New Adjacency Matrix and CEMiTool

Filipe Russo

June 09, 2019

Our own Adjacency Matrix

The use of the `adjacency` function from the `R` package `WGCNA` takes heavily into account the supposition that biological networks obey a power law, hence all the trouble of finding the soft thresholding power seen previously. We decided to trail a different path with Pearson Correlation and a FDR (False Discovery Rate) correction method so to construct *our own adjacency matrix*.

My fellow researcher at LABIS Rodrigo Dorado developed the code below, it takes our expression data and returns a new adjacency matrix.

```
library(parallel)
##source("https://bioconductor.org/biocLite.R")
##biocLite("qvalue")
##browseVignettes("qvalue")
library(qvalue)

## The main function to get the correlations and p_values.
## initial_matrix matrix The matrix to get the correlations and the p_values.
## titles array The name of the rows of the matrix.
## divided int The number of rows to get the correlations in every cluster.
## opt String Can be parallel or non-parallel.
## num_cores int The number of cores to execute the parallel option.
## Rodrigo Dorado
getCorrelationsPValuesParallel <- function(initial_matrix,
                                           titles,
                                           divided,
                                           opt = "parallel",
                                           num_cores = 0) {

  ## The function to get the correlation and the p_value.
  ## data_matrix matrix The matrix to get the correlations and the p_values.
  ## method String The method to use in the corrrelation.
  ## Rodrigo Dorado
  cor.P.Values <- function(data_matrix, method="pearson") {
    P_values              <- matrix(rep(0, ncol(data_matrix) ^ 2),
                                    nc = ncol(data_matrix),
                                    nr = ncol(data_matrix))
    colnames(P_values)    <- rownames(P_values) <- colnames(data_matrix)
    correlation           <- matrix(rep(1, ncol(data_matrix) ^ 2),
                                    nc = ncol(data_matrix),
                                    nr = ncol(data_matrix))
    colnames(correlation) <- rownames(correlation) <- colnames(data_matrix)
    for (i in 1:(ncol(data_matrix) - 1)) {
      for (j in (i + 1):ncol(data_matrix)) {
        result            <- cor.test(data_matrix[,i], data_matrix[,j], method = method)
        P_values[i,j]     <- P_values[j,i]    <- result$p.value
        correlation[i,j]  <- correlation[j,i] <- result$estimate
      }
```

```
  }
  return(list("correlation" = correlation, "P_values" = P_values))
}

## The main function to get the correlations and p_values in parallel mode.
## i int The divison to execute.
## data matrix All the matrix to get the correlations and p_values.
## options array The part of the principal matrix.
## type String Can be middle or all.
## combi array The posible combinatories of the values in options array.
## Rodrigo Dorado
getCorrelationMatrix_parallel <- function(i, data, options, type, combi) {

  ## The function to get the correlation and the p_value of the same row.
  ## data_matrix matrix The row to process.
  ## Rodrigo Dorado
  cor.P.Values.oneRow <- function (data_matrix){
    P_values            <- matrix(0, nc = 1, nr = 1)
    correlation         <- matrix(1, nc = 1, nr = 1)
    colnames(P_values)  <- rownames(P_values) <- colnames(data_matrix)
    colnames(correlation) <- colnames(data_matrix)
    rownames(correlation) <- colnames(data_matrix)
    return(list("correlation" = correlation, "P_values" = P_values))
  }

  ## The function to get the correlation and the p_value.
  ## data_matrix matrix The matrix to get the correlations and the p_values.
  ## type String Can be middle or all.
  ## divideSize int The division to get only one part of the result.
  ## method String The method to use in the corrrelation.
  ## Rodrigo Dorado
  cor.P.Values <- function(data_matrix, type, divideSize, method = "pearson") {
    P_values              <- matrix(rep(0, ncol(data_matrix) ^ 2),
                                    nc = ncol(data_matrix),
                                    nr = ncol(data_matrix))
    colnames(P_values)    <- rownames(P_values)     <- colnames(data_matrix)
    correlation           <- matrix(rep(1, ncol(data_matrix) ^ 2),
