Measuring and Maximizing Group Closeness Centrality over Disk-Resident Graphs

Junzhou Zhao[1] John C.S. Lui[2] Don Towsley[3] Xiaohong Guan[1]
[1]MOEKLINNS Lab, Xi'an Jiaotong University, Xi'an 710049, China
[2]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[3]School of Computer Science, University of Massachusetts Amherst, MA 01003, USA
{jzzhao,xhguan}@sei.xjtu.edu.cn cslui@cse.cuhk.edu.hk towsley@cs.umass.edu

ABSTRACT

As an important metric in graphs, group closeness centrality measures how close a group of vertices is to all other vertices in a graph, and it is used in numerous graph applications such as measuring the dominance and influence of a node group over the graph. However, when a large-scale graph contains hundreds of millions of nodes/edges which cannot reside entirely in computer's main memory, measuring and maximizing group closeness become challenging tasks. In this paper, we present a systematic solution for efficiently calculating and maximizing the group closeness for disk-resident graphs. Our solution first leverages a "probabilistic counting method" to efficiently estimate the group closeness with high accuracy, rather than exhaustively computing it in an exact fashion. In addition, we design an I/O-efficient greedy algorithm to find a node group that maximizes group closeness. Our proposed algorithm significantly reduces the number of random accesses to disk, thereby dramatically improving computational efficiency. Experiments on real-world big graphs demonstrate the efficacy of our approach.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Performance

Keywords

Group centrality, Greedy algorithm

1. INTRODUCTION

Node centrality[10] is an important measure in the analysis of networks. Many centrality measures such as degree, closeness, and betweenness, have been proposed to measure the importance of individual nodes in a network. While these measures are useful in finding the top-k most important individual nodes within a network, they are not suitable to address the question of finding a set of nodes of size k, such that these k *nodes as a group*, is the most important group in the network. Such a problem widely exists in scores of application domains. For instance, in online social networks, product retailers may want to locate k people to promote their products so as to maximize the number of potentially influenced customers. Due to the overlap of people's friend circles, simply returning the top k most influential people in the network is unlikely to be optimal.

Everett and Borgatti, in their seminal work [7], extended the idea of *individual centrality* to *group centrality*, which defines the importance of a node group in a graph. They illustrated the concepts of group degree, group closeness and so on for two graphs containing 20 and 14 nodes, respectively. Despite its conceptual novelty, group centrality lacks efficient calculation algorithms that can scale to large graphs containing hundreds of millions of nodes/edges such as the Facebook and Twitter networks. Such graphs call for efficient group centrality calculation methods.

In this work, we study how to efficiently measure and maximize *group degree* and *group closeness* of disk-resident graphs. We introduce these two metrics, and show that they are special cases of a *generalized group closeness* (which we call *group closeness* for short), in Section 2. When a graph cannot entirely fit in the computer's main memory, measuring and maximizing group closeness become challenging.

Challenge of Calculating Group Closeness in Big Graphs: Group closeness centrality measures how close a group of nodes is to all other nodes in a graph. Calculating group closeness requires calculating the shortest path length from each node in the group to all other nodes in the graph, i.e., solving the single-source shortest path (SSSP) problem for each group member. Dijkstra's SSSP algorithm can be efficiently implemented in $O(m + n \log n)$ time using state-of-the-art methods [9, 11], where n is the number of nodes and m is the number of edges of a given graph. However, for contemporary online social networks (OSNs), which include hundreds of millions of nodes/edges, this computational complexity is too large. If we want to find a node group to maximize the group closeness, we need to solve the all-pairwise shortest path (APSP) problem, where the time complexity becomes $O(nm + n^2 \log n)$. Furthermore, the above algorithms are all in-memory algorithms requiring data to fit in the main memory, which are not suitable for processing disk-resident graphs.

Our Solution: Instead of exactly calculating the shortest path length for each node, we propose a computationally efficient method to estimate group closeness centrality. Palmer et al.[19] use a probabilistic counting method [8] to approx-

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2579356.

imate the neighborhood function for each node in a graph, and the neighborhood function can be used to estimate the shortest path length. We leverage this method as a *preprocessing step* to efficiently estimate group closeness with time complexity $O(m)$ and $O(n \log n)$ extra space.

