Università della Svizzera italiana | Institute of Computing CI

Numerical Computing **2024**

Student: Lorenzo Galli **Discussed with:** Prof. Edoardo Vecchi, Gianmarco De Vita

Solution for Project 2

Due date: Wednesday, 23 October 2024, 11:59 PM

1. The reverse Cuthill–McKee ordering [10 points]

The reverse Cuthill-McKee (RCM) ordering is an algorithm used to reduce the bandwidth and profile of sparse matrices. In MATLAB, we use symrcm to compute the reverse Cuthill-McKee ordering for a symmetric sparse matrix. The goal is to reorder the rows and columns of the matrix so that the non-zero elements are clustered closer to the diagonal, minimizing the bandwidth.

The algorithm begins by identifying a pseudo-peripheral node, in other words a node that is kind of far away from the center of the graph. This node can be found by starting at a random node and performing a Breadth-First search (BFS) various times to find a node that is farther from the starting node.

After choosing the node, the BFS performed starting from this node and it explores all the neighboring nodes at the present depth before moving on to nodes at the next depth level. When visiting the neighbors of a node, the algorithm prioritizes the nodes with lower degrees (fewer connections), generating a list structure where nodes are ordered level by level, and within each level, the nodes with fewer neighbors are ordered first. At the end, reversing the list we will obtain the nodes with more connections closer to the diagonal.

Let's see how does our matrix look like before performing the reverse Cuthill–McKee ordering. In order to do that, I can display both the regular and the circular plot by using the commands SPY(A) and drawit;snapnow.

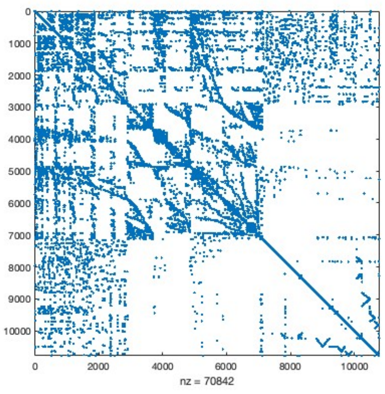


Figure 1: Regular Plot before symrcm()

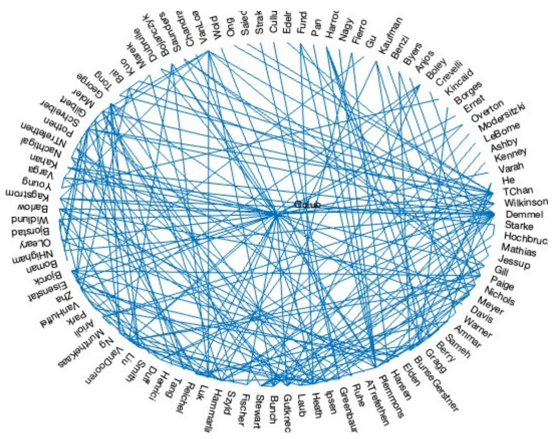


Figure 2: Circular Plot before symrcm()

As we can see, the graph is very sparse. In fact, if we calculate the sparsity, we will obtain the result sparsity = 6.1029e-04 = 0.00061, which means that only roughly 0.06% of the matrix entries are non-zero.

Now, we will be performing the reverse Cuthill–McKee ordering for this matrix, to see the difference in the result. In order to do that, we will use the function symrcm() in MATLAB:

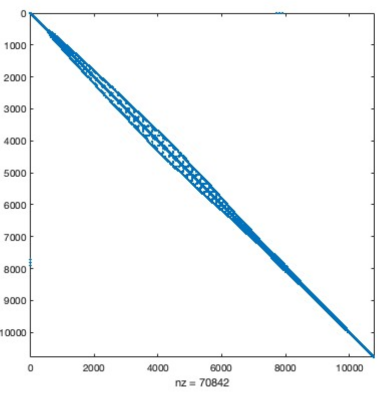


Figure 3: Regular Plot after symrcm()

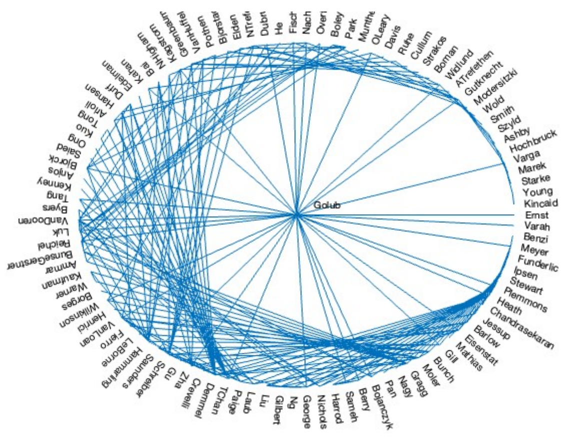


Figure 4: Circular Plot after symrcm()

We wanted to rearrange the data so that the connections are as close as possible to the circumference of the circle. As we have said, this corresponds to a symmetric permutation of matrix A, aimed at minimizing, or at least reducing, its bandwidth.
The symrcm() function used a permutation vector P that, applied to A, generated the reordered matrix PAP^T.