                                    nc = ncol(data_matrix),
                                    nr = ncol(data_matrix))
    colnames(correlation) <- rownames(correlation)  <- colnames(data_matrix)
    for (i in 1:(ncol(data_matrix) - 1)) {
      for (j in (i + 1):ncol(data_matrix)) {
        result            <- cor.test(data_matrix[,i], data_matrix[,j], method = method)
        P_values[i,j]     <- P_values[j,i] <- result$p.value
        correlation[i,j]  <- correlation[j,i] <- result$estimate
      }
    }
    if (type == 'middle') {
      return(list("correlation" = correlation, "P_values" = P_values))
    }
    if (type == 'all') {
      size <- nrow(correlation)
      return(list("correlation_inf" = correlation[(divideSize + 1) : size,
```

```
                                                1 : divideSize],
                    "P_values_inf" = P_values[(divideSize + 1) : size,
                                              1 : divideSize],
                    "correlation_sup" = correlation[1 : divideSize,
                                                    (divideSize + 1) : size],
                    "P_values_sup" = P_values[1 : divideSize,
                                              (divideSize + 1) : size]))
    }
  }

  cor <- NULL
  if (type == 'middle') {
    x <- options[i,'values_ini']
    y <- options[i,'values_fin']
    if (x == y) {
      cor <- cor.P.Values.oneRow(t(data[x:y,]))
    } else {
      cor <- cor.P.Values(t(data[x:y,]), type)
    }
  }
  if (type == 'all') {
    combi1    <- combi[i,1]
    combi2    <- combi[i,2]
    x1        <- options[combi1,'values_ini']
    y1        <- options[combi1,'values_fin']
    x2        <- options[combi2,'values_ini']
    y2        <- options[combi2,'values_fin']
    division  <- (y1 - x1) + 1
    cor       <- cor.P.Values(t(data[c(x1:y1, x2:y2),]), type, division)
  }
  return(cor)
}

## Get the entire result matrix of all the results got of the parallel function.
## rowsNumber int Number of rows in the data matrix.
## titles The row names of the data Matrix.
## options array The part of the principal matrix.
## middleTable matrix The results of the middle part of the entire result.
## boundTable matrix The results of the combinatories betwwen the options.
## option_parallel boolena If exists middle part.
## Rodrigo Dorado
getResultMatrix <- function(rowsNumber,
                            titles,
                            options,
                            middleTable,
                            boundTable,
                            option_parallel = TRUE) {
  Result              <- matrix(NA, nrow = rowsNumber, ncol = rowsNumber)
  Result_p            <- matrix(NA, nrow = rowsNumber, ncol = rowsNumber)
  row.names(Result)   <- titles
  colnames(Result)    <- titles
  row.names(Result_p) <- titles
  colnames(Result_p)  <- titles
```

```
  if(option_parallel) {
    for(i in 1:nrow(options) ) {
      x                   <- options[i, "values_ini"]
      y                   <- options[i, "values_fin"]
      Result[x:y, x:y]    <- middleTable[[i]]$correlation
      Result_p[x:y, x:y]  <- middleTable[[i]]$P_values
    }
  }
  for(i in 1:nrow(combinatorias) ) {
    comb1                   <- combinatorias[i, 1]
    comb2                   <- combinatorias[i, 2]
    x1                      <- options[comb1, "values_ini"]
    y1                      <- options[comb1, "values_fin"]
    x2                      <- options[comb2, "values_ini"]
    y2                      <- options[comb2, "values_fin"]
    Result[x1:y1, x2:y2]    <- boundTable[[i]]$correlation_sup
    Result[x2:y2, x1:y1]    <- boundTable[[i]]$correlation_inf
    Result_p[x1:y1, x2:y2]  <- boundTable[[i]]$P_values_sup
    Result_p[x2:y2, x1:y1]  <- boundTable[[i]]$P_values_inf
  }
  return(list("correlation" = Result, "p_values" = Result_p))
}

if(nrow(initial_matrix) < divided) {
  return(list("Error" = "Can not divide the matrix in a big nuber of the rows."))
}
init_time                 <- Sys.time()
row.names(initial_matrix) <- titles
rowsNumber                <- nrow(initial_matrix)
n                         <- ceiling(rowsNumber / divided)
initial_values            <- c()
final_values              <- c()
for(i in 1:n) {
  ini <- 1 + (divided * (i - 1))
  fin <- divided * i
  if (fin > rowsNumber) {
    fin <- rowsNumber
  }
