Using *k*-Core Decomposition to Find Cluster Centers for *k*-Means Algorithm in GraphX on Spark

Sheng-Tzong Cheng, Yin-Chun Chen, and Meng-Shuan Tsai
Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan
email: stcheng@mail.ncku.edu.tw, darkerduck@gmail.com, and stoya35893@hotmail.com

***Abstract*—Big data analysis is getting more and more attention these days. In social network applications, a large amount of data is in a graph structure form. As a result, more computation time is required for graph data analysis. In 2014, a framework of in-memory computing, Spark, was proposed for big data analysis. Through reusing the data in memory to solve the long computation time issue, Spark finishes a task in a shorter time compared to Hadoop. In addition, GraphX, a Spark API (Application Interface), provides a graphical interface and makes graph data analysis simple and efficient. This study presents an improved *k*-mean clustering method by integrating *k*-core decomposition, which is an important algorithm in community detection to find the center of each cluster. We implement the clustering algorithm with GraphX to get better performance and results compared to the original *k*-mean clustering method.**

Keywords—cloud computing; GraphX; Spark; k-core decomposition; graph-based k-means

I. INTRODUCTION

With the massive growth of computational data, cloud computing has become one of most popular disciplines in recent years. A typical example of cloud computing system like Hadoop [15], is based on MapReduce model [16] and Google file systems [16] to collect and analyze huge data.

For big data, analysis is the most important work. Techniques, such as machine learning, are used to train data and retrieve the most important parts form the data. Data reuse is also common in many iterative machine learning and graph algorithms, including PageRank [15], *k*-Core decomposition [3], and *k*-Means clustering [17]. However, for the framework of MapReduce, iterations of data computation become the critical bottleneck of the performance. Therefore, AMP (Algorithms Machines People) Lab at the University of California, Berkeley, proposes a new architecture, Spark [1], that not only improves the data processing over a parallel in-memory system, but also reuses inter-mediate results across multiple computations. Empirically, a program on Spark could run up to 100 times faster than that on Hadoop MapReduce.

Graph is a useful form to represent a massive amount of data in analysis, such as social network, biological information, business model, road and map, and collaboration network. However, traditional MapReduce framework makes it difficult to describe the graph-parallel computation in distributed system. Fortunately, Spark provides useful APIs (Application Interfaces) for GraphX [2] and MLib (Machine Learning Library). When users run applications on Spark, the APIs make the code development easy to use and build some algorithm efficiently. MLib provides machine learning algorithms. GraphX is a new large-scale distributed graph-parallel framework, such as Pregel [2], Graphlab [2], and PowerGraph [2], to provide useful graph algorithms. It also allows users to develop applications faster. In this paper, we consider the graph data analysis by using GraphX on Spark.

Graph clustering is one of the crucial problems in graph data analysis. It is used to group the vertices of a graph into clusters with as few edges as possible between them. It is related to unsupervised learning to divide a data set into some small classes without *a priori* information on how the classification should be done. *K*-Means clustering [17], which is an algorithm for graph clustering, uses vectors of characteristics to transform data into some clusters to find the nearest center. It usually gives *k* virtually random points as the initial centers of clusters. For each point, it finds the minimum Euler distance to a center. Then, each point is assigned to the cluster containing the nearest center. Before the next iteration, each cluster can re-compute a new center. The iteration repeats until there is no change for centers. In this work, we consider the *k*-Means algorithm for the graph clustering in GraphX.

The *k*-Core decomposition has been proposed to find the strongest communicators in a graph. Recently, *k*-Core decomposition for analysis of large networks has been reported [3]. We observe that the performance and results can be improved further if we combine the *k*-Core decomposition with the graph-based *k*-Means, in which the *k*-Core is used to find centers and the *k*-Means is used to find clusters. The contribution of this paper is to engineer the integration of *k*-Means clustering with *k*-Core decomposition for graph data analysis in GraphX on Spark. The rest of the paper is organized as follows. Background is given in Section II. The problem description and the system design are stated in Section III. Implementation details and experiment results are illustrated in Section IV, and conclusion remarks are drawn in Section V.

II. BACKGROUND

A. Spark

Spark is an open-source cluster computing system. It is the highest-level project in Apache Software Foundation. In November 2014, the world record of data sorting in the Sort Benchmark Competition was broken by Spark, while the previous record was made by Hadoop. As Hadoop took seventy-two minutes to finish the job, Spark only took less than thirty minutes to complete the sorting. Furthermore, it only used 207 sets of amazon E2 i2.8*large virtual machines, significantly less than 2100 sets used by Hadoop.