Challenge of Greedily Choosing Nodes from Disk: If we want to choose a subset of items from a population to maximize some given objective function, a widely used approach is the *greedy method:* choosing an item at each step to maximize the *increase* to the objective function. Unfortunately, for large-scale graphs residing on disk, the greedy method will cause too many random accesses on disk, which will induce frequent page faults and thereby increase the computational cost.

Our Solution: To address the disk random access problem, we design a novel I/O-efficient greedy algorithm for processing disk-resident graphs. Our method relies on the submodularity of the objective function (see Subsection 3.3) and its two important properties in order to reduce the number of disk random accesses, and so improve on computational efficiency.

The remainder of this paper is organized as follows. In Section 2, we generalize the definitions of group degree and group closeness and formulate the group closeness maximizing problem. Then we describe the algorithms for processing disk-resident graphs in Section 3. Experiments are conducted in Section 4. We summarize some related work in Section 5, and conclude with Section 6.

2. GROUP CENTRALITY MEASURES

In this section, we first review Everett and Borgatti's original definitions of group degree/closeness. Then we introduce a new notion of generalized group closeness. Lastly, we formulate the group closeness maximization problem and state its complexity.

2.1 Group Degree and Group Closeness

Everett and Borgatti define the *group degree centrality* of a node group as the number of non-group members that are connected to group members, namely

$$C_{\text{deg}}(S) = |\{v : (u,v) \in E \wedge u \in S \wedge v \notin S\}|, \quad S \subseteq V, \quad (1)$$

where $C_{\text{deg}}(S)$ denotes the group degree of a group S, while V and E are the sets of nodes and edges of a graph[1] G respectively.

Group degree centrality only considers one-hop neighbors, but *group closeness centrality* considers all nodes in the graph, and gives higher score to a group of nodes with smaller average distances to all other nodes. Everett and Borgatti define group closeness centrality as follows

$$C_{\text{close}}(S) = \frac{|V \backslash S|}{\sum_{v \in V \backslash S} d_{S,v}}, \quad S \subseteq V, \quad (2)$$

where $d_{S,v}$ is the distance between group S and a node v and defined as $d_{S,v} \triangleq \min_{u \in S} dist_{uv}$ where $dist_{uv}$ is the shortest distance between u and v. Therefore, group closeness centrality measures how close group members in S are to other non-members in a graph.

[1]We only consider undirected unweighted graphs in this work, although these metrics can be easily extended to directed graphs.

2.2 Generalized Group Closeness Centrality

Group degree in Eq. (1) and group closeness in Eq. (2) can be considered as distance scores measuring how close group S is to other nodes in the graph. The closer the group is to other nodes, the larger is the score. In general, let $g : \mathbb{R}_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$ be a monotonically decreasing non-negative function. We define the distance score from group S to other nodes that are within H hops to S in the graph as

$$C_H(S) = \sum_{v \in V} g(d_{S,v}) \mathbf{1}\{d_{S,v} \leq H\} \quad (3)$$

$$= \sum_{h=0}^{H} g(h)\left[N_h(S) - N_{h-1}(S)\right]. \quad (4)$$

Here $\mathbf{1}\{\cdot\}$ is the indicator function, $1 \leq H \leq \Delta$ is a given constant, and Δ is the diameter of G. $N_h(S)$ is the number of nodes within h hops to S in the graph, i.e.,

$$N_h(S) = \begin{cases} 0, & h < 0, \\ |S|, & h = 0, \\ |\{v : d_{S,v} \leq h\}|, & h \geq 1. \end{cases} \quad (5)$$

Thus, $C_1(S) = [g(0) - g(1)]|S| + g(1)N_1(S)$ measures the closeness of S to nodes within one hop of S. Therefore $C_1(S)$ can be used to approximate the group degree in Eq. (1). Similarly, $C_\Delta(S)$ measures the closeness to all the nodes in G, therefore it can be used to approximate the group closeness in Eq. (2).

