Algorithmic Methods of Data Mining
Project Report

Swati Choudhary(795920) : swati.choudhary@aalto.fi
Prakhar Verma (795904) : prakhar.verma@aalto.fi

December 2019

Contents

1 Introduction

The project aims to perform a graph partitioning algorithm with the use of Spectral Clustering on social network graphs. The goal is to minimize the number of cuts while maintaining cluster size balance. In this report we describe our implementation of a graph partitioning algorithm and apply it on 5 graphs of varying size. The graphs being used for the project are available at Stanford SNAP data.The required number of clusters is defined along with the graphs in the input files. The goodness of the partitions is evaluated using a given objective function. We use spectral graph partitioning as our main algorithm but make modifications on the existing algorithm to get better results and faster computation times.

The following 5 graphs are being used for the analysis :

- ca-GrQc
- Oregon-1
- soc-Epinions1
- web-NotreDame
- roadNet-CA

These graphs were large and strongly connected so sparse representations needed to be used.

In Section 2, we present background of the problem and existing studies on spectral partitioning, graph partitioning in general and on large symmetric eigenvalue problems. Section 3 describes the methods and algorithms used in detail and discuss the improvements we made on our initial implementation of the algorithm. In Section 4 we present our results of the algorithm and it's improvements and finally, in Section 5, we summarize the results of our report.

2 Problem Statement & Literature Review

Graph partitioning is the task of splitting the vertices of a graph into k clusters or communities $V_1, \ldots, V_k$, so that $v_i \cap v_j = \phi$ in such a way that, there are maximum number of edges within each cluster and very few between the clusters. This is the objective function that we have to minimize. Mathematically, it is defined as :

$$\phi\left(V_1, \ldots, V_k\right) = \sum_{i=1}^{k} \frac{\left|E\left(V_i, \bar{V}_i\right)\right|}{|V_i|} \tag{1}$$

Graph partitioning or graph clustering, has multiple applications for example in analyzing social networks and biological networks. The goal in graph partitioning is to find sets of vertices that are related. One way to achieve this

is by minimising the amount of edges between different communities. Graph partitioning is an NP-hard problem. The solutions for the problem are approximations and run faster than NP-hard by using heuristics.

Spectral clustering is an effective way to perform graph partitioning. In spectral graph theory, graphs are studied through eigenvalues and eigen-vectors of matrices that are derived from the graphs. One of these matrices is adjacency matrix A, where a value of 1 in A_{ij} implies an edge between vertices i and j. The eigen vectors of the matrix are used to map the vertices of the graph into a real line, having related vertices close to each others. The values of the eigen vectors represent different vertices. The clustering of the vertices is then fast, in 1 dimension with some clustering algorithm, like k-means [2].

One part of the spectral clustering algorithm is the computation of eigen-values and corresponding eigen vectors. However, in practice the problems can be very large and computing eigenvalues and/or -vectors for large symmetric matrices is a computationally expensive and intensive operation.

To understand the problem better, [1],[2],[3], were read and from there implementations were tried.

3 Algorithms

This section describes the different ways to do Spectral Clustering. We started our analysis with a simple spectral algorithm. We implemented the algorithm on python using networkx, scipy and the K-means algorithm from sklearn.

The general way of performing spectral clustering is by using Laplacian matrixes and finding the eigen decomposition of the laplacian matrix. Then use only k eigen values to perform clustering [3].

3.1 Spectral Clustering

The different ways in which we perform Spectral Clustering are listed below.

1. Unnormalized Spectral Clustering : Described in Algorithm 1
2. Symmetric Normalized Clustering : Described in Algorithm 2
3. Normalized Spectral Clustering using Random Walk Laplacian : Described in Algorithm 3
4. Normalized Spectral Clustering - Shi Malik : Described in Algorithm 4

Algorithm 1 Unnormalized Spectral Clustering

Input: Graph G and number of clusters k
Output: k optimal clusters of G
Compute Adjacency Matrix A
Compute Diagonal Degree Matrix D
Compute unnormalized laplacian matrix $\mathbf{L = D - A}$
Compute the k smallest eigen vectors of L
Form $U \in R^{nxk}$ with columns $u_1, \ldots u_k$
Cluster points y_i into $\{C_1, \ldots, C_k\}$ using Kmeans
Return clusters $A_1, \ldots A_k$ with $A_i = \{j \mid y_j \in C_i\}$

Algorithm 2 Symmetric Normalized Clustering

Input: Graph G and number of clusters k
Output: k optimal clusters of G
Compute Adjacency Matrix A
Compute Diagonal Degree Matrix D
Compute Identity Matrix I
Compute normalized laplacian matrix $\mathbf{L = I - } D^{-0.5}\mathbf{A}D^{-0.5}$
Compute the k smallest eigen vectors of L
Form $U \in R^{nxk}$ with columns $u_1, \ldots u_k$
Normalize U so that rows have norm 1
Cluster points y_i into $\{C_1, \ldots, C_k\}$ using Kmeans
Return clusters $A_1, \ldots A_k$ with $A_i = \{j \mid y_j \in C_i\}$

