CPBS 7711 MODULE 4 DAY 3 ASSIGNMENT: Aishwarya Mandava

MOTIVATION:

Fanconi Anemia (FA) is a rare genetic disorder inherited in an autosomal recessive pattern and is characterized by physical abnormalities, bone marrow failure, and increased risk of malignancy. Approximately 90% of the individuals with FA have impaired bone marrow function leading to decrease in the production of Red blood cells, White blood cells, and Platelets [1]. Previous studies have found 12 genes associated with FA, including BRCA and FANC genes that play a role in DNA damage response and repair mechanisms. The construction of functional networks involving these FA-associated genes could offer valuable insights into novel molecular mechanisms and pathways. Such analyses could prove valuable in identifying sets of mutually functionally related genes within the loci linked to FA, contributing to a deeper understanding of the disease and offering insights for potential therapeutic interventions.

COMPUTATIONAL PROBLEM:

We have a protein-protein interactions (PPI) network representing various interactions between the nodes/genes and a set of loci associated with FA disease. The computational objective is to prioritize groups of candidate genes from FA-associated loci on the basis of mutual functional relatedness, i.e., mutually connected gene groups. To address this task, an initial population of 5000 random FA associated subnetworks are subjected to a genetic algorithm to enrich dense subnetworks. The resultant optimized subnetwork is compared with non-informative subnetworks, similarly optimized using the genetic algorithm, to test for statistical significance. The final optimized FA associated population is used for computing the gene scores and subsequent ranking of the candidate genes. Visualizing and analyzing the top dense subnetworks would facilitate exploring dynamic interactions and provide insights into potential signatures within the protein-protein network within the specific context of FA disease-associated loci.

SPECIFIC APPROACH:

Given that we have the disease (FA) associated genes and the protein-protein interactions (PPI), the approach involves the initial generation of a population comprising 5000 random subnetworks. Each subnetwork is constructed to include a randomly chosen representative gene from each locus. The optimization of this initial population employs a Genetic Algorithm (GA) comprising mutation and mating steps.

In the mutation step, representative genes in each locus are swapped with other genes from the same locus with a 5% probability (with replacement) uniformly at random. The computation involves determining normalized weighted edge density for each subnetwork *i* denoted as s_i that is derived from cubically transforming edge weight d^3_i of the subnetwork.

$$s_i^* = \frac{s_i}{\sum_{j=1}^{5000} s_j}$$

The mating step involves sampling pairs of subnetworks with replacement, with the probability of selecting a parent subnetwork *i* equal to s_i^*. This results in child subnetworks by randomly selecting each gene in a locus from either parent. The GA optimization terminates when the average density of the child subnetworks fails to improve the average density of the parent population by more than 0.5%.

To test the statistical significance of the Genetic Algorithm-optimized population, a null case is generated through quantile-based binning. In this null case, FA genes within each locus are replaced with a non-FA gene from the same bin. The null case population undergoes optimization using the genetic algorithm, and after repeating this process 1000 times, the p-value is computed by comparing the test statistic in the FA-optimized population with that of the 1000 non-FA optimized populations and is equal to the number of times the null case test statistic is as extreme or more extreme as the test statistic of the optimized FA population.

We then score each candidate gene within the FA population by replacing the representative gene with every other gene in that locus, and subsequently computing the edge weights within each subnetwork. We then compute the "empty case" score by completely ignoring that specific locus. The difference between the candidate gene score and the empty case indicates the contribution of the candidate gene to the rest of the subnetwork. The final average gene scores are computed by taking an average of these candidate gene scores across the 5000 subnetworks.

SPECIFIC IMPLEMENTATION:

A) Data Description:
The protein-protein interactions (PPI) network was retrieved from the STRING database [2] in the tab-delimited format. This file has 1,972,248 interactions across various genes including both FA genes and non-FA genes. The FA disease genes were retrieved from the OMIM [3] database in the Gene Map Table (GMT) format. This file has 12 loci, and 584 FA disease genes. However, not all 584 interact with other genes, only 329 disease associated genes interact with other disease associated genes. **Table 1** shows disease associated genes within loci that are present in the input file and in the string file.

Locus	0	1	2	3	4	5	6	7	8	9	10	11
All FA genes (input file)	46	50	79	50	12	50	50	50	50	48	50	50
FA genes from STRING file (FA gene connected to FA gene)	28	27	54	31	4	27	29	27	28	24	25	26

Table1: Number of disease associated genes

B) Generating 5000 subnetworks with FA genes:
This step involves selecting a random gene from each loci to form a subnetwork. This is repeated 5000 times resulting in the generation of distinct subnetworks. There are 4.67e+16 potential ways of creating a

subnetwork by randomly selecting one gene per locus (note - these 12 genes may or may not have interactions with the other genes in that subnetwork) and each subnetwork could have at most 12C2 or (12*11)/2 or 66 edges. **Figure1** shows 4 loci and illustrates the 5000 random subnetworks generated from the 12 loci. The script uses min_edges parameter to select the minimum number of edges for the subnetwork to be considered for the population. Additionally, the script ensures to create unique subnetworks. By default it is set to 0, i.e., zero or more edges per subnetwork. Setting this higher ensures denser initial population, but the run time would be longer.