  initial_values  <- c(initial_values, ini)
  final_values    <- c(final_values, fin)
}
options       <- data.frame(option = 1:n,
                            values_ini = initial_values,
                            values_fin = final_values)
combinatorias <- t(combn(n, 2))
comb_number   <- nrow(combinatorias)
if(opt == "parallel") {
  ###parallel###
  option_parallel <- FALSE
  middleTable     <- c()
  total_cores     <- detectCores() - 1
  if(num_cores < 1) {
    num_cores <- total_cores
```

```
  }
  if(num_cores > total_cores) {
    num_cores <- total_cores
  }
  cl <- makeCluster(num_cores)
  if(divided > 1) {
    option_parallel <- TRUE
    middleTable     <- parLapply(cl,
                                 1:n,
                                 getCorrelationMatrix_parallel,
                                 initial_matrix,
                                 options,
                                 'middle')
  }
  boundTable <- parLapply(cl,
                          1:comb_number,
                          getCorrelationMatrix_parallel,
                          initial_matrix,
                          options,
                          'all',
                          combinatorias)
  stopCluster(cl)
  result <- getResultMatrix(rowsNumber,
                            titles,
                            options,
                            middleTable,
                            boundTable,
                            option_parallel)
  fin_time <- Sys.time()
  config <- list("number_cores" = num_cores,
                 "time" = fin_time - init_time,
                 "init_time" = init_time,
                 "finish_time" = fin_time)
  result$correlation[is.na(result$correlation)] <- 1
  result$p_values[is.na(result$p_values)]       <- 0
  return(list("correlation" = result$correlation,
              "p_values" = result$p_values,
              "config" = config))
  ###parallel###
} else {
  if(opt == "non-parallel"){
    ###NonParallel###
    result    <- cor.P.Values(t(initial_matrix))
    fin_time  <- Sys.time()
    config    <- list("number_cores" = NA,
                      "time" = fin_time - init_time,
                      "init_time" = init_time,
                      "finish_time" = fin_time)
    return(list("correlation" = result$correlation,
                "p_values" = result$P_values,
                "config" = config))
    ###NonParallel###
  }else{
```

```
      return(list("Error" = "Option does not exists."))
    }
  }
}

## Get the q_values and the new correlation.
## correlation matrix The original correlation matrix.
## P_values matrix The original p_values matrix.
## NaNFDRValue int, String, NA, NULL
## The value to put to the values that does not accomplished the comparation.
## comparation Double The value to comparate
## lambda int The lambda option of the qvalue function.
## Rodrigo Dorado
executeFDR <- function(correlation,
                       P_values,
                       NaNFDRValue = 0,
                       comparation = 0.05,
                       lambda = 0) {
  N                       <- nrow(correlation)
  M                       <- 2
  newCorrelation          <- matrix(NA, nc = N, nr = N)
  colnames(newCorrelation)  <- rownames(newCorrelation) <- colnames(correlation)
  q_value_result          <- qvalue(p = P_values, lambda = lambda)
  for (i in (1:N)) {
    newCorrelation[i,i] <- correlation[i,i]
    if (M <= N) {
      for (j in (M:N)){
        result <- correlation[i,j]
        if (q_value_result$qvalues[i,j] > comparation) {
          result <- NaNFDRValue
        }
        newCorrelation[j,i] <- newCorrelation[i,j] <- result
      }
      M <- M + 1
    }
  }
  return(list("newCorrelation" = newCorrelation, "qvalues" = q_value_result$qvalues))
}

datExprA <- read.csv("datExprA2.csv", sep = ",", header = TRUE)
rownames(datExprA) = datExprA$X
datExprA <- datExprA[ , -c(1)]
data <- t(datExprA)

resultProt        <- getCorrelationsPValuesParallel(data,
                                                    rownames(data),
                                                    10,
                                                    "parallel",
                                                    7)
resultCorrelation <- executeFDR(resultProt$correlation, resultProt$p_values)
adjMat <- resultCorrelation$newCorrelation
```