Figure 3 and Figure 4 show the impact of the reordering algorithm on the matrix and on the circular plot. Obviously, the number of nonzero (nz) remains the same, since the algorithm doesn't change the matrix values, but it just reduces the bandwidth.

Finally, we want to compute the Cholesky factorization of the original matrix (Figure 5) and the permuted matrix (Figure 6). We will visualize the Cholesky factors and comment on the number of nonzeros.

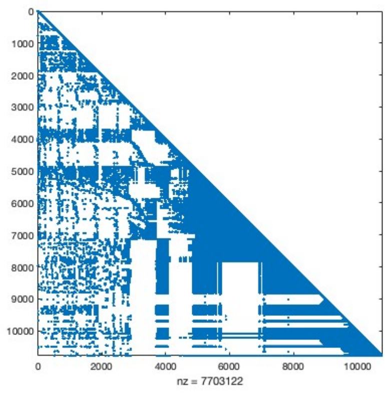


Figure 5: Cholesky factor before symrcm()

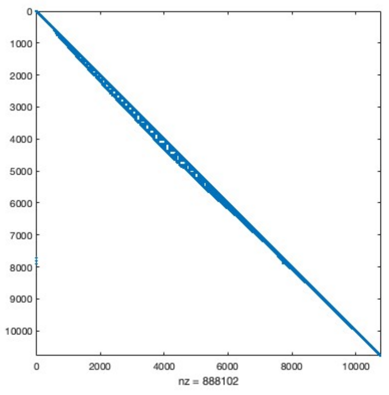


Figure 6: Cholesky factor after symrcm()

Here we find an interesting result: in the Cholesky factor of the original matrix we actually have more nonzeros than in the Cholesky factor of the permuted matrix.

In fact, the sparsity of Figure 5 is 0.0664, while the sparsity of Figure 6 is 0.0077. Why is that?

Sparsity of original matrix = 0.0664 means that approximately the 6.64% of the elements are non-zero. Sparsity of matrix permuted = 0.0077 means that approximately the 0.77% of the elements are non-zero. Since the size of both the Cholesky factor matrices is the same, the total number of elements in both matrices is constant. A lower sparsity value for the permuted matrix (0.0077) means that it presents a lower number of non-zero entries. However, even if the original matrix and the permuted matrix have the same number of non-zero entries, their arrangement structure is very different and the Cholesky factorization is very sensitive to this structure difference.

In fact, we know that in this case the Cholesky factorization of a matrix A produces a lower triangular matrix L such that A = LL^T. The structure of L depends on the original arrangement of the non-zero entries in A. For this reason, during the Cholesky factorization process, different non-zero entries could appear in L even if they were not present in the original matrix due to the triangularization process. As I said earlier, the reverse Cuthill-McKee ordering doesn't change the number of non-zero entries in the original matrix, but it can significantly affect how those entries interact during the factorization process. By permuting the matrix, the structure is modified in such a way that the factorization can become more efficient, leading to fewer fill-ins.

2. Sparse matrix factorization [20 points]

Let the matrix $A \in R^{n \times n}$ be symmetric and positive definite, with entries A_{ij} defined as follows:

For $i = 1$ or $i = n$ or $j = 1$ or $j = n$ and $j \neq i$:

$$A_{ij} = 1$$

For $i = j$:

$$A_{ii} = n + i - 1$$

This means that the diagonal entries are:

$$A_{11} = n, \quad A_{22} = n + 1, \quad A_{33} = n + 2, \quad \ldots, \quad A_{nn} = 2n - 1$$

For all other cases:

$$A_{ij} = 0$$

After having defined the matrix, I created a MATLAB script called A_Construct() (which will be requested later) to calculate the sparse matrix for n = 10:

$$A = \begin{bmatrix} 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 13 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 17 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 18 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 19 \end{bmatrix}$$

Using the nnz(A) function in MATLAB, we can easily know that the number of non-zero of this matrix is 44. This may seem a random number but it's not. Can we derive a general formula to compute the number of non-zero entries?

