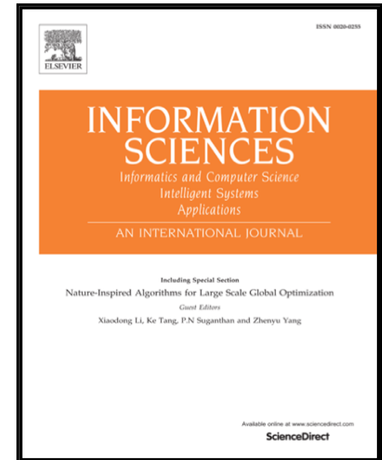
Accepted Manuscript



Community discovery by propagating local and global information based on the MapReduce model

Kun Guo , Wenzhong Guo , Yuzhong Chen , Qirong Qiu , Qishan Zhang

PII: S0020-0255(15)00464-8
DOI: 10.1016/j.ins.2015.06.032
Reference: INS 11626

To appear in: *Information Sciences*

Received date: 22 January 2015
Revised date: 23 April 2015
Accepted date: 16 June 2015

Please cite this article as: Kun Guo , Wenzhong Guo , Yuzhong Chen , Qirong Qiu , Qishan Zhang , Community discovery by propagating local and global information based on the MapReduce model, *Information Sciences* (2015), doi: 10.1016/j.ins.2015.06.032

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

Community discovery by propagating local and global information based on the MapReduce model

Kun Guo[a,c], Wenzhong Guo[a,c*], Yuzhong Chen[a,c], Qirong Qiu[b], Qishan Zhang[b]
[a] College of Mathematics and Computer Sciences, Fuzhou University, Fuzhou, China
[b] Management School, Fuzhou University, Fuzhou, China
[c] Fujian Province Key Laboratory of Network Computing and Intelligent Information Process, Fuzhou, China

ABSTRACT

Discovering communities in large-scale social networks efficiently and accurately is one of the challenges in social network data mining. We propose a clustering algorithm to discover social network communities based on the propagation of local and global information. Three strategies, namely, localizing propagation of affinity messages, relaxing self-exemplar constraints, and hierarchical processing, are employed in the algorithm to achieve reasonable time and space complexities in social networks. The local and global information is represented by the *k*-path edge centrality incorporated in the similarity calculation. The standalone algorithm is extended to provide parallel implementations based on the MapReduce model to accelerate processing in large-scale networks. Two well-known parallel computation frameworks, Hadoop and Spark, are adopted to implement the parallel algorithm. Experiments performed on artificial and real social network datasets show that the proposed algorithms can achieve near-linear time and space complexities with comparative clustering accuracy.

Key words: affinity propagation; community discovery; MapReduce model; social network

1. Introduction

Research on social network data mining has attracted increasing attention from both the academic and business communities because of the rapid growth of social network service applications such as Twitter, Facebook, and Flickr. Discovering communities based on common interests is particularly valuable given the millions of participants in such social networks. Online social network analysis can be considered as an extension of data mining and traditional clustering analysis [22]. From the business perspective, social networks are profitable for delivering targeted advertisements or disseminating recommendations among different social groups [28].

The large scale of social networks poses challenges from the viewpoint of clustering methods. First, only algorithms with near-linear ($o(n)$[1]) time and space complexities can process large numbers of social network vertices. Second, even when an algorithm with linear complexity is applied, a single computer is incapable of managing the tremendous computing workload involved. Social network data can be processed efficiently using distributed or parallel computation models such as message

* Corresponding author.
E-mail addresses: gukn123@163.com (K. Guo), fzugwz@163.com (W. Guo), yzchen1979@163.com (Y. Chen), qqrkyc@fzu.edu.cn (Q. Qiu), zhangqs@fzu.edu.cn (Q. Zhan).

[1] An algorithm with $o(n)$ time complexity means that its running time $T(n)$ satisfies $\forall k>0, \exists n_0, \forall n>n_0, T(n)\le k\cdot n$. Space complexity $o(n)$ can be defined similarly.

ACCEPTED MANUSCRIPT

passing interface (MPI) [35], bulk synchronous parallel (BSP) [44], and MapReduce [11].

MapReduce, a framework proposed by Google, is a popular parallel computing model [11]. The model accelerates the solution of complex problems by decomposing a big problem into several small ones and devising algorithms comprising parallel map and reduce tasks. Apache Hadoop [45], a popular open-source implementation of MapReduce, has proven to be effective for managing big data for the following reasons. First, an Apache Hadoop cluster can be extended easily to accommodate thousands of computers for providing high computation capability. Second, Apache Hadoop can process both structured and unstructured data. Third, the failure tolerance mechanism of the software allows jobs to be completed despite computational failures of some machines in the cluster. However, Hadoop has several limitations, for example, Support for data sharing among MapReduce jobs is limited, and thus, transforming an iterative algorithm into a Hadoop-based MapReduce model is difficult. Apache Spark [48] is another fast-growing open-source parallel computing framework that attempts to support a greater number of computation models and run faster than Hadoop. It is a powerful competitor and a promising substitute to Hadoop because of its abundant primitives, flexible computation workflows, and in-memory computing.

Parallel analysis of big data has become a significant and promising research topic with the advent of big data and the increasing popularity of parallel computation models and frameworks for processing large datasets. However, a challenge faced in designing parallel algorithms for the MapReduce model is transforming efficiently sequential, conditional, and loop structures into concurrent map and reduce tasks. This study focuses on discovering communities in large-scale social networks using the MapReduce model. The local affinity propagation (LAP) algorithm with near-linear time and space complexities is proposed. In addition, an extension of the LAP algorithm to the MapReduce (LAPoMR) model is proposed to process large-scale social networks efficiently. Design ideas for and complexity analysis of LAPoMR are discussed. Two concrete implementations of LAPoMR based on Hadoop and Spark are presented and compared. In the LAP algorithm, the similarity of vertices is measured according to the *k*-path edge centrality [33, 34]. The *k*-path edge centrality measures the importance of an edge by the probability with which the random simple *k*-paths traverse it. In LAPoMR, a parallel similarity calculation based on the *k*-path edge centrality is proposed. Experiments performed using artificial and real networks demonstrate the effectiveness of the algorithms from the viewpoint of community discovery. The key contributions of our work are summarized as follows.

(1) Three new strategies are proposed to develop the LAP algorithm, the time and space complexities of which are reduced significantly with comparable accuracy.

(2) Parallel LAPoMR is proposed by transforming the steps of the LAP algorithm into a series of map and reduce tasks. Theoretical analysis and experiment results demonstrate the notable performance speedup of the algorithm in large datasets.

(3) Two implementations of LAPoMR based on Hadoop and Spark are presented and their advantages and disadvantages are discussed.

(4) A new parallel similarity calculation method based on *k*-path edge centrality, which encompasses local and global information of social networks, is proposed.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the standard affinity propagation (AP) algorithm. Section 4 details the design and analysis of the LAP algorithm. Section 5 discusses adaptation of the LAP algorithm to MapReduce and its subsequent implementation on Hadoop and Spark. The parallel similarity calculation based on the *k*-path edge centrality is discussed in Section 6. Section 7 presents the results of experiments conducted using artificial and actual social networks. Section 8 concludes our study and presents an outline for future research.

2. Related work

Community discovery in social networks can be regarded as a type of complex network clustering [15]. Existing complex network clustering methods can be divided into methods based on optimization technologies and those based on heuristic rules.

The methods based on optimization technologies discover communities by iteratively approaching a predefined object function. Spectral clustering methods [41] transform community discovery into an optimization problem of a relaxed quadratic form. The approximate optimal solution is obtained by solving the eigenvectors of the corresponding Laplacian matrix. The modularity proposed by Newman and Girvan [36] is widely used as the object function for optimization in many algorithms such as the fast Newman algorithm [19], Clauset algorithm [8], Guimera–Amaral algorithm [20], extremal optimization algorithm [13], etc. The definition of the abovementioned modularity is as follows:

$$Q = \sum_i \left(e_{ii} - a_i^2\right) = Trace(\mathbf{e}) - \left\|\mathbf{e}^2\right\| \quad (1)$$

where $\mathbf{e} = (e_{ij})$ is a symmetric matrix, e_{ij} denotes the ratio of the edges between communities i and j to the total edges, and $a_i = \sum_j e_{ij}$ is the fraction of edges that connects to the vertices in community i. Trace($\mathbf{e}$) is the trace of $\mathbf{e}$, which measures the fraction of edges connecting vertices within the same community. $\|\mathbf{e}^2\|$ is a measure of the expected value of the same quantity with random connections among vertices. A high Q value usually indicates a strong community structure. Potts model–based algorithms discover communities by minimizing the Hamiltonian of a reduced energy system [42]. Algorithms based on vertex or edge centrality discover communities based on network structures [31, 34, 38]. Meo et al. proposed a new k-path centrality index [32] and applied it to community detection in large networks [31, 33]. Most optimization-based algorithms have a time complexity of $O(n^2)$, which is too high for large-scale network clustering. A promising way towards enhancing the modularity-based algorithms is the Louvain method [6]. It looks for smaller communities by optimizing modularity locally and aggregates nodes of the same community and builds a new network whose nodes are the communities. These two steps are repeated iteratively until the modularity value is maximized. In this way, the number of nodes to test quickly reduces and the computational cost is reduced in the same order.

The methods based on heuristic rules discover communities by designing specific heuristics. The Girvan–Newman (GN) algorithm is based on the heuristic rule that the "betweenness" [18] of the edges that connect two communities should be larger than that of the edges inside a community. Communities are discovered by removing the edges with the highest betweenness. The clique percolation method (CPM) algorithm discovers communities as a series of neighboring k-cliques and detects overlapping communities [37]. Label propagation algorithms (LPA) broadcast labels among vertices and group vertices with the same label into a community [46]. Algorithms based on random walks transform community discovery to random walks on a graph [39]. Based on the heuristic rule, the probability of a random walker staying in a community should be higher than that between communities. The time complexity of most algorithms is at least $O(n^2)$ except for LPA, which is applicable only to medium-scale networks.

Research on the MapReduce model has taken two directions. One direction directly exploits the model's power to process big data. The Hadoop-based diameter and radii estimator algorithm computes the diameters and radii of massive graphs [24]. Zhao et al. proposed a Hadoop-based parallel LPA [49]. Gao et al. introduced a MapReduce-based algorithm for e-mail network analysis [17]. Zhao et al. developed an algorithm based on Hadoop for relational subgraph analysis [50]. Schultz et al. applied MapReduce to detect important graph patterns in large-scale graphs [40]. Apache Mahout is a famous framework that implements many data mining algorithms on the Hadoop framework [4]. Spark MLib is a recently proposed library for the same purpose, albeit on the Spark framework [3]. Datameer is a powerful, one-stop service application for the integration, analysis, and visualization of big data for customers [10]. All studies in this direction have achieved significant reductions in running time, particularly in the context of large-scale networks.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

The second research direction for the MapReduce model is enhancing for or extending it to highly complex machine learning algorithms. Bu et al. proposed the HaLoop framework for implementing iterative algorithms efficiently on the MapReduce model [7]. Pregel and its open-source counterpart, Giraph, implement the MPI model to process large graphs rapidly [5, 29]. The Spark framework was proposed recently for supporting a number of computation models beyond those based on the MapReduce model, and it runs at least 10 times faster than Hadoop [14, 48]. Kajdanowicz et al. found the MapReduce model to be more suitable for large networks after its efficiency was compared with that of the BSP model [23]. Li et al. developed a local iteration MapReduce model and applied it for designing a parallel LPA [27]. Studies on the MapReduce model generally involve exploring the model's power and improving its performance in complex computation tasks.