Topological Overlap Matrix

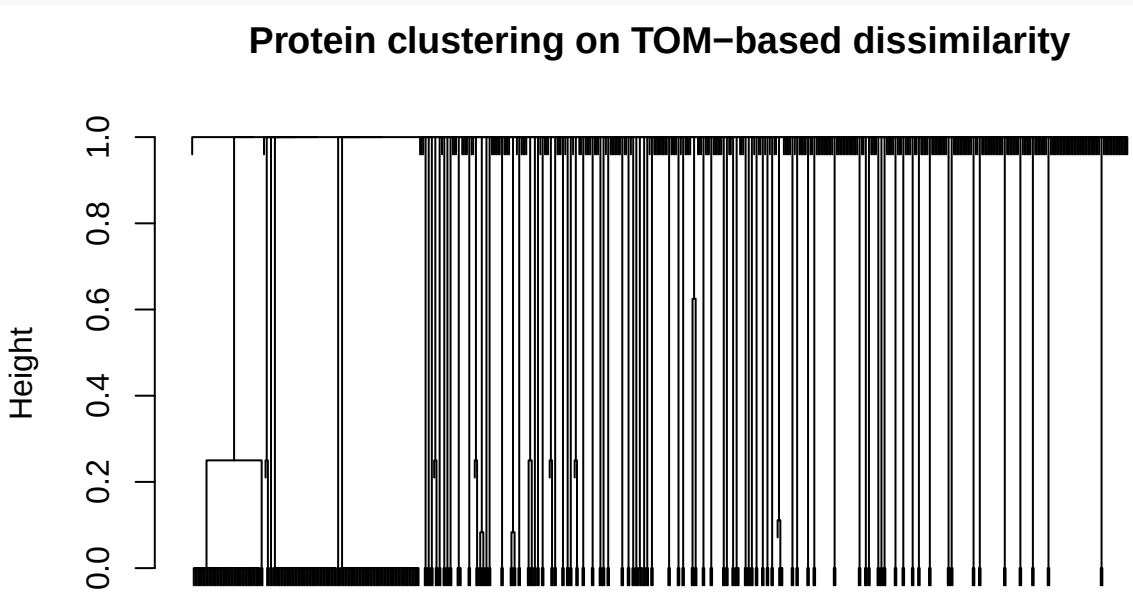
Now we pass our `adjMat` adjacency matrix to the `TOMsimilarity` function from the `R` package `WGCNA` and go on with the analysis just as we did in the previous report.

```
library(WGCNA)
```

```
# Turns adjMat matrix into TOM matrix
TOM = TOMsimilarity(adjMat = adjMat, TOMType = "signed")
```

```
## ..connectivity..
## ..matrix multiplication (system BLAS)..
## ..normalization..
## ..done.
```

```
dissTOM = 1 - TOM

# Call the hierarchical clustering function
geneTree = hclust(as.dist(dissTOM), method = "average")
geneTree$labels = names(datExprA)

# Plot the resulting clustering tree (dendrogram)
plot(geneTree, xlab = "", sub = "",
     main = "Protein clustering on TOM-based dissimilarity",
     labels = FALSE, hang = 0.04)
```




```
# We like large modules, so we set the minimum module size relatively high:
minModuleSize = 10

# Module identification using dynamic tree cut:
dynamicMods = cutreeDynamic(dendro = geneTree,
                            distM = dissTOM,
                            deepSplit = 2,
                            pamRespectsDendro = FALSE,
                            minClusterSize = minModuleSize)
```

```
##  ..cutHeight not given, setting it to 0.99  ===>  99% of the (truncated) height range in dendro.
##  ..done.
```

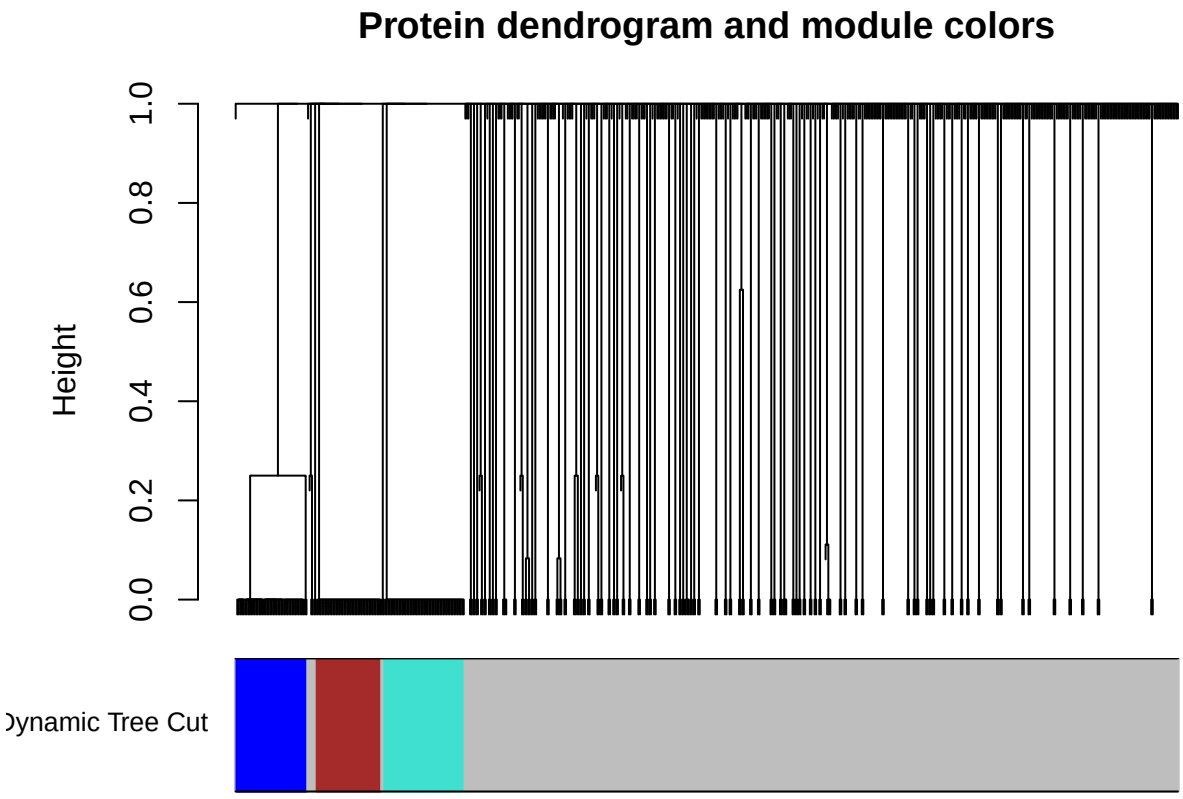
```r
table(dynamicMods)
```

```
## dynamicMods
##   0   1   2   3
## 463  51  45  41
```

```r
# Convert numeric lables into colors
dynamicColors = labels2colors(dynamicMods)
table(dynamicColors)
```

```
## dynamicColors
##      blue     brown      grey turquoise
##        45        41       463        51
```

```r
# Plot the dendrogram and colors underneath
plotDendroAndColors(geneTree,
                    dynamicColors,
                    "Dynamic Tree Cut",
                    dendroLabels = FALSE,
                    hang = 0.03,
                    addguide = TRUE,
                    guideHang = 0.05,
                    main = "Protein dendrogram and module colors")
```

