A Comparison of Graph Clustering Algorithms

A Project Report submitted as a requirement for 6th Semester Minor Project (CS 3404) in Integrated Dual Degree course in the Department of Computer Engineering.

Done by,

Abhijith V Mohan
10400EN001
IDD Part-III
Computer Science

under the guidance of Professor Bhaskar Biswas

CERTIFICATE

This is to certify that **Abhijith V Mohan**, roll number 10400EN001, has worked on the topic “A comparison of Graph clustering algorithms” under my direct supervision and guidance, the finding of which have been incorporated in this report. His work has been found satisfactory and is approved for submission

Dr Bhaskar Biswas,
Assistant Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology-(BHU), Varanasi.

Abstract

Over the past decade there has been a growing public fascination with the complex "connectedness" of modern society. This connectedness is found in many places: in the rapid growth of the Internet and the Web, in the ease with which global communications now takes place, and the ability of news and information as well as epidemics and financial crises to spread around the world with surprising speed and intensity. These are phenomena that involve networks, incentives, and the aggregate behaviour of groups of people, on the links that connect us and the ways in which each of our decisions can have subtle consequences for the outcome of everyone else.

Two graph clustering algorithms- the Markov Clustering and K-Cliques algorithm are compared in this project. The MCL algorithm is based on random walks while the K-clique algorithm is based on finding the maximal union of adjacent cliques of size K and classifying them as a cluster.

Contents

Acknowledgements

I would like to thank:
My Parents for supporting me through my education,
Professor Bhaskar Biswas for mentoring, support and encouragement.

Introduction

The modern science of networks has brought significant advances to our understanding of complex systems. One of the most relevant features of graphs representing real systems is community structure, or clustering, i. e. the organization of vertices in clusters, with many edges joining vertices of the same cluster and comparatively few edges joining vertices of different clusters. Detecting communities is of great importance in sociology, biology and computer science, disciplines where systems are often represented as graphs. This problem is very hard and not yet satisfactorily solved, despite the huge effort of a large interdisciplinary community of scientists working on it over the past few years.

Why do clustering?

Clustering or community detection is not just an academic curiosity but can have concrete applications. Some of the applications might be:

- Clustering Web clients who have similar interests and are geographically near to each other may improve the performance of services provided on the World Wide Web, in that each cluster of clients could be served by a dedicated mirror server. (Krishnamurthy and Wang, 2000)
- Identifying customer clusters with similar interests allows online retailers (such as flipkart.com) to set up efficient recommendation systems that can better guide the consumers through the list of items of the retailer and enhance their business opportunities. (Reddy et al, 2002)
- Ad-hoc wireless networks usually have no centrally maintained routing tables. Clustering the nodes enables one to route through efficient paths and at the same time have compact routing tables. (Streenstrup, 2001)
- Also, vertices with a central position in the clusters may have an important function of control and stability within the graph. For example, a central node in a social network cluster might be instrumental in the opinion formation and might exert a lot of influence on the cluster.
- In large parallel computing grids, the computing nodes can be clustered according to geographical proximity, ease of communication etc so that the processes belonging to the same task can be given to the nodes in the same cluster to minimise the overhead of communications.

As is evident from the few examples above, graph clustering finds applications in such wide ranging domains as economics, marketing, computer science, biology, etc. The datasets for which clustering should be applied tend to be huge. Thus, the importance of efficient algorithms for graph clustering cannot be overstated.

Background information

This section includes some background information such as some necessary definitions, terminology and metrics that quantify and qualify clusters in a graph.

A basic definition of cluster/community

The first problem in graph clustering is to look for a quantitative description of the word cluster or community in a graph. No definition is universally accepted. As a matter of fact, the definition usually depends on the specific system at hand and the applications one has in mind. However, we can arrive at one notion intuitively: there must be more edges “inside” the community than edges linking vertices of the community with the rest of the graph. This is the reference guideline at the basis of most definitions of community.

A simple definition of a cluster can be arrived at as follows in the next paragraphs.

