CS698 Project-Community Detection Algorithms

Liwen Dai
University of Waterloo
Computer Science
ID: 20552153
l7dai@uwaterloo.ca

Ying Yu
University of Waterloo
Computer Science
ID: 20661221
y288yu@uwaterloo.ca

ABSTRACT

The goal of our project is to implement two graph algorithms using Spark GraphX in Big Data platform. As a powerful data processing framework, Spark has been adopted by a wide range of big enterprises such as Netflix, Yahoo, to develop software products. Spark not only inherits the characteristics of map-reduce but provides many features that cannot be performed by MapReduce. We decide to learn an important component of Spark, GraphX, which outperforms most of the graph systems with flexibility and fault tolerance. In this course, we have learned the basic infrastructure of both map-reduce and spark in details. To detect community structure in graphs, many researchers have studied the problem and they have achieved great progress. Our project chooses two popular graph algorithms for community detection and implements them in both local environment and Altiscale using unweighted graph data from twitter and youtube. All of our work can be reached in github reposities. [9]

Keywords

Community Detection, Spark GraphX, Big Data

1. INTRODUCTION

With the size of network becoming larger and larger, study the graph networks can provide us a deeper understanding of the complex systems and get some new information from the network. Each active node in the network is represented as an entity. Community detection tries to find a "good" partition for a certain graph. From the previous researches, many networks contain nodes that share common characteristics or more dense connection than the rest of the network, we call these nodes "communities". Modularity remains one of the most popular measures in community detection, even though it is flawed and exists some problems. Many algorithms have been suggested to optimize modularity. The original algorithm created a full dendrogram and used modularity to decide on a cutting point. It was quite slow, running in $O(n^2m)$, where n is the number of nodes and m the number of links. After that, many algorithms were introduced to optimize modularity, such as spectral methods and greedy methods, and many other methods. One of the fastest and most effective algorithms is the Louvain algorithm [10], which is believed to be running in O(m). It has been shown to perform very well in comparative benchmark tests [12, 4]. The algorithm is largely independent of the objective function to optimize, and as such has been used for different methods. One of us choose to implement this algorithm. It has been implemented using C++ language by the first published author Vo and al. We will describe the steps and some modifications about the algorithm in the later section. The original algorithm runs in linear time with respect to the number of edges, but the running time depends on the choosing start point. The paper written by V. A. Traag[8] optimize this algorithm by choosing random neighbor algorithm and achieve a time complexity of O(n log k). They have experimented on benchmark tests and some real networks. The results show that this minor adjustment did reduce the running time, without losing much quality. From the modularity value is especially obvious. The common types of networks include social networks, information networks (e.g., hyperlinks between webpages on the World Wide Web), infrastructure networks (e.g., transportation routes between cities), and biological networks (e.g., metabolic interactions between cells or proteins, food webs, etc.). We decide to use the social media network considering its large volume and complexity. Since we want to test the efficiency of the algorithms in Altiscale. Given a network representation of a system, it can be useful to apply a coarse-graining technique in order to investigate features that lie between those at theâĂIJmicroscaleâĂİ. In social networks, communities can reveal groups of people with common interests, places of residence, or other similarities. The authors of [3] derived a generalization of modularity maximization, a popular clustering method for static networks, to multilayer networks. Modularity is a function that measures the âĂIJqualityâĂİ of a network partition which will be addressed in section 2. In our project, we choose two algorithms in community detection, Girvan Newman and Louvain Algorithm to implement. We address two main issues: (1) the choice of two algorithms, and (2) how to implement them in a big data platform. We discuss the first issue in section 2 and the second issue in section 3. In sections 4 and 5, we give an overview of existing results and how we evaluate the results. We conclude in section 6.