2.3 Group Closeness Maximization

Given the above metric, an important problem is to find a node group S in the graph that maximizes the group closeness $C_H(S)$. This problem can be formally stated as follows.

DEFINITION 1. (GROUP CLOSENESS MAXIMIZATION PROBLEM). *Given graph $G(V,E)$, H and $g(\cdot)$, find a set $S \subseteq V$ of at most K nodes that maximizes $C_H(S)$.*

For the group closeness $C_H(\cdot)$ we defined above, we have the following results (Proofs are included in [1]):

THEOREM 1. *The group closeness maximization problem is NP-hard.*

THEOREM 2. *Group closeness $C_H(\cdot)$ is a non-decreasing submodular function.*

Based on Theorem 1, finding an optimal solution is computationally difficult. But based on Theorem 2, we can exploit the submodular property of $C_H(\cdot)$ to find an approximation solution that has good performance guarantees. In particular, we have a polynomial time greedy algorithm (GA) to find an approximate solution which is within at least $1-1/e$ of the optimal solution [16], and the approximation is nearly optimal as no known polynomial time algorithm can achieve a better approximation factor. GA can be briefly stated as follows: The node, which maximizes the group closeness increment is chosen and put into the node group at each iteration; this produces a node group of size K after K iterations.

Although GA has polynomial time complexity, it requires that all of the data fit entirely in a computer's main memory. This condition is too stringent for large graphs containing hundreds of millions of nodes/edges, which are very common for today's popular OSNs. In this work, we develop a

novel algorithm that enables one to use a common PC with small memory capacity (1∼4GB) to efficiently find quality guaranteed solutions on a million-scale graph. This will be described in detail in the next section.

3. HANDLING DISK-RESIDENT GRAPHS

Before we describe our methods in detail, it is necessary to explain why the problem becomes challenging when the graph cannot fit in the computer's main memory.

3.1 Challenges of Handling Disk-Resident Graphs

In general, GA has polynomial time complexity. However, when we apply GA to compute the group closeness centrality, we have the additional complexity of calculating group closeness during each round of GA. In each GA round, we calculate the *reward gain* for each node $s \in V \backslash S$, i.e., $\delta_s(S) \triangleq C_H(S \cup \{s\}) - C_H(S)$. However, calculating $\delta_s(S)$ is computationally intensive. To see this, simply let $S = \emptyset$; then $\delta_s(S) = C_H(\{s\}) = \sum_{v \in V} g(dist_{sv})$. That is, we are required to solve SSSP for node s. As a result, we need to solve APSP in GA, which has time complexity $O(nm + n^2 \log n)$. This is obviously expensive (both computation and memory requirements) for a large graph with large n and m.

Another more serious challenge is that GA generates many random disk accesses, which in turns create frequent page faults and further increasing computational time[4]. To illustrate this issue, let us consider that we already use an efficient implementation of GA where nodes are maintained in a priority queue ordered in decreasing reward gain. Each time when a node s is added to S, we need to update the reward gains of affected nodes in the queue, and use an inverted index to quickly look up these affected nodes. An inverted index is a hash map that maps a node u to a list $L_u = \{v_1, v_2, \cdots\}$ and u is within H hops to $v \in L_u$. When s is selected, for each node u within H hops of s, we look up L_u and update $\delta_v(S)$ for $v \in L_u$ in the queue. If the index mapping is small, it can fit in main memory. Unfortunately, the inverted lists $\{L_u\}_{u \in V}$ are usually very large and have to be stored on disk. Since u is unlikely to be visited locally during the iterations in GA, this will cause many random disk accesses.

3.2 Efficiently Estimating the Group Closeness

We address the first challenge of calculating $C_H(\cdot)$ in this subsection. Since an efficient method to calculate $C_H(\cdot)$ can improve the efficiency of GA, we leverage a probabilistic counting method to efficiently estimate $C_H(\cdot)$ with high accuracy rather than exhaustively and exactly calculating it.

