Data Mining Programming Project – Using Eigenvectors for Graph Partition

Xiaobo Wu, 795962

xiaobo.wu@aalto.fi

1. Introduction

Graph partition is the reduction of a graph to sorts of smaller graphs by partitioning its nodes to mutually related groups. Problems of graph partitioning arise in various fields of computer science, engineering and other related areas. For example, partition the web into sets of related pages (web graph), find groups of scientists who collaborate with each other, find groups of related queries submitted in a search engine (query graph) and so on. There has been a huge amount of research on graph partitioning in the history of graph theory. And proposed methods that are able to obtain good partition result for large real-world graphs are generally derived using heuristics and approximation algorithms. As we used eigenvectors to solve this problem, our method is belonging to heuristics algorithms and it has a nice performance with a huge consuming of running time and memory space. In this project, I focus on using eigenvectors and K-means clustering method to partition a set of vertices V into k groups $V1$, $V2$..., Vk in an undirected graph G = (V, E). The detailed steps of algorithm I used was discuss clearly in the lecture of Data Mining. I implemented it by Python3 with the help of some famous packages, NumPy, Scikit-learn and SciPy.

2. Literature Review

2.1 Adjacency Matrix

Generally, we used adjacency matrix or adjacency list to represent a graph in computer language. In this project, because we need to use Laplacian matrix for computing eigenvectors, I choose to denote a graph in the adjacency shape. However, we need to care about that our graph for processing is pretty big, the edges of graph in roadNet-CA.txt has an amount that beyond 2,500,000. So, it's pretty hard to just using normal square matrix to store and analyze such a huge graph, in this case, I used sparse matrix as a data structure to construct the adjacency and diagonal matrix inside computer. From this web article [1], I know the detailed interpretation of sparse matrix. There are two reasons basically, one is for saving storage, there are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements. And another reason is for speed, computing time can be saved by logically designing a data structure traversing only non-zero elements. Furthermore, in this article, it describes how to implement spare matrix and other useful introductions.

2.2 Laplacian Matrix

The Laplacian matrix of a graph and its eigenvectors can be used in several areas of mathematical research and have a physical interpretation in various physical and chemical theories. From the Wikipedia of Laplacian matrix [2], I know the definition of it and get familiar with some important properties. Briefly, Laplacian matrix can be used to find many useful attributes of a graph, it can also be used to construct low dimensional embeddings, which can be powerful for various of machine learning algorithms. Then, I learn how to compute a Laplacian matrix of a graph from its diagonal matrix D and adjacency matrix A. The basic formula to compute a non-normalized Laplacian matrix is $L = D - A$. And normalized Laplacian matrix is calculated by equation $L' = I - D^{-1/2} A D^{-1/2}$.

In the paper *The Laplacian spectrum of graphs* [3], we know an efficient method emphasis on the second smallest Laplacian eigenvalues λ_2 and its relation to numerous graph invariants. From this paper, I have a deeper understanding of Laplacian matrix and its correlation with graph partition.

2.3 Eigenvector

There is lots of detailed introductions about definition and application of eigenvector in the Wikipedia [4]. First, we can have a look at its formal definition. If T is a linear transformation from a vector space V over a field F into itself and v is a vector that is not zero, then v is an eigenvector of T is T(v) is a scalar multiple of v. We can write it as the equation: $T(v) = \lambda v$.

2.4 Graph Partition

Just as the situation above. I learn the basic definition about graph partition from its Wikipedia [5]. It is the reduction of a graph to a smaller graph by partitioning its nodes into mutually exclusive communities. Edges of the original graph that cross between the groups will produce edges in the partitioned graph. Furthermore, I get to know some basic method about how to get a balance graph portioning from the paper Balanced Graph Partitioning that write by Konstantin Andreev and Harald

Racke [6]. In this article, they consider the problem of partitioning a graph into k communities of roughly same size while minimizing the capacity of the edges between different components of the cut. They presented a bicriteria approximation algorithm that for any constant amount of vertices v > 1 runs in polynomial time and guarantees an approximation ratio of $O(\log^{1.5} n)$. This algorithm is an approximation method for partitioning a graph, and we do not implement it in this project. But it's a good thinking which made me have a clearer idea about how to better use eigenvectors to solve this problem. Also, I will seek for implementing an approximation section for solving this problem in the future.

