Detecting Communities through Connected Components in Large Graphs using MapReduce

Mukund Seethamraju
Department of Undergraduate Studies
Indian Institute of Science
Email: mukund.seethamraju@gmail.com

***Abstract*—Label propagation algorithm is heavily used in finding connected components in complex and large networks. It is mainly based on calculating the positive similarity between the nodes of the graph. But, asserting the fact that negative similarities also contribute to the problem of finding connected components, a new ssl based algorithm has been proposed by Wei Liu and Tongtao Zhang [4]. The aim of this course project is to implement this algorithm on large graphs using MapReduce to find the connected components in the graph.**

***Keywords*—Graph Mining; Strongly Connected Component; LPA; BLP; MapReduce**

I. Introduction

In examining complex networks, such as social networks, community structure is characterised as a node of networks that can be grouped into specific sets of nodes in that each set of nodes is connected internally. Within a community network, the vertexes are completely connected to one another. This idea comes from the general principle that different pairs of nodes are more likely to be connected if they are members of the same community than nodes that do not share the same communities. SO connected components reveal communities in social network graphs. In this project, due to the time constraints, SSL based BLP algorithm has been studied but is not implemented. Another BLP algorithm which is based on Strongly Connected Components(SCC) is studied and will be implemented using MapReduce to find communities.

With increasing social media, a lot of data is available online which has yet to be explored. In this project, I have chosen Twitter graph which was crawled on entire Twitter Site in 2009 [3]. This is an user-follower graph with 41.7 million user profiles i.e. users and 1.47 billion social relations i.e. connections.

II. Algorithms Studied

A. Girvan-Newman Algorithm

Grivan-Newman Algorithm [5] commonly used hierarchical method for community detection. The algorithm works using edge betweenness centrality measures. It does so by calculating the betweenness measure of all nod es in the network and removing the edge with the highest betweenness. After this any edges affected by this removal have their betweenness recalculated and the previous step is repeated. This is done until there are no edges left and so only the communities themselves are left. This algorithm finally gives a dendogram. This method of community detection is quite simple and easily implemented. Unfortunately it is quite slow with a time of $O(m^2n)$. This is because the betweenness of the graph is recalculated multiple times. Not doing so can result in large errors so it is unavoidable.

Algorithm 1 The Girvan-Newman algorithm

Input : A connected graph G = (V, E)
Output: A hierarchical clustering of the graph G

```
while While condition do
    calculate betweenness values for edges e ∈ E;
    delete edge e with highest betweenness;
    if graph splits in more components then
        build a new cluster for each component;
        add each new cluster as subcluster to its parent cluster;
    end
end
return cluster hierarchy
```

B. BGLL Algorithm

BGLL Algorithm (Blondel et Al, 2008) is quite suitable for large networks because it displays linear complexity. However, it is a greedy algorithm that is based on an increase in modularity (which is defined as the fraction of the edges that fall within the given groups minus the expected such fraction if edges were distributed at random). This is done in two major phases. The first phase starts by initializing n communities where each node characterizes exactly one community and then moving each node to a neighbours community. Then the modularity is increased until it can no longer be increased any further by merging. From here, the second phase involves creating a new network with new nodes as communities that were formed in the first phase. The weights between new nodes then become the weights between the original communities. The algorithm then goes back to the first phase to iterate until the modularity cannot be increased any further.

Algorithm 2 BGLL algorithm

Data: undirected, connected, weighted graph inputGraph, threshold t, g ← inputGraph

```
while g can be contracted do
    // Phase 1: refine intermediate solution by moving vertices between clusters
    for vertices v_x do
        c[v_x] ← i // all vertices are placed in their own cluster
    end
    while totalIncrease > t do
        for vertices v do
            best ← v
            maxDeltaQ ← 0
            totalIncrease ← 0
            for neighbors w of v do
                deltaQ ← deltaQ from move of c from c[v] to c[w]
                if deltaQ ← maxDeltaQ then
                end
                maxDeltaQ ← deltaQ
                best ← w
            end
            c[v] ← c[best]
            totalIncrease ← totalIncrease + maxDeltaQ
        end
    end
    // Phase 2: contract graph as induced by partition from phase 1 g ← contract(g,c[])
end
```

C. Bi-Directional Label Propagation Algorithm

know it is easy to get unlabeled data easily than getting data labeled manually. So, Semi-Supervised Learning based BLP algorithm will be useful in practical cases. In this algorithm, initially k-NN graph is constructed using unsupervised learning techniques and then graphs are learnt using the available label information. Though the algorithm ensures consistent labeling along postitive edges, it doesn't ensure consistent labeling along negative edges. The algorithm as described in the paper[4] is presented below. Since, the proof and analysis of the algorithm involves heavy math, the theorems are skipped here.

