I will begin explaining the first part of the main script (where the .txt file from RegulonDB is processed), and then I will explain the functions created in '*NetworkFunctions.R'*, before going back to the rest of the main script.

To represent the networks, I have used a square adjacency matrix representation with named rows and columns, where the source of an interaction is the row and the target of an interaction is the column. So, the coordinates (row, column) represent an interaction going from row (acting as transcription factor) to column (acting as gene). Because of this, all of the algorithms have been designed to work with an adjacency matrix with these characteristics.

Main script (Network.R):

To begin with, I read the .txt file from RegulonDB, containing *E. coli* K-12 regulatory network (transcription factors versus genes), in an appropriate manner. I named the columns as "TF.id", "TF.name", "Gene.id", "Gene.name", "Effect", "Evidence" and "Evidence.type", because the file had no header.

To identify the genes and transcription factors (TF), there were two alternatives: using the identifiers (id) and using the names (name). I originally tried to use the ids, however the ID for a transcription factor in protein form is different from the ID of the gene that encodes that transcription factor. Therefore, I identified each TF using their names, turning them into lowercase because the TF.name column was in uppercase. I refer to these lowercase names as Source (in the case of the TF) and Target (in the case of the gene).

I then extracted all the Source names, removing duplicates, and did the same with the Target names. Because I am using an adjacency matrix representation, I wanted it to be a square matrix, so I joined the Source and Target names, removing duplicates again (those TF that were also in the gene column because they were the target of an interaction). The complete set of names was stored in all_names. I created an empty adjacency matrix (full of 0's) with dimensions all_names x all_names.

It is important to note that the regulatory network from RegulonDB contains, for each interaction, an evidence type, which can be null, Weak, Strong or Confirmed. According to RegulonDB [2]:

- Weak evidence is a single evidence with possible alternative explanations or false positives, as well as computational predictions.
- Strong evidence is a single evidence with a direct physical interaction or solid genetic evidence with low probability for alternative explanations.
- Confirmed evidence is given when the interaction is supported by at least two different strong evidences with mutually exclusive false positives.

Therefore, I distinguish in my analysis between four different cases:

- One where I ignore the evidence type, therefore including null, weak, strong and confirmed evidences (I refer to it with the prefix 'all').
- One where I remove null evidence, and only keep interactions with some evidence (weak, strong and confirmed, referring to it as 'wsc').
- One where I only keep strong and confirmed evidences ('sc').
- And one last one where I only keep confirmed evidences ('conf'). However, in this particular network, there are no confirmed evidences, so this network is empty.

Depending on the case, I create four hashes, where every type of evidence rejected corresponds to a 0 (no interaction), and every type included corresponds to a 1 (interaction).

In order to continue, I need to explain now the functions I created in '*NetworkFunctions.R'*.

Functions (NetworkFunctions.R):

Manipulation of an adjacency matrix:

- ***Populate:***

Arguments:

- adj_matrix: an empty adjacency matrix (full of 0's).
- hash: a list that, for each evidence type, contains the value that will be inserted in the matrix when an interaction has that evidence type.
- network_df: the dataframe created from the .txt file from RegulonDB.

This function's purpose is to populate an empty adjacency matrix (full of 0's). For each interaction in the network dataframe, it introduces a 1 in the corresponding position of the adjacency matrix if the evidence type of that interaction is accepted. If in the hash we have accepted that evidence type, it will input a 1, and else, it will not.

- ***Simplify_matrix:***

Arguments:

- adj_matrix: an adjacency matrix.

This function's purpose is to simplify an adjacency matrix, removing the ids (row and column names) that do not take part in any interaction, both as a source and as a target. In the adjacency matrix, those are names for which their row and their column are both empty (they have not got any 1, only 0's).

To do that, it searches for the empty rows and for the empty columns, finds the intersection of both sets, and then removes the rows and columns if they have the same name and are both empty.

Autoregulation motifs:

- ***Find_autoreg:***

Arguments:

- adj_matrix: an adjacency matrix.