 ISBN: 978-1-61208-529-6

In the original data flow of Hadoop MapReduce, the reduce function has to be performed after each map function. Spark critically improves the performance by changing the original inflexible data flow (of Hadoop MapReduce) to a new flexible framework in which several map functions could be done in memory before the reduce function is invoked. In this way, it can avoid many times of operations in the reduce stage and data is not required to write back to disk.

B. GraphX

Spark API provides a Pregel-like framework [4] to deal specifically with graphs. To compute graphs with strong correlation between the nodes, it needs to adjust Graph construction for running on a classic data-processing platform. GraphX combines two graph processings, Pregel and Graphlab [4], to make a new graph-parallel framework. GraphX is developed directly on Spark to obtain better performance than Graphlab.

1) VertexRDDs, EdgeRDDs, and Route Table: These three data structures are the most important components to compose a graph in GraphX. When a dataset of graph is stored into GraphX, the graph will be initialized to an edge table, which describes the information about a node linking to another node with a value. After that, it generates a vertex table by using the class named Graph.

A vertex table is always the place for storing and computing the result, such as collecting all data, generating sub-graphs, joining vertices to give new data, and filtering some vertices, etc. Two kinds of operators for user programing are possible. One operator is to view a loose graph as a table and allow users to modify data without strong relation. The other operator is to view a graph as a tight graph in which vertex relations are required to re-compute when a update is propagated. New subgraphs may be generated then.

2) Graph Parallelism: Google developed a super-step algorithm framework [4] for graph-parallelism in 2010. It is widely used to build graphs in distributed computing systems, because of convenience and efficiency just like using MapReduce. Users only need to complete three functions for super-steps.

This model follows the bulk synchronization to finish the computation. There are four steps for one process and the final step can only stop when every node is inactive. It is also the condition to start running the application for the next round.

Four steps are described as follows.

- *a)* A node receives some messages and transforms the status from inactive to active.
- *b)* Active nodes aggregate all messages to get a result for itself.
- *c)* If the result does not need to update to nodes, then change the status from active to inactive; otherwise, send message to other nodes.
- *d)* Repeat steps *a* to *c* until there is no message sending to nodes.

In GraphX, the algorithm starts to send an initial message to all of vertices. If vertices need to update their data, the vertices will turn the status to active just like step (1). Second, they filter all active vertices and compare with neighbors to decide whether messages shall be sent or not. They use triplet to compare the data in two vertices. In step (2), all messages in a vertex will be stored as a list that performs sequential processing to get a result. This result will be compared to the original data to control the status and data. In the end, GraphX will generate a new graph with some new RDDs (Resilient Distributed Datasets)for this result.

C. k-Means Clustering

This is a well-known data-clustering algorithm. Upon given every node vector of characteristics and the value of *k,* the algorithm can separate the set of data into *k* clusters [5]. In the basic mode, operations define the original data with characteristics, grouping those values, and vectorization. The dataset will be divided into *k* clusters, given *k* random centers with the same dimension. Secondly, each node finds a nearest center by Euler distance so that primitive clusters are formed. Thirdly, new centers are identified by averaging the sum of nodes in each cluster. Repeat these three steps. In traditional algorithm follows the math model: *S* is the center set, and *D(x, y)* is the distance between *x* and *y* for $y \in S$

$$\mathrm{m(S)} = \arg\min_{S} \sum_{i=1}^{k} \sum_{y \in S_k, x \neq y} D(x, y),\ k = 1, \ldots, N \quad (1)$$

D. k-Core Decomposition

In graph theory, *k*-Core decomposition [6] is usually used. It is a O*(m)* algorithm where *m* represents the number of edges in non-parallel computing. Its main goal is to find a strong subgroup, whose members play the role of communicators in the graph. Every node in the sub-graph needs to be at least degree of *k*. In this paper, we extend the *k*-Core decomposition to the graph computation in parallel.

E. Modularity

In recent years, the concept of "quality for graph clustering," proposed by Newman, has been widely used as a measure of performance [7]. Researchers usually use it to optimize the community that splits the network. If each community has dense relationship within the group and sparse relationship outside of the group, the value of modularity will be higher.