First of all, we know that all the diagonal entries are non-zero, so we know that the number of non-zero diagonal entries = n.

Now, we need to understand that the other non-zero elements basically represent the "border" of the matrix, excluding the element A_{11} and the element A_{nn}, that are already counted in the diagonal.

We can easily understand by looking at the matrix that we will have (n-1) entries under the first element, (n-1) entries above the last element, (n-2) entries between the first and last column in the upper raw, and (n-2) entries between the first and last column at in the lower raw. Obviously, we have to add the n non-zero entries of the diagonal.

We can calculate n + (n-1) + (n-1) + (n-2) + (n-2) = 5n - 6.

I created the function A_construct(n) that takes as input n and returns, as output, the matrix A and its number of non-zero elements nz.

Now, I want to test my function in a script ex2c.m for n = 10, and compare the results with those we obtained in the previous point. Furthermore, within the same script, we will visualise the non-zero structure of matrix A by using the command spy().

I wrote and run the script and, after generating the matrix with n = 10, I calculated the number of non-zero elements both with the MATLAB function and manually using the formula 5n - 6. As I was expecting, the results are identical. Using the command spy(A) we can also visualise the non-zero structure of matrix A:

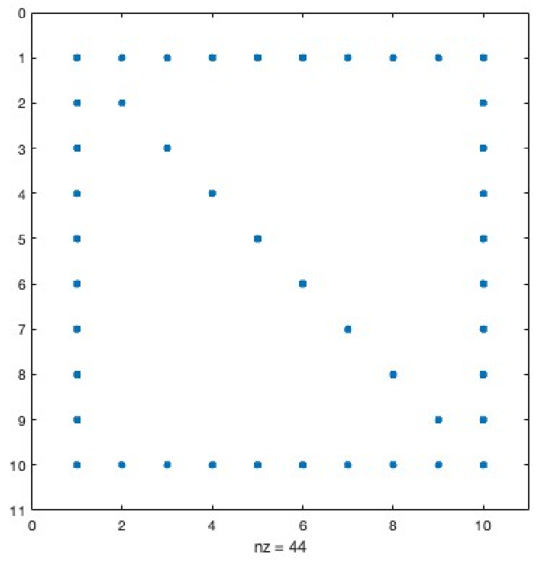


Figure 7: Non-zero (nz) structure of matrix A

As I was saying earlier, the non-zero elements of the matrix basically represent the "border" of the matrix plus the diagonal of the matrix.

I used again the spy() command, in order to visualize side by side the original matrix A and the result of the Cholesky factorization chol(A):

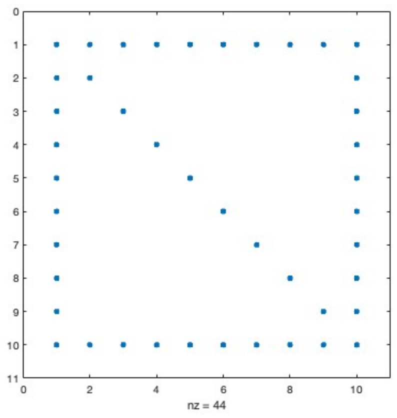


Figure 8: nz plot of matrix A

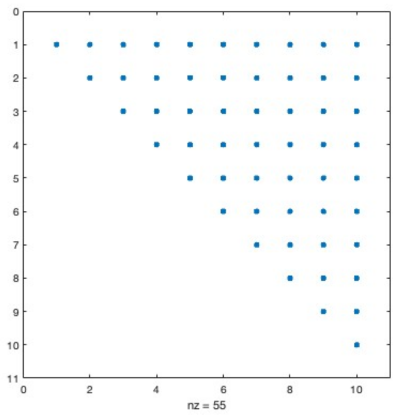


Figure 9: nz plot of matrix chol(A)

As we have seen, the Cholesky factorization is a mathematical method used to decompose a symmetric and positive-definite matrix A into an upper (or lower in the previous case) triangular matrix such that A = $R^T R$.

In this case, the original matrix A is sparse, meaning it has many zero elements. In fact, the off-diagonal elements are zero except for the boundary elements. During the factorization, some zero entries in A become non-zero in the factorized matrix, and this is called fill-in. Fill-in occurs because the factorization algorithm often introduces new non-zero entries that weren't in the original matrix, in order to maintain symmetry and positive-definiteness. In the new matrix the sparsity pattern becomes denser in certain areas, particularly in the upper triangular part.

In this case, as we can see, the number of non-zero elements increases from 44 to 55.

