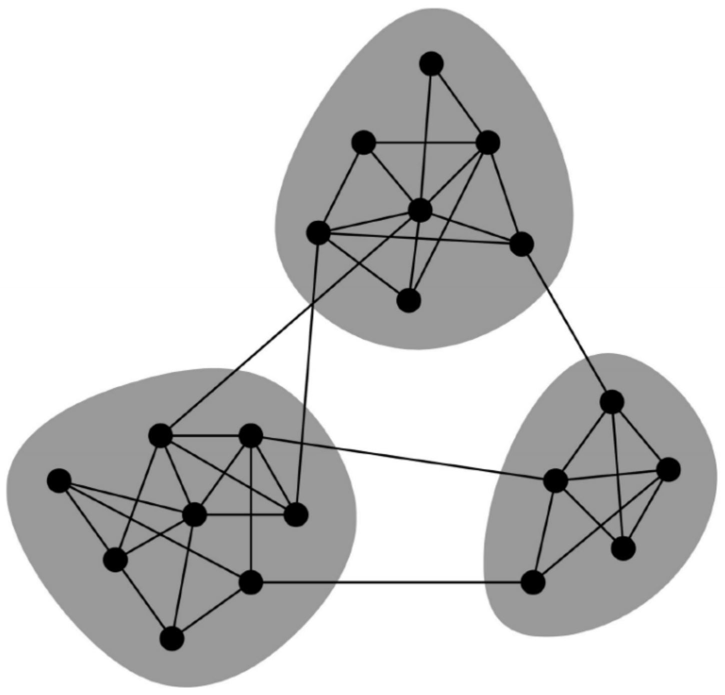
Community Structure in Networks
Final Project

DUE DATE: 20/9/2020 23:00

SUBMISSION IN PAIRS



1 Introduction

In complex networks, a network is said to have **community structure** if the nodes of the network can be grouped into groups of nodes with dense connections internally, and sparser connections between groups.

In this project you will implement an algorithm for detecting community structures (or **clusters**) in a network. The ability to detect such groups is of significant importance. For example, partitioning a protein-protein interaction network into clusters can provide a modular view of the network, with different groups of nodes performing different functions within the cell.

This document will first describe the mathematical basis for your algorithm, and then describe the code requirements and implementation. There will not be many implementation details, and no tester is provided – implementation and correctness are up to you. You will be graded for code modularity, design, readability, and performance.

2 Division Algorithm

This section describes the mathematical basis for your project. It does not describe your code, but rather the mathematical foundations your code will be based upon, as well as the algorithms you should implement and use.

We represent a network by a graph $G = (V, E)$, and let A be the adjacency matrix of G:

$$A = \begin{cases} 1 & i \text{ and } j \text{ are neighbors, } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We assume $\forall i : A_{ii} = 0$.

A given group of vertices in a network is considered a *community* if the number of edges within the group is significantly more than expected (by chance). We define the *modularity* of a group as the number of edges within the group minus the expected number of edges in a random graph with the same degrees.

Formally, let k_i be the degree of vertex i in G, and let $M = \sum_i k_i$. The expected number of edges between i and j is $\frac{k_i k_j}{M}$. Let S be a sub-group of vertices, $S \subseteq V$, we define the *modularity* of S as:

$$Q_S = \sum_{i,j \in S} \left(A_{ij} - \frac{k_i k_j}{M} \right)$$

Our goal in this project is thus to find a division that maximizes the modularity, that is, a division of the graph into groups such that the modularity Q, the sum of all group modularities, is close to maximal.

2.1 Dividing Into Two

Let us initially focus on a good division of the network into just *two* communities.

For a particular division of G into two groups, for each vertex i let $s_i = +1$ if it belongs to the first group and $s_i = -1$ if it belongs to the second group. Observe that:

$$\frac{1}{2}(s_i s_j + 1) = \begin{cases} 1 & i \text{ and } j \text{ are in the same group, } s_i = s_j \\ 0 & \textit{otherwise} \end{cases}$$

Thus we can express the modularity as:

$$Q = \frac{1}{2} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{M})(s_i s_j + 1)$$

Since $\sum_{i,j} A_{ij} = \sum_i k_i = M$:

$$Q = \frac{1}{2} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{M}) s_i s_j$$

Let $s = \{s_1, \dots, s_n\}$ be a column vector, and let B be the *modularity matrix*, $B_{ij} = A_{ij} - \frac{k_i k_j}{M}$. We can rewrite Q as:

