")
```

```
## [1] "----------------------------------"
```

```
# --- 5. Community Detection ---
print("--- Performing Community Detection (Louvain) ---")
```

```
## [1] "--- Performing Community Detection (Louvain) ---"
```

```
# Run Louvain algorithm
comm_g <- cluster_louvain(g_undir)
comm_g_star <- cluster_louvain(G_star)

# Compare Modularity
mod_g <- modularity(comm_g)
mod_g_star <- modularity(comm_g_star)
print(paste("Modularity (G):", sprintf("%.4f", mod_g)))
```

```
## [1] "Modularity (G): 0.8599"
```

```r
print(paste("Modularity (G*):", sprintf("%.4f", mod_g_star)))
```

```
## [1] "Modularity (G*): 0.4092"
```

```r
print("Higher modularity suggests stronger community structure.")
```

```
## [1] "Higher modularity suggests stronger community structure."
```

```r
# Compare Number of Communities
num_comm_g <- length(sizes(comm_g))
num_comm_g_star <- length(sizes(comm_g_star))
print(paste("Number of Communities (G):", num_comm_g))
```

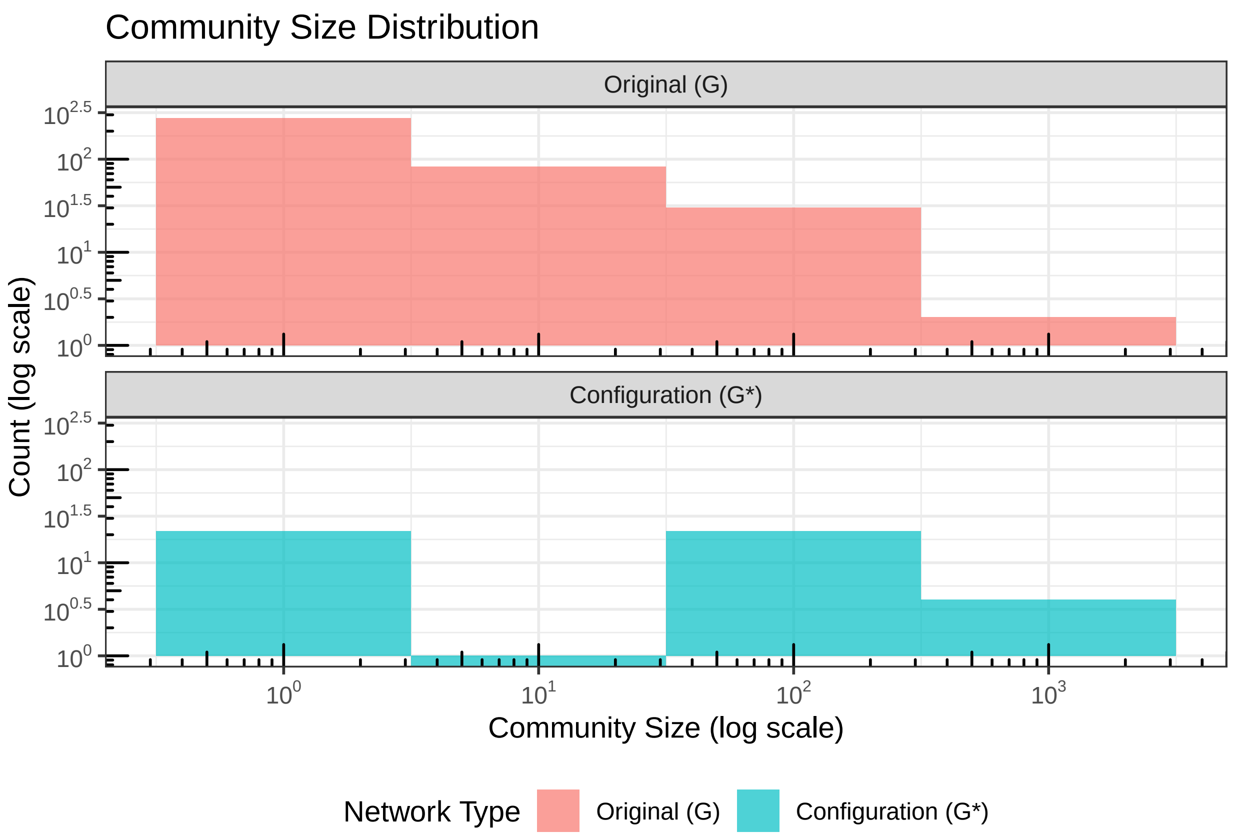
```
## [1] "Number of Communities (G): 391"
```

```r
print(paste("Number of Communities (G*):", num_comm_g_star))
```

```
## [1] "Number of Communities (G*): 48"
```

```r
# Compare Community Size Distributions
sizes_g <- sizes(comm_g)
sizes_g_star <- sizes(comm_g_star)

community_size_df <- data.frame(
  Size = c(as.numeric(sizes_g), as.numeric(sizes_g_star)),
  Network = factor(rep(c("Original (G)", "Configuration (G*)"),
                       times = c(length(sizes_g), length(sizes_g_star))),
                   levels = c("Original (G)", "Configuration (G*)"))
)

# Plotting community size distributions (histogram often works well)
plot_comm_sizes <- ggplot(community_size_df, aes(x = Size, fill = Network)) +
  geom_histogram(binwidth = 1, position = "identity", alpha = 0.7) + # Adjust binwidth as needed
  scale_x_log10(breaks = scales::trans_breaks("log10", function(x) 10^x),
                labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  scale_y_log10(breaks = scales::trans_breaks("log10", function(x) 10^x),
                labels = scales::trans_format("log10", scales::math_format(10^.x))) +
  labs(
    title = "Community Size Distribution",
    x = "Community Size (log scale)",
    y = "Count (log scale)",
    fill = "Network Type"
  ) +
  theme_bw() +
  theme(legend.position = "bottom") +
  annotation_logticks() +
  facet_wrap(~Network, ncol=1) # Separate panels often clearer for histograms

print(plot_comm_sizes)
```

```
## Warning in scale_y_log10(breaks = scales::trans_breaks("log10", function(x)
## 10^x), : log-10 transformation introduced infinite values.
```


Community Size Distribution
Original (G)
Configuration (G*)
Count (log scale)
$10^{2.5}$
$10^{2}$
$10^{1.5}$
$10^{1}$
$10^{0.5}$
$10^{0}$
$10^{0}$
$10^{1}$
$10^{2}$
$10^{3}$
Community Size (log scale)
Network Type
Original (G)
Configuration (G*)