3. Algorithms and methods

3.1 Generate adjacency matrix

After I read the vertices number n, value of k and edges in the *.txt file. I used scipy. sparse. lil_matrix to represent and store the adjacency matrix **adj** of graph **G.** For every edge **e** in the total edges, the connected two vertices **s, t** consists of **e.** So, we only need to traverse edges and set **adj[s, t]** and **adj[t, s]** both equal to 1, which means s and t and connected to each other. The code of this method is showed below:

```python
def generate_adj(graph, n):
    a = time.time()
    adj = lil_matrix((n, n), dtype=np.int16)
    for s, t in graph:
        adj[s, t] = adj[t, s] = 1
    t = time.time()
    print('Time for generate adjacency matrix:{}'.format(t - a))
    return adj
```

3.2 Compute Laplacian matrix

In Python's scipy package, there is a module named sparse.csgraph has a strong strength to deal with the computation of Laplacian matrix. We only need to input a sparse matrix just like the adjacency matrix adj we initialize before, and this function can compute the Laplacian matrix for us automatedly and quickly. The code for calculating Laplacian matrix is:

```python
def generate_lap(adj):
    s = time.time()
    lap = csgraph.laplacian(adj, normed=False)
    t = time.time()
    print('Time for generate laplacian matrix:{}'.format(t - s))
    return lap
```

3.3 Compute Eigenvector

Eigenvector computing is the most difficult and time-consuming part of our algorithm for graph partition. It's pretty hard and wasting time to get all the eigenvectors of a such big graphs that we are analyzing in this project. However, we only need the first k elements of eigenvectors and k is much less than the value of n. Hence, in this case we need to find a way to conveniently compute the first k eigenvectors. Also in scipy, the sparse.linalg module has an **eigs** function which can help us do this task beautifully.

It's a specific method for finding k eigenvalues and eigenvectors of the square matrix.

After we get the first k eigenvectors we transpose it from shape **k,n** to shape **n,k** and get the real value of them. Finally, we need to normalized it to make the computation follow more stable. The code is below:

```python
def get_U(lap, k):
    s = time.time()
    lap = lap.astype('float64')
    _, first_k = eigs(lap, k, sigma=0)
    U = first_k.real
    x = np.linalg.norm(U)
    U = U / x
    t = time.time()
    print('Time for get U:{}'.format(t - s))
    return U
```

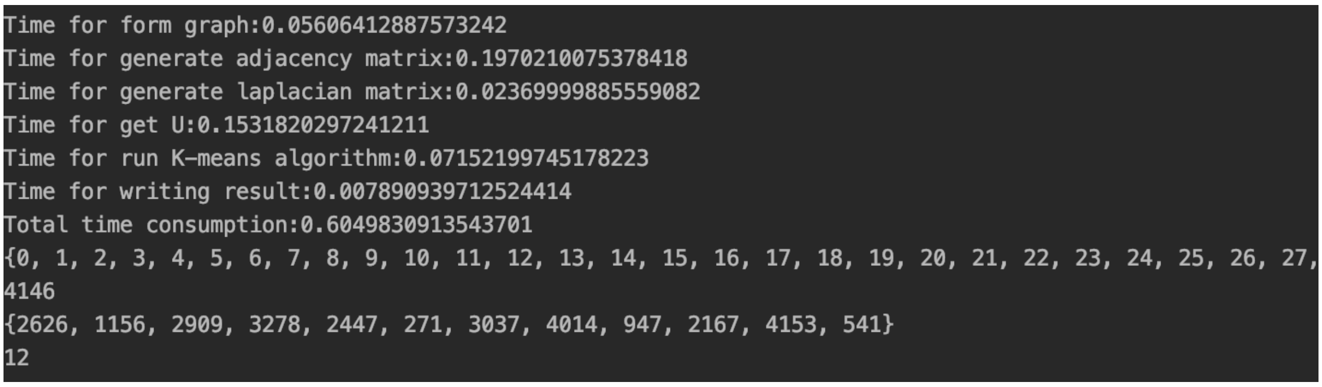
3.4 K-means clustering

K-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. In this project, instead if implementing a k-means method by myself, I used the KMeans function from sklearn. cluster. Sklearn is a simple and efficient tool for predictive data analysis. The code for calling this function is below:

```python
def k_means(data, k):
    s = time.time()
    kmeans = KMeans(n_clusters=k, algorithm='auto')
    kmeans.fit(data)
    t = time.time()
    print('Time for run K-means algorithm:{}'.format(t - s))
    return kmeans.labels_
```