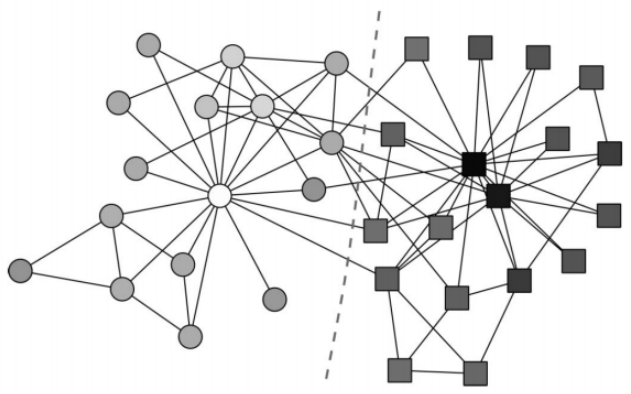
$$Q = \frac{1}{2} s^T B s$$

Our goal is to find a vector s, a division of G, that maximizes the modularity Q.

Note that B is a symmetric matrix, thus it is diagonalizable with n eigenvalues. In each row and column its elements sum to zero, thus B has the eigenvector $(1, \dots, 1)$ with a corresponding eigenvalue 0.

Let $\beta_1, \dots, \beta_n$ be the eigenvalues of B such that $\beta_1 \geq \dots \geq \beta_n$, and let $u_1, \dots, u_n$ be the corresponding normalized eigenvectors. For all vectors with a fixed norma c, the vector cu_1 achieves the maximum value for Q (which is $c^2\beta_1$). However, recall we look for a vector s whose elements are ± 1. Thus, we need to make s as close to being parallel to u_1 as possible, which is equivalent to maximizing the dot product $u_1^T \cdot s$. The maximum is achieve by setting $s_i = +1$ if the element i of u_1 is positive and $s_i = -1$ otherwise.

In other words, we need to obtain the *leading* eigenvector of B, i.e., the eigenvector of B with the largest eigenvalue. Then, vertices whose corresponding elements are positive go in one group, and the rest of the vertices go in the other group.



It is possible for there to be no positive eigenvalues of B. In this case, the leading eigenvector is the vector $(1, \dots, 1)$, corresponding to all vertices in a single group together. This is the correct result: in this case, the algorithm is telling us that there is no division of the network that results in a positive modularity. The modularity of an undivided network is zero, which is the best that can be achieved.

Additionally, it could happen that although the leading eigenvalue is positive, $s^T Bs$ is non-positive, i.e., the division represented by s has a worse modularity than a trivial division into one group.

The flow of the algorithm is as follows:

Algorithm 1 Divide a network into two modules

```
1. Compute leading eigenpair $u_1$ and $\beta_1$ of the modularity matrix $B$
2. if ($\beta_1 \leq 0$):
    2.1. The network is indivisible
3. Compute $s = \{s_1, \ldots, s_n\}$ where $s_i \in \{+1, -1\}$, according to $u_1$
4. if ($s^T Bs \leq 0$):
    4.1. The network is indivisible
5. return a division into two groups according to $s$
```

2.2 Dividing Into Modularity Groups

In the previous section we described a method that divides a network into *two* groups. Many networks, however, contain more than two communities, thus we would like to find good divisions of networks into a larger number of groups.

The standard approach to this problem, which we will adopt, is repeated division into two. We divide the network into two parts, then divide each of those parts, and so forth.

To divide a group, we cannot use the same method since it must take into consideration the entire graph, instead we need to look at the additional change ΔQ to the modularity. Suppose we divide a group g of size n_g in two, the change depends on pairs of elements in g. Prior to the division, the contribution of pairs in g to Q was $\sum_{i,j \in g} B_{ij}$. Let s be a vector of size n_g with ± 1 elements representing a division of g into two groups, and define δ_{ij} as follows:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

It follows that:

$$\begin{aligned} \Delta Q &= \frac{1}{2} \sum_{i,j \in g} B_{ij}(s_i s_j + 1) - \sum_{i,j \in g} B_{ij} \\ &= \frac{1}{2} \sum_{i,j \in g} (B_{ij} s_i s_j - \sum_{i,j \in g} B_{ij}) \\ &= \frac{1}{2} \sum_{i,j \in g} (B_{ij} - \delta_{ij} \sum_{k \in g} B_{ik}) s_i s_j \end{aligned}$$

Let $B[g]$ be the symmetric $n_g \times n_g$ sub-matrix of B corresponding to the rows and columns of g. Let $f_i^g = \sum_{j \in g} B[g]_{ij}$. Let $\widehat{B}[g]$ be a symmetric $n_g \times n_g$ matrix with elements:

$$\widehat{B}[g]_{ij} = B[g]_{ij} - \delta_{ij} f_i^g$$

We can rewrite ΔQ as:

$$\Delta Q = \frac{1}{2} s^T \widehat{B}[g] s$$

This equation has the same form as before, thus we can apply the same algorithm to maximize ΔQ, according to the leading eigenpair of $\widehat{B}[g]$.