However, it's important to notice that the process of fill-in during factorizations doesn't change the essential structure of the matrix, since The Cholesky factorization decomposes the matrix into an upper triangular matrix such that the original matrix can always be fully rebuilt from the new one. In fact, Cholesky doesn't change the actual relationships encoded by the edges of the matrix, but it just reorganizes the structure to make solving certain problems easier.

Using chol() to solve Ax = b when n = 100,000 could be an issue for many reasons.

First of all, when we use the Cholesky factorization on a large matrix, the number of non-zero elements can increase significantly due to fill-ins. Even if the original matrix is sparse, the Cholesky factor matrix will probably become much denser, increasing the storage requirements. In fact, since this factorization needs to store the upper triangular matrix somewhere in memory, we would need a lot of free space, and the amount of memory required could exceed the available physical memory.

For example, In a dense n×n matrix, storing all elements has a space complexity of $O(n^2)$ space, which for n = 100,000 means storing 10^{10} elements. If we assume that these elements are 8 bytes each, that would require 800 GB of memory, which is more than what a lot of systems can afford.

Furthermore, the Cholesky factorization has a time complexity of $O(n^3)$, and that means that calculating and storing so many elements could take a big amount of time.

Finally, for very large matrices, Cholesky factorization could face some numerical instability problems due to small errors in floating-point arithmetic, leading to inaccurate solutions.

However, there are some methods we can use to solve Cholesky factorization and in general large-scale systems more efficiently.

For example, we might prefer to use iterative methods, that for very large and sparse systems are often a better alternative because they don't have to face with complex systems to be solved. We could also use matrices that transform the original system into one that is easier to solve iteratively, like Jacobi. Finally, nowdays we could take advantage of parallel computing to parallelize both iterative solvers and direct methods through different cores at once.

3. Degree centrality [5 points]

In graph theory and network analysis, centrality refers to indicators which identify the most important vertices within a graph. Applications include, for example, identifying the most influential person(s) in a social network.

Here we are interested in the Degree centrality, which is conceptually simple. It's defined as the number of links incident upon a node (in case of a directed graph we consider both ingoing and outgoing edges). The degree centrality of a vertex v, for a given graph G:= (V,E) with $|V|$ vertices and $|E|$ edges, is defined as the numbers of edges of vertex v.

We now want to compute the degree centralities for the top 5 authors of the 104x104 sparse graph "housegraph.mat", and then we will include them in an ordered list, showing the authors, their coauthors and the degree centrality.

I created the script and the function DegreeCentrality(A), which is the same I used in the PageRank project, with a few improvements. After computing, the results are the following:

Author	**Degree Centrality**	**Coauthors**
1	63	1, 2, 3, 5, 10, 11, 32, 34, 35, 36, 41, 47, 48, 49, 51, 53, 55, 58, 59, 60, 66, 71, 72, 73, 76, 79, 81, 86, 91, 92, 96, 99
104	31	26, 32, 39, 44, 48, 51, 52, 56, 63, 69, 74, 78, 81, 83, 90, 104
86	27	1, 22, 23, 24, 25, 35, 41, 52, 81, 86, 91, 92, 96, 97
44	25	3, 32, 42, 43, 44, 45, 46, 54, 56, 59, 71, 88, 104
81	25	1, 3, 25, 41, 43, 59, 65, 67, 80, 81, 86, 98, 104

Table 1: Top 5 Authors by Degree Centrality and their Coauthors

As we were expecting, the most prolific author is Gene Golub, one of the organizers.

4. The connectivity of the coauthors [10 points]

We might be wondering how many coauthors do the authors have in common. We want to find a general procedure that allows us to compute the list of common coauthors of two authors, and expressing it in matrix notation (carefully explain your reasoning in the report).

In a coauthorship network, the matrix A is a square matrix where: A(i,j)=1 if authors i and j are coauthors. To find Common Coauthors we want to extract the coauthor vectors. For any two authors x and y, we can extract their coauthor vectors from the adjacency matrix by doing a(x) = A(x, :), which is the row of matrix A corresponding to author x and a(y) = A(y, :), which is the row of matrix A corresponding to author y.

The coauthors that are common to both authors x and y can be found by computing the element-wise multiplication (MATLAB) of their coauthor vectors a_x and a_y. Mathematically, this can be represented as:

$$\text{common coauthors} = a_x \cdot a_y^T$$

This operation returns a vector where a "1" at position i indicates that the author indexed at i is a coauthor of both x and y. In matrix notation this can be written as:

$$\text{Common Coauthors of Authors } x \text{ and } y = \left\{ i \mid \left(a_x \cdot a_y^T \right)_i = 1 \right\}$$

In MATLAB, the element-wise multiplication can be compute by using the operator ".*".
I created the script "commonCoauthors.m" that uses the formula of element-wise multiplication to compute the common coauthors of the pairs (Golub, Moler), (Golub, Saunders), and (TChan, Demmel). Who are the common coauthors?