Let $\mathcal{C}$ be a subgraph of a graph $\mathcal{G}$ with $|\mathcal{C}| = n_c$ and $\mathcal{G} = n$ vertices, respectively. We define the internal and

external degree of vertex $\boldsymbol{v} \in \boldsymbol{C}$, $\boldsymbol{k}_v^{int}$ and $\boldsymbol{k}_v^{ext}$, as the number of edges connecting v to other vertices of $\boldsymbol{C}$ or to the rest of the graph, respectively. If $\boldsymbol{k}_v^{ext} = 0$, the vertex has neighbours only within $\boldsymbol{C}$, which is likely to be a good cluster for v; if $\boldsymbol{k}_v^{int} = 0$, the vertex is disjoint from $\boldsymbol{C}$ and it should better be assigned to a different cluster. The internal degree $\boldsymbol{k}_{int}^{C}$ of $\boldsymbol{C}$ is the sum of the internal degrees of its vertices. Likewise, the external degree $\boldsymbol{k}_{ext}^{C}$ of $\boldsymbol{C}$ is the sum of the external degrees of its vertices. The total degree k^C is the sum of the degrees of the vertices of $\boldsymbol{C}$. By definition, $\boldsymbol{k}^C = \boldsymbol{k}_{int}^{C} + \boldsymbol{k}_{ext}^{C}$.
The intra-cluster density $\delta_{int}(C)$ of the subgraph $\boldsymbol{C}$ is defined as the ratio between the number of internal edges of $\boldsymbol{C}$ and the number of all possible edges, i.e.

$$\delta_{int}(C) = \frac{no.of\ internal\ edges\ of\ C}{n_c(n_c-1)/2}$$

Similarly, the inter-cluster density $\delta_{ext}(C)$ is the ratio between the number of edges running from the vertices of C to the rest of the graph and the maximum number of inter-cluster edges possible, i.e.

$$\delta_{ext}(C) = \frac{no.of\ inter\ cluster\ edges\ of\ C}{n_c(n-n_c)}$$

For $\boldsymbol{C}$ to be a community, $\delta_{int}(C)$ be expected to be appreciably larger than the average link density $\delta(\mathcal{G})$ of $\mathcal{G}$ (which is the ratio between the number of edges of G and the maximum possible number of edges n(n-1)/2) and $\delta_{ext}(C)$ to be much smaller than $\delta(\mathcal{G})$. Thus, the basic definition of a cluster or community can be arrived at as:
A cluster/community is a subgraph of a graph which has an intra cluster link density much higher and inter cluster link density much lower than the average link density of the graph.

Partitions and Covers

A *partition* is a division of graph in clusters, such that each vertex belongs to one cluster. A division of graph into overlapping (or fuzzy) communities is called *cover*. Of the 2 algorithms considered in this project, the MCL algorithm generates a partition and K-clique algorithm generates a cover of the graph.
There are also intermediate approaches such as hierarchical clustering which generates a dendrogram of the graph. Each node belongs to a cluster. Each of these clusters may themselves be part of bigger clusters thus forming a tree of clusters.

Quality Metrics

There are several Quality metrics that are used to quantitatively measure the quality of the clusters generated from various algorithms. Of these metrics, the most popular one is modularity, which was proposed by Girvann & Newman in 2004**.**
Modularity is based on the idea that a random graph is not expected to have a cluster structure, so the possible existence of clusters is revealed by the actual density of edges in a subgraph and the density inw would expect to have in the subgraph if the vertices of the graph were randomly attached. Modularity is calculated by the formula

$$Q = \sum_{c=1}^{n_c}\left[\frac{l_c}{m} - \left(\frac{d_c}{2m}\right)^2\right]$$

Here, n_c is the number of clusters, l_c the total number of edges joining vertices of module c and d_c the sum of the degrees of the vertices of c.

Implementation Details

The programs in this project were implemented in the Python programming language. The following 3rd party libraries were used for the implementation:

- **Networkx**, a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
- **Numpy**, a Python package that provides fast mathematical operations over n-dimensional arrays
- **Matplotlib**, an object oriented plotting library for Python.
- **Scipy**, a scientific computing package for Python.

The programs were run on an Intel Xeon quad core 3.0 GHz CPU with 16 GB RAM.