Thus, Algorithm 1 can be updated to any group g as follows:

Algorithm 2 Divide a group into two

```
1. Compute leading eigenpair $u_1$ and $\beta_1$ of the modularity matrix $\widehat{B}[g]$
2. **if** ($\beta_1 \leq 0$):
    2.1. The group $g$ is indivisible
3. Compute $s = \{s_1, \ldots, s_n\}$ where $s_i \in \{+1, -1\}$, according to $u_1$
4. **if** ($s^T \widehat{B}[g] s \leq 0$):
    4.1. The group $g$ is indivisible
5. **return** a division of $g$ into two groups according to $s$
```

Note that if g is the entire graph, then $\forall i : f_i^g = 0$ and thus $\widehat{B}[g] = B$, allowing us to use Algorithm 2 instead of Algorithm 1 in all cases.

Finally, we can use Algorithm 2 to repeatedly divide the module into groups, as described in the following algorithm:

Algorithm 3 Divide a network into modularity groups

```
1. Start with a trivial division into one group: $P \leftarrow \{\{1, \ldots, n\}\}$
2. Start with an empty output set of groups: $O \leftarrow \{\}$
3. Repeat until $P$ is empty:
    3.1. Remove a group $g$ from $P$
    3.2. Divide $g$ into $g_1, g_2$ with Algorithm 2
    3.3. **if** either $g_1$ or $g_2$ is of size 0:
        3.3.1. Add $g$ to $O$
    3.4. **else**:
        3.4.1. Add to $O$: any group ($g_1$ and/or $g_2$) of size 1
        3.4.2. Add to $P$: any group ($g_1$ and/or $g_2$) larger than 1
4. Output the division given by $O$
```

2.3 Modularity Maximization

In this section we describe a method to further optimize a division of into two groups. You should supplement this method into the previous algorithms.

Suppose $\{g_1, g_2\}$ is an initial division of g into two groups. Our goal is to optimize this division further. To achieve this, find a vertex v that, when moved the other group, will give the biggest increase in modularity of the complete network, or the smallest decrease if no increase is possible, and move it to the other group.

Repeat this process with the constraint that each vertex may only be moved once, until all vertices have been moved. Once done, from all the states of division into two groups during the operation, find the state that has the maximal modularity, and start again from this state.

We repeat the entire process iteratively until no further improvement is found, i.e., until the maximal state is the current one.

3 Leading Eigenpair

In order to find the leading eigenpair of a diagonalizable matrix we will use power iteration.

Power iteration was described in the previous assignments, and uses matrix-vector multiplications to find the dominant eigenvector, the eigenvector with the largest *absolute* eigenvalue.

Let k be the number of iterations until we stop, such that b_k is our approximation of the dominant eigenvector, an approximation of the corresponding dominant eigenvalue can be found as follows:

$$\lambda_1 = \frac{b_k \cdot Ab_k}{b_k \cdot b_k}$$

3.1 Matrix Shifting

Power iteration approximates the *dominant* eigenpair; however, in the division algorithm we search for the *leading* eigenpair. That is, we search for the eigenpair with the largest eigenvalue, while power iteration provides us with the eigenpair with the largest *absolute* eigenvalue.

✓ In order to correct this, we will use *matrix shifting* to ensure the eigenpair we find is the leading eigenpair.

Let C be a matrix with the eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$, and let the 1-norm of C be the sum of its largest column: $||C||_1 = \max_j(\sum\limits_i |C_{ij}|)$.

Let $C' = ||C||_1 \cdot I + C$, then C' has the same eigenvectors as C and the eigenvalues of C' are $\{\lambda_1 + ||C||_1, \ldots, \lambda_n + ||C||_1\}$. Thus, the dominant eigenpair of C' is $(\lambda_1 + ||C||_1, v_1)$, and the leading eigenpair of C is (λ_1, v_1).

4 Implementation

Your program should be named `cluster`.

It receives two command-line arguments. The 1st is an input filename, and the 2nd is an output filename. The input of your program is a network (a graph), and the output is a list of groups (the division).

Use the algorithms described in order to divide the network into modularity groups. Optimize your code with the tools you have learned. For example, can the modularity matrix B (and $\widehat{B}[g]$) be represented as a sparse matrix?

4.1 File Format

The input and output files are both binary files consisting *only* of integers.

Recall that binary files are not "human-readable" and cannot be edited in a text editor. There are no lines, whitespace, or other separators, rather these files consist only of a stream of bytes. The size of each value in the file is 4, the size of `int` on most machines, including Nova. This can be determined in code by `sizeof(int)`.