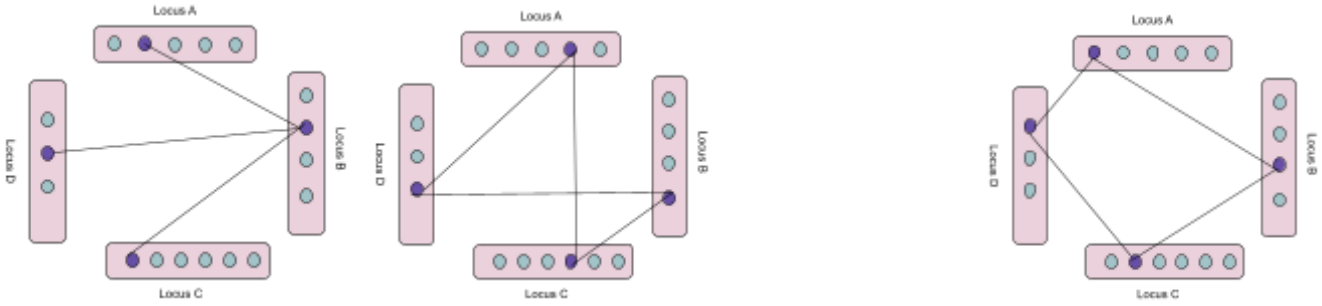


Figure1: 5000 random FA subnetworks

C) Genetic Algorithm:

Mutation: Within the mutation step, there is a 5% probability of swapping the representative genes in each subnetwork. This is achieved by generating a random number between 1 and 20 for each gene in the subnetwork and if the generated number is 1, the corresponding gene undergoes mutation. Additionally, the script ensures that the representative gene is not replaced by itself. The normalized edge weights and average edge weights are computed. **Figure 2** illustrates the mutation step with genes in yellow mutated with 5% probability.

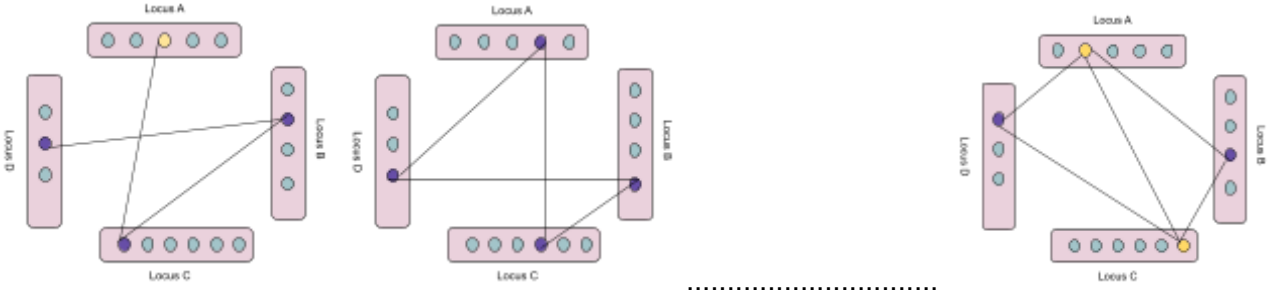


Figure 2: Mutation (Genes in Yellow are mutated with 5% probability)

Mating: Within this step, pairs of subnetworks are sampled from the population resulting in the preceding mutation step, and the probability of selection is determined by their normalized edge weights. From these selected pairs, a child subnetwork is generated by randomly selecting a gene for each locus from either parent. After 5000 such iterations, mean percent difference is calculated as:

((mean average density of the new generation - mean average density of the previous generation)*100)/(mean average density of the previous generation)

