**Motivating Problem**

Identifying functionally related genes of a disease of interest is crucial when understanding the pathways and which genes are associated with the disease phenotype. Determining if a set of candidate genes of a particular disease are statistically important compared to a set of non-associated genes is important in order to reject the null hypothesis where there is no difference between candidate genes and non-associated genes in a given disease. Within a network, if a gene has a weighted edge closer to 1, this could suggest the confidence that a edge exists between two genes than genes with an edge attribute closer to 0.

By assigning each gene with a connectivity score, where the higher the number, the more connections to other genes it has, we could quantify their contribution to the network as well. For disease-associated genes, one would expect a network where genes associated with a given disease have an overall average weight closer to 1 and a high connectivity score to demonstrate their relatedness with a high degree of edges. If genes are not associated with a disease, one would expect a network with an average weighted score closer to zero and a low connectivity score, since these genes would not work in concert with one another for a disease phenotype. Therefore, generating a network of the functional relatedness of genes allows for an easier way to identify candidate genes of a disease of interest by identifying the molecular pathways that underlie the pathophysiology of a disease.

**Computational Problem**

Given a full network file of genes (where the first two columns represent genes and the third column are their corresponding edge weights) and a file of disease-associated genes and their loci, generate a prix fixe network of random genes and enrich this network with a genetic algorithm to determine the most dense subnetworks. Once these networks are generated and enriched, generate a null-case subnetwork where true-candidate genes were replaced with non-informed genes in order to determine the statistical significance of the optimized prix fixe subnetworks by generating a p-value. Assign scores to each gene in the subnetwork based on its degree of edges within the network to characterize its functional relatedness in relation to the density of the network itself. These genes are ranked according to their score and added to the dataframe of genes, gene scores, p value. The top 10 subnetworks will be exported into .txt files with their corresponding p values.

**Specific Approach**

After reading in the full network file and the gene-locus file, randomly generate a 5000 subnetworks where each locus is represented by one gene that lies within it. Use a genetic algorithm on the prix fixe subnetworks in order to mutate each gene by 5% in the subnetwork where the edge count and selection score is calculated. These scores will be normalized by dividing each score by the total score and will proceed to the mating step where the two parent networks will be crossed to create a child network. An optimization function will then apply the mutation method until the average density of the network changes by less than or equal to 0.5.

To determine functional relatedness of each gene, the density of the networks with the gene included will be scored and an empty locus case where the gene is removed, and therefore, the locus is not represented will be scored based on the density without it. When subtracting the empty locus case from the density of the network with the locus included, the gene’s contribution is determined. Repeat this process for each network and then average the scores. Export the final rankings dataframe to show genes, gene scores, p value and their corresponding ranks. Export the top 10 subnetworks and their p-values, one subnetwork per file.

**Specific Implementation of Approach**

After the full network (STRING) file and the Fanconi anemia gene-locus file has been read in, a random subpopulation of FA genes is generated where one gene represents each of the 12 loci. One network would therefore have a total of 12 genes. The function random\_subnetworks creates a dictionary random\_nets and uses the FA\_genes dictionary where the keys are the loci and the values are genes associated with each loci to randomly choose a gene using dictionary comprehension and the random.choice function. The function will then return a list of 5000 dictionaries where each list is a random subnetwork and each locus is associated with a randomly selected gene. This output is now the prix\_fixe list of 5000 random subnetworks used for the input of the genetic algorithm. A limitation of this approach is the amount of memory usage required since the function generates the 5000 subnetworks and stores them in memory, so if the number of loci and genes are long, it could consume a significant amount of memory. This approach also attempts to randomize choices uniformly but some genes are more likely to be chosen more often than others (e.g., their frequency in the genome or the frequency in the full network file) and this function may not be an accurate representation of this. This function also does not track which genes have been selected in previous iterations, so the same gene can be selected multiple times, while others may not have been selected at all.