Algorithm 3 Normalized Spectral Clustering - Random Walk Laplacian

Input: Graph G and number of clusters k
Output: k optimal clusters of G
Compute Adjacency Matrix A
Compute Diagonal Degree Matrix D
Compute Identity Matrix I
Compute normalized laplacian matrix $\mathbf{L = I - } D^{-1}A$
Compute the k smallest eigen vectors $u_1, \ldots u_k$ of L
Form $U \in R^{nxk}$ with columns $u_1, \ldots u_k$
Normalize U so that rows have norm 1
Cluster points y_i into $\{C_1, \ldots, C_k\}$ using Kmeans
Return clusters $A_1, \ldots A_k$ with $A_i = \{j \mid y_j \in C_i\}$

Algorithm 4 Normalized Spectral Clustering - Shi Malik

Input: Graph G and number of clusters k
Output: k optimal clusters of G
Compute Adjacency Matrix A
Compute Diagonal Degree Matrix D
Compute unnormalized laplacian matrix **L = D - A**
Compute the k smallest eigen vectors $u_1, \ldots u_k$ of the generalized eigen problem $Lu = \lambda Du$
Form $U \in R^{nxk}$ with columns $u_1, \ldots u_k$
Normalize U so that rows have norm 1
Cluster points y_i into $\{C_1, \ldots, C_k\}$ using Kmeans
Return clusters $A_1, \ldots A_k$ with $A_i = \{j \mid y_j \in C_i\}$

3.2 KMeans

We tried KMeans algorithm with 3 modifications which are described below:

1. **Standard KMeans Algorithm:** KMeans algorithm from *sklearn.cluster* was used to cluster U into k clusters.

2. **KMeans with initialization:** Standard KMeans algorithm uses *KMeans++* as the default initialization method. We tried initializing KMeans clusters with k nodes with the maximum degree. However, it didn't show much improvement so this idea was dropped.

3. **Brute-force KMeans:** To obtain balanced clusters, KMeans algorithm was modified in a way that the nodes were assigned to the clusters in a sequential manner. In other words, nodes were assigned to the clusters one by one such that the cluster size remains almost the same. However, the objective function's value wasn't getting optimized from this so we didn't go ahead with it.

3.3 Greedy Algorithms

After k clusters were obtained from the standard KMeans algorithm, custom greedy algorithms were applied on them to optimize the cuts between the clusters and improve the objective function value. The greedy algorithms are discussed below:

1. Move the nodes with more outward edges than inwards to other cluster

2. Move the nodes which are farthest from the centroid of KMeans clusters to the other cluster

3. Move the nodes with minimum inward edges to the other cluster, so as to create balanced clusters

4 Experiments & Optimizations

We started to work with the smallest graph i.e *ca-GrQc* first. We did the basic spectral clustering algorithm as mentioned in Algorithm 1. We used the following python packages to implement it :

- pandas to do preprocessing on the graph
- numpy to compute the adjacency matrix, degree matrix, eigen vectors and values, and laplacian
- sklearn for kmeans clustering

With this, the result we got for *ca-GrQC* graph was 0.97. However, the same implementation was not working for the bigger graphs due to memory and time constraints. So we decided to optimize these parameters.

Also, in the competition leaderboard, there were better results, 0.06 being the best. We used this result as our baseline for comparison and performed further optimizations to improve the result.

Our baseline for comparison and trying more optimizations came from the fact that we were uploading our graph outputs in the competition regularly and comparing it with the 10 best results.

The following sections describe the various experiments conducted in detail.

4.1 Eigen Vector Selection

To calculate k-eigen vectors different implementations were tried and are mentioned below. The best result was given when **k smallest eigen vectors** were chosen.

1. K smallest real
2. K smallest magnitude
3. K smallest algebraic
4. K largest real
5. K largest magnitude
6. K largest algebraic

4.2 Laplacian Computation

Initially we calculated the adjacency matrix and the laplacian from the function provided by *networkx*.

However, by using it we were not getting a good value for the objective function and also there were some discrepancies in the values. So, we decided to write our own functions to calculate adjacency matrix, degree diagonal matrix and laplacian.

4.3 Normalized vs Unnormalized Eigen Vectors

We tried the above four algorithms both with normalized and unnormalized eigen vectors. Results for these are summarized in the Results section. Even though the algorithms say to normalize, we get a much better value for the objective function without normalizing the eigen vectors. This is because the *scipy.sparse.linalg.eigs()* in python gave us negative eigen values and on normalization these do not sum up to 1.

4.4 Space and Time Consumption

Since numpy was failing and giving errors for the bigger graph we decided to use sparse representations throughout to compute adjacency matrix and diagonal matrix. We used *scipy.sparse()* to compute and store these. This sparse representation saved time in the eigen decomposition as it calculates eigens over a sparse matrix rather than a big matrix.

4.5 Running kmeans n times and computing m eigen vectors instead of k

We also tried to compute more eigen vectors than needed and then from that we choose k eigen vectors and send these k eigen vectors to kmeans for clustering.