Common coauthors of Golub and Moler: `'Wilkinson', 'VanLoan'`

Common coauthors of Golub and Saunders: `'Golub', 'Saunders', 'Gill'`

Common coauthors of TChan and Demmel: `'Schreiber', 'Arioli', 'Duff', 'Heath'`

5. PageRank of the coauthor graph [5 points]

Now, we want to compute the PageRank value for all authors, providing a graph of all authors in descending order according to the PageRank. I used the power method version of the pagerank algorithm and the results are the following:

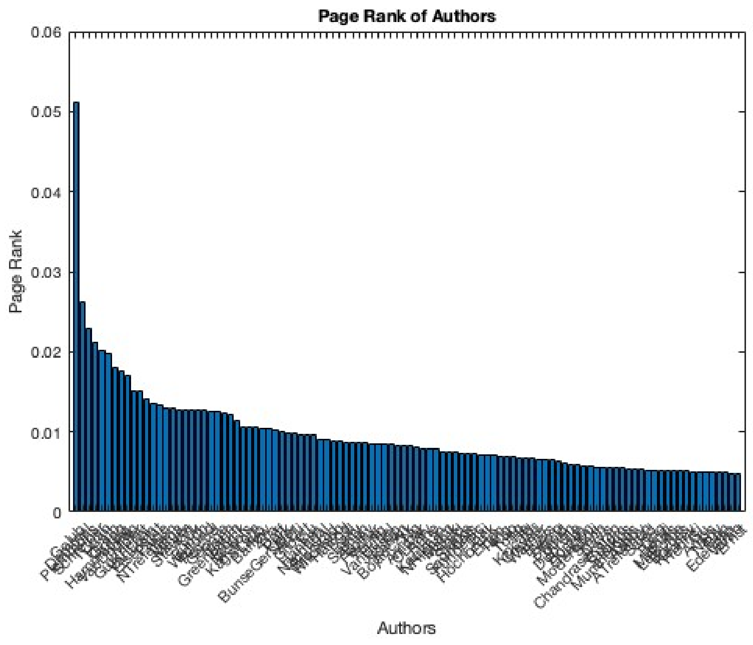


Figure 10: PageRank in descending order of the Coauthors matrix

Author	**PageRank**
Golub	0.051072
Demmel	0.026117
Plemmons	0.022895
Schreiber	0.021229
TChan	0.020140
Heath	0.019756
Gragg	0.018053
Hammarling	0.017682
VanDooren	0.017063
Moler	0.015188

Table 2: Authors Ranked by PageRank

As we can notice, the order of the authors with the higher PageRank is the same as the order of the authors with higher degree centrality. This is actually a very interesting thing because it's not guaranteed that highest degree centrality means highest PageRank as well.

As we saw in the previous project, after removing the nodes with the higher degree centrality, the PageRank didn't change drastically but just a little bit. That happened because having an high degree centrality means having a lot of connections but that doesn't mean necessarily that those connections are "important" in the PageRank calculation, since they might have a low PageRank.

However, in this case, due to how the matrix is structured, high degree centrality means also high PageRank. Probably this happens because in the coauthor matrix we are interested more in the number of coauthors than the actual importance of each of them.

6. Zachary's karate club: an introduction to spectral graph partitioning [35 points]

Before solving the requirements, I will briefly talk about what Laplace graph partitioning is, since it leads myself and the reader to a better understanding of the topic.

The goal of this section is to divide a graph into two distinct sets such that the number of edges between these two sets is minimized. This kind of problem is common in areas like image segmentation, clustering, and network analysis.

For a graph $G = (V, E)$, where V represents the vertices and E the edges, the Laplacian matrix is defined as:

$$L = D - A$$

where:

- D is the degree matrix (a diagonal matrix in which each diagonal entry D_{ii} represents the degree of vertex i),
- A is the adjacency matrix of the graph.

The Fiedler vector v_2 is the eigenvector corresponding to the second smallest eigenvalue λ_2 of the Laplacian matrix. The eigenvector v_2 is used to divide the graph into various sets. The components of the Fiedler eigenvector v_2 tells us how to make the partition of the vertices of the graph into two groups: vertices corresponding to positive entries of v_2 and vertices corresponding to negative entries of v_2.