2. ALGORITHMS

2.1 Girvan-Newman Algorithm

Girvan-Newman (GN) Algorithm [2] is a method to compute betweenness of edges on a graph. Betweenness of an edge (u, v) is defined as the number of pairs of vertices x, y such that the edge (u, v) lies on the shortest path between x and y. If there are more than one shortest path between x and y, edge (u, v) gets a credit, or a fraction of the betweenness for the x, y pair. A betweenness can be treated as an approximate measure of distance, and a high betweenness on an edge (u, v) suggests that u and v are in different communities.

GN algorithm visits each vertex x once and computes the number of shortest paths from x to every other vertex that go through each edge. For each x, GN performs two steps. Step 1 is a breath-first search (BFS) from root x, where a directed acyclic graph (DAG), the BFS tree is constructed. During the BFS process, each vertex y is marked with $depth(y)$, the number of edges on a shortest path (distance) between x and y, and $nsp(y)$, the number of shortest paths between x and y. For a directed edge (u, v) on the BFS tree, u is a parent of v and v is a child of u. Obviously, $depth(u) + 1 = depth(v)$. Note that the a child can have more than one parent, and a parent can also have multiple children. Moreover, the original graph G can be either directed or undirected, and not all edges from G are in the BFS tree.

To label the number of shortest paths from x to each other vertex, GN starts by labelling $nsp(x) = 1$. Then from top down on the BFS tree, $nsp(y) = sum(nsp(z)$ for each $z = parent(y))$. This completes step 1.

Step 2 is to calculate for each edge e the sum of fraction of shortest paths from root x that go through e. During the computation, vertices also get credits (fractions) of betweenness. The information flows bottom up on the BFS tree. Every vertex y gets a credit of 1, representing the shortest path from x to y. This credit is divided among edges and vertices above the BFS tree, since there can be multiple shortest paths from x to y. The calculation rules are:

1. Each leaf of the BFS tree gets a credit of 1.
2. Each non-leaf vertex u gets a credit of 1 plus the sum of credits of all edges (u, v), where (u, v) is a directed edge on the BFS tree.
3. An edge (u, v) on the BFS tree is given a share of credit from v proportional to the number of shortest paths from x to u. That is, $credit(v) \times nsp(u)/nsp(v)$.

This completes step 2, and GN has finished its visit to x as a root. After performing the credit calculation for all vertices as as the root, the sum of credits for each edge is its betweenness in the final result. Strictly speaking, this number needs to be divided by 2, since each shortest path is counted twice.

To find communities using betweenness, add edges to the graph by increasing order of betweenness, one a at a time. At each step, find the connected components as clusters. Edge removal is a similar idea: edges are removed one by one, in the order of decreasing betweenness, until the graph is divided into a suitable number of components. A short introduction of GN algorithm can also be found in the textbook `http://infolab.stanford.edu/~ullman/mmds/ch10.pdf`

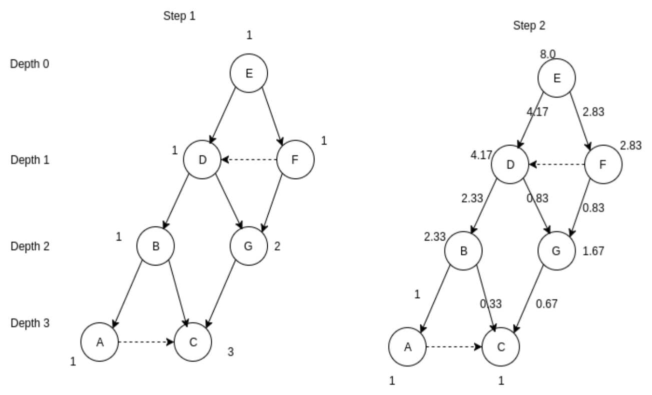

Figure 1: Girvan-Newman Algorithm Example

Vertex E as the root. Step 1: BFS marks vertices with depth and number of shortest paths to E. Step 2: credits are distributed along the BFS tree.