The modularity defined by $\mathrm{Q} = \frac{(\text{edges within communities} - \text{expectation of these edges})}{\text{sum of all degrees.}}$.

Q lies in the range [−1/2, 1). Without loss of generality, we assume that a graph has *n* nodes and *m* edges. Let the adjacency matrix for the graph be represented by *A*, where the element A_{vw} of matrix equals to zero meaning there is no edge between vertices *v* and *w*; otherwise, the element equals to one that means there is an edge between two vertices. The degree of vertex *v* is represented by *d(v).*

Consider a graph is split into *k* communities$\{C_i\}_{i=1}^{k}$, , and *Q* can be written as

$$\mathrm{Q} = \frac{1}{2\mathrm{m}} \sum_{\mathrm{i}=1}^{\mathrm{k}} \sum_{vw \in C_i} A_{vw} - p(v, w) \quad (2)$$

where, $p(v, w) = \frac{d(v)d(w)}{2\mathrm{m}}$.

We can rewrite (2) to

$$Q = \sum_{i=1}^{k}\left(\sum_{vw\in C_i}\frac{A_{vw}}{2m} - \frac{1}{2m}\sum_{vw\in C_i}\frac{d(v)d(w)}{2m}\right)$$
$$= \sum_{i=1}^{k}\left(\sum_{vw\in C_i}\frac{A_{vw}}{2m} - \left(\sum_{v\in C_i}\frac{d(v)}{2m}\right)^2\right) \quad (3)$$

Modularity indicates how good the result is. The value of modularity often drops in the range from 0.3 to 0.7, which means the result is moderate. For the experiment in section IV.B.3 about the social circle in Facebook, we can see that a graph can get a high value of modularity when the number of clusters equals 9. In this paper, we use modularity as the performance index to conduct experiments.

III. System Design

In this section, we give the problem description and focus on the details of our system design. First, we clarify our problem. Then, we discuss the steps of system flow chart. Finally, the algorithm is given in details.

A. Problem Description

When people use graph to represent the real-world data, graph representation and graph clustering become crucial issues. Graph clustering focuses on finding the sub-graph with high relatives. It relies on the edges to reflect the relation and connection of vertices. Normally, the matrix is considered as the data structure for graph clustering. However, as a graph becomes larger, the matrix computation makes the performance worse.

Similarly, an adjacency matrix is not suitable for the classic *k*-Means algorithm. In fact, the number of edges in real-world graph is far less than the square of the number of vertices. Using the row of a huge sparse matrix to be feature vector makes *k*-Means algorithm heavy. It will compute many huge vectors, but they have many useless values leading to resource waste. Therefore, in this paper, we only use the idea of "finding the minimum distance sum to *k* center of clusters" and rebuild the *k*-Means algorithm with the *single source shortest path* algorithm and *k*-Core decomposition.

B. Scheme Overview

Fig. 1 shows the system flow chart to express our graph-based *k*-means algorithm. We run the code in Spark, which provides strong and flexible framework to deal with distributed parallel computing. It hides complex details of application development so that programmers can write functions with operations such as Map and Reduce. The GraphX adopts the graph-parallel processing model into Spark and transforms the big graph into some RDD table. The Pregel-like super-step module is a simplified coding for the iterative graph algorithms. Now, we explain the system systematically and introduce properties with Spark and GraphX in steps.

1) Spark connects to HDFS (Hadoop Distributed File System). If programmers want to run the project in parallel-computing mode, they should put the data into HDFS, which is a Hadoop database. Spark uses HDFS for not-local data to get better performance.

2) GraphX needs to generate edge RDD first for graph generation.

3) We separate graph generation from initializing the values. The values are initialized in the beginning of loops every time.

4) K-means algorithm is to select the nearest center for clustering the data set. In this paper, we choose the single source shortest path module for finding nearest center for each vertex. Given *k* centers, run single source shortest path for each vertex to find which center is the nearest center for this vertex.

5) *K* clusters are obtained from the result of step 4. Then, run the *k*-Core decomposition for each cluster. In the end, this step will find *k* new centers. The reason why we use *k*-Core decomposition to find the center of clusters is originating from the property: a core with the value of *k* is a group in which every member is connecting to at least *k* members. The vertex in-group with the biggest original degree value will be picked as the center.

6) Repeat steps *3* to *5* until the group of centers remains unchanged.