Below, we describe the input and output file formats. Sample files are provided to you. Use these samples to check your file formats, not the correctness of your code!

4.1.1 Input File

The first value represents the number of nodes in the network, $n = |V|$.

The second value represents the number of edges of the first node, i.e., k_1. It is followed by the k_1 indices of its neighbors, in increasing order.

The next value is k_2, followed by the k_2 indices of the neighbors of the second node, then k_3 and its k_3 neighbors, and so on until node n.

4.1.2 Output File

The first value represents the number of groups in the division.

The second value represents the number of nodes in the first group, followed by the indices of the nodes in the group, in increasing order.

The next value is the number of nodes in the second group, followed by the indices of the nodes in group, then the number of nodes and indices of nodes in the third group, and so on until the last group.

4.2 Checking Zeros

As in previous assignments, due to numerical representations of real values, we will use $\epsilon = 0.0001$ as zero. For example, if we want to check if a value is positive, we will use the following:

```
#define IS_POSITIVE(X) ((X) > 0.00001)
```

4.3 Performance Considerations

There can be various bottlenecks in your code, such as the calculations in the power iteration or repeatedly finding a vertex with minimal ΔQ for the division optimization.

You are not given a tester with time measurements, but are instead expected to analyze your code and optimize it by finding these bottlenecks. Store values to avoid repeated calculations, try to access contingent memory, minimize memory allocations, avoid repeated array indexing, etc.

You may freely compare your performance with other students, so long as you do not share the specific bottlenecks you encountered or enhancements you performed.

4.4 Error Handling

Your code should handle all possible errors that may occur (e.g., memory allocation, file access). File names can be either relative or absolute and can be invalid (the file or path might not exist or could not be opened). Given that the input file exists and readable, you may assume it is in the correct format.

Since you are dealing with numeric procedures, numeric problems are to be expected. You should protect your code from problems such as division by zero, infinite loops, etc. You may exit with an error message on each case which leads to it.

Unlike the previous assignments, you may **not** use `assert`. On any error, print a descriptive error message and `exit` or `return` from `main` with a value **other than 0**. On a successful execution, your `main` function should `return 0`.

4.5 Coding and Compilation

Your code should be gracefully partitioned into files (modules) and functions. The design of the program, including interfaces, function declarations, and partition into modules, is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

Header files should contain a main comment that describes the module, its purpose, and interface. Source files should be commented at critical points in the code and at function declarations. Please avoid long lines of code, and shorten lines that exceed 80 characters. Functions should not exceed roughly 60 lines of code.

In your implementation, pay careful attention to the use of constant values and proper use of memory. Do not forget to free any memory you allocated. You should especially aim to allocate only necessary memory and free objects (memory and files) as soon as possible.

You should create your own makefile, which compiles all relevant parts of your code and creates an executable named `cluster`. Your code should pass the compilation test with no errors or warnings, which will be performed by running the `make all` command in a UNIX terminal on `nova`. Your code should compile and run on `nova`. You should also write a `make clean` command, and construct the makefile as seen in class and with all relevant flags.

4.6 Submission

The project will be submitted by moodle. The grading may also involve testing of the code in a frontal meeting, to be set as necessary. Both students should be familiar with a significant part of the code.

The zipped file must contain your `makefile`, and all of your source and header files. All files must be in the root, **no folders**.

MAC users: create (or check) your zip files under Linux, as otherwise hidden auxiliary files are automatically added and will reduce your grade.

Both students should submit a zip file named `id1_id2_project.zip`, replacing `id1` and `id2` with the actual 9-digit ID of both partners.

Submission in pairs is mandatory! If you wish to submit alone, contact the TA with a valid reason to get approval.

4.7 Remarks

- For questions regarding the project, please post in the forum.
- The forum is an integral part of the project. Subscribe to it and follow its messages.
- You may freely discuss and compare your outputs with fellow students; however, note that some inputs may produce different outputs depending on your code and random factors. It is strictly forbidden to discuss any implementation details such as modules, code design, etc. Similarities will result in failing the course, and no resubmission is allowed.
- **The submission date and time is final!** Late submissions are not allowed and will not be approved. There will be no extensions under any circumstances, except miluim or hospitalization. Sick notes do not allow for extensions. You are expected to complete and submit the project at least two weeks ahead of the submission date, and failure to do so is your responsibility. Any submission beyond the deadline will result in a 10 points deduction per day (or part of it), until the submission closes. Check moodle for the exact deadline and closing time of the submission.
- Borrowing from others' work is unacceptable and bears severe consequences.

GOOD LUCK