The basic idea is to estimate $N_h(S)$ in Eq. (5) and use Eq. (4) to obtain $C_H(S)$. In [19], Palmer et al. use the Flajolet-Martin (FM) sketch method[8] to estimate $N_h(u) \triangleq N_h(\{u\})$ for a node u in a graph. We extend this approach to estimate $N_h(S)$ for a group S. FM-sketch is a *probabilistic counting method* that encodes the *counting* information in a *bit-string*. The method is efficient and requires little extra space.

We first describe how to estimate $N_h(u)$ using the method in [19]. First, a bit-string $M_h(u)$ of $\log n$ length is generated to *encode* $N_h(u)$ for each node $u \in V$, and $M_h(u)$ is obtained by iteratively doing the following bitwise-OR operation

$$M_{h+1}(u) = M_h(u) \oplus \underbrace{M_h(v_1) \oplus \cdots \oplus M_h(v_{d_u})}_{v_i:(u,v_i)\in E,\ i=1,\ldots,d_u},$$

where $\oplus$ denotes the bitwise-OR operation of two bit-strings, d_u is the degree of node u, and $M_0(u)$ is initialized as the binary representation of a uniformly distributed random number. $N_h(u)$ is *decoded* from $M_h(u)$ by $N_h(u) = 2^r/0.77351$, where r is the position of the lowest '0' bit in $M_h(u)$.

We can leverage this method to estimate $N_h(S)$. Suppose we have obtained $M_h(u)$, $\forall u, \forall h$, then a bit-string $M_h(S)$ encoding of $N_H(S)$ can be calculated as follows

$$M_h(S) = \bigoplus_{u \in S} M_h(u).$$

$N_h(S)$ is then decoded in the same way as $N_h(u)$. To increase estimation accuracy, we can store N bit-strings per node per hop and decode $N_h(S)$ by

$$N_h(S) = \frac{1}{0.77351} 2^{\frac{1}{N}\sum_{i=1}^{N} r_i}.$$

$C_H(S)$ is obtained using Eq. (4). Notice that $\delta_s(S)$ is also easy to calculate using such a method. Because $M_h(S \cup \{s\}) = M_h(S) \oplus M_h(s)$, $C_H(S \cup \{s\})$ is easily obtained after decoding and hence $\delta_s(S)$. Obtaining all bit-strings requires time $O(m)$ and extra space $O(n \log n)$.

3.3 An I/O-Efficient Greedy Algorithm

Next, we present an I/O-efficient greedy algorithm to overcome the disk random access problem. The new algorithm leverages two properties of submodular functions to reduce I/O costs.

The first property of submodularity comes from one of its equivalent definitions, which we state as follows.

PROPERTY 1 ([16, PROPOSITION 2.1]). *For a submodular function $F(\cdot)$, let $\delta_s(S) \triangleq F(S \cup \{s\}) - F(S)$. If $S \subseteq T$, then $\delta_s(S) \geq \delta_s(T)$, for all $s \in V \backslash T$.*

This property tells us that as GA proceeds, the reward gain of each node cannot increase. It can be used to reduce the number of reward gain calculations in each round and thereby reduce the I/O cost. For example, if the recently updated node already has the largest gain in the queue, then there is no need to calculate gains for the other nodes. Because their gains will only become smaller according to Property 1.

To further reduce I/O costs, we leverage the following second property of submodularity.

PROPERTY 2. *Suppose at each step t of GA, we can choose a node s_t whose gain δ_{s_t} is at least a fraction λ of the maximum gain $\delta_{s_t^*}$, i.e., $\delta_{s_t} \geq \lambda \delta_{s_t^*}$. Then we can guarantee that the final solution $S_K = \{s_1, \ldots, s_K\}$ satisfies*

$$F(S_K) \geq (1 - \frac{1}{e^\lambda}) F(OPT),$$

where OPT is the optimal solution maximizing F.