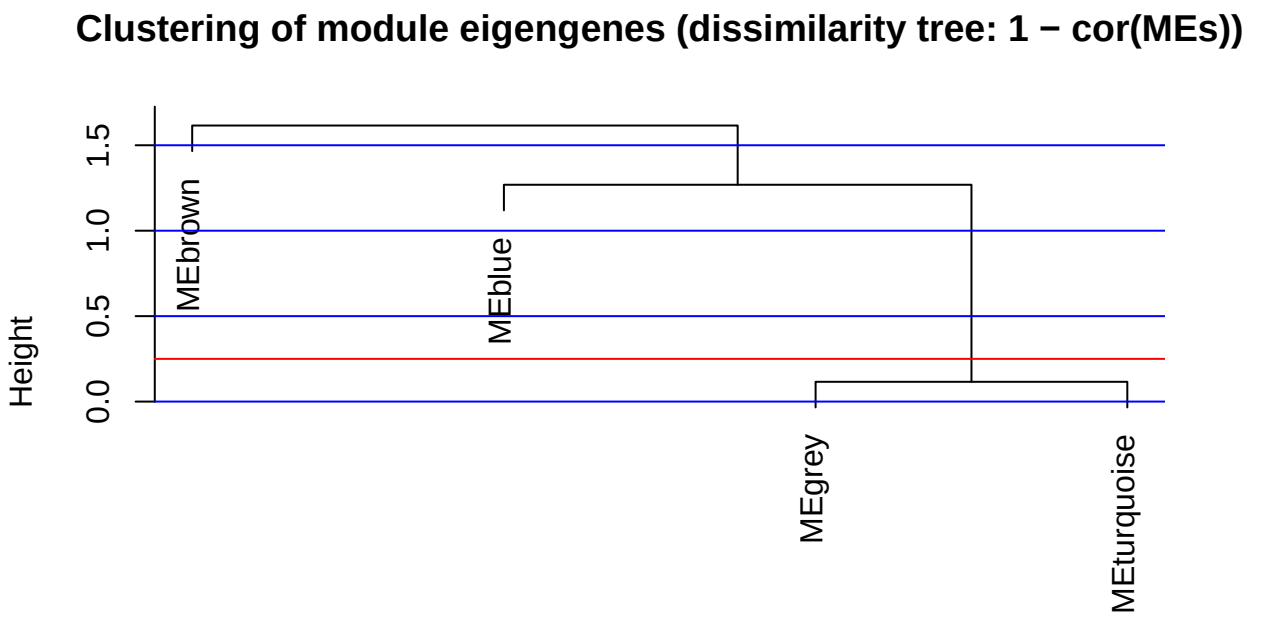


The `cutreeDynamic` function returns our 600 proteins in a 4 modules partition. Note how large is the grey module, it represents a collection of uncorrelated proteins that couldn't be grouped together elsewhere.

For the next step we try to merge modules with an intermodule correlation of at least 0.75.

```r
# Calculate eigengenes
MEList = moduleEigengenes(datExprA, colors = dynamicColors)
```

```
MEs = MEList$eigengenes

# Calculate dissimilarity of module eigengenes
MEDiss = 1 - cor(MEs);

# Cluster module eigengenes
METree = hclust(as.dist(MEDiss), method = "average")

# Plot the result
plot(METree,
     main =
     "Clustering of module eigengenes (dissimilarity tree: 1 - cor(MEs))",
     xlab = "",
     sub = "")

# Correlation of at least 0.75 necessary to merge modules
MEDissThres = 0.25

# Plot the cut line into the dendrogram
abline(h = MEDissThres, col = "red")
abline(h = 2, col = "blue")
abline(h = 1.5, col = "blue")
abline(h = 1, col = "blue")
abline(h = 0.5, col = "blue")
abline(h = 0, col = "blue")
```