3. The standard AP algorithm

Brendan and Delbert [16] proposed the standard AP algorithm as a clustering algorithm by propagating messages between a pair of vertices in a graph. The number of communities is not a prerequisite, and a symmetric similarity measure is unnecessary. Thus, the algorithm is suitable for a wide variety of applications. The most important parameter of the standard AP algorithm is *p*, i.e., the self-similarity of each vertex because it significantly influences the number of communities discovered. Two types of messages are transferred among the vertices during iterations of this algorithm. These are responsibility message $r(i,k)$, which expresses the support of vertex *i* to another vertex *k*, and availability message $a(i,k)$, which expresses the suitability of vertex *k* as the exemplar of another vertex *i*. The main procedures of the standard AP algorithm include updating $r(i,k)$ and $a(i,k)$ iteratively until convergence, as follows:

$$r(i,k) \leftarrow s(i,k) - \max_{k' s.t. k' \neq k} \{a(i,k') + s(i,k'))\} \tag{2}$$

$$a(i,k) \leftarrow \begin{cases} \min\left\{ \begin{matrix} 0, r(k,k) + \\ \sum_{i' s.t. i' \neq \{i,k\}} \max\{0, r(i',k)\} \end{matrix} \right\} & i \neq k \\ \sum_{i' s.t. i' \neq \{i,k\}} \max\{0, r(i',k)\} & i = k \end{cases} \tag{3}$$

In Equation (2), $s(i,k')$ represents the similarity between vertices *i* and k'. In the standard AP algorithm, this similarity is set as the negative of the Euclidean distance between the two vertices. When applied to community discovery in social networks, structural similarity measures such as the Jaccard index [15] are used frequently to describe similarity among vertices. The Jaccard index considers the similarity between two vertices as the ratio of the number of shared neighbors to the total number of neighbors, as expressed below:

$$Jaccard(i,k) = \frac{|NB(i) \cap NB(k)|}{|NB(i) \cup NB(k)|} \tag{4}$$

where $NB(i)$ denotes the neighbor set of vertex *i*, and $0 \leq Jaccard(i,k) \leq 1$ can be verified easily.

4. The LAP algorithm

The standard AP algorithm propagates messages between each pair of vertices regardless of whether they are connected. Therefore, the time complexity of the standard AP algorithm is $O(n^2)$, which is unacceptable for large networks with millions or even billions of vertices. For example, for a moderate network with 10^6 vertices, the memory and running time requirements are of the order of 10^{12}. In social networks, users typically exchange information with only a small part of the network (relatives, friends, colleagues, etc.). Hence propagating responsibility and availability messages to everyone in the network is unnecessary. Another weakness of the standard AP algorithm is the self-exemplar constraint that forces each

ACCEPTED MANUSCRIPT

exemplar to select itself as its own exemplar. Sumedha and Weigt [43] pointed out that the performance of the standard AP algorithm can be improved by relaxing the constraint. This section presents the design ideas, detailed implementation, and complexity analysis of the LAP algorithm.

4.1 Design ideas

Two design strategies are proposed to address the weaknesses of the standard AP algorithm.

Strategy 1 Messages are propagated only between connected vertices, i.e., a vertex can only exchange messages with its neighbors.

Strategy 2 An exemplar is free to select any vertex as its exemplar regardless of whether it is selected as its own exemplar.

The first strategy employs the characteristics of social relationship. Thus, the quantity of messages spread across a network is proportional to the total number of neighbors of all vertices, which has been established as useful in the literature [33]. The total number of neighbors of all vertices is twice the number of network edges when the network is connected, which is frequently true in social networks. Therefore, the time complexity of the algorithm can be reduced to be near linear with respect to network size, except that the network is an incomplete graph. Near-linear time complexity is always achieved because a social network is seldom a complete graph. Moreover, the strategy shortens the propagation path of a message. Therefore, inadequate exchange of messages may adversely affect the algorithm's clustering accuracy. However, in experiments, minimal loss of accuracy and increase in community size were observed.

The second design strategy aims to discover a more reasonable scheme for assignment of vertices to communities than that used in the standard AP algorithm by relaxing the self-exemplar constraint. A vertex specifically belongs to a community that is a member of another community, i.e., the community may form hierarchical structures. When the strategy is applied, the equations for responsibility and availability calculations are transformed as follows:

$$r(i,k) \leftarrow s(i,k) - \max_{k's.t.k' \neq k \cap k' \in NB(i)} \{a(i,k') + s(i,k')\} \tag{5}$$

$$a(i,k) \leftarrow \begin{cases} \min\left\{ \begin{matrix} 0, r(k,k) + \\ \sum_{i's.t.i' \neq \{i,k\} \cap i \in NB(k)} \max\{0, r(i',k)\} \end{matrix} \right\} & i \neq k \\ \sum_{i's.t.i' \neq \{i,k\} \cap i \in NB(k)} \max\{0, r(i',k)\} & i = k \end{cases} \tag{6}$$

The equation for selecting the exemplar of a vertex *i* is now given as

$$e(i) = \arg\max_{ks.t.k \in NB(i)} \{r(i,k) + a(i,k) \mid r(i,k) + a(i,k) > 0\} \tag{7}$$

A subtle problem may arise when the constraint on exemplars is relaxed. This issue is expressed in Property 1.

Property 1. Relaxation of the self-exemplar constraint can lead to a cyclic structure in exemplars, which can be eliminated by selecting one vertex in the circle as the exemplar of all other vertices.

Description: As shown in Fig. 1, exemplars may form a circle, which causes the algorithm to run exemplar selection ceaselessly. The solution is to break the circle at a specific vertex, thereby transforming it into a path. The vertex that functions as the self-exemplar is also selected as the exemplar of all other vertices in the path, as shown in Fig. 2. After removing the arrow between vertices 2 and 3 and reversing the direction of the arrow between vertices 1 and 2, vertex 1 is selected as the exemplar of vertices 2 and 3. Thus, the circular structure is broken.

ACCEPTED MANUSCRIPT

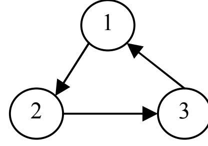


Fig. 1. Circular structure of the exemplars

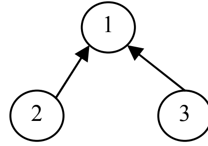


Fig. 2. Exemplar selection after breaking the circle

A common method for managing large social networks is to partition it into layers and then process each layer in a hierarchical manner [1, 6, 12, 31, 33]. The AP algorithm can be run iteratively. The exemplars discovered in one iteration are regarded as vertices and used to form a super-network for the next iteration. The proportion of the network to be processed in one iteration decreases rapidly, and thus, large networks can be managed efficiently. Hence, we suggest an alternative strategy to design the new LAP algorithm.

Strategy 3 Networks can be processed in a hierarchical manner.

The new LAP algorithm is developed considering all aforementioned strategies.

Algorithm 1. LAP Algorithm

Input: network $G=(V,E)$, where V is the vertex set and E is the edge set, the maximum iterations *maxIter*, the stable *convIter*, preference p

Output: $\{e(i)\}$, where $e(i)$ denotes the exemplar of vertex i

1. for each vertex i, calculate the similarity $s(i,k)$ between it and its neighbor k, and build the similarity vector set $Set_s=\{\mathbf{s}_i=(s(i,k))\}$;
2. $Set_e^{old} = \Phi$;
3. **while** ***true*** **do**
4.   *stc*=0;
5.   **for** n=1 to *maxIter*
6. $Set_e= \Phi$;
7.   **for** i=1 to $|V|$ **do**
8.   calculate $\mathbf{r}_i=(r(i,k))$ and $\mathbf{a}_i=(a(i,k))$ for each vertex i according to equation (5) and equation (6), where $k=\{1,2,\ldots, |NB(i)|\}$;
9.   **end for**
10.   for each vertex i, calculate $k=e(i)$ according to equation (7) and add vertex k to Set_e;
11.   **if** $Set_e = \Phi$ **or** $Set_e^{old} - Set_e \neq \Phi$ **then**
12.   *stc*=1;
13.   **else**

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

14. stc=stc+1;
15. **end if**
16. **if** stc=$convIter$ **then**
17. **break**;
18. **end if**
19. **end for**
20. call procedure RemoveCircles(V, Set_e) to remove the circles in the exemplars;
21. **if** $Set_e^{old} \neq \Phi$ **and** Set_e^{old} - $Set_e = \Phi$ **then**
22. **break**;
23. **end if**
24. $Set_e^{old} = Set_e$;
25. prune Set_s to hold only the exemplars;
26. **end while**
27. output Set_e;

The LAP algorithm mainly relies mainly on two loops to find exemplars after the similarity vector $\mathbf{s}_i$ for each vertex i is calculated: (1) the outermost while loop and (2) inner for loop spanning steps 5–19. The latter loop runs in iterations similar to those in the standard AP algorithm to find the exemplars of the vertices in network G. However, the LAP and standard AP algorithms have two key differences. The first difference is presented in steps 7–9, wherein only responsibility and availability messages from or to neighbors are calculated for each vertex i according to strategy 1. Therefore, the time and space complexities of the iterations are reduced from O(n^2) in AP to O(m) (m = |E|). The second difference lies in step 10, wherein an exemplar is selected for a vertex without considering the self-exemplar constraint according to strategy 2. Steps 11–19 are only used for convergence judgment.

The outer while loop comprises three tasks. First, RemoveCircles() is run to remove possible circles in the exemplars. Second, the loop is stopped if the exemplars found in two consecutive loops are the same, which is determined in steps 21–23. Third, if the exemplars vary, network G is pruned to retain only the exemplars and their edges for the next run of the inner for loop. Set_s is pruned instead of network G because the network structure is represented by Set_s, and the computation concerns only Set_s.

The details of RemoveCircles() are as follows:

Procedure 1. RemoveCircles()

Input: V, {e(i)}

1. **for** each $i \in V$ **do**
2. initial hash table H;
3. j=i;
4. **while** $j \neq e(i)$ **do**
5. **if** H contains vertex j **then**
6. **break**;
7. **end if**
8. put vertex j to H;
9. j= e(i);
10. **end while**
11. **if** H is not empty **then**
12. **for** each vertex $k \in H$ **do**
13. e(k)=j;
14. **end for**
15. **end if**
16. **end for**

ACCEPTED MANUSCRIPT

We provide the property for this procedure as follows.

Property 2. RemoveCircles() can remove circles in the exemplars properly.

Description: RemoveCircles() checks whether the exemplar for each vertex i is a self-exemplar. If it is, then hash table H would be empty or without a circle. Otherwise, the while loop spanning steps 4–10 is executed to find a path from vertex i to its real exemplar. Two situations are considered. First, a self-exemplar is found at the end of the path, i.e., no circle is formed. In this case, the exemplar of each vertex in the path is set to the self-exemplar by following steps 11–15. Second, a vertex is met twice during the search for the real exemplar, which indicates that a circular path exists. In this case, the exemplar of each vertex in the path is set to a specific vertex by following steps 11–15.

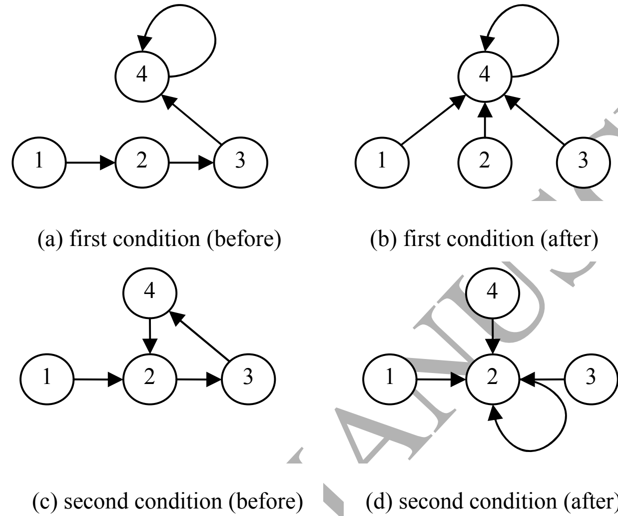


Fig. 3. Solutions to the two conditions

Fig. 3 shows an example of the solutions to the two conditions. We do not distinguish which vertex is selected as the exemplar because, as found in our experiments, this choice has no effect on the discovered communities.