2.2 Louvain Algorithm

Louvain Algorithm was first raised by the Etienne Lefebvre who first developed it during his Master thesis at UCL (Louvain-la-Neuve) in March 2007. It was first published in the paper Fast unfolding of communities in large networks by Vincent D Blondel et al [10]. This method can be applied to weighted and directed graph. Our project only focuses on the unweighted and undirected graph provided by twitter. The Louvain algorithm is a clustering algorithm which can find hierarchical community structure based on the computation and optimization of modularity. According to the previous study and experiment results, it indeed achieves a good efficiency and accuracy.

2.2.1 Modularity

The main idea of community detection is to partition the network into clusters of densely connected nodes, which are called communities. Several algorithms have been proposed to find good partitions in the increasing complicated networks. The classical algorithms can be divided to two main classes. One is the Graph Partition based on the computer graphic, another is the hierarchical clustering. One of the famous algorithms in the first class is spectral clustering [6] approach using the eigenvectors of matrices. The second class includes Agglomerative method and Divisive method, the main idea is to do hierarchical clustering based on the similarity of the vertices. The quality of the partitions resulting from these methods is often measured by the so-called modularity of the partition. The modularity of a partition is a scalar value between -1 and 1that measures the density of links inside communities as compared to links between communities [4, 11]. For unweighted networks, the modularity is defined as:

$$Q = \frac{1}{2m} \sum_{i,j} \Big[A_{ij} - \frac{k_i k_j}{2m} \Big] \sigma(c_i, c_j) \tag{1}$$

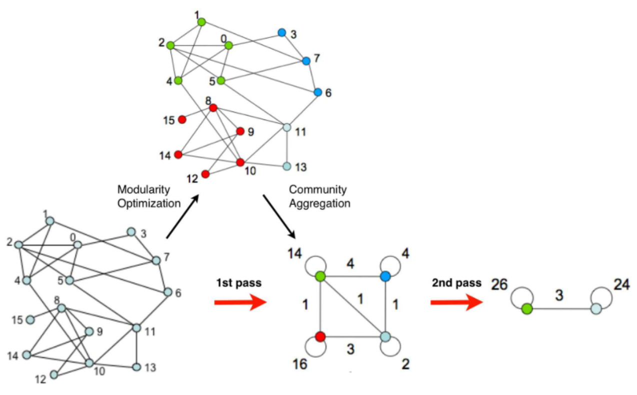

Figure 2: The basic idea of louvain algorithm

2.2.2 *Modularity Gain*

After the initialization of the graph, each community has been assigned a community id. Each vertex is treated as a community, we need to evaluate the quality of the community [5]. Here we use the corresponding modularity gain [1] is:

$$\Delta Q = \left[\frac{\sum in + K_{i,in}}{2m} - \left(\frac{\sum out + k_i}{2m}\right)^2\right] - \left[\frac{\sum in}{2m} - \left(\frac{\sum out}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \quad (2)$$

Here, $\sum in$ is the sum of weights from all internal edges, $\sum out$ is the sum of weights from the edges that connected with the internal vertices. $k_i$is the sum of weights from the edges connected to the vertex i.$K_{i,in}$ is the sum of weights from the all edges in the community C connected with i.

2.2.3 *Introduction of Louvain Algorithm*

There are two passes in the algorithm and each pass contains two phases. The first phase is to optimize modularity only allowing local changes of communities; the second one is to aggregate communities and build a new network of communities. The passes are repeated iteratively until no increase of modularity is possible.

Briefly, the Louvain algorithm works as follows. The algorithm initially starts out with a partition where each node is in its own community i, which is the initial partition. So, initially, there are as many communities as there are nodes. The algorithm moves around nodes from one community to another, to try to improve the modularity value. The modularity gain is the difference in moving node i to another community c. In particular, implying that if the modularity gain is improved. At some point, the algorithm can no longer improve modularity by moving around individual nodes, at which point it aggregates the graph, and reiterates on the aggregated graph. We repeat this procedure as long as we can improve Q value.

From Figure1, we can see how the algorithm works clearly.