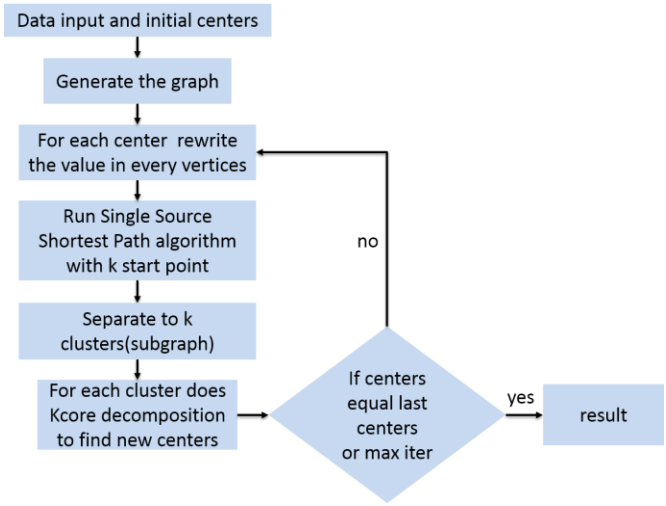

Figure 1. System flow chart.

C. k-Core Decomposition

In this section, we describe the details of implementing k-Core decomposition on GraphX. The k-Core decomposition has the following property:

$$\forall u \in V:\ k - core(u) = k \leftrightarrow \begin{cases}\text{There exist a maximum subgraph } V_k \\ \text{such that } \forall v \in V_k:\ deg(v) \geq k, \text{ and} \\ \text{There is no subgraph } V_{k+1} \\ \text{such that } \forall v \in V_{k+1}:\ deg(v) \geq k+1\end{cases} \quad (4)$$

This algorithm is mainly to find a sub-graph with the strongest relationship of *k*. It means every member in this sub-graph has at least *k* neighborhoods. Furthermore, there is no greater sub-graph where every member has more than *k* neighborhoods. Therefore, if we find a vertex that has the highest degree in this sub-graph, it will be a good candidate for the center in a cluster.

 ISBN: 978-1-61208-529-6

We tried to implement two versions with two different methods. The time complexity of both methods is $O(|V|^2)$. We can use a pair likes (a, b) to be data in a vertex table and the data will be changed to an integer value, K, representing the number of cores as the output.
Fig. 2 presents the pseudo code of *k*-core decomposition.

```
Procedure  k-core decomposition
      1:    Input
      2:     Graph: data in vertex is (degree, bool)
      3:    Output
      4:     Graph: data in vertex is K
      5:    Pseudo Code
      6:     While
      7:        Initial the Pregel send initial MSGs to all node
      8:        Graph.Vertex update (intitial MSGs)
      9:        MSGs = message merge (all message sent)
      10:       While messages.count > 0
      11:          Graph.Vertex update (messages)
      12:          MSGs = message merge (all message sent)
      13:       End while
      14:       K += 1
      15:    End while
      16:    Vertex update stage(messages)
      17:       If (message.bool ) messages.degree =
            max(origin, new  degree)
      18:       Message.bool = origin && new bool
      19:    Message send stage
      20:       If ( ! v1.bool || ! v2.bool)
      21:          empty
      22:       Else if(v1.degree == k && v2.degree > k)
      23:          Send to v2 (1,true)
      24:          Send to v1 (0,false)
      25:       Else
      26:          Iterator.empty
      27:    Message merge stage
      28:   (Sum, a.bool && b.bool)
```

Figure 2. Pseudo code for k-Core decomposition.

GraphX uses triplets to compare two nodes that need updates. The time complexity of the methods is $O(|V|^2)$.

IV. IMPLEMENTATION AND EXPERIMENT

In this section, we describe the experimental environment and illustrate the results obtained for modularity and run time. We use the same terminology for the original *k*-means algorithm to compare the performance of the original algorithm with that of our revised method.

A. Experiment environment and setting

In the experiment, we consider a peer-servicing cloud-computing platform, which contains six homogeneous virtual machines. The hardware and software specifications are detailed in Tables I and II, respectively. In Table III, the setting of our configuration of Spark is given. Because some RDDs are only used once, we do not need the original setting of memory fraction (0.75), which means the memory splits most of the space for storing RDD. When the system executes several iterations, the driver's java garbage collection is always too late to recycle the resource. We observe that the driver's memory stores a lot of DAG (Directed Acyclic Graph) and RDD in memory and therefore, only little space is left for allocating work. It may block the operation of iterative loop, so that we reduce the fraction from 0.75 to 0.4 and increase the memory for drivers.