Figure 3 illustrates the mating step

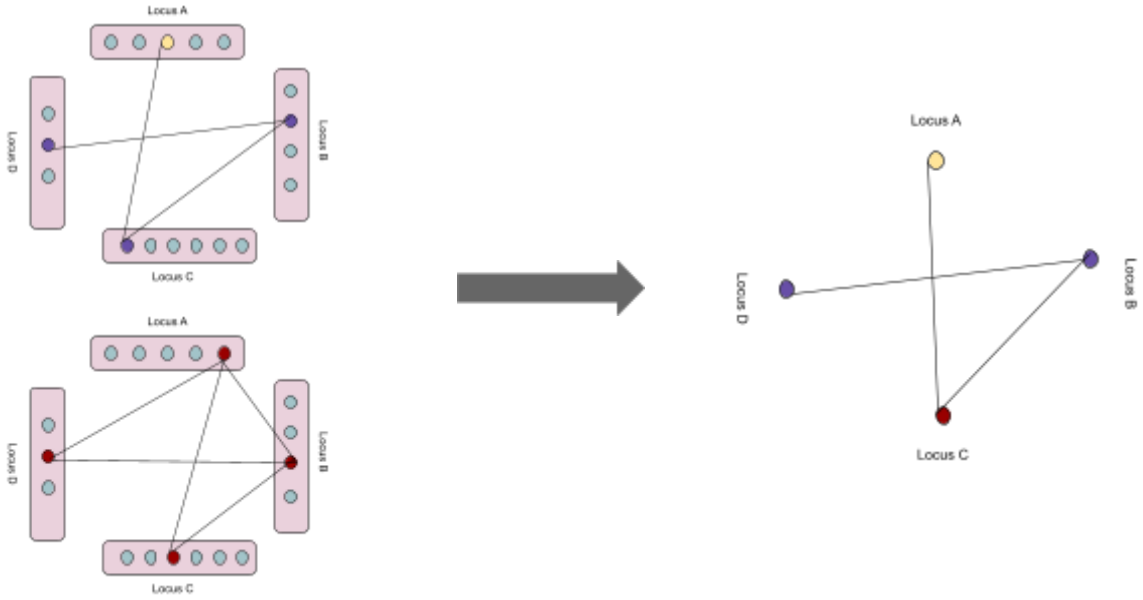


Figure3: Mating step with a pair of subnetworks sampled and a child subnetwork by selecting a gene from each loci from either parent (purple genes are from parent 1, yellow gene is mutated in the previous step, and red genes are from parent 2)

Optimization: The Genetic algorithm terminates when the mean percent difference between the mean population density from the current generation and the mean population density from the previous generation is less than 0.5%. The mating population from the previous generation is used as the final population of disease associated genes.

D) Generating a null case:

The initial population of 5000 random FA subnetworks were used to create the null case. This involved replacing each FA disease-associated gene with a non-FA gene within a similar range of edges. Quantile-based binning was used to categorize both FA and non-FA genes into 100 bins based on their edge numbers, with a minimum frequency of 0, a maximum frequency of 10101, and a bin width of 101. While using average edge weights per gene is more reasonable than the number of edges, this approach led to numerous bins containing 0 non-FA genes. This presented a challenge as replacing an FA gene within such bins is impractical, for instance, 13 bins out of 50 bins do not have non-FA genes.

The population of substituted non-FA subnetworks underwent optimization through the genetic algorithm. In the mutation step, non-FA genes were randomly assigned to 12 loci in each iteration of the null case. The final optimized generation, comprising 5000 subnetworks in each iteration, was used to compute the test statistic. The test statistic, represented by the average edge weight, was compared for each subnetwork of the optimized FA population against the test statistics of 1000 null case populations. The p-value is determined as the number of times the null case test statistic was as extreme or more extreme than that of the FA subnetwork. **Figure 4** illustrates the null case step.

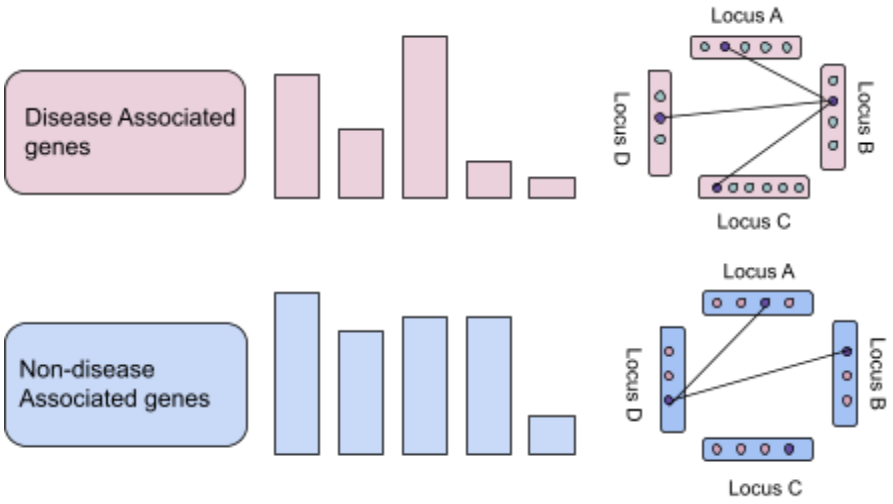
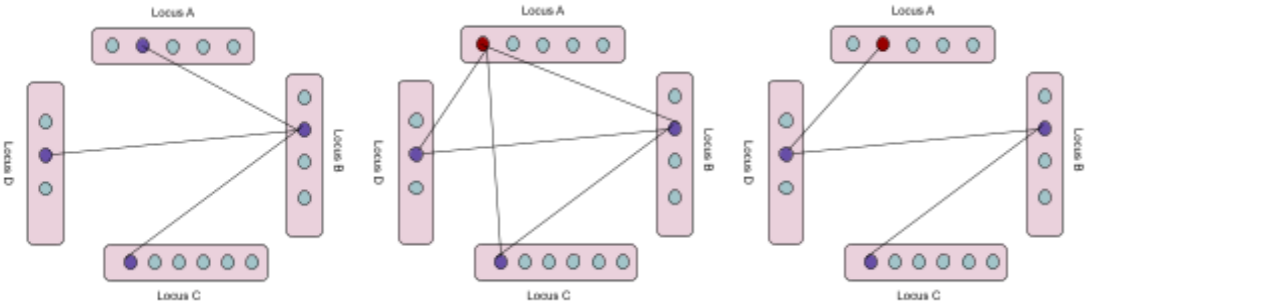