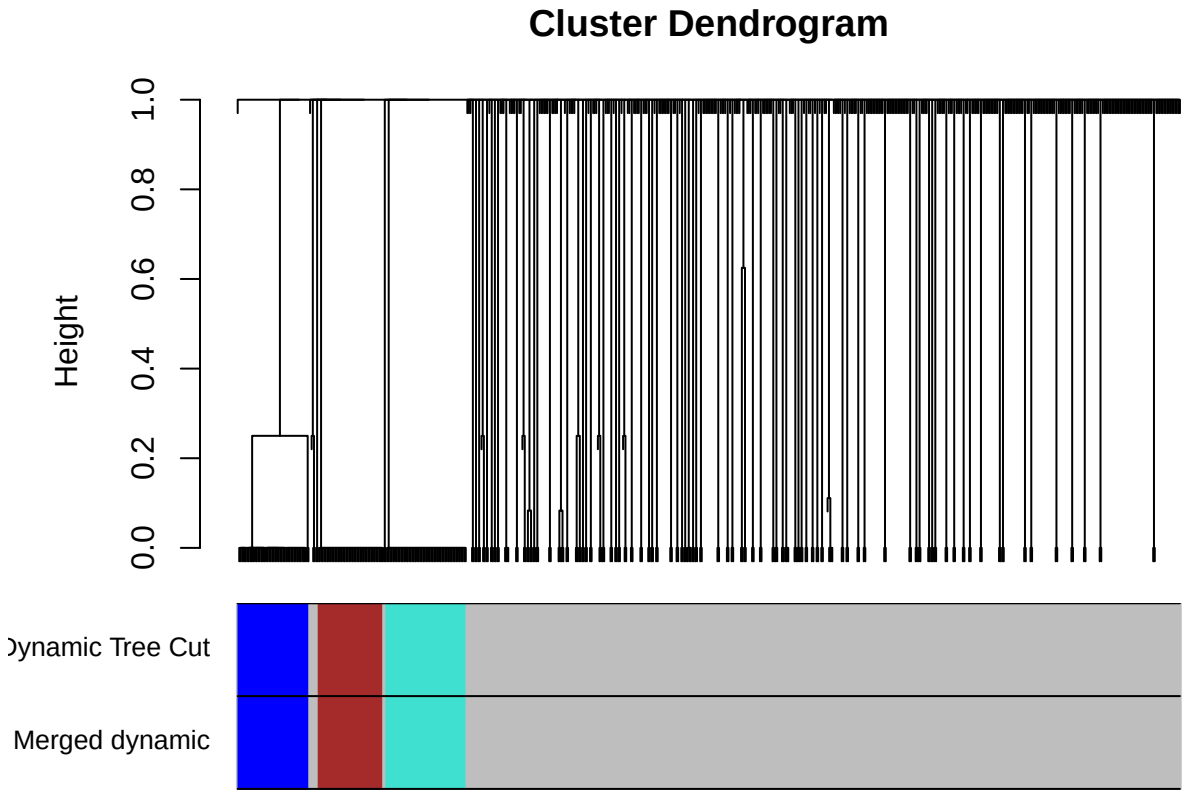



```
# Call an automatic merging function
merge = mergeCloseModules(datExprA,
                          dynamicColors,
                          cutHeight = MEDissThres,
                          verbose = 3)
```

```
##  mergeCloseModules: Merging modules whose distance is less than 0.25
##    multiSetMEs: Calculating module MEs.
##      Working on set 1 ...
```

```
##       moduleEigengenes: Calculating 4 module eigengenes in given set.
##     Calculating new MEs...
##     multiSetMEs: Calculating module MEs.
##       Working on set 1 ...
##       moduleEigengenes: Calculating 4 module eigengenes in given set.
```

```
# The merged module colors
mergedColors = merge$colors

# Eigengenes of the new merged modules:
mergedMEs = merge$newMEs

# Plot the comparision between Dynamic Tree Cut and Merged Dynamic
plotDendroAndColors(geneTree,
                    cbind(dynamicColors, mergedColors),
                    c("Dynamic Tree Cut", "Merged dynamic"),
                    dendroLabels = FALSE,
                    hang = 0.03,
                    addguide = TRUE,
                    guideHang = 0.05)
```




```
# Rename to moduleColors
moduleColors = mergedColors
table(mergedColors)
```

```
## mergedColors
##      blue     brown      grey turquoise
##        45        41       463        51
```

```
# Construct numerical labels corresponding to the colors
colorOrder = c("grey", standardColors(50))
moduleLabelsDynamic = match(dynamicColors, colorOrder) - 1
MEs = mergedMEs

moduleLabelsMerged = match(mergedColors, colorOrder) - 1

library(clues)
adjustedRand(moduleLabelsDynamic, moduleLabelsMerged)
```

```
##    Rand      HA      MA      FM Jaccard
##       1       1       1       1       1
```

```
Partition_C <- dynamicColors
```

From the result of the `adjustedRand` function from the `R` package `clues` we confirm what the plots already indicate: the Dynamic Tree Cut partition is equal to the Merged dynamic one. We stored this partition in the `Partition_C` variable.