3. IMPLEMENTATION

3.1 Girvan-Newman Algorithm

Considering scalability, when the graph is large, it may be infeasible to visit every vertex on the graph as a root. An approximation was used such that only K vertices are chosen as roots to compute the edge credits. For each edge the sum of the K credits was used as the betweenness, since only relative values are needed for community detection. It was concluded that, for a graph with bounded diameter, picking $K \in O(\log n)$ roots is enough for good estimation of vertex centrality [11]. Though not exactly the same we edge betweenness, the similar idea was used to set the number of roots K.

Both step 1 and step 2 were implemented using the *pregel* API of GraphX. If clear with the usage of *pregel* API, they are relatively straightforward to code up. *pregel* API passes messages along edges on the graph to vertices. The type of vertex attribute VD is set to (depth, nsp, credit, tmp), where tmp stores the received credits in the current iteration, which is added to credit later.

Before step 1 all depth are set to $infinity$ except the root. The message for step 1 is defined as (depth(src), nsp(src)), where src is the source vertex ID of the edge triplet which generates this message. A message is sent to dst vertex only if: (1) $depth(src) < infinity$; and (2) $depth(src) == infinity$. Messages are passed though out-edges. When BFS is done, for each edge (u, v), u is the parent of v if and only if $depth(u) = depth(v) + 1$. Note that each vertex only processes 1 (merged) message, besides the initial message, so a the vertex program is called twice in step 1. The max number of iteration is set to the estimated graph diameter.

For step 2 each vertex program can be called multiple times. Because the BFS tree can be asymmetric, a vertex doesn't wait for all its children to be ready. That comes the purpose the tmp field in VD. In this step, messages are passed bottom-up, to the in-edge direction. Note that a message generated by an edge triplet (u, v) is sent to the u if only u is the parent of v, where message is just a credit, equal to $tmp(v) \times nsp(u)/nsp(v)$. When a vertex program receives a merge message, tmp is set to the received credit, where the old tmp is added to the credit field. So in order to parallelize the original GN algorithm, more message runs are required here.

3.2 Louvain Algorithm

If we simply partition the vertices and distribute the calculation of modularity gain ΔQ across the distributed environment, the community change of one vertex cannot be known by other vertices. And the vertices may be updated in one slave and updated back in another slave, which will cause died circle. To count ΔQ, it requires a significant amount of communication and synchronization, which limits the degree of parallelism. What's more, in parallel environment, when vertices compute their ΔQ, they can only see a limited number of their neighbors' pre-existing community membership. The convergence property of the original greedy policy does not work. Vertices may end up exchanging obsolete community membership with little gain in modularity, resulting in the infinite movement of vertices, without reaching convergence. So we follow the method in paper [10], this paper provides a solution with an in-depth examination of the inherent computation and communication balance as well as a heuristic that can dynamically control the vertices' movement to different community membership, get rid of obsolete or non-contributing vertices, and eventually converge with high-modularity communities.

3.3 Parallel Louvain Algorithm

To make the algorithm to run in parallel, we need to update the information of each vertex synchronously in each iteration, based on the information from the previous iteration to update the current vertex information. So we defined a new vertexData data structure, which add the message of community and the neighbours.

```
class VertexData() extends Serializable{
  var degree: Int = 0
  var community: Long = 0
  var communityDegreeSum: Long = -1
  var neighDegree: Int = 0
  var neighCommunity: Long = -1
  var neighCommunityDegreeSum: Long = -1
  var edgeCount: Long = -1
}
}
```

We set the input graph with the Long edge type, the saveLevel is to save the graph including vertices and edges after each phase of the process.

1.Each vertex is assigned a unique community with its own id. Each vertex attempts to increase graph modularity by changing to a neighboring community or remaining in its current community

2.Repeat the first step until no progress have been made. Here, the progress is measured by looking at the decrease in the number of vertices that change their communities on each pass. If the progress is less than the min value we have set or more than the progressCounter times, we exit this level.