This process is basically called spectral clustering, in fact the graph is partitioned into clusters using the eigenvectors of the Laplacian matrix, by using the eigenvector v_2, as we have just seen. In a few steps summary, what we need to do is:

1. For a given graph, calculate the Laplacian matrix L.
2. Find the eigenvalues and eigenvectors of L.
3. Identify λ_2 and v_2, which are the second smallest eigenvalue λ_2 and its corresponding eigenvector v_2.
4. Use the signs of the entries in v_2 to partition the graph into sets.

Example

I want to do an example real quick: let's consider an undirected graph with 6 vertices A, B, C, D, E, F, and the following edges:

(A, B), (A, C), (B, D), (C, D), (D, E), (D, F), (E, F)

The adjacency matrix A is the following:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

The degree matrix D is instead a diagonal matrix where each diagonal entry D_{ii} represents the degree (number of edges) of each vertex i:

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

And the Laplacian matrix L is calculated as $L = D - A$:

$$L = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & 0 & -1 & 0 & 0 \\ -1 & 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & -1 & 4 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{bmatrix}$$

Finally, we want to calculate the eigenvalues and eigenvectors of L, which are:

$$\lambda_1 = 0,\ \lambda_2 = 0.585,\ \lambda_3 = 2,\ \lambda_4 = 3,\ \lambda_5 = 4,\ \lambda_6 = 5.414$$

And consequently, the eigenvector corresponding to $\lambda_2 = 0.585$ (the Fiedler vector v_2) is about:

$$v_2 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ -0.35 \\ -0.5 \\ -0.5 \end{bmatrix}$$

In conclusion, to partition the graph, we look at the sign of the entries in v_2:

- Positive entries: A, B, C
- Negative entries: D, E, F

In fact, the two sets will be:

$$\text{Set } 1 = \{A, B, C\}, \quad \text{Set } 2 = \{D, E, F\}$$

This quick part of theory might be useless for the goal of the project, but it's very helpful to me to understand what I will be doing later.

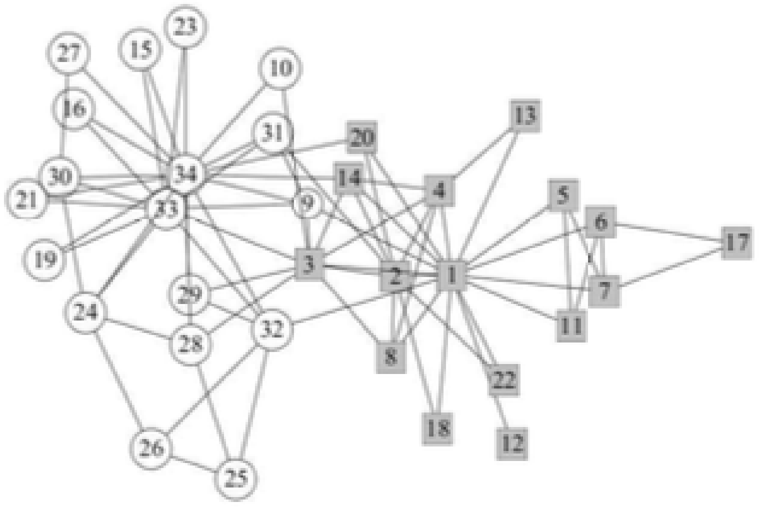
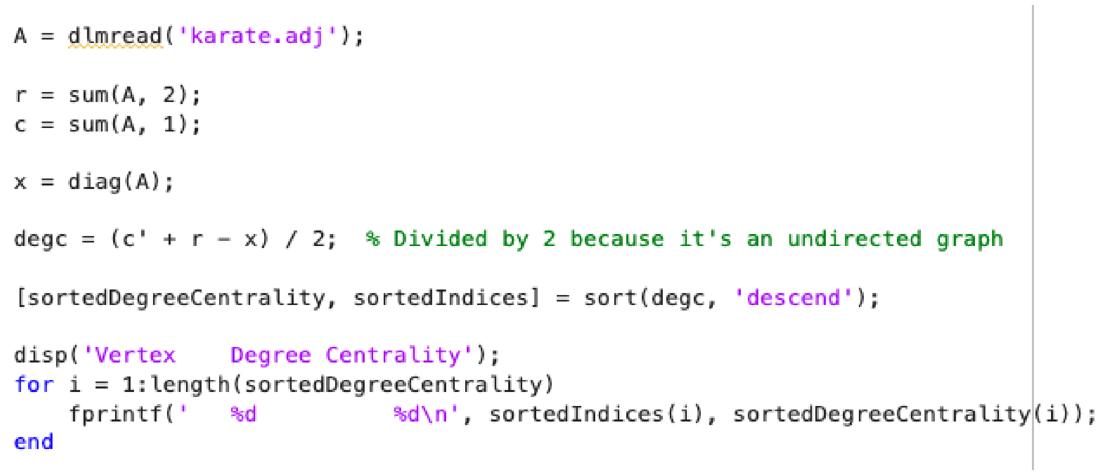


Figure 11: The social network of a karate club at a US university.