Figure 4: Generating a null case and subjecting it to the genetic algorithm (assigning random loci numbers to non-disease associated genes)

E) Gene Scoring:

In the final optimized population of FA subnetworks, wherein each subnetwork is generated by randomly selecting a representative gene from each locus, the first step computes the gene score equivalent to the number of edges in that specific subnetwork. Subsequently, each representative gene is replaced with every other gene within that specific locus and re-compute the gene score based on the edge counts. For instance, for loci L, comprising of Gi genes (g1, g2, g3, …., gn), the representative gene g* is replaced with g1 through gn and re-compute their associated edge counts. For the empty case, the edge counts were computed by completely eliminating the loci L. The final gene score for this subnetwork is the difference between the candidate gene scores and edge density of the empty locus case. Genes with higher gene scores imply high connectivity to the other loci in that subnetwork, while genes with lower scores imply lower connectivity to the other loci in that subnetwork.

The final gene scores are applied across all the 5000 subnetworks in the final population and an average gene score is computed for each gene. In cases where a gene at a locus is not found in the network file, an “NA” score is assigned, implying its absence in the network.



.....

Figure 5: Candidate gene scores

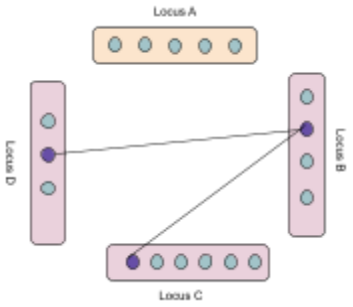


Figure 6: Empty locus case

RESULTS:

A) 5000 subnetworks:
There are 1024 edges that connect FA gene to FA gene. The 5000 random subnetwork generation uses a constraint of 0, i.e., zero or more edges in a subnetwork for it to be considered for the initial population.

B) Genetic Algorithm:
The initial population had a mean edge density of 1.006. The genetic algorithm ran for 13 iterations with the final iteration less than 0.5%. The optimized mean population density is 8.785. Table 2 shows mean percent population density, mean current population density, mean difference, and percentage difference. The optimized subnetwork had a maximum of 20 edges and a minimum of 2 edges in the subnetworks. Table 3 shows the top ten subnetworks arranged in descending order of edge numbers. Figure 7 shows distribution of the first generation.

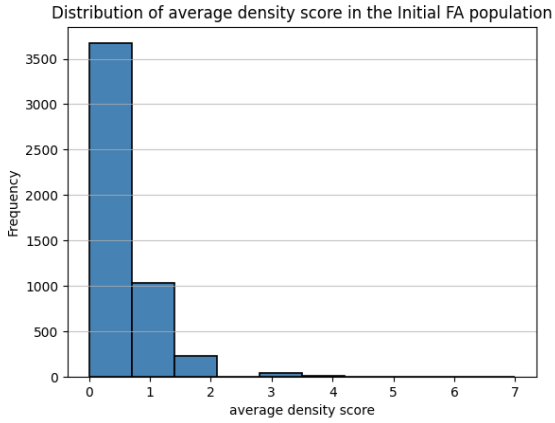
Generation	Mean parent population density	Mean current population density	Mean difference	% difference
1	1.006	0.889	-0.117	-11.630
2	0.889	1.945	1.055	118.673
3	1.945	3.88	1.937	99.598
4	3.88	5.843	1.960	50.480
5	5.84	7.146	1.303	22.311
6	7.146	7.952	0.805	11.268
7	7.952	8.342	0.390	4.910
8	8.342	8.482	0.140	1.681
9	8.482	8.590	0.107	1.265
10	8.590	8.662	0.071	0.837
11	8.662	8.751	0.089	1.036