After all the vertices have been labeled with a community, we save the result of the community id into text file. Here, the minimal progress number is the number of vertices that must change communities for the algorithm to consider progress, the progressCounter is the number of time the the algorithm can fail to make progress before exiting. In the update stage of the each level modularity gain computation, we need to reconstruct the graph, Figure 2 [12]shows the idea of graph reconstruction with an example, where a weighted graph with 10 vertices is partitioned across 2 nodes (vertex 0, 2, 5, 7, 8 on node 1 and vertex 1,3,4,6,9 on node 2). The same color vertices belong to theh same community. Each node sends a message to the remote In Table and updates its In Table. For example, node 2 first scans ((3, 8), 2) and sends ((8, 8), 2) to node 1, which will be hashed (inserted/updated) into the third bucket in its In Table. When all the messages have been delivered, the In Tables now represent the full super graph, as shown in the bottom left. In this way we transform the graph relabeling problem into an all-to-all communication with hashing.

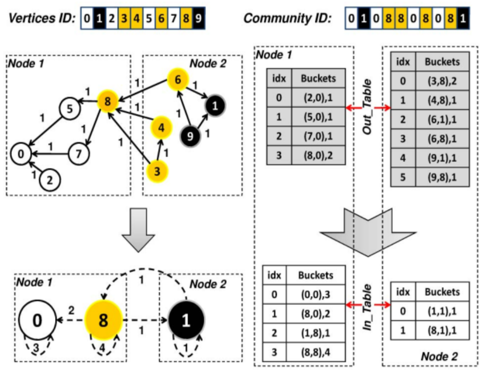

Figure 3: The process of graph reconstruction

3.Compress the graph. Treating each community as a single node, repeating the first four steps to complete the compress stage.

4.Repeat until modularity will not be improved.

4. EXPERIMENTAL RESULTS

4.1 Playing with Massive Twitter Graph

In our proposal we planned to explore the twitter following graph, consisting of 41 million vertices and 1.46 billion edges. The data is available at `http://konect.uni-koblenz.de/networks/twitter`. Each vertex represents a user and each directed edge represents a follow. Stored as a plain text edge list file, the data is 22.3 GB in size. It turned out that, the giant graph size became a big problem for graph processing, even that loading the graph would only took about 3 minutes.

4.1.1 *Graph Compression*

Each line of the original twitter graph data is an edge, (sourceId, destId) in plain text. This format made us tempted to perform some compression on the data, to reduce the size of the giant graph. The graph was re-represented as adjacency lists: each line is a source vertex id followed by a sorted list of destination vertex ids. An approach similar to what we did in assignment 3 was used: the sorted list was encoded using delta compression and VByte encoding. Note that the java WritableUtils library can also be used in spark and scala. Then the RDD of bytes was stored as an object file. As a result, the compressed graph was only 5.4 G in size, and can be correctly decompressed. The compression and decompression were also fast, and could be completed within minutes.

Though the data compression ratio of 413% looks high, the actual impact is not that significant. Firstly the memory size of altiscale platform is still more of a bottleneck than the disk storage, because in order to process the graph, it still must be decompressed in memory. Secondly the same method cannot be used to compress the betweenness result. Nevertheless, when implementing graph compression, we discovered a performance problem in GraphX.

4.1.2 *Buggy CollectNeighbours in GraphX*

Before graph compression, the list of neighbour must be collected for each vertex. Since GraphX provides APIs of $collectNeighbors$ and $collectNeighbouIds$, it was natural to use them for the goal. However, when $collectNeighbouIds$ was applied to the twitter graph, it wasn't even able to finish after running overnight (more than 6 hours). While we intuitively believed that collecting neighbours for a vertex with degree d should only take $O(d)$ time, GraphX made a mistake such that it would take $O(d^2)$ instead. Indeed this would have little impact on small or sparse graphs. However, although the twitter graph is generally sparse, with average degree of 70.5, few vertices have very high out-degrees, with a top value of 3.0 million. Therefore a routine with running time quadratic to the degree will not be able to finish in reasonable amount of time.