4.2 Similarity calculation

In [33] and [34], Pasquale et al. proposed that the edges can be weighted according to their centrality to enhance community detection. Specifically, edge weights are represented by the k-path edge centrality, which considers both local and global information of the network. The k-path edge centrality of an edge is the sum of the probabilities with which a message originating from any vertex traverses the edge which passes only random simple k-paths. A simple k-path is a path in a network made up of k random unique edges at most. The probability of selecting an edge for a path is proportional to its current weights (initially set to 1.0), which can be expressed as Equation (8).

$$\Pr(e_i) = \frac{w(e_i)}{\sum_{e_i \notin T_e} w(e_i)} \tag{8}$$

where e_i is the neighboring edge of the last edge in the path, $w(e_i)$ is the weight of e_i, and T_e is the set of traversed edges in the path. An algorithm called WERW-Kpath was proposed in [34] to calculate the edge weights based on k-path edge centrality, the main steps of which can be summarized as follows:

(1) **for** i = 1 to ρ **do**

(2) randomly select the first edge for path P_i;

(3) **for** j = 2 to k **do**

(4) select the next edge for P_i according to Equation (8);
(5) update edge weights;
(6) **end for**
(7) set the weight of each edge in P_i to $w(e_j)/\rho$;
(8) **end for**

Here, ρ decides the number of random paths used for calculating *k*-path edge centrality. As suggested in [34], a value of $|V|\log|V|$ is sufficient to achieve satisfactory accuracy. By considering the edge weights based on *k*-path edge centrality, we can develop a similarity measure similar to the Jaccard index, as follows:

$$s(i,k)=\frac{\sum_{j\in NB(i)\cap NB(k)} w_{ij}+w_{kj}}{\sum_{j\in NB(i)} w_{ij}+\sum_{j\in NB(k)} w_{ki}} \quad (9)$$

where $NB(i)$ denotes the neighbor set of vertex i and w_{ij} is the weight of edge e_{ij}. Therefore, in the first step of the LAP algorithm, the similarity of two vertices is calculated according to Equation (9).

4.3 Complexity Analysis

Property 3. The time and space complexities of the LAP algorithm are $O(m \times \log n)$ and $O(m)$, respectively, where n is the number of vertices and m is the number of edges.

Description: Let $n = |V|$, $m = |E|$. The similarity calculation is $O(m)$, as demonstrated in [34]. The maximum number of iterations of the inner for loop is $O(maxIter)$. Updating responsibility and availability messages and determining the exemplars in each iteration require $O(m)$ time. Therefore, the time complexity of the inner for loop is $O(maxIter \times m)$.

The number of iterations required by the outer while loop is determined by the period in which the exemplars become stable. The number of exemplars (or communities) decrease by at least half after each iteration because the situation wherein each vertex selects itself as its exemplar can be avoided by setting parameter p to a value smaller than 0. Therefore, the maximum number of iterations of the outer while loop is $O(\log n)$.

Calculating Set_s takes $O(m)$ time. The time complexity of RemoveCircles() is $O(n \times l_p)$, where l_p is the length of the longest path from a vertex to its real exemplar. The maximum time required to prune Set_e is $O(m)$. In summary, the total time complexity of the LAP algorithm is $O(\log n \times maxIter \times (m + n \times l_p))$, which can be reduced to $O(m \times \log n)$ for $n \leq m$ and $l_p \ll n$. Thus, the LAP algorithm scales near linearly with network size.

By adopting strategy 1, the space occupied by the similarity, responsibility, and availability vectors of all vertices is $O(m)$. Occupation of the hash table and the path in RemoveCircles() is $O(n)$. Therefore, the total space complexity of the LAP algorithm is $O(m)$. This finding proves that the LAP algorithm, too, has near-linear space complexity.

5. Local affinity propagation based on the MapReduce model

Distributed or parallel computation models such as MapReduce are widely applied to process large networks that cannot be incorporated into a single machine. Determining and transforming steps of the LAP algorithm to ensure that they can run in parallel are central to adapting the algorithm for the MapReduce model. The design ideas of LAPoMR (LAP algorithm over MapReduce) and the detailed procedures of the algorithm are discussed in this section. The complexity of the algorithm is analyzed as well.

5.1. Design ideas

The strength of the MapReduce model lies in its parallel execution of independent tasks. Therefore, parallelization of

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

many steps of the LAP algorithm increases performance (speed). As mentioned previously, the biggest challenge in the design of MapReduce algorithms is the transformation of sequential, conditional, and loop structures into concurrent map and reduce tasks efficiently. We consider four critical steps that for transforming sequential, conditional, and loop structures into parallel map or reduce tasks.

(1) Similarity calculation

According to Equation (9), the similarity calculation of a pair of vertices requires only information about the neighbors of said pair. Thus, similarity calculations for all pairs of vertices can be performed independently so long as the neighbors of each pair of vertices are presented. A sketch of the MapReduce model for similarity calculation is then developed, as shown in Fig. 4.

The map tasks are used to collect pairs of vertices and their neighboring sets. The reduce tasks are run to calculate the similarity of each pair of vertices. The final reduce tasks are used to gather similarity from all neighbors of each vertex i and build similarity vector $\mathbf{s}_i$.

(2) Responsibility calculation

Equation (5) shows that the responsibility calculation involving a vertex and its neighbors depends only on the availability sent from and the similarity with its neighbors. Therefore, a sketch of the MapReduce model for the responsibility calculation is developed, as shown in Fig. 5.

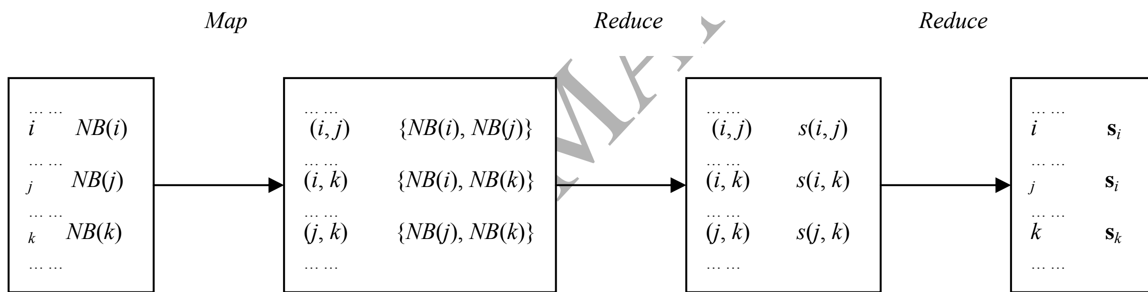


Fig. 4. Similarity calculation based on the MapReduce model

ACCEPTED MANUSCRIPT

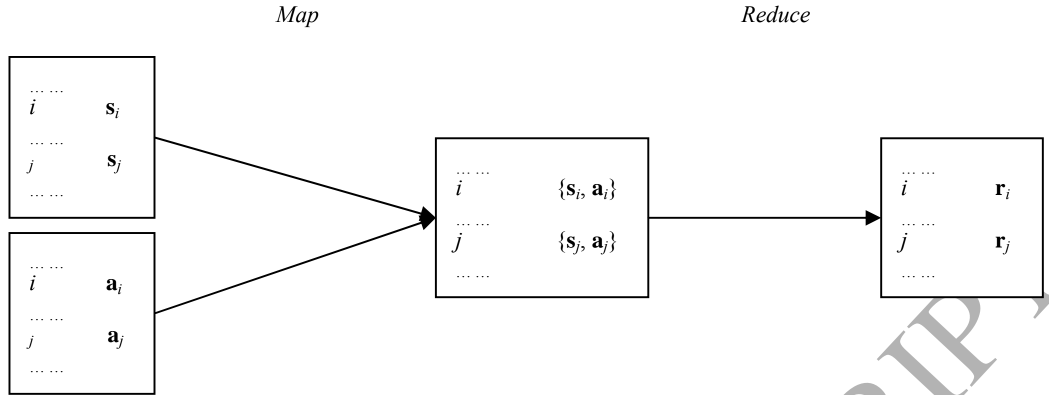


Fig. 5. Responsibility calculation based on the MapReduce model

In the map step, the similarity vector $\mathbf{s}_i$ and availability vector $\mathbf{a}_i$ of each vertex i are determined for computation using Equation (5). In the reduce step, the responsibility $r(i,k)$ of each neighbor k to vertex i is determined and packed into the responsibility vector $\mathbf{r}_i$.

(3) Availability calculation

The availability calculation is similar to the responsibility calculation, but a fundamental difference exists between them. Equation (6) reveals that the computation of the availability vector $\mathbf{a}_i$ does not depend directly on the corresponding responsibility vector $\mathbf{r}_i$. The calculation of element $a(i,k)$ requires collecting the kth element of $\mathbf{r}_j$, where j traverses all neighbors of vertex i. This procedure is similar to transposing a matrix consisting of all $\mathbf{r}_i$s (it is indeed the transposition of matrix $\mathbf{R} = (\mathbf{r}_i)$ if all vertices have the same number of neighbors). Therefore, we should extract all $r(i,k)$s with the same k to compute $a(i,k)$, and then all $a(i,k)$s with the same i should be collected to build vector $\mathbf{a}_i$ for any vertex i. A sketch of the MapReduce model of the availability calculation is shown in Fig. 6.

Given its complexity, the availability calculation is divided into two stages. The first stage performs *transposition* and calculates $a(i,k)$ for each neighbor k, according to Equation (6). The second stage determines $a(i,k)$ values for each vertex i and generates the corresponding availability vector $\mathbf{a}_i$.

(4) Exemplar calculation

According to Equation (7), an exemplar vector $\mathbf{e}_i = ((r(i,k) + a(i,k)))$ should be computed initially for each vertex i. The neighbor corresponding to the maximum positive element of $\mathbf{e}_i$ is then selected as its exemplar. Therefore, a sketch of the MapReduce model for exemplar calculation with a design similar to that of the responsibility calculation model is developed, as shown in Fig. 7.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

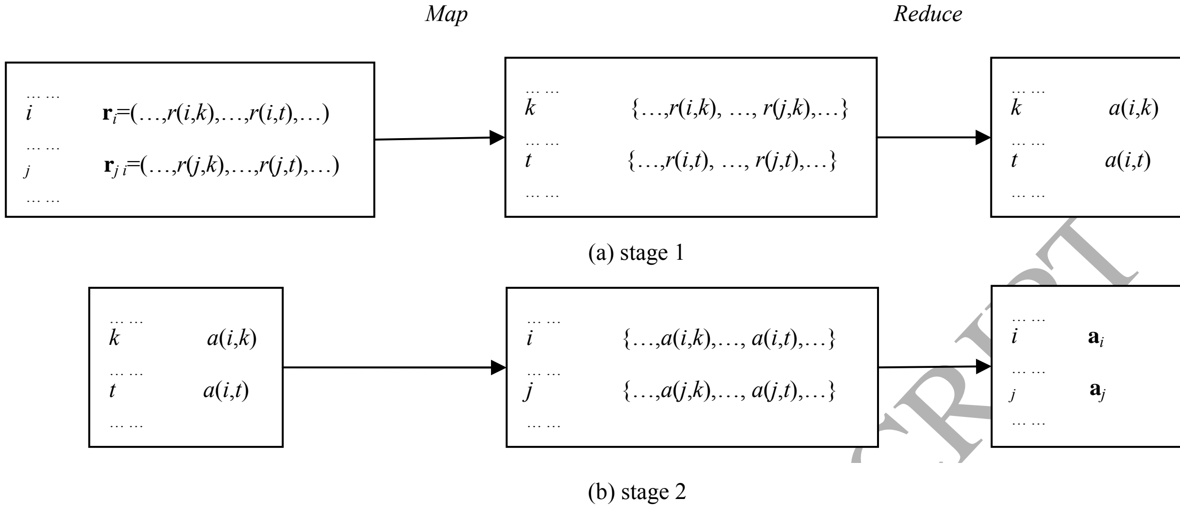


Fig. 6. Availability calculation based on the MapReduce model

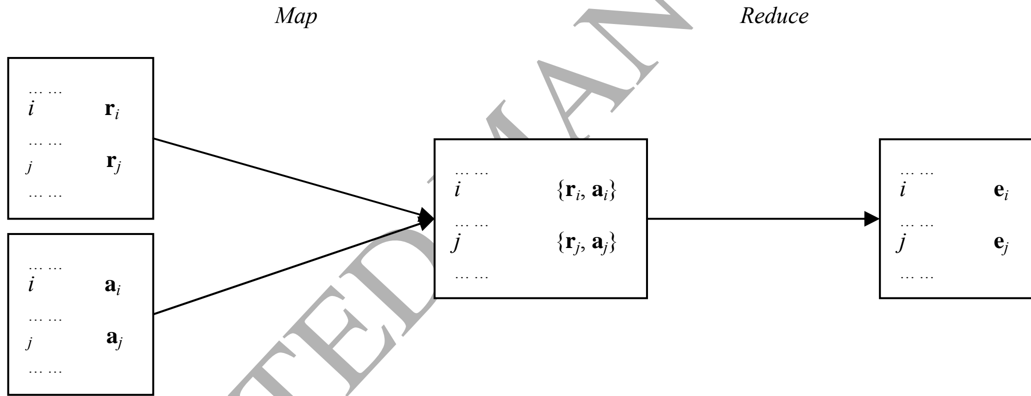


Fig. 7. Responsibility calculation based on the MapReduce model

The responsibility vector $\mathbf{r}_i$ and availability vector $\mathbf{a}_i$ corresponding to each vertex i are collected in the map step. In the reduce step, the exemplar vector $\mathbf{e}_i$ is calculated according to Equation (7).