Figure 11 shows the social network of a karate club at a US university in the 1970. At the end of the study, the club formally split into two separate organizations (white circles vs. grey squares). We want to load the adjacency matrix karate.adj to compute our studies.

First of all, we want to rank the five nodes with the largest degree centrality, by using a MATLAB script:



```
A = dlmread('karate.adj');

r = sum(A, 2);
c = sum(A, 1);

x = diag(A);

degc = (c' + r - x) / 2;  % Divided by 2 because it's an undirected graph

[sortedDegreeCentrality, sortedIndices] = sort(degc, 'descend');

disp('Vertex    Degree Centrality');
for i = 1:length(sortedDegreeCentrality)
    fprintf('   %d          %d\n', sortedIndices(i), sortedDegreeCentrality(i));
end
```

Figure 12: Script for karate.adj degree centrality

The five nodes with the largest degree centrality and their degree centrality are the following:

Node	Degree Centrality
34	17
1	16
33	12
3	10
2	9

Table 3: Top 5 nodes by Degree Centrality

Now, we want to rank the five nodes with the largest eigenvector centrality.

The eigenvector centrality is a measure of the influence of a node in a network. Differential from the degree centrality, that only counts the number of connections of each node, eigenvector centrality considers not only the quantity of the connections, but also the quality and the influence of those connections. In fact, a node is considered more central if it's connected to other well-connected nodes.

This concept is kind of similar to the PageRank idea. In fact, a node has an high eigenvector centrality if it's connected to other nodes that also have an high eigenvector centrality. This means that the influence of a node is derived not only from its immediate connections but also from the connections of its neighbors. For example, in a social network, a person with many connections might have an higher eigenvector centrality if those connections are to influential people.

For our graph, I calculated the normalized eigenvector centralities for the five nodes with larger degree centrality, and the results are the following: What are their (properly normalized) eigenvector centralities?

Node	Eigenvector Centrality
34	0.0750
1	0.0714
3	0.0637
33	0.0620
2	0.0534

In order to answer the next question as well, we can compare the two rankings that we got. If we compare side by side the two tables that we just obtained, it's interesting to notice that the order is a bit different. In fact, even if node 33 has an higher degree centrality than node 3, this one has an higher eigenvector centrality. We might be wondering how is that possible, and the explanation is the following:

Node 3 is connected to node 1, 2 and 33, which are three of the nodes with higher degree centrality. On the other hand, from the top 5 nodes, node 33 is only connected to node 3 and 34, which means one important connection less than the other one.

This behaviour is pretty similar to the one used to compute the PageRank algorithm.

Node	Degree Centrality
34	17
1	16
33	12
3	10
2	9

Node	Eigenvector Centrality
34	0.0750
1	0.0714
3	0.0637
33	0.0620
2	0.0534

To conclude our analysis, we want to use spectral graph partitioning to find a near-optimal split of the network into two groups of 16 and 18 nodes, respectively.

After listing the nodes in the two groups, we want to know how does spectral bisection compare to the real split observed by Zachary (Figure 11). Comment on the results obtained.

I used the code informations given to calculate the two groups division by using the spectral graph partitioning. I actually created two different scripts, GraphPartitioning1.m and GraphPartitioning2.m, quite similar but with some differences.

The first code creates a graph by using randomness, by randomly assigning edges, while the second code analyzes the real structure of Zachary's karate club graph. The first code presents randomness in both group assignments and edge connections, which might not keep the actual trueness of the network. In fact, the groups are defined arbitrarily, while the second code creates groups starting from the properties of the graph, ensuring that the groups are formed based on the connectivity patterns.

For this reason, in order to give a better understanding of it, in this report I will be using the results given by GraphPartitioning2.m, that more reflects the example that I've illustrated at the beginning of this section. However, similar results can be obtained by using the other algorithm as well, which is the book suggestion.