Algorithm 3 Bidirectional Label Propagation (BLP)

Step 1: Construct a k-NN graph G(V,E,W) upon $\{X = x_1, x_2, ...x_n\} \subset R^d$. The first l examples are labeled as $y_i \in \{1, ...C\}$ Use weighted adjecency matrix to set W. Set a class indicator matrix $Y \in R^{l \times C}$ where $Y_{iC} = 1$ if $y_i = c$ and $Y_{iC} = -1$ otherwise.

Step 2: Transductive inference:

for $c = 1, 2, ..., C$ **do**

compute K^c from W,

compute $f^*_{v,c}$ with K^c and $Y_{.c}$

$F^*_v = [f^*_{z1}, ...f^*_{zC}]$

Inductive Inference:

for $c = 1, 2, ..., C$ **do**

compute $K^C_{z.}$

compute f^*_{zc} with K^c, $K^C_{z.}$ and $Y_{.c}$

Step 3: For an unlabeled example x_i $(l = 1 \leq i \leq n)$,predict its labels by $y(x_i) = \underset{1 \leq c \leq C}{\operatorname{argmax}} [F^*_u]_{i-l,c}$.

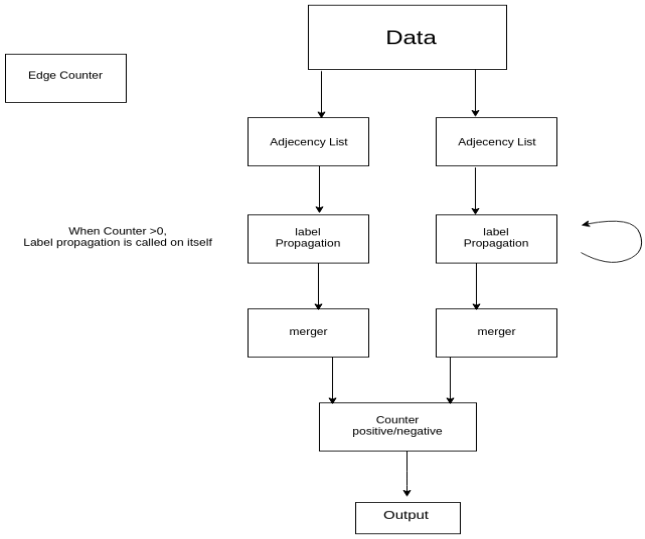
For a novel test example z, predict its label by $y(z) = \underset{1 \leq c \leq C}{\operatorname{argmax}} [F^*_z]_{1,c}$

III. Label Propagation with MapReduce

strongly connected component (SCC) of a directed graph is defined as a maximal subset of vertices in which there is a directed path from any vertex to any other. In social network graphs, SCC resembles groups of users who share a lot in common, SSC can be used to find the communities in graphs. In this algorithm, BLP algorithm will be used to find SCC.

To search for the strongly connected components look at the Bi-Directional Label Propagation Algorithm. This algorithm works in 4 steps. The first is a positive label propagation (PLP) step. Here a node is given a label depending on its value. Then its children are given labels the same way. If a child node has a label greater than that of its parent node, the label assigned is changed to the same as the parent node. This is repeated throughout the network. After this, in step 2, the original network is reversed. This means that all directed edges are turned the other way. On this new reversed network negative label propagation (NLP) is performed. This is the same as the PLP however the labels assigned are given a negative value. Following this, in the final step, the labels of the nodes are checked. The nodes that have the same positive and negative labels make up strongly connected components in the network.

Whilst the algorithm is split into four clearly defined steps, the implementation of these steps is slightly different. I created a workflow diagram to map out exactly each step in the process. This can be seen in Figure. As the diagram indicates, the Data first is sent through the Adjacency List Algorithm, where positive and negative are generated, and then to Label Propagation to Merger to the Positive/Negative Counter Combiner, where the positive and negative labels are put together, and then finally the output is emitted. The Label Propagation code is set so that the Label Propagation is called on itself when the Counter > 0.