After parallelization of the four calculations, the main body of the LAP algorithm can now be transformed into a parallel LAP algorithm based on the MapReduce model.

5.2. The LAPoMR algorithm

This subsection details the LAPoMR algorithm that conforms to the design ideas presented in the previous subsection. However, implementing this algorithm with the two well-known MapReduce frameworks, i.e., Hadoop and Spark, involves certain subtleties. The Hadoop framework requires that the algorithms running on it should be separated into a series of MapReduce jobs, wherein each job is divided into a map stage and a reduce stage. Therefore, the similarity calculation shown in Fig. 4 should be separated into two jobs, with the map tasks of the second job employing the identity mapper pattern [45] to transfer the input directly to the output. In addition, we have to rely on the Hadoop distributed file system

ACCEPTED MANUSCRIPT

(HDFS) or a database to share data between successive jobs, which may severely affect the performance of the algorithm. In contrast, Spark provides flexible solutions. This framework offers plenty of primitives (e.g., *map*, *reduce*, *fold*, *foreach, groupBy*, *join*, etc) to allow for a wider variety of operations than the MapReduce model and supports data sharing by caching data in memory or disks, or both. These advantages make Spark a good candidate for implementing the LAPoMR algorithm. We implement the LAPoMR algorithm on both Hadoop and Spark to compare the performance of the frameworks.

Details of the implementation of the LAPoMR algorithm on the two frameworks are presented below.

Algorithm 2. LAPoMR Algorithm (Hadoop)

Input: network G=(V,E), where V is the vertex set and E is the edge set, the maximum iterations *maxIter*, the stable *convIter*, preference p

Output: {$e(i)$}, where $e(i)$ denotes the exemplar of vertex i

1. //SIMILARITY CALCULATION
2. map (key i, value $NB(i)$)
3. for each neighbor k, output (key (i,k), value $NB(i)$); (if $i>k$, change key to (k,i))
4. output (key(i,i), value -1);
5. reduce (key (i,k), values {$NB(i)$,$NB(k)$})
6. **if** $i==k$ **then**
7. *output* (key i, value p);
8. **else**
9. calculate $s(i,k)$ according to equation (4);
10. output (key i, value (k,$s(i,k)$));
11. output (key k, value (i,$s(i,k)$));
12. **end if**
13. map (key i, value (k,$s(i,k)$)) //Identity Mapper pattern
14. output (key i, value (k,$s(i,k)$));
15. reduce (key i, values {(k,$s(i,k)$)|$k\in NB(i)$})
16. output (key i, value {(k,$s(i,k)$)});
17. Set_s={(i, {(k, $s(i,k)$)})};
18. $Set_e^{old} = \Phi$;
19. **while true do**
20. **for** *iter*=1 to *maxIter* **do**
21. //RESPONSIBILITY CALCULATION
22. map (key i, value {(k, $s(i,k)$)}) and (key i, value {(k, $a(i,k)$)})
23. calculate $r(i,k)$ according to equation (5);
24. output (key i, value (k, $r(i,k)$,0));
25. output (key k, value (i, $r(i,k)$),1);
26. reduce (key i, values {(k, $r(i,k)$,0)}) and (key k, values {(i, $r(i,k)$,1)})
27. output (key i, value {(k,$r(i,k)$)});
28. output (key k, value {(k,$r(i,k)$)}); //prepare for availability calculation
29. //AVAILABILITY CALCULATION
30. map (key k, value {(k, $r(i,k)$)})
31. calculate $a(i,k)$ according to equation (6);
32. output (key i, value (k, $a(i,k)$));
33. reduce (key i, values {(k, $a(i,k)$)})
34. output (key i, value {(k,$a(i,k)$)});
35. //EXEMPLAR CALCULATION
36. map (key k, value {(k, $r(i,k)$)}) and (key k, value {(k, $a(i,k)$)})

ACCEPTED MANUSCRIPT

37. output (key i, value (k, $r(i,k)$, $a(i,k)$));
38. reduce (key i, values {(k, $r(i,k)$, $a(i,k)$)})
39. calculate $k=e(i)$ according to equation (7) and add vertex k to Set_e;
40. call procedure RemoveCircles(V, Set_e) to remove the circles in the exemplars;
41. **if** $Set_e = \Phi$ **or** $Set_e^{old} - Set_e \neq \Phi$ **then**
42. stc=1;
28. **else**
43. $stc=stc+1$;
44. **end if**
45. **if** $stc=convIter$ **then**
46. **break**;
47. **end if**
48. **end for**
49. **if** $Set_e^{old} \neq \Phi$ **and** $Set_e^{old} - Set_e = \Phi$ **then**
50. **break**;
51. **end if**
52. $Set_e^{old} = Set_e$;
53. prune Set_s to hold only the exemplars;
54. **end while**

Algorithm 3. LAPoMR Algorithm(Spark)

Input: network $G=(V,E)$, where V is the vertex set and E is the edge set, the maximum iterations *maxIter*, the stable *convIter*, preference p

Output: $\{e(i)\}$, where $e(i)$ denotes the exemplar of vertex i

1. //SIMILARITY CALCULATION
2. map (key i, value $NB(i)$)
3. for each neighbor k, output (key (i,k), value $NB(i)$); (if $i>k$, change key to (k,i))
4. output (key(i,i), value -1);
5. reduce (key (i,k), values $\{NB(i),NB(k)\}$)
6. **if** $i==k$ **then**
7. output (key i, value p);
8. **else**
9. calculate $s(i,k)$ according to equation (4);
10. output (key i, value $(k,s(i,k))$);
11. output (key k, value $(i,s(i,k))$);
12. **end if**
13. reduce (key i, values $\{(k,s(i,k))|k\in NB(i)\}$)
14. output (key i, value $\{(k,s(i,k))\}$);
15. $Set_s=\{(i, \{(k,s(i,k))\})\}$;
16. $Set_e^{old} = \Phi$;
17. **while true do**
18. **for** $iter$=1 to *maxIter* **do**
19. //RESPONSIBILITY CALCULATION
20. map (key i, value $\{(k, s(i,k))\}$) and (key i, value $\{(k, a(i,k))\}$)
21. calculate $r(i,k)$ according to equation (5);
22. output (key i, value (k, $r(i,k)$));
23. reduce (key i, values $\{(k, r(i,k))\}$)

ACCEPTED MANUSCRIPT


```
24.                 output (key i, value {(k,r(i,k))});
25.  //AVAILABILITY CALCULATION
26.             map (key i, value {(k, r(i,k))})
27.                 output (key k, value (i, r(i,k)));
28.             reduce (key k, values {(i, a(i,k))})
29.                 output (key k, value {(i,r(i,k))});
30.             map (key k, value {(k, r(i,k))})
31.                 calculate a(i,k) according to equation (6);
32.                 output (key i, value (k, a(i,k)));
33.             reduce (key i, values {(k, a(i,k))})
34.                 output (key i, value {(k,a(i,k))});
35.  //EXEMPLAR CALCULATION
36.             map (key k, value {(k, r(i,k))}) and (key k, value {(k, a(i,k))})
37.                 output (key i, value (k, r(i,k), a(i,k)));
38.             reduce (key i, values {( k, r(i,k), a(i,k))})
39.                 calculate k=e(i) according to equation (7) and add vertex k to Set_e;
40.             call procedure RemoveCircles(V, Set_e) to remove the circles in the exemplars;
41.             if Set_e = Φ or Set_e^old - Set_e ≠ Φ then
42.                 stc=1;
29.             else
43.                 stc=stc+1;
44.             end if
45.             if stc=convIter then
46.                 break;
47.             end if
48.         end for
49.         if Set_e^old ≠ Φ and Set_e^old - Set_e = Φ then
50.             break;
51.         end if
52.         Set_e^old = Set_e;
53.         prune Set_s to hold only the exemplars;
54.  end while
```

Although algorithms 2 and 3 appear similar, two key differences exist between them. The first difference is in the similarity calculation. The Hadoop implementation needs two MapReduce jobs to achieve this, whereas the Spark implementation can reduce directly a batch of reduce task results. The flexibility afforded by the Spark primitives serves a major function at this point. The second difference is in both responsibility and availability calculations. To reduce the number of MapReduce jobs and save time, the Hadoop implementation integrates availability calculation steps with the responsibility calculation steps. This integration increases the design complexity of the associated map and reduces tasks. An additional 0 or 1 has to be appended to the values of the mappers of the responsibility calculation jobs to distinguish responsibility calculation data from availability calculation data. Otherwise, the reducers will be unable to determine whether the value $r(i,k)$ received by it is for calculating the responsibility vector $\mathbf{r}_i$ or the availability vector $\mathbf{a}_i$. By contrast, the Spark implementation can separate responsibility and availability calculations easily and clearly without employing extra MapReduce jobs because flexible primitives such as *flatMap*, *groupByKey*, and *mapValues* can complete the computation.

Two patterns are applied in the LAPoMR algorithm. The first is the identity mapper pattern [45], which is used in the second job of the similarity calculation in the Hadoop implementation to transfer input to the reducer directly. The second is

ACCEPTED MANUSCRIPT

the map-side join pattern [45], which is adopted in the availability and exemplar calculations to join the responsibility and availability vectors in both implementations. Unlike the responsibility calculation, the availability calculation must employ two MapReduce jobs because the responsibility vectors must be transposed to calculate $a(i,k)$ for each k and transpose them back to calculate $a(i,k)$ for each i. This transposition cannot be executed in a single job even in the Spark framework.

5.3. Complexity analysis

Property 5. The time and space complexities of the LAPoMR algorithm are $O(maxIter \times \log n \times m/n_m)$ and $O(m)$, respectively, where n is the number of vertices, m is the number of edges, and n_m is the number of machines.

Description: The index of each vertex is an integer that enables the default partitioner of Hadoop and Spark to distribute workloads uniformly. Therefore, the time complexity of the similarity, responsibility, availability, and exemplar calculations can be reduced to $O(m/n_m)$. The outer while loop requires $O(\log n)$ time, as discussed in Section 4.3. The total time complexity of the LAPoMR algorithm is $O(maxIter \times \log n \times m/n_m)$. The space complexity of the LAPoMR algorithm is $O(m)$ because the entire network is required to be stored before running the MapReduce jobs.

6. Parallel similarity calculation based on the MapReduce model

The similarity calculation in the LAPoMR algorithm requires transformation of the WER-Kpath algorithm into a parallel algorithm. In this section, we describe the design and implementation of the parallel similarity calculation based on the *k*-path edge centrality. When a network is represented as an adjacency list, the selection of edges for a path can be replaced with the selection of neighbors of the vertices in the path. We consider three critical steps that can be transformed into parallel map or reduce tasks.