TABLE I. RECEIVER ENVIRONMENT

Item	Content
OS	Ubuntu 15.10 Desktop 64bit
Spark	2.0.0
Java	1.7.0_101
Scala	2.11.8
Maven	3.3.9

TABLE II. HARDWARE SPECIFICATION OF RECEIVER

Item	Content
CPU	Intel(R) Xeon(R) E5620 @2.40GHz x 2
RAM	8 GB
Hard Drive	80GB
Network Bandwidth	1Gbps

TABLE III. CONFIGURATIONS OF SPARK

Item	Content
Number of executor	6
Memory size of the driver	6GB
Memory size of each executor	6GB
Memory fraction	0.4
Running mode	Standalone

B. Experiment Results

We conduct experiments on three real-world datasets from SNAP [8] and UCI Network Data Repository [9]-[13]. Each dataset is represented as a graph with vertices and edges. To revise the original *k*-means algorithm, we transfer the data to an adjacency matrix, because the algorithm requires the vertices' features for clustering. Each column can be considered as features of a vertex in the adjacency matrix. The vertices connecting to same vertex should be assigned to the same cluster. After finishing clustering, we calculate the modularity to evaluate the performance of both our revised method and the original *k*-means with adjacency matrix. Spark has a software version of the common *k*-means algorithm in Mlib, which is an API for machine learning.

The runtime of an experiment does not include the time of transferring original data to a matrix nor the time of calculating the modularity. From the experimental results, we see that even though the runtime of our proposed method is longer than the common *k*-means in Mlib, the value of modularity is much higher than the original *k*-means. All of the experiments started with centers randomly picked.

 ISBN: 978-1-61208-529-6

1) The dolphin social network

This is a famous social network dataset for graph clustering. There is a network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand [9]. In Fig. 3 (a), we can see that the modularity gets higher when the number of clusters increases. For example, our method gets an averaged value of modularity up to 0.3924 upon splitting to five clusters. Comparing the results reported in [14], they get the highest scores on four clusters. Although the case of four clusters is not the best result for our method, our proposed method still gets an averaged Q to 0.387703. This is helpful for observing a big group in real world.

In Fig. 3 (b), the runtime of our proposed method gets significantly high from 5 to 6 (for the number of clusters). Taking into account the modularity, the optimum case for the number of clusters is five.

2) The social cycle in Facebook

In this section, the dataset is much bigger than the datasets in the formal two experiments. The dataset is provided by J. McAuley and J. Leskovec [11]. They collected data from using an APP (Application) named social circle. There are 4039 vertices and 88234 edges. One edge between two vertices means two vertices are friends. In Fig. 4 (b), we can see that the runtime does not grow up when the data size increases if the memory is still enough to handle the experiments. This result also proves that Spark is fit for large-size data rather than small-size data.

The Stanford website [8] not only provides the dataset for researchers but also provides some basic analytic results. The average clustering coefficient they provided is 0.6055. In our method, we can find when the k = 4, 7, 8, 9, 10, the average modularity is higher than 0.6055 and the highest modularity is appearing when k = 9, which is the same number of clusters for the data on website. We think this data has a clear relationship between groups, because both methods obtain good result. In our method, when k is 5 or 6, the performance is worse than the case when k is 4. It is because that the four-cluster structure is similar to the nine-cluster graph. We can see from Fig. 4 (a) that the modularity reaches a local peak when k is 4 .

3) Gnutella Network

This is the experiment with Gnutella peer-to-peer file sharing network from August 2002. Nodes represent hosts in the Gnutella network topology and edges represent connections between the hosts [12][13]. There are 8717 vertices and 31525 edges in the graph. Compared to the second experiment, this graph has double the number of vertices but the number of edges is much less. On Stanford website [8], we can see that for this kind of graphs with strongly-connected components, only a small group can reach each other. It also has bad coefficient in clustering. In k-means with adjacency matrix, the modularity is almost approaching zero, however, as shown in Fig. 5 (a), our proposed method still gets the modularity average higher than 0.2. Although this result is slightly less, our method still performs well for k-means with loose grouping dataset.