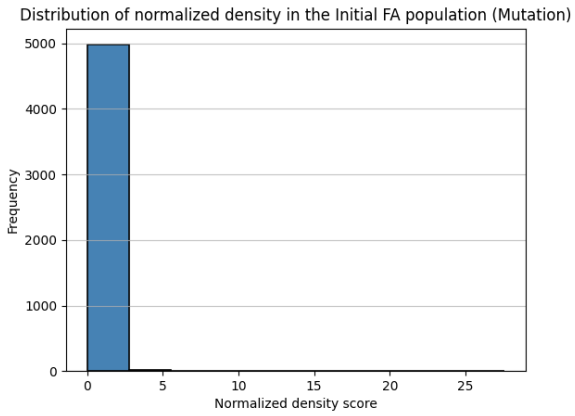
12	8.751	8.785	0.033	0.381

Table 2: Genetic Algorithm optimization (13 generations)

Subnetwork	Number of edges
1830	20
783	18
142	18
896	17
3980	17
3723	17
3002	17
734	16
669	16
58	16

Table 3: Top 10 subnetworks sorted by descending order of number of edges





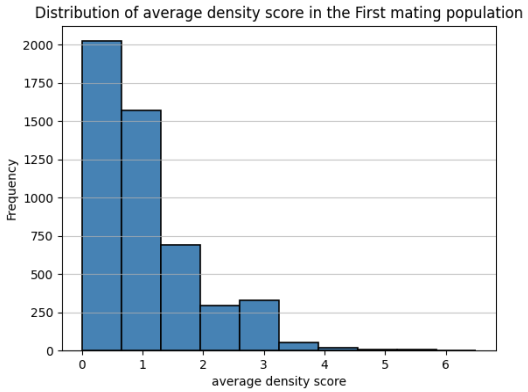


Figure 7: a) Average density of the initial FA population (top left) b) Normalized weights for the first FA population (top right) c) Average density score for the mating step in the first generation (bottom left)

C) Gene Scores: Figures 8 and 9 show the distribution of gene scores across loci.