(1) Random selection of starting vertex for each path

Initially, the first vertex of each path is selected based on the degree of the vertex. Map tasks are employed to determine the degrees of all vertices. Then, in the reduce task, ρ starting vertices are selected according to their degrees. The higher the degree of a vertex, the greater is the likelihood of its selection. Without doubt, vertices with high degrees can appear on more than one path. A sketch of the MapReduce model for random selection of the first vertex for a path is shown in Fig. 8.

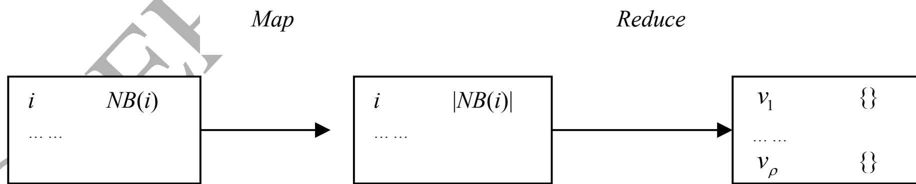


Fig. 8. Starting vertex selection for paths based on the MapReduce model

Here, $NB(i)$ is extended to include the weight of the edge connecting each neighbor. Specifically, the elements of $NB(i)$ are tuples in the form of $<j, w_{ij}>$ that encapsulate not only the neighbor j but also the weight of edge e_{ij}. The outputs of the reduce task are the starting vertices v_1–v_ρ (as the output keys) and their predecessor lists (as the output values, initially empty). The constraint is that there should be only one reduce task for selecting the real high-degree vertices. Otherwise, the selected vertices would represent high-degree vertices in each data split, processed by individual reduce tasks.

(2) Selection of next vertex for each path

ACCEPTED MANUSCRIPT

After the first vertex is chosen, it is natural to select the next vertex for each path. A sketch of the MapReduce model corresponding to this step is shown in Fig. 9.

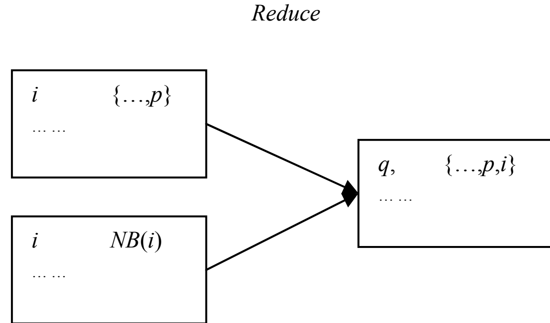


Fig. 9. Next vertex selection for paths based on the MapReduce model

Suppose vertex *i* is the last vertex in some path. The next vertex should satisfy three requirements: (1) it should be *i*'s neighbor; (2) it should not have been selected before in the path; (3) it should be selected according to Equation (8). When neighbor *q* is selected, it is added to the path. The vital consideration here is that the newly chosen *q* is output as the key and the input key *i* is appended to the predecessor list that is output as value. In this way, we can easily collect paths ending with *q* and *q*'s neighbors using key *q* to start the next round of selection. Two conditions lead to termination of the selection process: number of the edges in the path reaches *k*, or all neighbors exist in the path. Fulfillment of any aforementioned condition implies that the *k*-path is built.

(3) Update of edge weights

When edges are added to the paths, their weights are updated in a parallel fashion. A sketch of the MapReduce model corresponding to edge weight updates is shown in Fig. 10.

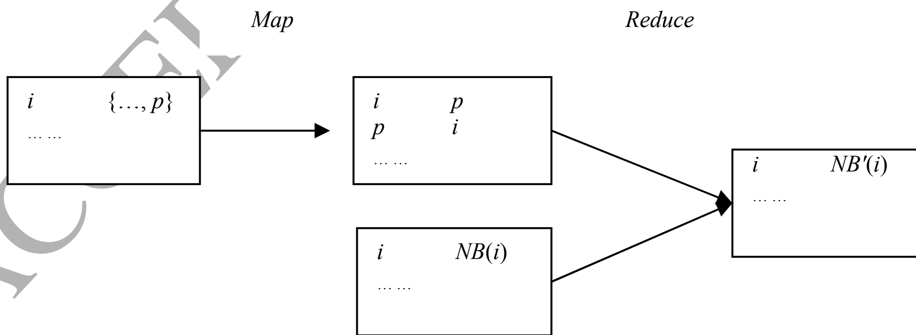


Fig. 10. Update of edge weights based on the MapReduce model

Using the special design of path representation introduced above, it is easy to extract a new edge e_{ip} from the key and the

ACCEPTED MANUSCRIPT

last vertex in the output value list. Combined with the old neighbor information list of i, the neighbor tuple <p, w_{ip}> can be easily updated to <p, $w_{ip}+1/\rho$>.

6.1 PKpath algorithm

This subsection details the PKpath algorithm based on the design ideas presented in the previous subsection. Similar to the LAPoMR algorithm, owing to the availability of a greater number primitives, the Spark implementation of the PKpath algorithm is more flexible than its Hadoop counterpart. The details of the PKpath algorithm implementation in the two frameworks are presented below.

Algorithm 4. PKpath Algorithm (Hadoop)

Input: network $G=(V,E)$, where V is the vertex set and E is the edge set, the maximum path length k
Output: $\{(i, NB'(i))\}$, where $NB'(i)=\{<j,w'_{ij}>\}$ and w'_{ij} is the k-path centrality of edge e_{ij}

1. //STARTING VERTEX SELECTION
2. map (key i, value $NB(i)$)
3. output (key i, $|NB(i)|$);
4. reduce (key i, values $\{|NB(i)|\}$)
5. select ρ vertices with high degrees;
6. output (key v_i, value {}); //i=1… ρ
7. **for** $iter$=1 to k **do**
8. //NEXT VERTEX SELECTION
9. map (key i, value $P_{i,p}$) // Identity Mapper pattern, $P_{i,p}=\{\ldots,p\}$ is a path ended with i and p is i's predecessor
10. output (key i, value $P_{i,p}$);
11. map (key i, value $NB(i)$) // Identity Mapper pattern
12. output (key i, value $NB(i)$);
13. reduce (key i, values { $P_{i,p}$, $NB(i)$})
14. select the next vertex q for each path ended with i according to equation (8);
15. output (key q, value $P_{q,i}$);
16. //EDGE WEIGHT UPDATE
17. map (key i, value $P_{i,p}$)
18. output (key i, value p);
19. output (key p, value i);
20. reduce (key i, values {p, q, …})
21. output (key i, value{p, q, …});
22. map (key i, value $NB(i)$) // Identity Mapper pattern
23. output (key i, value $NB(i)$)
24. map (key i, value{p, q, …}) // Identity Mapper pattern
25. output (key i, {p, q, …})
26. reduce(key i, values {$NB(i)$, {p, q, …}}
27. update the weights of the neighbors of i in $NB(i)$ according to the newly added edges;
28. output (key i, value $NB'(i)$)
29. **end for**

Algorithm 5. PKpath Algorithm (Spark)

Input: network $G=(V,E)$, where V is the vertex set and E is the edge set, the maximum path length k
Output: $\{(i, NB'(i))\}$, where $NB'(i)=\{<j,w'_{ij}>\}$ and w'_{ij} is the k-path centrality of edge e_{ij}

1. //STARTING VERTEX SELECTION
2. map (key i, value $NB(i)$)
3. output (key i, $|NB(i)|$);
4. reduce (key i, values $\{|NB(i)|\}$)

ACCEPTED MANUSCRIPT

5. select ρ vertices with high degrees;
6. output (key v_i, value {}); //i=1… ρ
7. **for** *iter*=1 to k **do**
8. //NEXT VERTEX SELECTION
9. reduce (key i, values { $P_{i,p}$, $NB(i)$}) // $P_{i,p}$={…,p} is a path ended with i and p is i's predecessor
10. select the next vertex q for each path ended with i according to equation (8);
11. output (key q, value $P_{q,i}$);
12. //EDGE WEIGHT UPDATE
13. map (key i, value $P_{i,p}$)
14. output (key i, value p);
15. output (key p, value i);
16. reduce (key i, values {p, q, …})
17. output (key i, value{p, q, …});
18. reduce(key i, values {$NB(i)$, {p, q, …}}
19. update the weights of the neighbors of i in $NB(i)$ according to the newly added edges;
20. output (key i, value $NB'(i)$)
21. **end for**

Two differences exist between algorithms 4 and 5. The first pertains to the next vertex selection. The Hadoop implementation employs two map tasks first to collect the data for the reduce task because a job in Hadoop must be started with map tasks. In contrast, after using its *join* primitive to join the required data, the Spark implementation can directly start the reduce task. The second difference is in the edge weight update. For the same reason, the results of the first reduce task can be sent directly to the second reduce task in the Spark implementation, whereas two extra map tasks are employed in the Hadoop implementation. The reductions in the number of map/reduce tasks and the memory-cache mechanism ensures that the Spark implementation performs better than the Hadoop implementation, as was demonstrated experimentally in this study. When each edge is weighted according to its k-path centrality, the weights can applied to the similarity calculation procedure in algorithms 2 and 3.

6.2 Complexity analysis

Property 6. The time and space complexities of the PKpath algorithm are $O(m/n_m)$ and $O(m)$, respectively, where n is the number of vertices, m is the number of edges, and n_m is the number of machines.

Description: The time and space complexities of the WER-Kpath algorithm are $O(k \times m)$ [34]. Therefore, when the computation is carried out on n_m machines, the time complexity will be $O(k \times m/n_m)$. However, the space occupation remains $O(m)$ because the entire network must be stored before running the algorithm. As suggested in [34], k is usually an integer far smaller than m. Therefore, the time and space complexities of the PKpath algorithm are $O(m/n_m)$ and $O(m)$, respectively.

7. Experiments

Comprehensive experiments were conducted on both artificial and real social networks to evaluate the performance of the proposed algorithms. The artificial networks were generated using the benchmark network generator developed by Lancichinetti and Fortunato [25]. The real social networks were downloaded from the SNAP project website [30]. Table 1 summarizes the details of the experimental network, with n, m, μ, k, and c denoting the number of vertices and network edges, mixing parameter required by the network generator, average degree of the vertices, and community size, respectively.

ACCEPTED MANUSCRIPT

Using the network generator, we can develop artificial networks of sizes varying from 1 *k* (one thousand) to 1 *m* (one million), wherein 10% of the vertices are mixed. The degree of a vertex varies from 20 to 50, and the size of a community varies from 10 to 100. All directed networks are transformed into undirected networks by considering two directed edges between a pair of vertices as a single edge.

Clustering accuracy and running time are the most widely used indices to evaluate a clustering algorithm. In this study, the clustering accuracies and running times of the proposed algorithms were compared under different network sizes and numbers of machines. The clustering accuracy was measured using normalized mutual information (NMI) [9, 26, 47] if the community structures of a network were known or the modularity *Q* [15] if the information was absent. The equations for NMI and *Q* are as follows:

$$NMI(X \mid \mathrm{Y}) = 1 - [\mathrm{H(X \mid Y)} + \mathrm{H(Y \mid X)}] / 2 \tag{10}$$

where *X* and *Y* denote the set of true communities and the set of discovered communities, respectively.

$$Q = \sum_{c=1}^{n_c} [\frac{l_c}{m} - (\frac{d_c}{2m})^2] \tag{11}$$

where m, n_c, l_c, and d_c is the edge number, the number of communities, the number of internal edges in community *c*, and the sum of degrees in community *c*, respectively.