For runtime shown in Fig. 5 (b), our proposed method is faster than k-means with adjacency matrix for the cases when k is from two to six. Actually, the edges are less dense so that the k-mean algorithm spent time in computing many bad features. We observe that when the data size grows, k-means algorithm not only needs a large space and time to transfer data for adjacency matrix but also needs more time to divide the graph into small clusters.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a graph-based k-means algorithm on Spark. Given a dataset, a graph could be structured and fed into this algorithm for solving the clustering problem. We also implemented the k-core decomposition with GraphX API to find the centers of clusters.

By spending more runtime, our proposed method would be able to find clusters with higher modularity. If we take the time for data processing into account, the total elapsed time of our method will be approaching to that of k-means. It is because the matrix structure used by k-means needs more processing time especially for the sparse matrix of many vertices with few edges. We find that k-core decomposition still is a good method for finding centers of clusters.

The initial random centers have a large effect on the performance of the k-means algorithm and our proposed method. Runtime can be greatly reduced if the two methods start with good initial points. Furthermore, even if the two methods start with the same initial centers, they probably have big difference in the clustering and the result value of modularity. We also find that vertices with many neighbors have great dominating effects. If we can select initial centers nearby these vertices, we can gain a better result.

Future work is to improve the performance of our proposed method further. The distributed k-core decomposition could be improved by integrating with Pregel algorithm. The original Pregel algorithm wastes a lot of time in sending initial iterators that have impact on the performance of GraphX. We plan to use the internal code of GraphX Pregel API to reconstruct the k-core decomposition.

REFERENCES

[1] M. Zaharia, et al., “Spark: cluster computing with working sets” Hot Cloud, vol. 10, 2010, pp.10-10.

[2] R. Xin, J. Gonzalez, M. Franklin, and I. Stoica “GraphX: A Resilien t Distributed Graph System on Spark”, AMPLab, EECS, UC Berkele y 2013

[3] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-Core Deco mposition of Large Networks on a Single PC," Proc. VLDB Endowm ent, vol. 9, no. 1. 2015.

[4] G. Malewicz, et al., “Pregel: a system for large-scale graph processin g,” Proc. 2010 ACM Intl. Conf. on Management of data, 2010.

[5] J. Hartigan and M. Wong, “A k-means clustering algorithm,” Applie d Statistics, vol. 28, 1979, pp. 100-108.

[6] A. Montresor, F. Pellegrini, and D. Miorandi, “Distributed k-core dec omposition,” IEEE Trans. Parallel Distrib. Syst., vol. 24, no. 2, 2013, pp. 288–300.

[7] M. Newman, “Modularity and community structure in networks,” Pr oc. Natl. Acad. Sci. USA, vol. 103, no. 23, 2006, pp. 8577-8582.

 ISBN: 978-1-61208-529-6

[8] Stanford Large Network Dataset Collection, http://snap.stanford.edu/data/index.html [Jul, 6, 2016].
[9] D. Lusseau, et al., "The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations," Behavioral Ecology and Sociobiology," vol. 54, 2003, pp. 396-405.
[10] W. W. Zachary, "An information flow model for conflict and fission in small groups," J. Anthropological Research, vol. 33, 1977, pp. 452-473.
[11] J. McAuley and J. Leskovec, "Learning to Discover Social Circles in Ego Networks," NIPS, 2012.
[12] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph Evolution: Densification and Shrinking Diameters," ACM Trans. Knowledge Discovery from Data, vol. 1, no. 1, 2007.
[13] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design," IEEE Internet Computing Journal, 2002.
[14] D. Lusseau and M. E. J. Newman, "Identifying the role that individual animals play in their social network," Proc. R.SOC.LONDON B, 271:S477, 2004.
[15] "Hadoop," http://hadoop.apache.org/.
[16] "MapReduce," http://research.google.com/archive/mapreduce.html.
[17] J. Hartigan and M. Wang, "A K-Means Clustering Algorithm," J. Royal Statistical Society, vol. 28, no.1, 1979, pp. 100-108.