Next, a genetic algorithm will be generated to optimize the prix\_fixe subnetworks. The mutation function takes two arguments, the prix\_fixe subnetworks and the FA\_genes dictionary to create a new subnetwork where there is a 5% chance that each gene in each subnetwork will be replaced with a new gene from the same locus to introduce randomness to explore new solutions. A class named Subnetwork with a function nested inside of it will calculate the total edge count for a subnetwork and iterates over each gene in the self.genes list. For each gene, it uses the get method of the gene\_counts dictionary to find the count of edges associated with that gene. If a gene is not found in the dictionary, get returns 0. The counts are then summed up to give the total edge count for the subnetwork. These counts are used to calculate a fitness score for a subnetwork, based on the total number of connections in the subnetwork. The normalize\_scores function takes a list of Subnetwork objects as input and calculates the fitness of each subnetwork which is then normalized to a probability that reflects its proportion of the total fitness of the population. This probability is then used to select individuals for reproduction. A form of genetic crossover is generated in the mate function where it combines subnetworks from the parent subnetworks to create a child subnetwork by iterating over the genes of the two parent subnetworks in pairs, and for each pair, randomly choosing one of the genes to include in the child subnetwork. The crossing function creates a new population of subnetworks by mating the pairs of the parent subnetworks which are chosen randomly and the probability of selection being proportional to the subnetworks's selection score. The optimize function then takes prix\_fixe and the child\_subnetworks and repeatedly applies the mutation operation until the average density of the network changes by less than or equal to 0.5 to find a more optimal network structure.

Next, genes from the string file are placed into bins based on their number of connections, implemented by the bin\_genes function. Bins are of equal-sized count and genes are binned in a quantile manner, with each bin with a size of 10 connections (e.g., 0-9, 10-19, etc.). The function replace\_genes then replaces each gene in each subnetwork of prix\_fixe with genes in the same bin that are considered uninformed genes (genes with no relation to the disease phenotype). The output subnetwork of uninformed genes is then exposed to the genetic algorithm using the optimize function. The average\_density function calculates the density of a network by summing the weights of the edges in each subnetwork and divides them by the maximum possible weight of the network. Next, compare\_statistics compares the density of two subnetworks, the informed network and the uninformed network, and performs this test for a specified number of trials. If the network of uninformed genes has a higher density than the network of true-candidate genes, then a count is added to higher\_count and can be interpreted as the p value. This would help test the null hypothesis to reject the idea that the densities of the two subnetworks would be equal but have differing p values where the informed network has a small p value, indicating the densities are in fact significantly different. A limitation to this step is that it takes quite some time to run and a p value was not able to be calculated since the run time takes too long. This code would need to be modified to become more efficient to deal with larger networks.

The next step is to score the genes based on how much the gene contributes to the connectivity of the network. The function gene\_scoring takes three arguments (prix\_fixe, FA\_genes and num\_networks) as inputs where it initializes an empty dictionary gene\_scores to store the scores for each gene. It does this by converting each network in prix\_fixe to a set for faster membership tests and then iterates over each subnetwork within prix\_fixe. After, it iterates over each locus in FA\_genes and selects a random gene designated as g\_star from the network and creates a new network where g\_star is removed. This will then serve the empty locus case where the contribution of a gene to the connectivity of a network is calculated without its presence in a network. Within this function, a helper function compute\_density calculates the density of a subnetwork where the density is defined as the number of edges within the subnetwork and is stored in full\_densities. The density of the empty locus case (new\_network) is calculated and then subtracted from the original network to get the contribution of g\_star to the connectivity of the network. The function will then record the contribution of g\_star and normalize by the number of loci in gene\_scores. Some limitations of this function are its performance since the function has to iterate over each network and each locus which can be very slow if the number of networks or loci is large. It also computes the density of each network twice, so the code would need to be updated so the performance can be optimized. The function also selects a random gene from each network to be removed, and some genes may be selected multiple times and other genes may not be selected at all. This could lead to some bias in the network and not be an accurate representation of the connectivity of each gene and their gene scores. One last limitation that was noticed was that content from FA\_df was used within the function but not passed as an argument, rendering the function less reusable since it depends on the specific structure of FA\_df.