Algorithm 5 Random times k means with n eigens

Input: U, clusters, n , initial_fixed_k
Output: k optimal clusters of U
for `i in (0, n)` **do**
    `Sample eigen values from U keeping initial_fixed_k values`
    `fixed and randomly picking (k - initial_fixed_k) values`
    `current_clusters = KMeans(U_sampled)`
    `current_objective_value = objective_function(current_clusters)`
    if current_objective_value < objective_value **then**
        `objective_value = current_objective_value`
        `clusters = current_clusters`
Return clusters $A_1, \ldots A_k$ with $A_i = \{j \mid y_j \in C_i\}$

5 Results

In this section we present the performance of our implementation of the algorithm and the performance increases gained by the modifications done to the implementation of the algorithms.

5.1 Kmeans run only once along with greedy

We ran k means clustering only once along with the greedy algorithms. For normalized and unnormalized eigens the results are summarized in Figure 1.
The '-' in the figure means that the algorithm did not complete running or was taking a very long time to run so we did not compute those values.
The best results obtained from this set is in blue color in the figure.

		Un-normalized Laplacian		Random Walk Laplacian		Symmetric Normalized Laplacian		Shi Malik	
		Normalize Eigen	Un-normalize Eigen	Normalize Eigen	Un-normalize Eigen	Normalize Eigen	Un-normalize Eigen	Normalize Eigen	Un-normalize Eigen
roadNet-CA	Without Greedy algorithms	-	-	0.2838	0.2737	0.2776	1.777	0.2755	0.2745
	Greedy algorithms	-	-	0.2803	0.2714	1.7835	**0.2644**	0.2726	0.2764
Oregon-1	Without Greedy algorithms	1.7345	**0.722**	1.187	0.7397	1.193	20.3553	1.1933	0.7397
	Greedy algorithms	1.7345	**0.722**	1.187	0.7397	1.193	20.3553	1.1933	**0.7222**
ca-GrQc	Without Greedy algorithms	0.3906	0.0836	0.3558	**0.0757**	0.5263	0.2305	0.5263	**0.0757**
	Greedy algorithms	0.3906	0.0836	0.3497	**0.0757**	0.3497	0.1636	0.3497	**0.0757**
web-NotreDame	Without Greedy algorithms	-	-	0.1272	**0.0431**	0.1087	2749.8714	0.1313	0.0514
	Greedy algorithms			0.0965	**0.0431**	-	-	-	-
soc-Epinions1	Without Greedy algorithms	4.812	**0.7548**	3.3163	0.7652	6.4361	0.9518	6.4361	0.7652
	Greedy algorithms	4.812	**0.7548**	4.9544	0.7652	6.4361	0.9518	6.4631	0.7652

Figure 1: Results

5.2 Kmeans run n times along with greedy

When we run for this set of experiment, instead of running kmeans for all types of laplacian and doing for both normalized and unnormalized eigen vectors, we pick from Figure 1 the best combination and then run k means n times for that combination. So for example in web-NotreDame, the best result came for unnormalized eigen using Random Walk Laplacian, so this configuration was used to run k means n times.

The results from this set of experiments is summarized in the table below. Here there are two running times, as the first time represents the time taken to calculate the best configuration and the second time is the time taken to run k means n times and optimze the existing value.

Graph	Obj. Fn. Value	Config. Time	Kmeans time	Total
ca-GrQc	0.0627414293	125 s	1.23 s	126.23 s
Oregon-1	0.6003758692	604 s	30.2 s	634.2 s
soc-Epinions1	0.5423031827	1.5 h	0.5 h	2.0 h
web-Notredame	0.033314062	9.2 h	1.2 h	10.4h
roadnet-CA	0.2644429793	18 h(*)	4.5h	22.5 h

* : Explicitly terminated, didn't complete for all configurations

5.3 Best Configuration

Below are the configurations which gave us the best results. Further research can result in better configurations too.

Graph	kmeans iterations	Eigen vectors
ca-GrQc	10	6
Oregon-1	30	15
soc-Epinions1	60	40
web-Notredame	50	30
roadnet-CA	1	50

6 Competition Results

The results obtained from the competition are summarized below.

Graph	Objective Function Value	Position
ca-GrQc	0.0627414293	1 (tied with 30 others)
Oregon-1	0.6003758692	1 (tied with 20 others)
soc-Epinions1	0.5423031827	6
web-Notredame	0.033314062	14
roadnet-CA	0.2644429793	17

7 Conclusion

Our initial implementations improved a lot in terms of the objective function value because of the optimizations and greedy algorithms which we did. The idea to run kmeans multiple times and use a randomized amount of additional

eigenvectors on top of the k smallest eigenvectors improved the objective function value by quite a significant amount. The additional eigenvectors provide some extra information on the internal structure of the graph and the partitions produced by the K-means algorithm when using these additional eigenvectors happened to produce a smaller value for the objective function.

References

1. A Tutorial on Spectral Clustering

2. Bin Luo, Richard C Wilson, and Edwin R Hancock. Spectral embedding of graphs.Pattern recognition, 36(10):2213–2230, 2003.

3. Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In Advances in neural information processing systems, pages 849–856, 2002.