This function's purpose is to return a vector with the names of the rows/columns that are involved in an autoregulation motif. To do that, each row name of the adjacency matrix gets added to the vector if the diagonal (where that element would interact with itself) has a 1.

- ***Count_autoreg:***

Arguments:

- autoreg: a vector of names, result of the previous function.
- net_df: the dataframe created from the .txt file from RegulonDB.

This function's purpose is to classify the autoregulation motifs found by find_autoreg according to their effect type: positive for induction ('+'), negative for repression ('-'); and other, for the cases where, at the same time, an element induces itself and represses itself ('+' and '-'), or when there is an effect different from '+' or '-'.

To do that, for each name in autoreg, it searches for the effect of the autoregulatory interaction in the network dataframe ("Effect" column) and classifies it accordingly. In the end, it returns a list with the number of total autoregulation motifs, and the number of positive, negative and other autoregulation motifs.

Feed Forward Loop motifs:

- ***Find_feed_forw:***

Arguments:

- adj_matrix: an adjacency matrix.

This function's purpose is to find all of the feed forward loop (FFL) motifs in the adjacency matrix. From all of the possible triangular motifs in which three elements (A, B and C) participate, it only retrieves the FFLs: this means that the interactions A->B, B->C and A->C must exist, but there should not be any other interaction between any two elements.

My implementation ignores autoregulation when looking for feed forward loops, so a FFL where a member autoregulates itself is included. However, a motif formed by the previous interactions (A->B, B->C and A->C) and one or more of C->A, C->B or B->A will be excluded.

To do this, the general steps followed are:

- For each row name in the adjacency matrix (start element, or A in the previous example), it retrieves the column names that are targets of A, and removes itself from the target list in case of autoregulation.
- In a feed forward loop, A interacts with both B and C, so the function discards A and goes to the next row name if A has less than two targets.
- For each target of A (which is a possible B), it retrieves the column names that are targets of B.
- It discards this possible loop if B also interacts with A, and removes itself from the target list in case of autoregulation.
- For each target of B (possible C), it discards this possible loop if C interacts with B or if C interacts with A.
- If at the end there are still any targets of B, it includes in a results dataframe the FFL Start = A, Mid = B and End = C for each A,B,C that satisfy all of the conditions.

To count the different types of FFL, the next function is used.

- ***Count_feed_forw:***

Arguments:

- feed_forw: a dataframe in which each row is an FFL, containing the Start, Mid and End names, result of the previous function.
- net_df: the dataframe created from the .txt file from RegulonDB.

This function's purpose is to classify the FFL motifs found by find_feed_forw according to their type. FFLs can be coherent or incoherent, and each of them can have four subtypes (1-4). If we define the three interactions of the FFL as: SM (Start -> Mid), ME (Mid -> End), SE (Start -> End), then the coherent FFLs are:

- Coherent type 1 (C1): SM +, ME +, SE +
- Coherent type 2 (C2): SM -, ME +, SE –
- Coherent type 3 (C3): SM +, ME -, SE –
- Coherent type 4 (C4): SM -, ME -, SE +

And the incoherent FFLs are:

- Incoherent type 1 (I1): SM +, ME -, SE +
- Incoherent type 1 (I2): SM -, ME -, SE –
- Incoherent type 1 (I3): SM +, ME +, SE –
- Incoherent type 1 (I4): SM -, ME +, SE +

To do that, for each interaction, it searches for its effect in the network dataframe ("Effect" column) and classifies it accordingly. In the same way as count_autoreg, I am including in other the FFLs in which at least one interaction is different from positive or negative, or if it has multiple effects at the same time.

In the end, it returns a list with the number of coherent, C1, C2, C3, C4, incoherent, I1, I2, I3, I4, other and total FFL motifs, where coherent is the sum of C1-4, incoherent is the sum of I1-4, and total is the sum of coherent, incoherent and other.