A. Edge Counter

The Edge Counter is very similar to word-count example which we have studied. In this the total number of lines is calculated As the data contains one edge per line, this is an accurate count of how many edges there are in the data. In mapper, the input key value pairs are user-id and follower-id respectively. For every <key, value> pair mapper outputs <key = "no. of edges", 1>. This is the input for reducer and reducer returns the sum of keys of its input. Thus we will get the total number of edges in the graph.

B. Adjacency Lists

the Adjacency List Mapper, all of the data is placed into the array. The data consists of rows of user-follower pairs. Each row is in a text format with the follower separated from the user by space. Each row is parsed and the user-follower pair is sent to the Reducer. Each row only emits one user and its follower; other rows can also contain this user-id with other followers. The key and value sent to the Reducer are the user and list of its followers respectively. Hence, a user with a list of followers will arrive at the Reducer.
The Adjacency List reducer takes the key (user) and value (list of followers), and puts all the followers into a string. From here, the actual list of followers for each user is outputted. The Reducer emits the userid as a key, a positive label in the form +userid, and an adjacency list formed from the list of followers that it receives from the Mapper. The adjacency list is separated from the label by a space, and a comma separates each of the followers within the list.

C. Adjacency List Reversed Mapper and Reducer

In the Adjacency List Reversed Mapper, the pair being emitted is reversed. The follower is emitted as a key and the value is its corresponding list of users. The Adjacency List Reversed Reducer follows the same logic as the Adjacency List Reducer, but instead of positive labels, negative labels are assigned in the form of userid. The adjacency list developed through these steps is characterized by a list of corresponding users for a follower, rather than followers of one user in the output.

D. Label Propagation

list is given as the input for label propagation. The Mapper emits the userid, positive or negative label, the adjacency list, and the original label for each node. Then, for each node from its adjacency list, the Mapper emits the userid and label as the key and value. The counter should be accessible from the Reducer, and it incremented whenever one of the nodes is changed. To detect a change, an extra field is emitted from the Mapper. This is the original label so that the reducer is able to detect the change. After the Label Propagation performs its first iteration, it enters the while loop, which runs only when the counter > 0. It only stops running when no nodes are changed, or rather when nothing else is added to the counter. Each iteration uses the output of the previous one as its input, and generates the output.

E. Merger

The Merger Mapper takes user-label-adjacency list lines from the output of Label Propagation and emits the label and user as key, and a value to group them later in the Reducer through the means of a label. The Merger Reducer essentially groups users by their label and outputs this label as the key and a list of corresponding users with a , separator as the value.

F. Combiner

The combiner mapper takes the positive and negative output generated from the merger. For each row, there is a list of users separated by , as a value and positive or negative labels as a key. There is an array that holds these nodes (users). The while loop looks through the array and while there are values present in this array, it looks at each token and sends the tokens to the Combiner Reducer with the corresponding user. For each user in the list, it takes a positive/negative label and emits its absolute value as the key, and the userid as the value.
In the Combiner Reducer, the absolute value label is the key, and the user value is checked if it appears twice for the label. If so, it is added to this labels connected nodes graph. In other words, some nodes from the Mapper are sent to the reducer twice, once with the positive label and once with the negative label. For those that do display this trait, it indicates they form a connected map. The first if statement checks if any of the nodes for this label appeared twice . If it does, the Reducer writes this as a result and outputs it. For each user, the label is emitted together with the value as the user. If there is no match, then the Reducer will not output anything.

ACKNOWLEDGMENT

I would like to thank the Professors and TA for helping throughout the course.

REFERENCES

[1] Lu Lv and Lei Xie, *Enumerate Strongly Connected Components of Large-scale Graph with MapReduce*, Intelligent Telecommunications Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing,100876

[2] Iwata, Tomoharu, and Kevin Duh. "Bidirectional Semi-supervised Learning with Graphs." Machine Learning and Knowledge Discovery in Databases Lecture Notes in Computer Science (2012): 293-306. Web.

[3] Rappa, Michael, and Paul Jones. WWW 2010: Proceedings of the 19th International Conference on World Wide Web, Raleigh, North Carolina, USA. New York, NY: ACM, 2010. Web.

[4] Liu W, Zhang TT. Bidirectional label propagation over graphs. Int J Software Informatics, Vol.7, No.3 (2013): 419433. http://www.ijsi.org/1673-7288/7/i168.htm

[5] Girvan, M. "Community Structure in Social and Biological Networks." Proceedings of the National Academy of Sciences of the United States of America 99.12 (2002): 7821-826. JSTOR. Web. 01 May 2016.