PROOF. Please refer to [1]. □

The second property uses λ as a parameter to trade off the solution quality and computational efficiency. Therefore, instead of exhaustively searching a node that has the

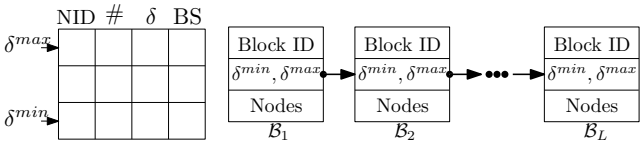
maximum gain at each GA step, we can search for a node for which the gain is at least a fraction λ of the maximum gain, and the solution quality is still bounded. Since this property can reduce the search scope during each GA step, I/O cost decreases.

Based on above two properties, we now design an I/O-efficient greedy algorithm.

Preprocessing: In the preprocessing step, we use the method of the previous subsection to generate $N \times H$ bit-strings for each node, which allows us to calculate the reward gain efficiently. Then we use external sorting methods to sort the nodes in decreasing order of reward gain. Next, the sorted data is split into blocks where each block is of size at most B. B is selected as large as possible while allowing the block to reside in main memory. We only load one block at a time into memory. A block maintains bit-strings for a set of nodes, as shown in Fig. 1a and includes the following fields:

- NID represents the node ID;
- δ denotes the reward gain of the node;
- # is the round number in which round δ is updated;
- BS represents the bit-strings of the node;
- δ^{min} and δ^{max} are two pointers pointing to the nodes having the minimum and maximum gain in the block.

A block is stored as a single file on a disk and it is named after its block ID. Different blocks are not necessarily contiguous on disk. In addition, we maintain a block-meta list in main memory, which records the meta information $\langle$Block_ID, δ^{min}, δ^{max}, Nodes$\rangle$ for each block, as shown in Fig. 1b.



(a) A block. (b) Block-meta list.

Figure 1: Block structure of the data.

Iterations: The algorithm, as depicted in Alg. 1, uses the "*Read-Compute-Write*" framework.

- *Read:* Load the first block into memory (Lines 2, 18).
- *Compute:* To search a node s such that $\delta_s \geq \lambda\delta_{s^*}$, check the head node in the queue and update its gain if necessary (Lines 5-8). Note that δ_{s^*} is unknown, however, the maximum gain δ^{max} in the current block is an upper bound of δ_{s^*} according to Property 1. Hence, if a node c such that $\delta_s \geq \lambda\delta^{max}$, it is placed into S (Lines 9-11); otherwise it is placed back into the queue (Line 13).
- *Write:* If the maximum gain in the current block is smaller than in the next block (Line 16), the current block is written back to disk (Line 17), and a new block is read from disk (Line 18). During the WriteToDisk operation, each node v in the queue is appended to a block i such that $\delta_i^{min} \leq \delta_v \leq \delta_i^{max}$.

Our I/O-efficient greedy algorithm leverages Properties 1 and 2 to load blocks *on demand* from the disk, thus reducing I/O cost.

4. EXPERIMENTS

In this section, we conduct experiments on typical real-world graphs to demonstrate the efficacy of our method.

Algorithm 1: I/O-Efficient Greedy Algorithm.

Input: Approximate ratio λ and group size K.
Output: Node group S.

```
1  C = ∅, t = 1;
2  Load the first block into priority queue Q;
3  while |S| < K, do
4      v = Q.Pop();
5      if #_v < t, then                                  /* Update */
6          δ_v = C_H(S ∪ {v}) − C_H(S);
7          #_v = t;
8      end
9      if δ_v ≥ λδ^max, then                          /* If succeed */
10         S = S ∪ {v};
11         t = t + 1;
12     else                                            /* If failed */
13         Q.Insert(v);
14     end
15     Update δ^max and δ^min;
16     if δ^max < δ_2^max, then                  /* Write and reload */
17         WriteToDisk(Q);
18         Load the first block into priority queue Q;
19     end
20 end
```

Function WriteToDisk(Q):

```
1  Delete the first block from block-meta list and disk;
2  foreach element v in Q do
3      Find the block i s.t. δ_v ∈ [δ_i^min, δ_i^max];
4      Write v to the end of block i on disk;
5  end
```

4.1 Datasets and Experimental Environment

The datasets are four real-world graphs of different sizes, an ArXiv High Energy Physics citation network (HEPTH), a Youtube social network, a LiveJournal social network and a Twitter follower network. Table 1 summarizes the basic information of these four graphs.