The final output is a dictionary of genes as keys and their corresponding scores as the values. The dictionary was converted into a data frame and the associated loci to each gene was added to the dataframe. The function top\_10\_subnetworks finds the top 10 most dense subnetworks by finding the subnetworks with the highest number of edges. The final\_ranking function then combines the ranks, scores, p values and locus of each gene into a dataframe for easy visualization. This dataframe is then exported to a .txt file , Day3\_Output.gmt. The function pval\_output exports 10 different files, where each file contains a subnetwork and the associated p values.

**Pseudocode**

1. Generate a random prix\_fixe population from a defined number of networks:
   1. DEFINE a function random\_subnetworks:
      1. Input a dictionary FA\_genes:
      2. Initialize an empty dictionary random\_nets
         1. For each locus in the keys of FA\_genes:
         2. Select a random gene from the list of genes at that locus in FA\_genes
         3. Add the locus and the selected gene to random\_nets
      3. Initialize an empty list to store the random subnetworks
      4. Repeat 5000 times:
         1. Initialize an empty list for the current subnetwork
         2. For each locus in the keys of FA\_genes:
            1. Select a random gene from the list of genes at that locus in FA\_genes
            2. Add the selected gene to the current subnetwork
         3. Add the current subnetwork to the list of random subnetworks
      5. Return the list of random subnetworks
2. DEFINE a function mutation:
   1. or each of the subnetworks in prix\_fixe,
   2. for each gene in the subnetwork,
   3. random integer between 1 and 100
      1. if integer is less than or equal to 5, mutate the gene
      2. find locus of gene -> locus
      3. get length of locus -> locus size
      4. get random gene from locus:
         1. new random integer between 1 and locus size
         2. new gene = locus[new random integer]
         3. newsubnet.append(new gene)
      5. else, newsubnet.append(gene)
3. DEFINE a function crossing:
   1. Convert each subnetwork into an object:
      1. Initialize the genes attribute with the genes list
      2. Calculate the edge count using calculate\_edge\_count
      3. calculate the selection score as the cube of the edge count
   2. Normalize the selection scores of the subnetwork:
      1. DEFINE a function normalize\_scores
         1. calculate the total selection score of all subnetworks
         2. FOR each subnetwork, divide its selection score by the total score
   3. DEFINE a function mate:
      1. Create a new list of genes by iterating over the genes of the two parent subnetworks and randomly select one gene from the pair.
      2. RETURN an object with the new list of genes
   4. DEFINE a function crossing:
      1. Create a new population of subnetworks by repeating following steps 5000 times:
         1. Select two parent subnetworks from the current population with the probability of selection being proportional to the subnetwork’s selection score.
4. DEFINE a function bin\_genes
   1. Convert gene\_counts dictionary to a dataframe
      1. Create columns ‘Gene’ and ‘Connections’
   2. Calculate the bin edges as a range from 0 to the maximum value in ‘Connections’ with a step size of bins
      1. Create a list of labels for bin by iterating over the bin edges and for each edge i, create a string in the format ‘i-(i+bins-1)’
      2. Add a new column ‘Bin’ to the dataframe
   3. RETURN dataframe
   4. Set bins to 10
5. DEFINE a function replace\_genes
   1. For each subnetwork in prix\_fixe:
      1. Create a new set of genes by repeating the following steps for the number of genes in the subnetwork:
         1. Randomly select a key from non\_fa\_genes\_in\_bins and add to the set
      2. RETURN a new list of subnetworks where each subnetwork is a set of genes created in the previous step
6. DEFINE a function calculate\_density:
   1. Initialize total\_weight and max\_possible\_weight to 0
   2. FOR each subnetwork in the input list:
      1. Filter string dataframe and include only rows where both Gene1 and Gene2 are in the current subnetwork.
         1. Store in a dataframe
      2. Add the sum of the ‘Weight’ column of the new dataframe to total\_weight
      3. Calculate the max\_possible\_weight
   3. TURN total\_weight divided by max\_possible\_weight IF max\_possible\_weight greater than 0, otherwise 0.
7. DEFINE a function compare\_statistics:
   1. Initialize higher\_count to 0