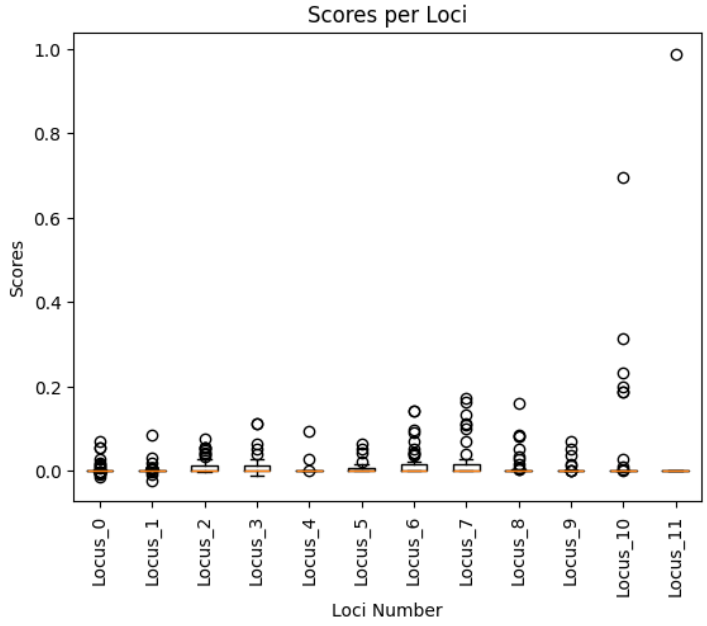


Figure 8: Distribution of gene scores across loci

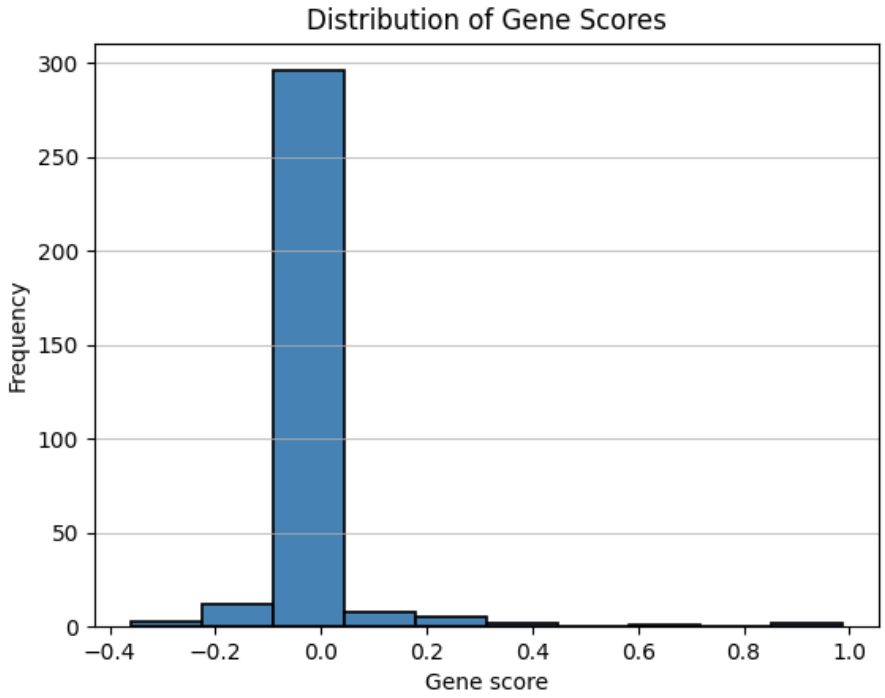


Figure 9: Distribution of gene scores

DISCUSSION:

Genes with high connectivity to other loci in the subnetwork have high scores, and similarly, genes with low connectivity to other loci in the subnetwork have low scores.

LIMITATIONS:

1. The null case does not use average edge weights for generating bins. While using average edge weights per gene is more reasonable than the number of edges, this approach led to numerous bins containing 0 non-FA genes. This presented a challenge as replacing an FA gene within such bins is impractical, for instance, 13 bins out of 50 bins do not have non-FA genes.
2. The optimization of nonFA networks takes several hours to complete.

REFERENCES:

PSEUDOCODE:

// Function to create dictionary of input file (loci_diction) and FA network (fa_diction), sets of all fa genes (fa_set) i.e., 584 genes; FA genes that are in the STRING file (string_set), i.e., 508 genes; and FA genes that are only connected to other FA genes (fa_string_set) i.e., 329 genes

FUNCTION data_to_diction(input_path,string_path):