Network	n	m	Δ	B	Blocks	Preproc.
HEPTH	27K	352K	10	5K	6	$<$ 1min
Youtube	1M	3M	14	5K	228	$<$ 1min
LiveJournal	5M	49M	16	5K	1038	8min
Twitter[13]	40M	258M	18	10K	2296	55min

Table 1: Dataset summary

We perform the experiments on a Linux Ubuntu 12.04 desktop with a dual-core 2.13GHz Intel Processor, 2GB of main memory and a standard 180GB, 7200rpm SATA HDD.

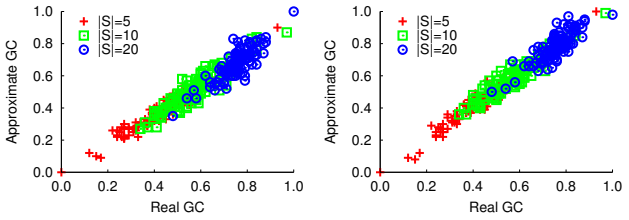
When calculating group closeness, we set $g(0) = 1$ and $g(h) = 1/h$ for $h \geq 1$.

4.2 Validation of the Group Closeness Approximation

First, we show that the proposed method in Section 3.2 can well approximate group closeness in the HEPTH network. We generate 3×100 node groups with size $|S| = 5, 10$, and 20, respectively. Nodes in each group are randomly chosen from the network. Because the network is small, we can calculate the exact values of group closeness for these node groups.

Figure 2 shows the scatter plots of the exact group closeness values and the approximate group closeness values. Both

metrics are normalized to the range $[0, 1]$. These two metrics show strong correlations, and if we increase the number of bit-strings per node per hop from $N = 16$ to $N = 32$, the Pearson Correlation Coefficient (PCC) increases from 0.962 to 0.975. This indicates that our proposed method in Section 3.2 can well approximate true group closeness.



(a) $N = 16$ (PCC=0.962) (b) $N = 32$ (PCC=0.975)

Figure 2: Approximate validation (GC denotes group closeness and $H = 7$).

4.3 Performance and Scalability

Next, we compare the performance and efficiency with existing methods. In order to let the existing greedy algorithm handle big disk-resident graphs with small memory, the greedy algorithm is designed to scan the disk multiply times, and a node maximizing the reward gain is selected after each scan. We use this approach as the baseline method.

Figures 3(a) and 3(e) show group closeness values for different group sizes using the baseline method and the proposed method. In the proposed method, we set $\lambda = 1.0$ and 0.5 respectively. When $\lambda = 1.0$, the proposed method performs exactly the same as the baseline method, and when $\lambda = 0.5$, the proposed method performs worse than the baseline method, which is expected due to Property 2.

Figures 3(b) and 3(f) show the computation times to find node groups of different sizes using the two methods. We see that the baseline method is more time consuming than our method. Fig. 3(b) also reveals that our method requires longer times for $\lambda = 1.0$ than for $\lambda = 0.5$, but in Fig. 3(f), their performance are comparable.

To clearly see the effect of λ on performance, Figs. 3(c) and 3(d) show how λ impacts page faults (# new block loads into memory). We observe that increasing λ results in more page faults. Figs. 3(d) and 3(g) also show the trade-off effects of λ for selecting a group of 20 nodes. When λ increases, we obtain better solutions, but at the cost of longer execution times.

4.4 Observations on Big Graphs

In this section, we present patterns of node groups maximizing the group closeness on Livejournal and Twitter. We calculate group closeness using $\lambda = 0.5$, and $H = 1, 2, 3$ respectively. The preprocessing step for Livejournal costs eight minutes and for Twitter is 55 minutes (for $H = 3$). Applying the I/O-efficient greedy algorithm on Livejournal takes three minutes for $H = 3, \lambda = 0.5$, and for Twitter 30 minutes (for a group size 1000).