Markov Cluster Algorithm

The Markov Clustering algorithm, also known as MCL algorithm, was first proposed by Van Dongen in his Phd thesis in 2000. This method simulates a peculiar process of flow diffusion in a graph.
One starts from the transfer matrix of the graph T. The element T_{ij} of the transfer matrix gives the probability that a random walker, sitting at a vertex j, moves to i. The sum of the elements of each column of T is one.Each iteration of the algorithm consists of two steps. In the first step, called *expansion*, the transfer matrix of the graph is raised to an integer power e (usually e=2). The entry M_{ij} of the resulting matrix gives the probability that a random walker. starting from vertex j, reaches i in p steps. The second step, which has no physical counterpart, consists of raising each entry of the matrix M to some power i, where i is real valued. The operation, called *inflation*, diminishes weights between pairs of vertices with low values of diffusion flow which are likely to be in different clusters. The resulting matrix is column normalized so that it remains a valid transfer matrix. The process repeats for a fixed number of iterations or until a steady state is reached. A stable matrix is obtained which represents a graph whose connected components are the communities of the original graph.
As of now, MCL is one of the most used algorithms in bioinformatics. Its simplicity is the main reason for its success. Due to the matrix multiplication, the algorithm scales as $O(n^3)$. However, it lends itself naturally to parallelization and hence the actual runtime can be improved by using parallel computing.

Algorithm

1. T_{nxn}: weight matrix of a weighted undirected graph.
   e: expansion parameter
   i: inflation parameter
2. Add self loops to each node, i.e.

$$T_{ii} = 1, for 1 \leq i \leq n$$

3. Column normalize T, i.e.

$$T_{ij} = \frac{T_{ij}}{\sum_{i=1}^{n} T_{ij}}$$

4. Expand by taking the e^{th} power of the matrix.

$$T = T^e$$

5. Inflate by taking i^{th} power of each element of the matrix.

$$T_{ij} = T_{ij}^i$$

6. Repeat steps 4 and 5 until a steady state is reached.
7. Interpret cluster from the resulting matrix by interpreting it as a graph and finding the connected components.

Code

```
import numpy as np

def norm_mat(a):
    return a / a.sum(axis=0)[:np.newaxis]

def inflate(a, i):
    return norm_mat(a ** i)

def expand(a, e):
    return np.linalg.matrix_power(a, e)

def mcl_clusterize(a, e, i, steps=20):
    a = norm_mat(a)
    for step in range(steps):
        a = expand(a, e)
        a = inflate(a, i)
    return a

def interpret_clusters(a):
    clusters=[]
    flag=[False for i in range(len(a))]
    for row in range(len(a)):
        clusternodes=[]
        for col in range(len(a[0])):
            if not flag[col] and a[row][col]!=0:
                clusternodes.append(col+1)
                flag[col]=True
        if clusternodes:
            clusters.append(clusternodes)
    return clusters
```

Observations

For n=100, e=2, i=2

```
{4 11 36 49 54 71 80}
{5 42 76}
{2 7 8 10 16 17 18 22 31 44 48 61 77 85 90 91 96 97}
{6 9 12 14 15 19 21 23 24 25 26 27 29 30 32 35 38 39 40 41 43 46 47 50 51 55 56 58 59 60 62
68 78 81 83 87 88 92}
{13 75}
{20}
{1 3 28 33 52 53 63 64 65 67 69 70 72 73 74 86 94 95 98}
{34 57}
{37 89}
{45}
{66}
{79 82}
{84}
{93}
{99}
{100}
```

There are 16 clusters. Out of these, there are 4 clusters with at least 5 nodes and 3 with at least 10.
For n=100, e=2, i=5

```
{4 11}
{5 76}
{8}
{9}
{2 10 16 17 31 44 48 77 91 96 97}
{6 12 14 15 19 23 24 25 26 27 29 30 32 35 38 39 40 41 43 46 47 50 51 55 56 58 59 60 62 68
78 81 83 88 92}
{13}
{7 18}
{20}
{21}
{22}
{33 53 64 72 74 86}
{34}
{36}
{37 89}
{42}
{45}
{49 54 80}
{1 3 28 52 65 67 69 70 73 94 95 98}
{57}
{61}
{63}
{66}
{71}
{75}
{79}
{82}
{84}
{85}
{87}
{90}
{93}
{99}
{100}
```

There are 34 clusters. Out of these, 4 have at least 5 nodes and 2 have more than 10 nodes.
With an increase in the value of the inflation parameter, the cluster granularity increases. This is because with higher values of inflation parameter, the smaller probability transitions are diminished to zero.