   2. REPEAT the following steps a user defined number of times (trial):
      1. Call calculate\_density function and store result in informed\_density
      2. Call calculate\_density function and store result in non\_informed\_density
      3. If non\_informed\_density greater than informed\_denstiy, increment higher\_count by 1
   3. RETURN higher\_count divided by trials
8. DEFINE a function gene\_scoring:
   1. pass three required arguments list prix\_fixe, a dictionary FA\_genes, and an integer num\_networks:
   2. Define a helper function compute\_density that takes a subnetwork and a full\_network:
      1. Filter the full\_network to only include rows where both genes are in the subnetwork
      2. Return the number of rows in the filtered network
   3. Initialize an empty dictionary gene\_scores
   4. Get a list of loci from the keys of FA\_genes
   5. Calculate the full density of each network in prix\_fixe using compute\_density and store in full\_densities
   6. Convert each network in prix\_fixe to a set for faster membership tests
   7. For each index i from 0 to num\_networks:
      1. If the size of the network at index i in prix\_fixe is less than the number of loci:
      2. Print a warning message
      3. Continue to the next iteration
   8. For each index j and locus in the list of loci:
      1. Select a random gene g\_star from the network at index i in prix\_fixe
      2. Create a new network from the network at index i in prix\_fixe and remove g\_star from it
      3. Calculate the density of the new network using compute\_density and store in empty\_density
      4. Subtract empty\_density from the full density of the network at index i to get the contribution of g\_star
      5. Record the contribution of g\_star to the connectivity of the loci in gene\_scores
   9. Return gene\_scores
9. DEFINE a function top\_10\_subnetworks:
   1. DEFINE a nested function compute\_density that takes a subnetwork and a full network as input:
      1. Filter the full network to include only rows where both Gene1 and Gene2 are in the subnetwork.
      2. Store the result in a dataframe.
      3. RETURN the sum of the ‘Weight’ column of the dataframe.
   2. Calculate the density of each network in prix\_fixe by calling compute\_density with each network and FA\_df as input.
      1. Store result in a dataframe
   3. Convert each network in prix\_fixe to a set for faster membership tests.
   4. Initialize an empty list top\_10.
   5. Repeat the following steps 10 times:
      1. Find the maximum density in `full\_densities` and its index.
      2. Store them in top\_density and top\_index
      3. Find the network in prix\_fixe at top\_index.
      4. Store it in top\_network.
      5. Append a tuple of top\_network and top\_density to top\_10.
      6. Remove top\_density from full\_densities and top\_network from prix\_fixe.
   6. RETURN top\_10.
10. DEFINE a function final\_ranking:
    1. Add a new column 'Rank' to gene\_scores\_df that contains the rank of each row when sorted by the 'Score' column in descending order.
    2. Add a new column 'Locus' to gene\_scores\_df that contains the locus of each gene. The locus is obtained by mapping the index of gene\_scores\_df to the reverse\_dict dictionary.
    3. Return gene\_scores\_df.
11. DEFINE a function calculate\_p\_values\_top\_10:
    1. Initialize an empty dictionary p\_values\_top\_10
    2. Create a list all\_scores that contains the scores of all networks in top\_10.
    3. FOR each network and its score in top\_10:
       1. Create a list permuted\_scores that contains num\_permutations random scores selected from all\_scores.
       2. Calculate the percentile rank of the score in permuted\_scores and divide it by 100 to get a p-value.
       3. Add the network and its p-value to p\_values\_top\_10.
    4. Return p\_values\_top\_10.
12. DEFINE a function pval\_output:
    1. FOR each network and its p-value in pval\_top\_10:
       1. OPEN a new file with a name based on the index of the network and its p-value.
       2. FOR each gene in the network:
          1. write the gene to the file followed by a newline.
    2. RETURN nothing.
13. CALL the pval\_output function with pval\_top\_10 as input.

**References**

Data formats. Data formats - GeneSetEnrichmentAnalysisWiki. (2020, December 15). <https://software.broadinstitute.org/cancer/software/gsea/wiki/index.php/Data_formats>

Taşan, M., Musso, G., Hao, T. et al. Selecting causal genes from genome-wide association studies via functionally coherent subnetworks. Nat Methods 12, 154–159 (2015). <https://doi.org/10.1038/nmeth.3215>