Rand Index: Partition_B & Partition_C

Now, we will compare the `Partition_B` constructed in the previous report with the `Partition_C` constructed in this one.

```
proteins <- read.csv("proteins.csv", sep = ",", header = TRUE)
Partition_B <- proteins$Partition_B
# remember we use their numeric labelled counterparts for the adjustedRand() function
adjustedRand(match(Partition_B, colorOrder) - 1, moduleLabelsDynamic)
```

```
##       Rand         HA         MA         FM    Jaccard
## 0.46624374 0.03604860 0.03770832 0.42468687 0.24129694
```

Our Hubert–Arabie Adjusted Rand Index was 0.03604860, which according to the heuristics proposed by the researcher Douglas Steinly means a poor recovery. So the `Partition_B` comprised of the 600 proteins grouped in 4 modules can be consired very different from the `Partition_C` comprised of the 600 proteins grouped in 4 modules.

CEMiTool

CEMiTool (Co-Expression Modules identification Tool) is a systems biology method that easily identifies co-expression gene modules in a fully automated manner. We will give it a try with our proteomics data.

```
library(CEMiTool)
library(dplyr)

# loading the data
proteomics <- read.csv("datExprA.csv", sep = ",", header = TRUE)
rownames(proteomics) = proteomics$X
proteomics <- proteomics[ , -c(1)]

# preparing the data
test <- t(proteomics)
test <- as.data.frame(test)
ids <- rownames(test)
```

```
test <- test %>% mutate(ID = ids)
test <- test[, c(10, 1:9)]

# using the cemitool function
cem <- cemitool(test[, -c(1)])
```

```
##    Power SFT.R.sq    slope truncated.R.sq mean.k. median.k. max.k. Density
## 1      1 3.47e-02 -0.54900        0.41700  15.900    15.900  20.50  0.4190
## 2      2 8.64e-03  0.22900       -0.23600   9.080     8.930  12.80  0.2390
## 3      3 1.14e-05  0.00453       -0.12000   5.980     5.680   9.45  0.1570
## 4      4 1.02e-01 -0.20900       -0.00331   4.290     4.140   7.69  0.1130
## 5      5 2.29e-01 -0.33700        0.16200   3.260     3.180   6.49  0.0859
## 6      6 3.18e-01 -0.60400        0.21200   2.580     2.580   5.61  0.0679
## 7      7 5.08e-01 -0.63100        0.45200   2.100     1.940   4.92  0.0553
## 8      8 4.81e-01 -0.70200        0.35000   1.750     1.590   4.35  0.0460
## 9      9 4.40e-01 -0.73300        0.29100   1.480     1.290   3.90  0.0389
## 10    10 1.54e-01 -2.86000       -0.00259   1.270     1.030   3.52  0.0334
## 11    12 1.56e-01 -2.67000        0.02100   0.964     0.779   2.92  0.0254
## 12    14 1.60e-01 -2.55000        0.03630   0.757     0.598   2.47  0.0199
## 13    16 2.19e-01 -3.72000        0.13900   0.609     0.438   2.11  0.0160
## 14    18 1.33e-01 -2.17000       -0.10000   0.500     0.326   1.83  0.0131
## 15    20 1.33e-01 -2.02000       -0.09270   0.416     0.246   1.59  0.0110
##    Centralization Heterogeneity
## 1          0.1270         0.145
## 2          0.1030         0.250
## 3          0.0964         0.340
## 4          0.0941         0.419
## 5          0.0895         0.487
## 6          0.0841         0.547
## 7          0.0782         0.599
## 8          0.0723         0.646
## 9          0.0671         0.688
## 10         0.0624         0.727
## 11         0.0543         0.796
## 12         0.0475         0.856
## 13         0.0417         0.909
## 14         0.0369         0.957
## 15         0.0326         1.000
```

```
cem
```

```
## CEMiTool Object
## - Number of modules: 0
## - Modules: null
## - Expression file: data.frame with 757 genes and 9 samples
## - Selected data: 39 genes selected
## - Gene Set Enrichment Analysis: null
## - Over Representation Analysis: null
## - Profile plot: null
## - Enrichment plot: null
## - ORA barplot: null
## - Beta x R2 plot: null
## - Mean connectivity plot: null
```

By running our `test` in *CEMiTool* on `R` and online, we get the same result: "No beta value found. It seems

that the soft thresholding approach used by CEMiTool is not suitable for your data.". The tool developers further explain:

"The beta value is a parameter that lies in the core of the weighted gene co-expression network analysis (WGCNA). Originally, this parameter needed to be defined by the user. Therefore, the original CEMiTool R package implemented an automatic beta value selection procedure that uses the gene expression data to select the best value on behalf of the user. In some cases, however, the CEMiTool automatic procedure fails to find the best solution and cannot keep on with the co-expression analysis and this error is raised."