K-Clique Clustering (Clique percolation)

A clique in a graph G is a complete subgraph of G; that is, it is a subset S of the vertices such that every two vertices in S are connected by an edge in G. A maximal clique is a clique to which no more vertices can be added.
The clique percolation method builds up the communities from k-cliques, which is a clique consisting of k nodes Two k-cliques are considered adjacent if they share k − 1 nodes. A community is defined as the maximal union of k-cliques that can be reached from each other through a series of adjacent k-cliques. Such communities can be best interpreted with the help of a k-clique template (an object isomorphic to a complete graph of k nodes). Such a template can be placed onto any k-clique in the graph, and rolled to an adjacent k-clique by relocating one of its nodes and keeping its other k − 1 nodes fixed. Thus, the k-clique communities of a network are all those sub-graphs that can be fully explored by rolling a k-clique template in them, but cannot be left by this template.
Since even small networks can contain a vast number of k-cliques, the implementation of this approach is based on locating the maximal cliques rather than the individual k-cliques. Thus, the complexity of this approach in practice is equivalent to that of the NP-complete maximal clique finding, (in spite that finding k-cliques is polynomial). This means that although networks with few million nodes have already been analyzed successfully with this approach, no prior estimate can be given for the runtime of the algorithm based simply on the system size.

Algorithm

1. Find all the k-cliques in the graph.
2. For every k-clique, find the maximal union of adjacent k-cliques. (Two k-cliques are adjacent if they share k-1 nodes).
3. All the nodes belonging to such a union belong to a cluster.

Code

```python
from collections import defaultdict
import networkx as nx

def k_clique_communities(G, k, cliques=None):
    if cliques is None:
        cliques = nx.find_cliques(G)
    cliques = [frozenset(c) for c in cliques if len(c) >= k]
    # First index which nodes are in which cliques
    membership_dict = defaultdict(list)
    for clique in cliques:
        for node in clique:
            membership_dict[node].append(clique)
    # For each clique, see which adjacent cliques percolate
    perc_graph = nx.Graph()
    for clique in cliques:
        for adj_clique in _get_adjacent_cliques(clique, membership_dict):
            if len(clique.intersection(adj_clique)) >= (k - 1):
                perc_graph.add_edge(clique, adj_clique)

    # Connected components of clique graph with perc edges
    # are the percolated cliques
    for component in nx.connected_components(perc_graph):
        yield(frozenset.union(*component))

def _get_adjacent_cliques(clique, membership_dict):
    adjacent_cliques = set()
    for n in clique:
        for adj_clique in membership_dict[n]:
            if clique != adj_clique:
                adjacent_cliques.add(adj_clique)
    return adjacent_cliques
```

Observations

An optimum value of k must be known in order to generate a good clustering. Too small values of k will lead to large sized clusters with little or no relevance between the nodes. Too large values of k will lead to no clusters.
Also, it must be noted that K-clique percolation generates a cover rather than a partition, so some nodes may not be in any cluster and some other nodes may be in more than one cluster.

For n=100, k=10, 2 clusters are formed.

```
{2 6 10 77 16 17 22 89 90 91 80 31 96 97 38 44 48 49 50 54 56 61 95}
{6 12 13 14 15 19 23 24 25 26 27 29 30 32 35 38 39 40 41 43 46 47 50 51 55 56 58 59 60 62
68 75 78 81 83 88 92}
```

Conclusion

The Markov Clustering and K-clique percolation algorithms were implemented for the blogger dataset. A comparison between the two algorithms is drawn in the following table.

Markov Clustering	**K-Clique Percolation**
Generates a partition	Generates a cover
Based upon random walks	Based upon cliques
O(n^3) time complexity	NP-complete
Highly scalable, simple implementation	Because of it being NP-complete, unsuitable for large scale applications

The 9614 node blogger dataset contains of 5 clusters with atleast 100 nodes in them and 2 huge clusters with more than 1000 nodes in each of them (3108 & 2011 nodes respectively). Thus the cluster structure in the dataset mainly consists of 2 clusters.

Bibliography

1. Van Dongen, S. (2000), “Graph Clustering by Flow Simulation”. PhD Thesis, University of Utrecht, The Netherlands.
2. Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek, “Uncovering the overlapping community structure of complex networks in nature and society” ,Nature 435, 814-818, 2005, doi:10.1038/nature03607
3. Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008
4. Eric Jones, Travis Oliphant, Pearu Peterson and others. “SciPy: Open Source Scientific Tools for Python”, 2001. http://www.scipy.org.