The detail of this performance problem was verified in the source code of GraphX, which can be found at `https://github.com/apache/spark/tree/master/graphx/src/main/scala/org/apache/spark/graphx` .

The issue is that $collectNeighbors$ was implemented using $aggregateMessages$, where the $mergeMessage$ operation is specified as concatenation of two arrays of neighborIds. This is problematic because the message size is not a constant. With no guarantee on the sequence of merge, the total time complexity becomes $O(d^2)$ since each merge takes $O(m + n)$ for two arrays of lengths m and n. By some further examination, it turns out that $aggregateMessages$ hits the worst case, where m keeps growing and n is always 1.

Once realized this issue, we performed mapreduce operations on the raw RDDs instead to collect neighbours, which worked fast and well. A general lesson is that messages had best be in constant size. However, inevitably GraphX looks no longer that reliable.

4.1.3 Degree Distribution, BFS and GN

Though a graph was given on the data source website, we recomputed the degree distribution of twitter graph. A hidden point discovered was that, 5.9 million, 14% of vertices have no incoming edge, and 1.5 million, 3.7% of vertices have no outgoing edge. This showed that large number of idle twitter accounts exist: they either follow no one, or not be followed by any one. The impact to graph analysis is that, we definitely don't want to start our BFS search from an vertex without any outgoing edge. Our solution was to randomly select vertices of top out-degree to be roots of BFS. We also verified that, 71 vertices have out-degree higher than 1 million. While we can understand some popular users can have millions of followers, we strongly doubt whether these 71 users had really clicked the follow button for so many times.

A simple BFS from vertex 1, for depth and the number of shortest paths to each vertex only, on the twitter graph was able to finish in about 40 minutes (step 1 of GN). It turned out that from vertex 1, 1.0 million of vertices have depth of 1; 1.56 M vertices have depth of 2; 2.66 million with depth 4; 0.26 million with depth 5; 27 thousand with depth 6; 2854 with depth 7; 305 with depth 8; 52 with depth 9; seven with depth 10; two with depth 11; a single one with depth 12; in addition to 6.6 million of unreachable vertices. As a result, it seemed that setting the max number of iterations to be 6 when performing BFS would be enough for a good approximation.

At last, a betweenness result from one random root (step 2 of GN) was extracted in 4 hours. Although this was not enough to detect communities on the twitter graph, we would expect the complete GN algorithm to finish within several days (for $K = 30$).

4.2 Detecting Communities Using Betweenness

We tried to detect communities using betweenness filtering followed by computing strongly connected components (SCC). We couldn't succeed. Even on a small graph with 38.7 MB data size, which can be found at `https://snap.stanford.edu/data/com-Youtube.html` , the computation of SCC couldn't finish in 3 hours, despite that betweenness values were quickly extracted. We wouldn't say that the SCC API by GraphX is buggy, but its documentation provides no running time explanation. We would expect that it would take as many iterations as the length of the longest path in the graph. This number becomes unknown after edges with high betweenness are been removed. In addition, we didn't know how to choose a good cut-off threshold for betweenness, and we don't know how many communities will be given in the final result.

Here we propose another solution to detect communities using betweenness. The prerequisite is that all vertex Ids can fit into the memory of a single (driver) machine, which is the case for the twitter graph. Another requirement is that a stream of edges are sent to the driver in the order of increasing betweenness. The driver uses a UnionFind data structure [7] to detect communites while edges are inserted one by one. On average each edge insertion takes only constant time but no extra space. The number of communities can also be traced at each step in constant time. The space requirement is a giant array for all vertex Ids. The procedure returns when the number of communities becomes satisfying.