Several researchers[15, 21] have suggested using degree as an alias for other node centrality metrics. Hence, we ask the following question: *Are the top k largest degree nodes good aliases for group of size k maximizing C_H?* To answer the question, we choose node groups of various sizes and compare their overlaps with the top k largest degree nodes in the same network. Fig. 4 shows the results on the two networks respectively. Surprisingly, the overlap is low and becomes even lower as k and H increase. For example, for $H = 1$, the node group of size 200 contains only about 10% of the top-200 largest degree nodes.

In conclusion, group closeness is a useful metric that cannot be simply represented by the existing degree metric.

5. RELATED WORK

There is a vast literature on scaling up the single node centrality calculation over large graphs, but little work on scaling up group centrality calculation.

Scaling up the Single Node Centrality Calculation: Many single node centralities such as closeness and betweenness require solving SSSP first. For closeness centrality, Eppstein and Wang[6] only calculate the distances to a number of sampled nodes, where the time complexity is reduced to $O(\frac{\log n}{\epsilon^2}(m + n \log n))$ and an error bound is given by applying Hoeffding's inequality. Okamoto et al. [17] leverage the above result and present a similar algorithm. However these are in-memory methods for single node closeness centrality and not suitable for our setting.

Recently, there are increasingly many works scaling centrality calculations by distributing the computation using MapReduce [5]. For example, Kang et al. [12] develop a parallel graph mining tool to estimate single node centrality on Hadoop. Oktay et al. [18] present a method to estimate pair-wise nodes shortest distance using MapReduce. Sariyüce et al. [20] present a distributed framework for calculating closeness centrality incrementally over dynamic graphs. However, developing distributed graph algorithms remains a challenging task, and there is still a need for optimizing graph algorithms on a single machine [14].

Scaling up the Greedy Algorithm: The greedy algorithm is a heuristic approach used to solve many NP-hard problems such as the travelling salesman problem and the set cover problem. Despite its importance, relatively little effort has focused on scaling it up for large datasets. Recently, Cormode et al. [4] present a variation of the greedy algorithm for the set cover problem on large datasets. However, our problem cannot be easily converted to a set cover problem and hence we cannot apply their method.

Another approach to scale up the algorithm is parallelization. The greedy algorithm is inherently sequential in nature. To relax this constraint, Berger et al. [2] conduct a study of the set cover problem, in which multiple processors can randomly cover sets and avoid covering the same elements redundantly. Inspired by Berger's work, Chierichetti et al. [3] develop an algorithm for the max-cover problem in combining with the MapReduce framework. These two methods require data fitting in main memory so that multiple processors can randomly access data; if data residents on disk, random access on disk will cause I/O costs. Therefore they are not suitable in our setting.

6. CONCLUSION

Group closeness centrality is an important metric in measuring how close a group of nodes to other nodes in a graph. However, it is not easy to calculate/maximize when graphs contain hundreds of millions of nodes/edges which cannot entirely fit in the computer's main memory. We present a systematic solution for efficiently calculating group closeness