Length of the longest cascade:

The following functions are used to find the length of the longest cascade (in number of interactions): if the longest cascade is A->B->C, its length will be 2.

Finding the longest cascade is the hardest and most complex part of this exercise. It is equivalent to the problem of finding the longest path in a graph. This problem is simple if the graph is guaranteed to be acyclic (which means that it does not contain any cycle) [6, 7]. However, natural networks and especially random networks are not guaranteed to be acyclic.

If the graph can be cyclic, the problem is no longer simple, and we have to check every single possible path in order to choose the longest one. This is computationally expensive, and scales with the size of the graph. To get an exact solution, it is unavoidable to check every single path, however there may be some optimizations that could be done [6, 7]. For my implementation, I used the brute force algorithm without any optimization.

This is composed by two functions: a recursive function, path_fn, and the initial function, find_longest_path.

- ***Path_fn:***

Arguments:

- node_i: the current node.
- input_list: the list with the current parameters, contains:
  - adj_matrix: the adjacency matrix.
  - visited: a vector of all of the nodes that contains if the node has been visited or not.
  - current_path_length: the length of the current path, from the start to the current node.
  - best_path_length: the highest value of current_path_length.

This is the recursive function, adapted from [8]. For the current node (i), we have information about which nodes have been visited, the length of the current path, and the length of the longest path that has been found until now.

The function sets the current node (i) as visited and gets the child nodes (the targets of the current node) that have not been visited yet. For each non-visited child node (j), it increases the current path length by 1, checks if it is longest that the best_path_length found until now (if so, it replaces it), and calls itself, using the child node (j) as the new current node. After that, all of the recursive calls have been resolved, so it decreases the current path length by 1, because we are back to the original current node (i).

When there are no more child nodes for the current node, it marks the current node as not visited, and that particular call of the function finishes. When this happens, if the function was called by itself, we are back to the previous node. If the function was called by find_longest_path (explained below), we are back at the start, and the function gets called again for the next possible starting node in the network until there are no more starting nodes left.

- ***Find_longest_path:***

Arguments:

- adj_matrix: the adjacency matrix.

This function's purpose is to call the previous function for every possible starting node in the adjacency matrix. It keeps track of the best path length found by every call of path_fn that has happened until that moment. At the end, when every node has been a starting node and all of the possible paths have been checked, it returns the best path length.

Random network simulation and analysis:

- ***Simulate_and_analyze_networks:***

Arguments:

- num_sim: the number of times a random network will be simulated and analyzed.
- adj_matrix: the adjacency matrix of the real network.
- pos_freq: the frequency of '+' effects in the real network.
- neg_freq: the frequency of '-' effects in the real network.
- other_freq: the frequency of other effects in the real network, as defined before.
- plot_name: the name of the png file that will be created if the graph is plotted.
- plot_graph: Boolean value, if True, it plots a graph of the first random network generated, using the igraph library.

This function's purpose is to simulate a number of random networks with the same characteristics as the real network passed in the form of an adjacency matrix, and then analyze the abundance of each motif and the length of the longest cascade.

To do this, it first gets the characteristics of the real network: its size (the number of nodes) and its number of edges (the number of interactions, or 1's in the adjacency matrix). Other characteristics such as the frequency of positive interactions, negative interactions and other interactions are passed as arguments of the function (pos_freq, neg_freq, and other_freq), because they need the network dataframe and cannot be calculated using just the adjacency matrix.

Then, for each simulation, it uses this information to simulate a random network. It first creates an empty adjacency matrix (full of 0's) of the same size as the real adjacency matrix. After that, it creates random row and column coordinates: it samples with replacement the numbers from 1 to the size of the network, as many times as there are interactions. This is done for the row coordinates, and again for the column coordinates. They then get combined, so that each pair of row, column coordinates corresponds to a position in the matrix, and an interaction (a 1) is added to that position. At this point, we have a random adjacency matrix with the same size and roughly the same number of interactions (because there could be repeated coordinates) as the original adjacency matrix.