4.3 Running time analysis of Louvain Algorithm

Below is the table which contains the detain information about the dataset and the running time of the louvain algorithm. We try to run the bigger twitter dataset on Altiscale, bu the result seems to be not ideal. Since the intemediate computation for modularity gain takes a long time during the first stage, the memory may not be enough to run such a big graph which contains billions of nodes. Based on the log provided by Yarn in ALtiscale, the reduce operation will take a large amount of time even though the graphs have been cached. From the previous study, randomly choosing the start vertex will not have much impact on the final community result, but it will significantly influence the running time due to the possibility to get the best communities from the important vertices from start. So we choose to start the minimal vertex which is what the cleaned dataset has done, to keep the result consistent and comparable. From the table, we can see that the dataset from Amazon, which is the relationship between customers, although it is a bit larger than the twitter dataset, it runs faster than the twitter and get a slightly higher modularity value. The higher

Table 1: Running time result of three datasets

DataSet	Twitter	Amazon	Youtube	LiveJournal
Size/M	11.3	12.6	38.7	500
Vertices	0.465M	0.335M	1.135M	3.997M
Edges	0.874M	0.925M	2.987M	34.68M
Total edge weight	1669594	1851744	5975248	69362378
Circles	12	24	28	N/A
time/s	323.08	199.597	11334.885	N/A
Qvalue	0.5614647570219136	0.6552802409568756	0.6067892457681281	N/A

modularity value is, the more dense the community will be, the result only show the modularity value of the first compression level, since I set the progress counter to 4. For the small dataset, it usually complete the process in one circle and the total weight become 0 when the second compress level starts.

5. CONCLUSION AND FUTURE WORK

We implement two graph algorithms for community detection and test our code on Altiscale. After many attempts and experiments, we finally choose some relatively small datasets from stanford large network dataset collection. For the Louvain algorithm, we get the final community result and modularity values of each graph. The Higher modularity value is, the more densely connected the vertices in the graph are. Due to the time limit and the memory limitation on Altiscale, we are not able to run dataset at a magnitude of GB, since a graph with millions of nodes will take at least several hours or even several days to be finished. Girvan-Newman Algorithm is fast in betweenness calculation for small graphs, and can be expected to complete on the massive twitter graph within days. To detect communities using betweeness, removing edges and computing SCCs could be problematic. Instead, edges should be added one by one, in the order of increasing betweenness, to a Union-Find structure, which needs to be stored in a single driver node.

At last, GraphX provides convenient interface, but we'd better not use them like black-box magic. As we discovered a performance bug in GraphX's source, we should be careful when using a interface that we are unfamiliar with.

6. REFERENCES

[1] I. F. T. V. Gergely Palla, Imre DerÃl'nyi. Uncovering the overlapping community structure of complex networks in nature and society. *Nature 435*, 435:814–818, 2005.

[2] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[3] S. W. M. M. e. a. MARYA BAZZI, MASON A. PORTER. Community detection in temporal multilayer networks, with an application to correlation networks. *2016 Society for Industrial and Applied Mathematics*, 14(1):1–41, 2016.

[4] K. K. Mingming Chen. Community detection via maximization of modularity and its variants. *IEEE Trans*, (1):46–65, 2014.

[5] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[6] J. Ruan and W. Zhang. An efficient spectral algorithm for network community discovery and its applications to biological and social networks. *IEEE Computer Society*, pages 643–648, 2007.

[7] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[8] V. A. Traag. Faster unfolding of communities: Speeding up the louvain algorithm. *PHYSICAL REVIEW*, page 032801, 2015.

[9] University of Waterloo. *https://github.com/lwdai/bigdata2017w-project,https://github.com/outerforce/test*, April 2017. We referenced the Spark API to complete this project `https://spark.apache.org/docs/2.0.2/index.html`.

[10] e. a. Vincent D. Blondel, Jean-Loup Guillaume. Fast unfolding of communities in large networks. *J. Stat*, page P10008, 2008.

[11] D. E. J. Wang. Fast approximation of centrality. *Graph Algorithms and Applications 5*, 5:39, 2006.

[12] F. P. J. A. G. Xinyu Que, Fabio Checconi. Scalable community detection with the louvain algorithm. *IEEE Computer Society*, 2015.