Table 1

Experimental networks details

Network	Description
Artificial networks	n = 1*k*~1*m*(i.e. 1,000 to 1,000,000) μ = 0.1 k = 20~50 c = 10~100
Real social networks	
CA-HepPh	Undirected collaboration network of Arxiv High Energy Physics, *n*= 12008 *m*=118505
Email-Enron	Undirected email communication network from Enron, *n*= 36692 *m*=183831
web-Stanford	Directed web graph of Stanford.edu, *n*= 281903 *m*=3985272
com-youtube	Undirected Youtube online social network, *n*=1134890 *m*=2987624

We compared the LAP algorithm with two well-known community discovery algorithms, the BGLL algorithm [6] and the Clauset algorithm [8], on a single machine. In addition, we compared the LAPoMR algorithm with the Mahout and Spark implementations of the KMeans algorithm (called KMeans(Mahout) and KMeans(Spark) hereinafter) in terms of their clustering accuracy and running time on a cluster. The LAP-based algorithms exhibited fast convergence, and thus, we set *maxIter* = 10 and *convIter* = 2; *p* was set to −10.0 for ensuring that all algorithms were insensitive to varying *p*; for the *k*-path edge centrality calculation, *ρ* and *k* were set to *n*log*n* and 5 separately, as suggested in [24]. For the KMeans algorithm, *K* was set to the true cluster number and the input was a sparse matrix (for space saving) with its elements representing the similarity between two connected vertices, as calculated using Equation (4). All experiments were performed on a cluster composed of eight machines virtualized using VSphere 5.5. All machines were of the same configuration as follows: 2.0 GHz 2-core CPU, 2 GB RAM, and CentOS 6.4 (64 bit).

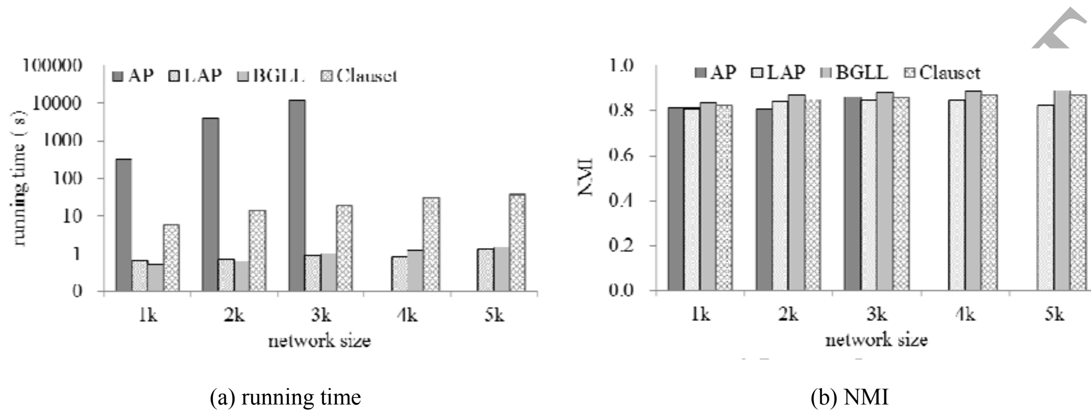
7.1 Experiment results on artificial networks

(1) Performance of the algorithms on single machine

The performances of the LAP, standard AP, BGLL, and Clauset algorithms were compared. The experimental results are shown in Fig. 11. The standard AP algorithm ran out of memory on a single machine for network sizes greater than 3*k* because of its O(n^2) space complexity. Therefore, it was not used in the experiments involving 4*k* and 5*k* artificial networks.

ACCEPTED MANUSCRIPT

To display the difference among the algorithms in terms of running time, the scale of the vertical axis was set to the power of 10. As can be seen in Fig. 11(a), the running time of the LAP, BGLL, and Clauset algorithms is nearly negligible when compared with that of the standard AP algorithm. The LAP and BGLL algorithms finished the work in less than 2 seconds in all artificial networks. The Clauset algorithm took about tens of seconds. By contrast, the standard AP algorithm required a considerable amount of time to finish the same work because of its quadratic time complexity. The clustering accuracy of the LAP algorithm is slightly inferior to that of the other algorithms, as shown in Fig. 11(b), because of the local affinity propagation strategy employed in it. However, the loss in NMI is less than 0.06.

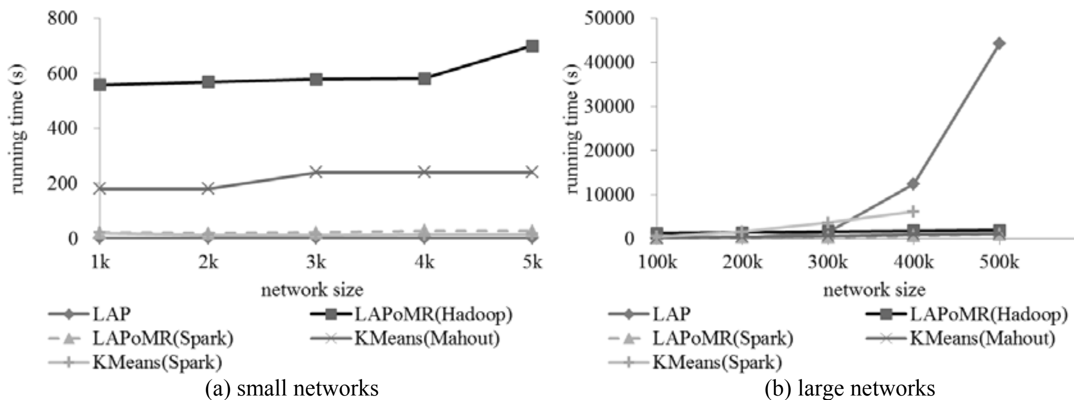


(a) running time (b) NMI

Fig. 11. Running time and accuracy of AP, LAP, BGLL and Clauset under varying network sizes

(2) Effect of network size

The running time and clustering accuracy of the LAP, LAPoMR(Hadoop), LAPoMR(Spark), KMeans(Hadoop), and KMeans(Spark) algorithms with varying network sizes are shown in Figs. 12 and 13. The LAPoMR(Hadoop), LAPoMR(Spark), KMeans(Hadoop), and KMeans(Spark) algorithms were run on a cluster with 8 machines. The experimental networks were partitioned into two groups of small and large networks to reveal similarities and differences in algorithm performance under different network scales.



(a) small networks (b) large networks

Fig. 12. Running time of LAP, LAPoMR(Hadoop), LAPoMR(Spark), KMeans(Hadoop) and KMeans(Spark) under varying network sizes

As shown in Fig. 12(a), the LAPoMR(Hadoop) and KMeans(Mahout) algorithms are slower than the other algorithms when run in small networks. Running time varied minimally with respect to network size, which was contrary to our expectations (investigation of this phenomenon is discussed later in this section). As shown in Fig. 12(b), the running time of

ACCEPTED MANUSCRIPT

the LAPoMR algorithm in large networks was close to that of the LAP algorithm, even surpassing it for network sizes larger than 300 *k*. Therefore, the near-linear time complexity of the LAP and LAPoMR algorithms is prominent in sufficiently large networks. In addition, we deduce that the workloads of framework maintenance and data exchange among machines affect small networks more than large networks. The KMeans(Mahout) algorithm ran out of memory in networks larger than 100 *k*. Moreover, it was the slowest algorithm. The KMeans(Spark) processed all networks except the largest one. However, in large networks, it required more time than the other algorithms except KMeans(Mahout). The input matrix to the KMeans algorithm was composed of *k*-dimensional vectors, where *k* was the average degree of the vertices. Therefore, when both network size and average degree are large, the performance of the MapReduce–based KMeans algorithm may be greatly affected. Moreover, the Mahout implementation of the KMeans algorithm is always inferior to the Spark implementation. Similarly, in all experiments, the performance of the LAPoMR algorithm implementation in Spark was consistently better than that of its implementation in Hadoop. The results demonstrate the efficiency of the Spark framework because of its abundant MapReduce primitives and the full use of both memory and disks for caching.

The NMI values of the algorithms under different network sizes are shown in Fig. 13. The values of the LAP and LAPoMR algorithms are similar and are consistently better than that of the KMeans algorithms. This finding indicates that the strategies employed by the LAP algorithm are effective and well scalable. The basic ideas of the two LAPoMR algorithms are the same as that of the LAP algorithm, although the former employ MapReduce for enhancing their parallel processing capability. The absence of the results pertaining to the KMeans algorithms in Fig. 13(b) is owing to the insufficient memory error encountered by these algorithms on large networks. Moreover, the accuracy of the KMeans(Spark) algorithm is always better than that of the KMeans(Mahout) algorithm.

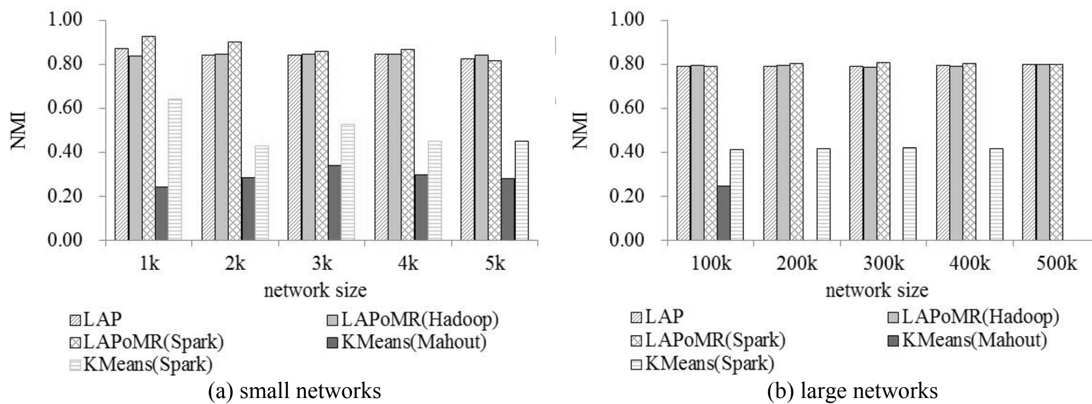


Fig. 13. Clustering accuracy of LAP, LAPoMR(Hadoop), LAPoMR(Spark), KMeans(Hadoop) and KMeans(Spark) under varying network sizes

To investigate the cause underlying the experimental results shown in Fig. 9(a), experiments were performed to measure the actual running time of the algorithms (i.e., effective work time) and the time for extra system maintenance and data exchange of the Hadoop/Spark framework (i.e., extra time). In addition, we defined an index called effective ratio to measure the amount of effective work done by the Hadoop/Spark framework. The effective ratio was calculated by dividing the effective work time with the extra time in the first iteration of the first round. The experimental results are shown in Fig. 14.

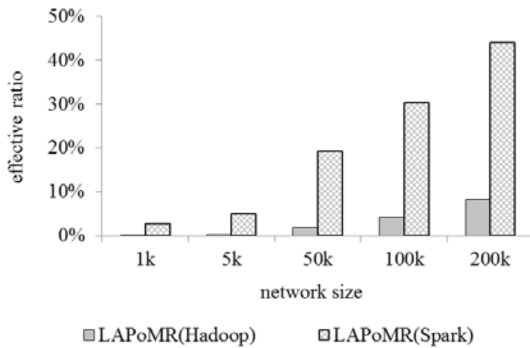


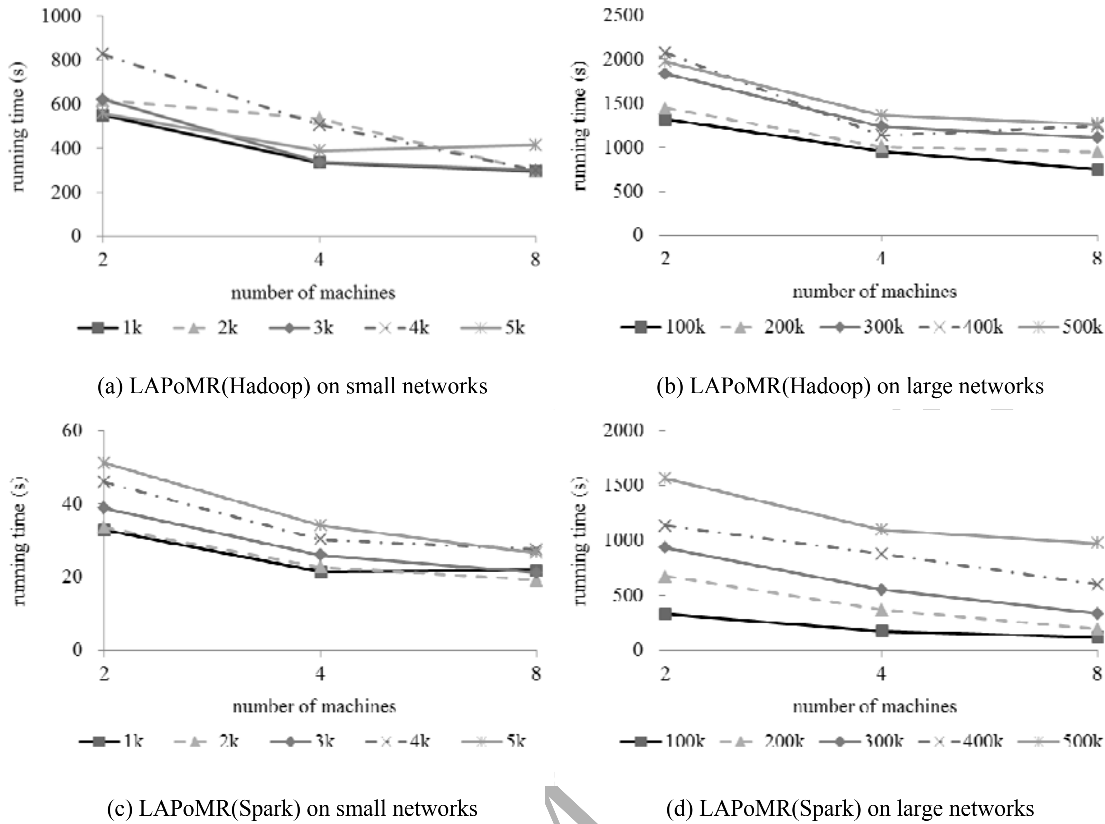
Fig. 14. Effective ratio of LAPoMR(Hadoop) and LAPoMR(Spark)