4. Experimental results

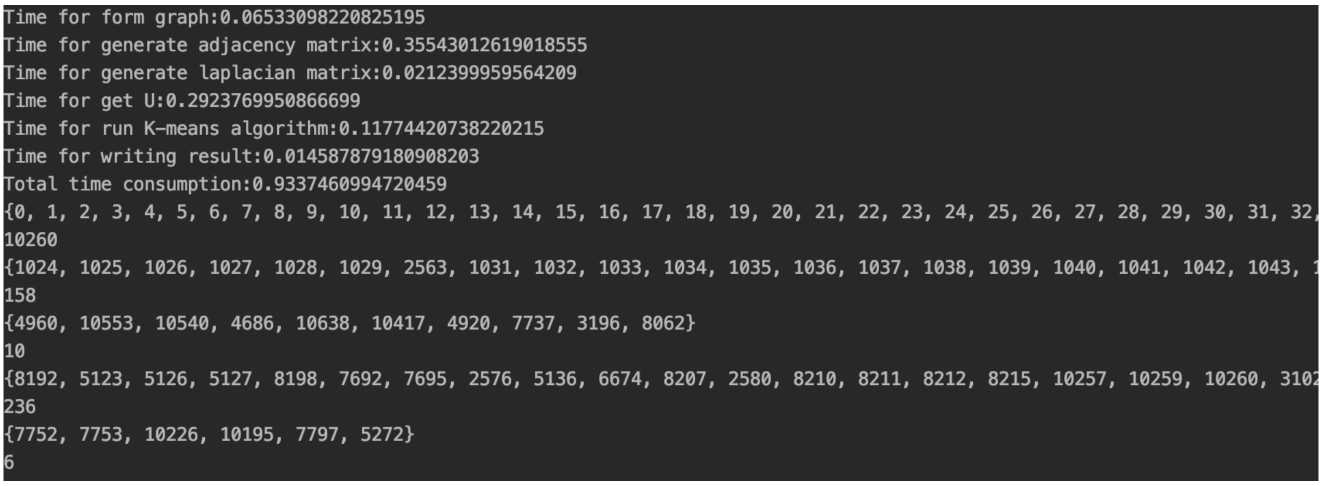
I tried several different data structures and implementing methods. Finally got a fast and stable implement section for solving this problem. For example, the first graph **ca-GrQc** which has 4158 vertices and 13428 edges, it only takes less than 0.7 second to come up with its partition. We can have a look at it's operation result:



```
Time for form graph:0.05606412887573242
Time for generate adjacency matrix:0.1970210075378418
Time for generate laplacian matrix:0.02369999885559082
Time for get U:0.1531820297241211
Time for run K-means algorithm:0.07152199745178223
Time for writing result:0.007890939712524414
Total time consumption:0.6049830913543701
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
4146
{2626, 1156, 2909, 3278, 2447, 271, 3037, 4014, 947, 2167, 4153, 541}
12
```

Operation result of partitioning of graph **ca-GrQc**

As for other graphs, the second graph Oregon-1 takes about 0.9 second to analyzing it:



```
Time for form graph:0.06533098220825195
Time for generate adjacency matrix:0.35543012619018555
Time for generate laplacian matrix:0.0212399959564209
Time for get U:0.2923769950866699
Time for run K-means algorithm:0.11774420738220215
Time for writing result:0.014587879180908203
Total time consumption:0.9337460994720459
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
10260
{1024, 1025, 1026, 1027, 1028, 1029, 2563, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043,
158
{4960, 10553, 10540, 4686, 10638, 10417, 4920, 7737, 3196, 8062}
10
{8192, 5123, 5126, 5127, 8198, 7692, 7695, 2576, 5136, 6674, 8207, 2580, 8210, 8211, 8212, 8215, 10257, 10259, 10260, 3102
236
{7752, 7753, 10226, 10195, 7797, 5272}
6
```

The last three graph is much bigger than the first one. After several times code optimizations, it takes about 5000 seconds to compute the graph soc-Epinions, 2000 seconds to compute graph web-NotreDame and 1000 seconds for graph roadNet-CA. The specific partitioning result can be reviewed in the *output.txt files.

5. Conclusion

It's really interesting and rewarding to do a programming project like this. At first, I think it's easy to implement such a method to divide the graph. However, since the data amounts of our graph especially the last two graphs are pretty huge. A bad implementation will lead to a really long time consuming in running the algorithm. The first section of my code is just a bad one, I used the simple and normal data structure to store the matrices, it will cost a huge memory when initialize its object in computer, even for dealing with the last two graphs, the program will report an Memory error directly, which means my computer do not have enough ram memory to solve this problem. Then, I started to optimize my code, using as advanced and suitable data structure as possible. After several times try, my code got an excellent running speed and acceptable space requirement. Through this project, I have a deeper understanding of graph partitioning theory as well as the Laplacian matrix. Moreover, my skills in using Python language to solve the practical problems has been further improved. Thanks to all the teachers and TAs, it's such a great course and I learned a lot with your help.

6. References

1. https://www.geeksforgeeks.org/sparse-matrix-representation
2. https://en.wikipedia.org/wiki/Laplacian_matrix
3. Bojan Mohar. The Laplacian Spectrum of Graphs. University of Ljubljna.
4. https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors
5. https://en.wikipedia.org/wiki/Graph_partition
6. Konstantin Andreev, Harald Racke. (2006). Balanced Graph Partitioning. Theory of Computing Systems.