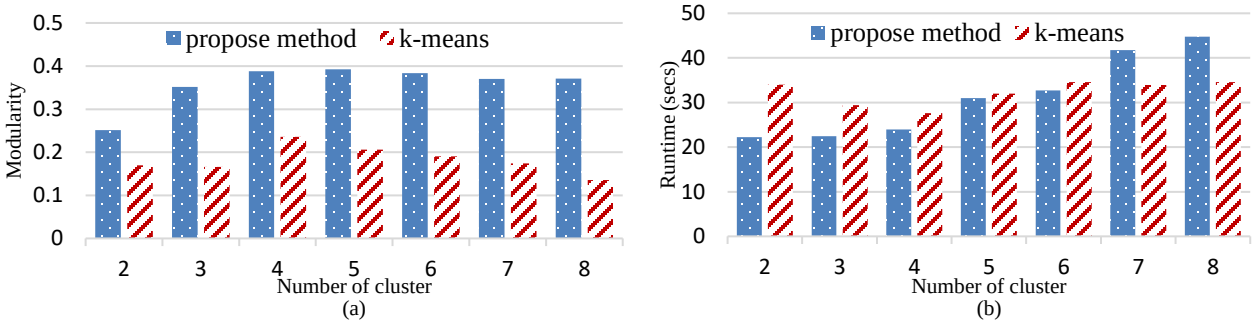

Figure 3. Modularity and runtime of dolphin social network.

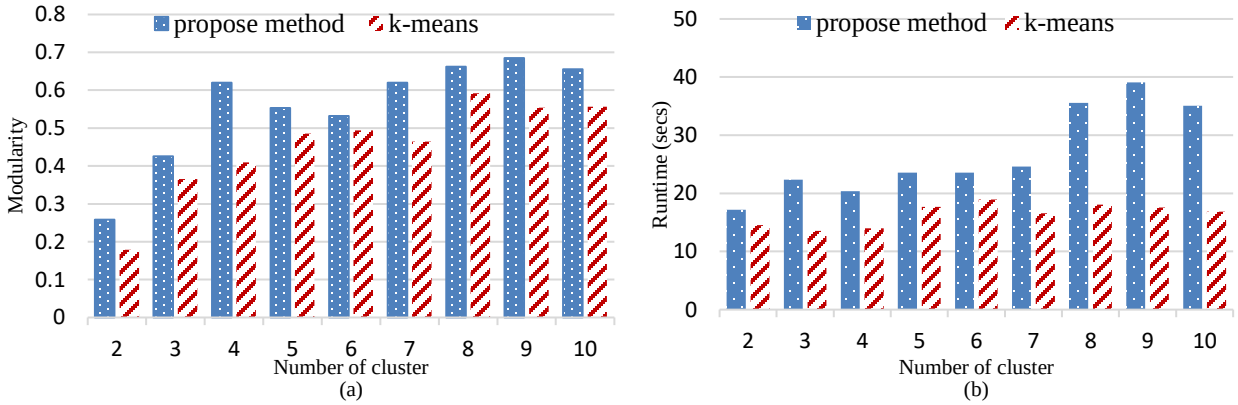

Figure 4. Modularity and runtime of Facebook social network.

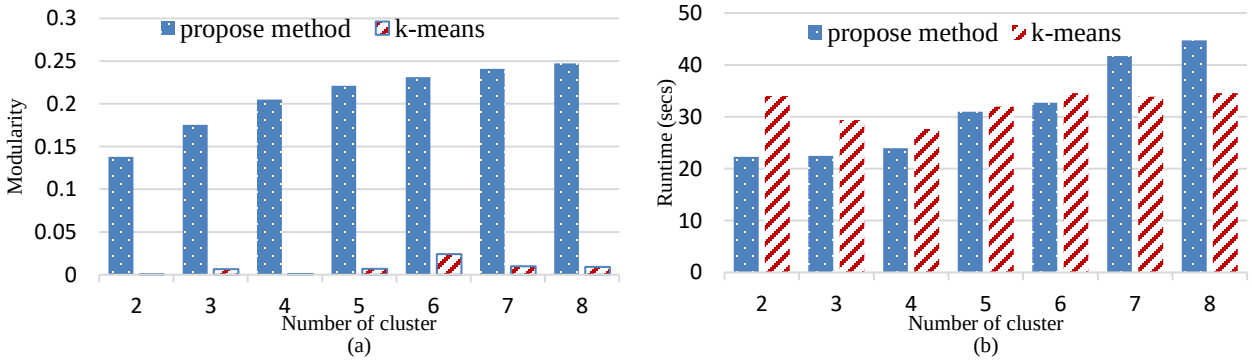

Figure 5. Modularity and runtime of Gnutella network.