    OPEN input_path as input:

```
            CREATE loci_diction as loci ID as key and genes as values
            CREATE fa_set with all the genes
            INITIALIZE fa_diction empty dictionary
            INITIALIZE fa_string_set empty set
            INITIALIZE string_set empty list
            INITIALIZE all_genes_set
        OPEN string_path as string_file:
            FOR edge in string_file:
                SORT columns of each row

                IF gene in first column and second column are in fa_set:
                    UPDATE genes to fa_string_set
                    IF genes in first column is in fa_diction.keys():
                        IF genes in two columns not in fa_diction inner keys:
                            UPDATE inner dictionary to the key
                    ELIF first column not in fa_diction.keys():
                        ADD inner dictionary to the key
                ELIF gene in first column or second column are in fa_set:
                    IF gene in first column is in fa_set:
                        APPEND gene in first column to string_set list
                    IF gene in second column is in fa_set:
                        APPEND gene in second column to string_set list

                ADD gene in first column to all_genes_set
                ADD gene in second column to all_genes_set
            ADD fa_set to all_genes_set
        CONVERT string_set list to a set
RETURN loci_diction, fa_set,fa_diction,fa_string_set,string_set,all_genes_set


// Function to create dictionary string file dictionary
FUNCTION string_to_diction(string_path):
        INITIALIZE string_diction empty dictionary
        INITIALIZE string_gene_set empty set
        WITH open(string_path,'r') as string_file:
            FOR edge in string_file:
                ASSIGN sorted(edge.strip().split('\t')) to genes
                UPDATE genes[:2] to string_gene_set

                IF first column gene IN string_diction:
                    IF ((second column gene NOT IN (string_diction[genes[0]].keys()) AND
string_diction[genes[0]].keys()):
                        ASSIGN edge weight to string_diction[genes[0]][genes[1]]
```

```
                ELIF first column genes NOT IN string_diction:
                    ASSIGN {genes[1]:genes[2]} to string_diction[genes[0]]
    RETURN string_gene_set, string_diction


// Function to create the first parent population with 5000 random subnetworks
FUNCTION first_parent_population(loci_diction,fa_diction,num_iterations=5000,min_edges=0):

    INITIALIZE subnetwork_diction empty dictionary
    INITIALIZE subnetwork_diction_edge_counts empty dictionary
    INITIALIZE first_parent_subnetwork empty dictionary
    INITIALIZE i to 0

    SET random seed to 123

    WHILE len(subnetwork.keys()) LESS THAN num_iterations:
        INITIALIZE iteration_fa_genes empty list
        INITIALIZE random_index_set empty list

        FOR key,value in loci_diction.items():
            ASSIGN random index between 0 and len(value)-1 to random_index
            APPEND value[random_index] to iteration_fa_genes
            APPEND random_index to random_index_set

        IF NOT ANY(iteration == values FOR values in first_parent_subnetwork.values()):
            ASSIGN iterations_fa_genes to first_parent_subnetwork['subnetwork_i']

        INITIALIZE temp_list empty list
        FOR outer_g, inner_d in fa_diction.items():
            FOR inner_g,edge_value in inner_d.items():
                IF outer_g AND inner_g IN first_parent_subnetwork['subnetwork_i']:
                    APPEND([outer_g,inner_g,edge_value]) TO temp_list

                IF len(temp_list) >= min_edges:
                    ASSIGN len(temp_list) to subnetwork_diction_edge_counts[subnetwork_i]
                    ASSIGN temp_list to subnetwork_diction['subnetwork_i']

        i+=1

    RETURN first_parent_subnetwork,subnetwork_diction


// Function to generate a random between a range excluding a specific number
FUNCTION generate_random_excluding(range_start,range_end,exclude_number):
```

```
    FOR num in range(range_start,range_end+1):
        IF num NOT EQUAL TO exclude_number:
            ASSIGN num to valid_numbers
    RETURN random.choice(valid_numbers)


// Function to mutate the parent subnetwork
FUNCTION ga_mutation(loci_diction,fa_diction,parent_subnetwork,num_iterations=5000):

    INITIALIZE subnetwork_diction_updated empty dictionary
    INITIALIZE subnetwork_mutated_weighted empty dictionary
    INITIALIZE subnetwork_normalized_weighted empty dictionary

    SET random seed to 123

    FOR subnetwork_index,subnet_genes_list IN parent_subnetwork.items():
        FOR locus,mut_genes in enumerate(subnet_genes_list):
            ASSIGN random integer between 1 and 20 to prob_5
            IF prob_5 EQUAL TO 1:
                ASSIGN index of mut_genes in locus to mutate_gene
                ASSIGN random number excluding mutate_gene to rand_replace
                ASSIGN rand_replace index gene to parent_subnetwork['subnetwork_i']


    FOR subnetwork_index,genes_list IN parent_subnetwork.items():
        ASSIGN temp_list_2 empty list

        FOR outer_gene,inner_diction IN fa_diction.items():
            FOR inner_gene,edge_value IN inner_diction.items():
                IF ALL(current_fa_gene IN genes_list FOR current_fa_gene IN
(outer_gene,inner_gene)):
                    IF(outer_gene IN genes_list) AND (inner_gene IN genes_list):
                        APPEND([outer_g,inner_g,edge_value]) TO temp_list_2

        ASSIGN temp_list_2 TO subnetwork_diction_updated
        ASSIGN sum of edge weights in temp_list_2 to subnetwork_mutated_weighted
        ASSIGN cubic sum of edge_weights in temp_list_2 to subnetwork_normalized_weighted

    ASSIGN sum of subnetwork_normalized_weighted.values() to sum_norm_weights_parent
    ASSIGN normalized subnetwork_mutated_weighted to normalized_weights_parent