Then, using the frequencies of each effect (positive, negative, or other), for each interaction in the random network, it samples with replacement '+', '-' or 'other' as many times as interactions are in the network, in order to simulate the effects of each interaction. It creates a dataframe with each interaction in the random network and the corresponding randomly generated effect. This way, not only am I simulating the interactions, but also their effects, with the same frequencies as the ones in the real network.

After that, if plot_graph is set to TRUE, and we are in the first simulation, it creates a directed igraph graph from the random adjacency matrix and plots it into a png file.

Then, the analysis of the random network takes place. Using the previous functions, it finds the number of autoregulation motifs (distinguishing by type), the number of FFLs (again, by type) and the length of the longest cascade. It appends those results into the results dataframes that will be returned at the end of the last simulation.

Function tests (NetworkFunctionsTest.R):

Briefly, this script contains a testing function, which tests the complicated functions (find_autoreg, find_feed_forw and find_longest_path). It compares the expected results with the actual results, and prints error messages if they differ.

Then, there are some tests, with different adjacency matrixes and combinations of motifs, in order to test different cases such as empty matrix, full matrix, autoregulations, FFL with and without autoregulations, triangular motifs with more interactions than the ones in FFL (which should not be classified as an FFL), cyclic graphs, non-cyclic graphs, and so on. Every test passed correctly. I also included at the end a test that was intentionally wrong to check the failure, which also worked as expected.

Main script continuation (Network.R):

Now that the functions have been explained, I will continue with the main script.

After the hashes where every type of evidence rejected corresponds to a 0 (no interaction), and every type included corresponds to a 1 (interaction) were created, I populated four adjacency matrixes, one for each hash. After populating them, I simplified them to remove the genes that are neither source nor target of an interaction. Because there are no 'Confirmed' evidences in the network, there are three different cases in the end:

- tf_gene_all_adj: the matrix disregarding evidence (null, weak and strong).
- tf_gene_wsc_adj: the matrix for only weak and strong evidence interactions.
- tf_gene_sc_adj: the matrix for only strong evidence interactions.

In order to visualize them in a graph, I converted each adjacency matrix to an igraph graph. I created a function that, for each node in the graph, checks if the node is part of the Source names (a transcription factor), to be able to color the TFs in purple and the non-TFs in orange.

I ended up not representing the matrix with all the evidences (null, weak and strong), because there were so many nodes and edges that I could not manage to get a clean representation.

Then, I applied the algorithms from *'NetworkFunctions.R'*. I repeated the same process for the three adjacency matrixes ('all', 'wsc' and 'sc'), with little differences.

I calculated the size and number of interactions of each network. I then got all of the effects from the network dataframe for the appropriate evidence types, and calculated the frequencies of positive, negative and other effects. Then I calculated the number and types of autoregulation motifs, the number and types of FFL motifs and the length of the longest path.

Afterwards, I called simulate_and_analyze_network to create the random networks and analyze their characteristics. In the first case, when considering 'all' of the evidences, the computations were so slow that I only did 1 simulation. In the other cases ('wsc' and 'sc'), I did 100 simulations for each.

At the end, I stored every result, from the real and from the random networks, into .csv files. I then represented the data from these files using the *'ResultsPlotting.R'* script, and I analyzed the statistical significance using '*StatisticalAnalysis.R'*.

Statistical analysis (StatisticalAnalysis.R):

I could not do a proper statistical analysis with all the evidence, because I only did 1 simulated random network.

For the others, I analyzed the statistical significance for the autoregulation motifs and the FFLs, using two metrics. The first is the Z-score, a significance metric which, if greater than 2.0, regards a motif as statistically significant (the probability of that motif to appear an equal or greater number of times in a random network than in the real network is less than 0.01). The second is the abundance, similar to the z-score, and ranges between -1 (a motif is under-represented in the real network compared to the random networks) and +1 (a motif is over-represented in the real network compared to the random networks) [1].