Our proteomics dataset differs sensibly from the dataset shown in *CEMiTool*'s tutorial. Their dataset is comprised of 25498 genes across 81 samples, while our (uncleaned) dataset is comprised of 757 proteins across 9 samples. That's probably what is interfering with the auto-detection of the beta value (soft thresholding power).

Let's take a look when *CEMiTool* runs properly:

```
tutorial <- read.csv("cemitool-expression.tsv", sep = "\t", header = TRUE)
cem2 <- cemitool(tutorial[, -c(1)])
```

```
##    Power SFT.R.sq   slope truncated.R.sq mean.k. median.k. max.k. Density
## 1      1 0.343000  0.8210          0.711  229.00   234.000  361.0 0.30100
## 2      2 0.000537  0.0206          0.583  110.00   104.000  224.0 0.14500
## 3      3 0.140000 -0.3080          0.714   64.70    56.000  155.0 0.08490
## 4      4 0.426000 -0.5610          0.876   42.20    36.200  114.0 0.05540
## 5      5 0.587000 -0.7980          0.891   29.50    25.500   88.3 0.03870
## 6      6 0.650000 -0.9720          0.907   21.60    18.100   70.1 0.02830
## 7      7 0.657000 -1.0900          0.869   16.40    12.600   56.8 0.02150
## 8      8 0.686000 -1.1100          0.857   12.80     9.010   46.6 0.01680
## 9      9 0.701000 -1.1000          0.834   10.20     6.640   38.7 0.01340
## 10    10 0.706000 -1.0800          0.808    8.32     5.040   32.5 0.01090
## 11    12 0.926000 -0.9000          0.929    5.80     3.030   23.5 0.00761
## 12    14 0.919000 -1.0700          0.949    4.26     1.910   21.2 0.00559
## 13    16 0.921000 -1.1600          0.955    3.25     1.250   19.3 0.00427
## 14    18 0.899000 -1.2200          0.900    2.56     0.857   17.7 0.00336
## 15    20 0.922000 -1.2300          0.928    2.07     0.597   16.3 0.00272
##    Centralization Heterogeneity
## 1          0.1740         0.308
## 2          0.1500         0.470
## 3          0.1180         0.565
## 4          0.0943         0.633
## 5          0.0774         0.689
## 6          0.0639         0.739
## 7          0.0532         0.787
## 8          0.0445         0.834
## 9          0.0375         0.883
## 10         0.0318         0.933
## 11         0.0233         1.040
## 12         0.0222         1.150
## 13         0.0211         1.270
## 14         0.0199         1.390
## 15         0.0187         1.520
## ..connectivity..
## ..matrix multiplication (system BLAS)..
## ..normalization..
## ..done.
##  ..cutHeight not given, setting it to 0.995  ===>  99% of the (truncated) height range in dendro.
```

```
##  ..done.
##  mergeCloseModules: Merging modules whose distance is less than 0.2
##    Calculating new MEs...
```

```
glimpse(cem2@module)
```

```
## Observations: 763
## Variables: 2
## $ genes   <chr> "12620", "23286", "217", "16814", "19199", "19539", "9...
## $ modules <chr> "M3", "M6", "M2", "M1", "M5", "M6", "M2", "M1", "M6", ...
```

```
table(cem2@module$modules)
```

```
##
##             M1             M2             M3             M4             M5
##            246            131             86             65             53
##             M6 Not.Correlated
##             51            131
```

We see the `tutorial` dataset comprised of 25498 genes across 81 samples was turned into a 763 genes Partition grouped in 7 modules, where 131 of said genes are not correlated. It means only 3% of the original genes were actually used in the network. If it had worked with our `proteomics` dataset in the same proportion as it did with the `tutorial` dataset we would have had a network with roughly 23 proteins and probably no more than one module.

Saving the Data

Finally, we store the `Partition_C` variable in our `proteins` dataframe, which we save for further analysis.

```
proteins <- read.csv("proteins.csv", sep = ",", header = TRUE)
proteins <- proteins %>% mutate(Partition_C = Partition_C)
write.csv(proteins, file = "proteins.csv", row.names = FALSE)
proteins <- read.csv("proteins.csv", sep = ",", header = TRUE)
```