    RETURN normalized_weights_parent, subnetwork_diction_updated, subnetwork_mutated_weighted,
parent_subnetwork
```

```
// Function for mating a mutated subnetwork
FUNCTION
ga_mating(fa_diction,normalized_weights_parent,subnetwork_mutated_weighted,num_iterations=5000,min_
edges=0):
    SET random seed to 123
    INITIALIZE mate_population empty dictionary
    INITIALIZE mate_weighted empty dictionary
    INITIALIZE iteration to 0
    INITIALIZE mate_parent_subnetwork empty dictionary

    WHILE len(mate_population.keys()) LESS THAN num_iterations:
        ASSIGN random indices to subnet_pair   //Based on weights or probabilities
        INITIALIZE sub_parent empty dictionary

        INITIALIZE mate_parent_subnetwork['subnetwork_i'] empty list

        FOR r in range(12):
            ASSIGN random integer 1 or 2 to pair_random
            APPEND parent_subnetwork[subnet_pair[pair_random-1][r]] to
mate_parent_subnetwork['subnetwork_i']

        INITIALIZE temp_list_3 empty list

        FOR outer_gene,inner_diction in fa_diction.items():
            FOR inner_gene,edge_value in inner_diction.items():
                IF outer_gene in mate_parent_subnetwork['subnetwork_i'] AND inner_gene in
mate_parent_subnetwork['subnetwork_i']:
                    APPEND outer_gene,inner_gene,edge_value to temp_list_3


        ASSIGN temp_list_3 to mate_population['subnetwork_i']
        ASSIGN sum of edge weights in temp_list_3 to mate_weighted

        INCREMENT iteration by 1

    ASSIGN sum of mate_weighted.values() to tot_mate_weighted

    ASSIGN average mate_weighted.values() to mean_current_weights
    ASSIGN average subnetwork_mutated_weighted.values() to mean_parent_weights

    ASSIGN difference between mean_current_weights and mean_parent_weights to mean_delta
```

```
    ASSIGN percentage difference to mean_delta_perc

    RETURN mate_population,mate_parent_subnetwork,mate_weighted,mean_delta,mean_delta_perc

//Function to a dictionary of gene frequencies
Function gene_frequency(string_diction,string_gene_set)
    INITIALIZE gene_freq dictionary
    FOR outer_key,inner_diction in string_diction.items()
        FOR inner_key,inner_value in inner_diction.items()
            IF outer_key is in string_gene_set
                INCREMENT gene_freq[outer_key] by 1
            IF inner_key is in string_gene_set
                INCREMENT gene_freq[inner_key] by 1
    RETURN gene_freq


//Function to create a nested dictionary of bin sizes for fa genes and non fa genes
Function bin_dictionary(gene_freq,num_bins,filter_bin_data)
    Bin_width = (maximum(gene_freq.values)-minimum(gene_freq.values))/num_bins

    INITIALIZE an empty dictionary bin_diction
    FOR i in range of num_bins
        Lower_bound = minimum(gene_freq.values)+i*bin_width
        Upper_bound=maximum(gene_freq.values)+i*bin_width
        INITIALIZE outer_keys in bin_diction
        INITIALIZE inner_keys in bin_diction
        FOR gene,freq in gene_freq.items
            IF freq >= lower_bound and freq <= upper_bound
                IF gene is in fa_set
                    APPEND gene to bin_diction fa_genes
                ELSE
                    APPEND gene to bin_diction non_fa_genes


// Function to match the bins for a given FA gene
Function match_fa_nonfa_by_bucket(nested_dict,disease_gene)
    FOR out_g, inner_dict in nested_dict.items()
        ASSIGN out_g to bin_cat
            IF inner_dict is dictionary
                CALL match_fa_nonfa_by_bucket(inner_gene,disease_gene)
                IF result is not NONE
                    RETURN out_g
```

```
//Function to generate nonFA subnetwork from FA
FUNCTION generate_nonfa_from_fa(optimized_FA_subnetwork,random_seed,bin_diction):
    INITIALIZE nonFA_parent empty dictionary
    INITIALIZE nonFA_parent_size empty dictionary
    INITIALIZE iteration to 0
    INITIALIZE nonFA_parent_genes empty dictionary
    INITIALIZE all_nonFA_null empty set

    SET random seed to random_seed
    FOR subnet_index,list_genes in optimized_FA_subnetwork.items():
        INITIALIZE iteration_nonfa_genes empty list
        INITIALIZE nonFA_parent_genes['subnetwork_'+str(iteration+1)] empty list

        FOR genes in list_genes:
            CALL match_fa_nonfa_by_bucket(bin_diction,genes) and ASSIGN to bin
            ASSIGN bin_diction[bin]['non_fa_genes'] to non_fa_values

            ASSIGN random integer to random_index_nonfa
            APPEND non_fa_values[random_index_nonfa] to iteration_nonfa_genes
            ADD non_fa_values[random_index_nonfa] to all_nonFA_null

        ASSIGN iteration_nonfa_genes to nonFA_parent_genes['subnetwork_'+str(iteration+1)]

        INITIALIZE temp_list_4 empty list

        FOR outer_gene,inner_dict in bin_diction.items():
            IF outer_gene IN iteration_nonfa_genes:
                FOR inner_gene,inner_edge IN inner_diction.items():
                    IF inner_gene IN iteration_nonfa_genes:
                        APPEND outer_gene, inner_gene, inner_edge to temp_list_4

        INCREMENT iteration by 1
        ASSIGN temp_list_4 to nonFA_parent['subnetwork_'+str(iteration+1)]
        ASSIGN len(temp_list_4) to nonFA_parent_size['subnetwork_'+str(iteration+1)]

    RETURN nonFA_parent,nonFA_parent_size,nonFA_parent_genes

// Function to create random loci for nonFA genes

FUNCTION generate_random_loci_NonFA(nonFA_parent_genes):

    ASSIGN {'Locus_{}'.format(l): [] for l in range(12)} to non_fa_loci
    INITIALIZE all_nonfa_subnet_genes empty set
```

```
FOR nonfa_sub,non_fa in nonFA_parent_genes.items():
        FOR nonfa_locus,non_fa_g in enumerate(non_fa):
                ADD non_fa_g to all_nonfa_subnet_genes
                ASSIGN non_fa.index(non_fa_g) to nonfa_index
                APPEND non_fa_g to non_fa_loci['Locus_{}'.format(nonfa_index)]

FOR i in all_genes_set:
        IF i not in all_nonfa_subnet_genes:
                ASSIGN a random integer between 0,11 to rand_nonfa_loci
RETURN non_fa_loci
```