Only a small fraction of the total time was spent on running the algorithms in a small network, as shown in Fig. 14. Most work time was spent on starting, stopping, scheduling the jobs and tasks, and exchanging data among machines. Effective work time increased as the network scale increased. Effective work time was comparable with extra time, and the effective ratio could be increased to approximately 50%, as shown in Fig. 14. In addition, the effective ratio of the Spark implementation was higher than that of the Hadoop implementation. A large network size revealed further the contrast between the two implementations. This phenomenon was ascribed to the following two factors. The aforementioned additional map tasks and special key/value design of the Hadoop implementation incurred greater expenses than those of the Spark implementation. The data exchanged between MapReduce jobs should be stored in the HDFS within Hadoop, which reduces the data processing speed of the Hadoop implementation. The situation worsened with increasing network size. The Spark framework provides flexible MapReduce primitives, such as *flatMap*, *groupByKey*, and *mapValues*. Therefore, the implementation of the LAPoMR algorithm in the Spark framework was more compact and efficient than that in the Hadoop framework. Moreover, the Spark framework prefers to store data in memory. Spark stores data on disks only in cases where memory is insufficient. Owing to the aforementioned advantages, the Spark implementation achieved better results than the Hadoop implementation.

(3) Effect of number of machines

The running times of LAP, LAPoMR(Hadoop), and LAPoMR(Spark) under different numbers of machines are shown in Fig. 12. The experimental results can be explained from two viewpoints. From the network size perspective, the increase in the number of machines did not reduce running time proportionally when the network was small, as shown in Figs. 15(a) and 15(c). This phenomenon was mainly ascribed to the fact that most of the time was spent on the extra work of the framework itself, as shown in Figs. 12 and 14. Therefore, the expected acceleration owing to the addition of machines to the cluster was weakened. The time reduction in Fig. 15(c) is more noticeable than that in Fig. 15(a) because of the efficient implementation of the Spark framework. For large networks, as shown in Figs. 15(b) and 15(d), the speedup due to the addition of machines to the cluster was significant. When the number of machines was increased from 2 to 8, the running time of the LAPoMR(Hadoop) algorithm on the 500*k* network decreased from 2000 to 1200 s. The LAPoMR(Spark) algorithm exhibited a significant time reduction from 1500 to 1000 s. Thus, the strength of the parallel computation frameworks is demonstrated. The Spark implementation is 2–10x faster than the Hadoop implementation, which is consistent with the results in [48].

ACCEPTED MANUSCRIPT



(a) LAPoMR(Hadoop) on small networks (b) LAPoMR(Hadoop) on large networks

(c) LAPoMR(Spark) on small networks (d) LAPoMR(Spark) on large networks

Fig. 15. Running time of LAPoMR(Hadoop), and LAPoMR(Spark) under varying numbers of machines

We can draw three conclusions from Fig. 15. First, parallel computation frameworks such as Hadoop or Spark may be unsuitable for small networks. These frameworks were designed mainly for increasing efficiency in the parallel processing of large-scale datasets. Therefore, with small networks, the reduction in running time as the number of machines is increased is not significant. Second, there is a tradeoff between the speedup brought about by parallel computation and the cluster scale. As can be seen from Figs. 12(b) and (d), when the number of machines was increased from 2 to 4, the running times of both the Hadoop and Spark implementations of the LAPoMR algorithm were nearly halved. However, the speedup effect weakened when the number of machines was increased from 4 to 8. The greater the number of machines added to the cluster, the greater is the amount of communication and administration work required. As can be seen in Fig. 14, the additional work weakens the benefits of parallel computation. Third, the Spark framework has a greater number of flexible parallel primitives and a more efficient cache mechanism; consequently, it performs better than the Hadoop framework. Thus, the design of MapReduce primitives and synthetic application of fast, massive memory are important.

(4) Number of discovered communities

Table 2 lists the number of communities discovered by the algorithms against the true number of communities. Two inferences can be made using Table 2. First, in all networks, the number of communities determined by the algorithms was slightly higher than the true number. This finding can be attributed to the effect of localizing message propagation, which limits the extent to which a vertex can broadcast its influence. Second, the numbers of communities discovered by the LAP and LAPoMR algorithms were nearly the same. Both algorithms conformed to the strategies introduced in Section 4.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

However, the LAP algorithm was run on a single machine, whereas the LAPoMR algorithm was run on a cluster of machines.

Table 2

Number of communities discovered by the algorithms

Network size	True number of communities	LAP	LAPoMR(Hadoop)	LAPoMR(Spark)
1k	25	28	28	28
2k	51	59	55	56
3k	72	82	87	78
4k	101	116	115	115
5k	125	142	146	150
100k	2571	2748	2720	2720
200k	5169	5431	5407	5549
300k	7804	8362	8226	8401
400k	10374	10968	10955	11058
500k	12935	13702	13471	13761

7.2 Experiment results on real social networks

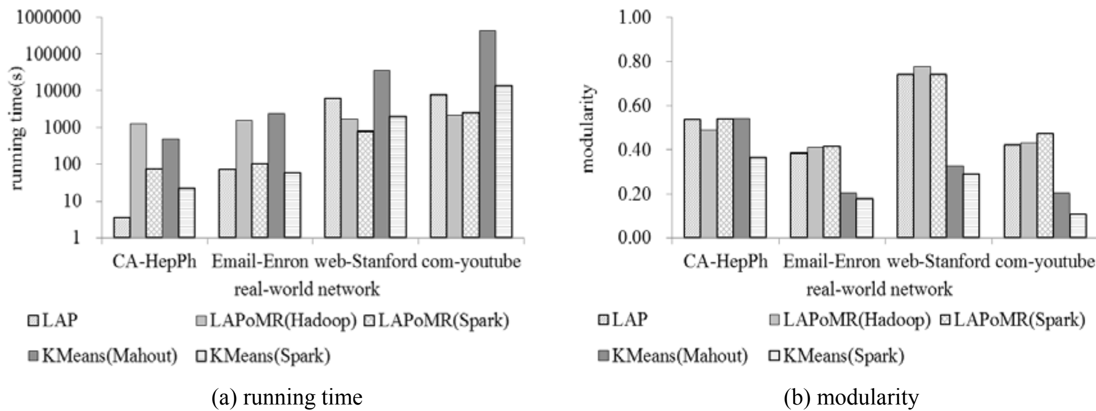
Real social networks differ significantly in scale and contain complex community structures that can hardly be simulated using computer programs. Therefore, experiments were performed on real social networks to evaluate the performance of the algorithms in the real-world environment.

(1) Running time and clustering accuracy

The running times and clustering accuracies of the LAP, LAPoMR(Hadoop), LAPoMR(Spark), KMeans(Mahout), and KMeans(Spark) algorithms in different real social networks are shown in Fig. 16. The NMI index was replaced with the modularity index for accuracy measurement because the number of communities in real networks is unknown. The cluster used for the experiments included 8 machines.

ACCEPTED MA

ACCEPTED MANUSCRIPT



(a) running time (b) modularity

Fig. 16. Running time and accuracy of LAP, LAPoMR(Haddop), LAPoMR(Spark), KMeans(Hadoop) and KMeans(Spark) in real networks

Because the running times of the algorithms vary enormously in different networks, the scale of the vertical axis of Fig. 16(a) was changed to the power of 10, as in Fig. 11. The running times of the algorithms in the CA-HepPh and Email-Enron networks were significantly lower than those in the web-Stanford and com-youtube networks, as shown in Fig. 16(a). The running time of the LAPoMR(Spark) algorithm was comparable with that of the LAP algorithm for the same network, whereas the running time of the KMeans(Mahout) was slower than that of all other algorithms. The first phenomenon may be attributed to the skewed degree distribution of the web-Stanford network, although the number of vertices in the network was not large. The network contained a few vertices with large numbers of neighbors, which considerably augmented the number of edges in the network. This feature is typical of complex networks called *scale-free* networks [15]. The second phenomenon is attributed to the advantages of the Spark framework over the Hadoop framework and the inefficiency of the MapReduce-based KMeans on large networks, as discussed in the preceding subsections.

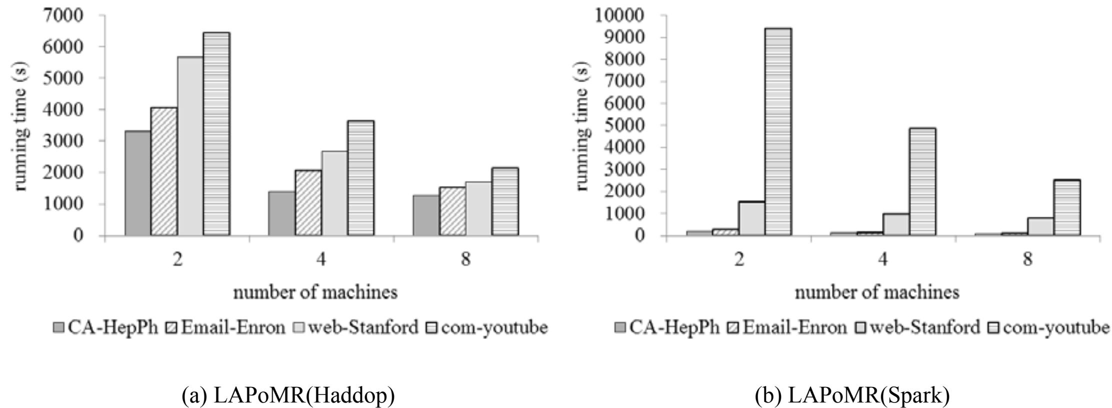
The modularities of the algorithms in various real networks were different, as shown in Fig. 16(b). The Email-Enron network consists of complex community structures, and thus, all the algorithms attained the lowest modularity in this network. The web-Stanford network has a skewed degree distribution that helps identify the core vertices of the network. Therefore, clustering accuracy in this network was the highest among the three networks. The three LAP algorithms achieved nearly the same modularities in the same network and were consistently superior to the KMeans algorithms, except in the CA-HepPh network. This result can be explained using the mechanisms employed by the algorithms. The performance of the two KMeans algorithms varied greatly in different networks, owing largely to the randomness inherent in the KMeans algorithm.