After running the script, the two groups based on the sign of the entries in the Fiedler eigenvector v_2 are the following:

Group 1 Nodes
17
6
7
5
11
12
1
13
18
22
4
8
2
14
20
3
9
31

Table 4: Group 1 Nodes

Group 2 Nodes
10
29
32
34
28
33
25
24
26
23
16
21
15
19
30
27

Table 5: Group 2 Nodes

This spectral bisection compared to the real split observed by Zachary (Figure 11) is literally identical. This is a great thing and it happened because we used the sorted entries of the Fiedler eigenvector v_2 to compute the partitioning. If we were to use the algorithm that used randomness, the result would have been a bit different, since it wouldn't have properly respected the structure of the graph.

Finally, let's plot the two important images of the spectral graph partitioning:

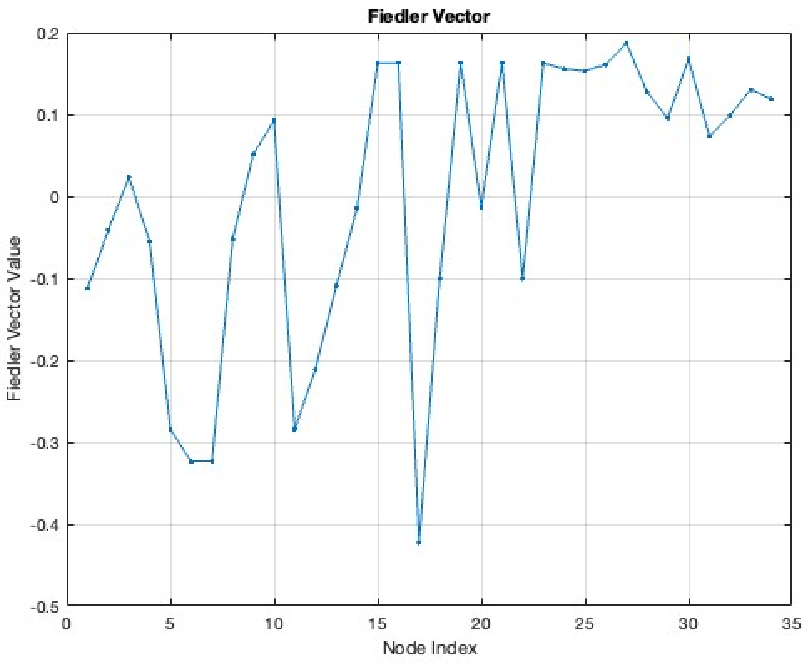


Figure 13: The second smallest eigenvector

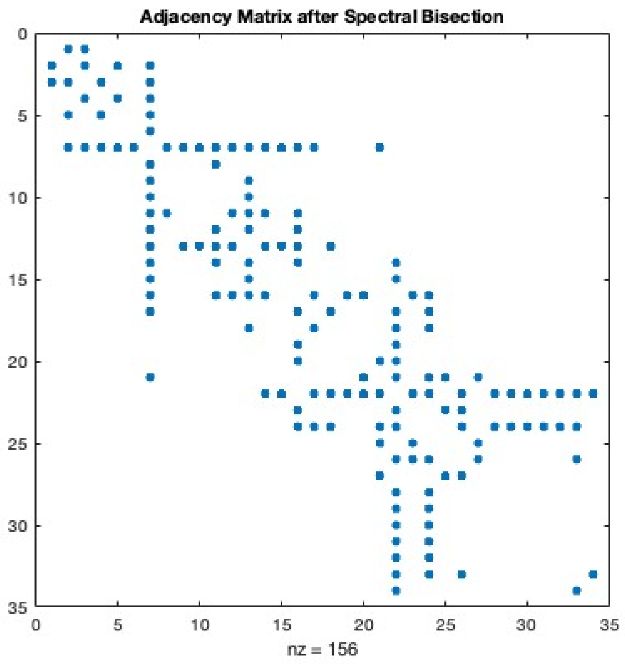


Figure 14: The results after the partitioning of our artificial dataset

In Figure 14 we can finally clearly observe our partitioning. In the spectral analysis of a graph, the second smallest eigenvalue is very helpful in finding a partitioning that splits the graph into two subgraphs while at the same time minimizing the edges between these two subgraphs.

Sources:

https://ch.mathworks.com/help/matlab/math/partition-graph-with-laplacian-matrix.html

https://www.stat.berkeley.edu/ mmahoney/s15-stat260-cs294/Lectures/lecture06-10feb15.pdf

https://www.cs.purdue.edu/homes/dgleich/demos/matlab/spectral/spectral.html

https://www.youtube.com/watch?v=cxTmmasBiC8t=23s