(2) Effect of number of machines

Experiments were carried out in real networks to evaluate the effect of the variation of the number of machines in the cluster. The results shown in Fig. 17 are similar to those in Fig. 15, and they reiterate the effectiveness of increasing of the cluster size for decreasing the processing time. The greater the number of machines in the cluster, the shorter is the time required to run the tests. However, the reduction in time achieved by increasing the cluster size from 4 machines to 8 machines is not as much as the reduction achieved when increasing the cluster size from 2 machines to 4 machines. This result agrees well with the conclusion drawn from Fig. 15. Moreover, with the increase in real network complexity from CA-HepPh to Email-Enron to web-Stanford to com-youtube, the running time increases as well. However, by comparing Fig. 17(a) with Fig. 17(b), we find that the Spark implementation is consistently faster than the Hadoop implementation in all networks. The efficient mechanism of the Spark framework plays a major role here.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT



(a) LAPoMR(Haddop) (b) LAPoMR(Spark)

Fig. 17. Running time of LAPoMR(Haddop) and LAPoMR(Spark) in real networks under varying numbers of machines

(3) Number of discovered communities

Table 3 lists the number of communities discovered by the algorithms against the true number of communities on the real networks. Two inferences can be made using Table 3. First, the numbers of communities discovered by the LAP and LAPoMR algorithms were nearly the same. Second, the KMeans algorithms preferred to discover fewer communities, which may be the main cause of their poor performance.

Table 3

Number of communities discovered by the algorithms

Network	LAP	LAPoMR(Hadoop)	LAPoMR(Spark)	KMeans(Mahout)	KMeans(Spark)
CA-HepPh	1295	1241	1265	545	482
Email-Enron	2408	2384	2423	761	1485
web-Standard	10954	8867	10965	4989	4497
com-youtube	21360	21205	21389	1669	1669

8. Conclusion

In anticipation of the era of big data, parallel computation models and frameworks are receiving increasing attention. Given that many machine-learning algorithms such as logistic regression, support vector machines, and *k*-means rely on iterative refinement to obtain optimal solutions [21], it is important to provide mechanisms for efficient parallel iteration. Modern parallel frameworks such as Hadoop and Spark are moving in this direction [2, 14, 23, 48]. Compared with serial algorithms, parallel algorithms that run on clusters of computers have two advantages: the running time can be decreased considerably by processing the data in parallel and the large availability of memory and disk space beyond that in a single computer ensures that data at the TB or even PB scale can be processed.

Discovering communities in social networks is a typical application that requires iterative processing of tremendous amounts of data. This study proposed a LAP algorithm with near-linear time and space complexities. The algorithm was

ACCEPTED MANUSCRIPT

adapted to the MapReduce model via its implementation in the Hadoop and Spark frameworks. Furthermore, a parallel similarity calculation scheme based on *k*-path edge centrality, which considers the local and global information of entire networks, was designed. Experiments were performed on artificial and real networks to demonstrate the community discovery performance of the proposed algorithms in both small and large social networks.

Several issues persist and require further research. First, the strategies employed in the LAP and LAPoMR algorithms achieve fast convergence at the expense of minor loss in clustering quality. This loss can be reduced by incorporating additional strategies from other perspectives. Second, alternatives to *k*-path edge centrality can be tested as vertex or edge weighting schemes for improving similarity measurement. Third, Hadoop and Spark have their respective advantages and disadvantages. Therefore, improving the efficiency of LAP algorithm implementation by considering other parallel computation models or frameworks should be expected in future studies.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China under Grants No. 61103175 and No. 61300104, the Key Project of Chinese Ministry of Education under Grant No.212086, the Fujian Province High School Science Fund for Distinguished Young Scholars under Grand No.JA12016, the Program for New Century Excellent Talents in Fujian Province University under Grant No. JA13021, the Fujian Natural Science Funds for Distinguished Young Scholar under Grant No. 2014J06017, and the Natural Science Foundation of Fujian Province under Grant No. 2013J01230.

References

[1] Y. Y.Ahn, , P. B James., L.Sune. Link communities reveal multi-scale complexity in networks. Nature, 466.7307 (2010): 761-764.

[2] Apache, Apache Hadoop nextGen MapReduce (YARN). http://hadoop.apache.org/docs/ current /hadoop-yarn/hadoop- yarn-site/YARN.html, 2014.

[3] Apache, Machine learning library (MLib). http://spark.apache.org/docs/latest/mllib-guide.html, 2014.

[4] Apache, Apache Mahout. https://mahout.apache.org, 2014.

[5] Apache. Giraph: open-source implementation of Pregel. https://giraph.apache.org, 2014.

[6] V. D. Blondel, J. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of Statistical Mechanics: Theory and Experiment. 2008 (10) (2008) P10008.

[7] Y. Bu, B. Howe, M. Balazinska, M. D. Emst, Haloop: efficient iterative data processing on large clusters, In: Proc. VLDB'10, Singapore, 2010, 24-30.

[8] A. Clauset, M.E.J. Newman, C. Moore, Finding community structure in very large networks, Phys. Rev. E. Stat. Nonlin. Soft Matter Phys. 70 (2004) 066111.

[9] L. Danon, D.-G. Albert, J. Duch, A. Arenas, Comparing community structure identification, J. Stat. Mech. Theory Exp. (2005) P09008.

[10] Datameer Corporation, Datameer. http:// http://www.datameer.com, 2014.

[11] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters. Communications of the ACM. 51 (1) (2008) 107–113.

[12] I. S. Dhillon, Y. Guan, B. Kulis, Weighted graph cuts without eigenvectors: a multilevel approach, IEEE Transactions on Pattern Analysis and Machine Intelligence. 29 (11) (2007) 1944–1957.

[13] J. Duch, A. Arenas, Community detection in complex networks using extremal optimization, Physical Review E. 72 (2) (2005) 4.

[14] C. Engle, A. Lupher, R. Xin, M. Zaharia, H. Li, S. Shenker, I. Stoica. Shark: fast data analysis using coarse-grained distributed memory, In: Proc. of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12), 2012, pp. 689-692.

[15] S. Fortunato, Community detection in graphs, Physics Reports. (103) (2010).

[16] B. J. Frey, D. Dueck, Clustering by passing messages between data points, Science. 315 (5814) (2007) 972–976.

[17] R. H. Gau, T. C. Hsieh, S. W. Tsai, C. P. Cheng, An implementation framework of mapreduce email social network analysis, In: Proc. 6th ACM workshop on Wireless Multimedia Networking and Computing, 2011, 67–70.

[18] M. Girvan and M. E. J. Newman, Community structure in social and biological networks, Proceedings of the National Academy of Sciences of the United States of America, 99 (2002) 7821–7826.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[19] R. Guimera, M. Sales-Pardo, L. A. N. Amaral, Modularity from fluctuations in random graphs and complex networks, Physical Review E. 70 (2) (2004) 025101.

[20] R. Guimera, L. A. N. Amaral, Functional cartography of complex metabolic networks, Nature. 433 (7028) (2005) 895–900.

[21] J. Han, M. Kamber. Data mining: concepts and techniques, third edition, Morgan Kaufmann, Burlington, Massachusetts, 2011.

[22] J. Hopcroft, New directions in computer science, In: Proc. Computing in the 21st Century, Tianjin, 2012.

[23] T. Kajdanowicz, W. Indyk, P. Kazienko, J. Kukul. Comparison of the efficiency of MapReduce and bulk synchronous parallel approaches to large network processing, In: Proc. of the 2012 IEEE 12th International Conference on Data Mining Workshops (ICDMW'12), 2012, pp. 218-225.

[24] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, J. Leskovec, HADI: mining radii of large graphs, ACM Transactions on Knowledge Discovery from Data. 5(2) (2011) 1–24.

[25] A. Lancichinetti, S. Fortunato, Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities, Physical Review E. 80 (1) (2009) 1–8.

[26] A. Lancichinetti, S. Fortunato, J. Kertész, Detecting the overlapping and hierarchical community structure in complex networks, New Journal of Physics. 11 (3) (2009) 033015.

[27] Q. Li, Z. Wang, W. Wang, Yimin Liu, P. Wang, T. Yu, LI-MR: a local iteration map/reduce model and its application to mine community structure in large-scale networks, In: 2011 IEEE 11th International Conference on Data Mining Workshops, 2011, 174–179.

[28] A. Machanavajjhala, A. Korolova, A. Das Sarma, Personalized social recommendations: accurate or private? Proceedings of the VLDB Endowment. 4 (7) (2011) 440–450.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel: a system for large-scale graph processing, In: Proc. of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10), 2010, pp. 135-146.

[30] J. McAuley, J. Leskovec, Learning to discover social circles in ego networks, In: Proc. of 26th Annual Conference on Neural Information Processing Systems 2012, 548–556.

[31] P. D. Meo, E. Ferrara, G. Fiumara, A. Provetti. Generalized louvain method for community detection in large networks, In: Proc. of the 11th International Conference on. IEEE Intelligent Systems Design and Applications (ISDA), 2011, pp. 88-93.

[32] P. D. Meo, E. Ferrara, G. Fiumara, A. Ricciardello. A novel measure of edge centrality in social networks. Knowledge-Based Systems 30 (2012) 136-150.

[33] P. D. Meo, E. Ferrara, G. Fiumara, A. Provetti. Mixing local and global information for community detection in large networks. Journal of Computer and System Sciences, 2013: 21.

[34] P. D. Meo, E. Ferrara, G. Flumara, A. Provetti, Enhancing community detection using a network weighting strategy, Information Sciences, 222 (2013) 648-668.

[35] MPI Forum, The Message Passing Interface (MPI) standard. http:// www.mcs.anl.gov /research/projects/ mpi/standard.html, 2013.

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

[37] G. Palla, I. Derényi, I. Farkas, T. Vicsek, Uncovering the overlapping community structure of complex networks in nature and society, Nature. 435 (7043) (2005) 814–818.

[38] X. Qi, E. Fuller, Q. Wu, Y. Wu, C. Q. Zhang, Laplacian centrality: A new centrality measure for weighted networks, Information Sciences, 194, (2012) 240-253.

[39] M. Rosvall, C. T. Bergstrom, Maps of random walks on complex networks reveal community structure, Proceedings of the National Academy of Sciences of the United States of America. 105 (4) (2008) 1118–1123.

[40] J. Schultz, J. Vierya, E. Lu. Analyzing patterns in large-scale graphs using MapReduce in Hadoop, In: Proc. of 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC), 2012, pp. 1457-1458.

[41] M. Shiga, I. Takigawa, H. Mamitsuka, A spectral approach to clustering numerical vectors as nodes in a network, Pattern Recognition. 44 (2) (2011) 236-251.

[42] S. W. Son, H. Jeong, J. D. Noh, Random field Ising model and community structure in complex networks, The European Physical Journal B. 50 (3) (2006) 431–437.

[43] M. L. Sumedha, M. Weigt, Unsupervised and semi-supervised clustering by message passing: soft-constraint affinity propagation, The European Physical Journal B. 66 (1) (2008) 125–135.

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

[44] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM. 33 (8) (1990).

[45] T. Whit, Hadoop the definitive guide (2nd edition) , O'Reilly, Inc. 2010.

[46] Z. H. Wu, Y. F. Lin, S. Gregory, H. Y. Wan, S. F. Tian, Balanced multi-label propagation for overlapping community detection in social networks, Journal of Computer Science and Technology. 27 (3) (2012) 468–479.

[47] H. Yin, L. Jing, N. Yue, Detecting local communities within a large scale social network using Mapreduce, International Journal of Intelligent Information Technologies (IJIIT) 10.1 (2014): 57-76.

[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, In: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12), 2012, pp. 2-2.

[49] W. Zhao, V. Martha, X. Xu, PSCAN: a parallel structural clustering algorithm for big networks in mapreduce, In: Proc. of the IEEE 27th International Conference on Advanced Information Networking and Applications, 2013, pp. 862–869.

[50] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, M. V. Marathe. SAHAD: subgraph analysis in massive networks using Hadoop, In: Proc. of the 26th IEEE International parallel and Distributed Processing Symposium, 2012), pp. 390-401.

ACCEPTED MANUSCRIPT