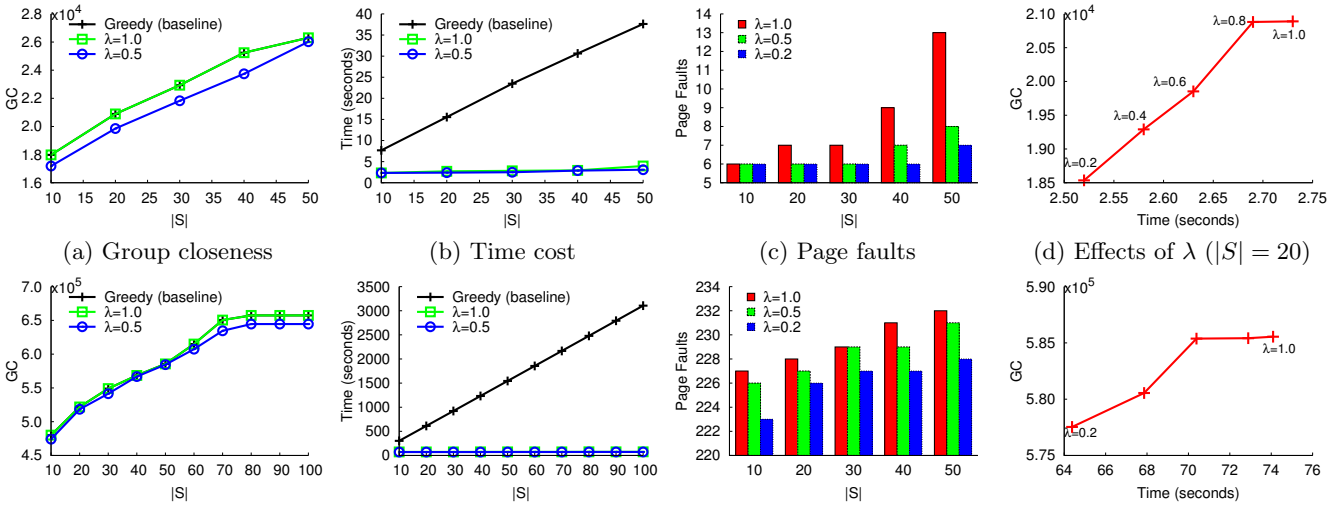


Figure 3: Scalability and performance. Top row is on HEPTH network and bottom row is on Youtube network. For both networks, we set $H = 7, N = 32$.

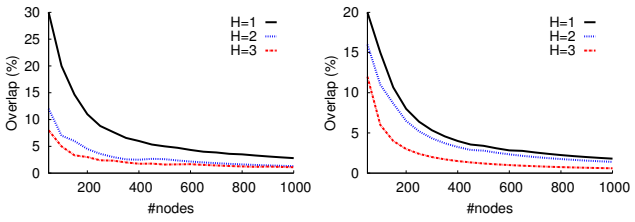


Figure 4: Overlap between group of nodes and top degree nodes ($\lambda = 0.5$).

centrality over disk-resident graphs. Our solution leverages the FM-sketch method to efficiently approximate the group closeness and exploits two properties of submodular functions to maximize the group closeness measure. The experiments demonstrate the efficacy of this approach.

Acknowledgement

This work was partly supported by the NSF grant CNS-1065133 and ARL Cooperative Agreement W911NF-09-2-0053.

7. REFERENCES

[1] ————. Technique report. `http://nskeylab.xjtu.edu.cn/dataset/jzzhao/NodeGroup_TR.pdf`, 2013.

[2] B. Berger, J. Rompel, and P. W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *JCSS*, 49(3), 1994.

[3] F. Chierichetti, R. Kumar, and A. Tomkins. Max-Cover in Map-Reduce. In *WWW*, 2010.

[4] G. Cormode, H. Karloff, and A. Wirth. Set cover algorithms for very large datasets. In *CIKM*, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[6] D. Eppstein and J. Wang. Fast approximation of centrality. *Journal of Graph Algorithms and Applications*, 2004.

[7] M. G. Everett and S. P. Borgatti. The centrality of groups and classes. *Journal of Mathematical Sociology*, 23(3):181–201, 1999.

[8] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31(2), 1985.

[9] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34:596–615, 1987.

[10] L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1:215–239, 1978.

[11] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[12] U. Kang, S. Papadimitriou, J. Sun, and H. Tong. Centralities in large networks: Algorithms and observations. In *SDM*, 2011.

[13] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[15] A. S. Maiya and T. Y. Berger-Wolf. Online sampling of high centrality individuals in social networks. In *PAKDD*, 2010.

[16] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of approximations for maximizing submodular set functions. *Math. Prog.*, 14, 1978.

[17] K. Okamoto, W. Chen, and X.-Y. Li. Ranking of closeness centrality for large-scale social networks. *Frontiers in Algorithmics*, 5059, 2008.

[18] H. Oktay, A. S. Balkir, I. Foster, and D. D. Jensen. Distance estimation for very large networks using mapreduce and network structure indices. In *Workshop on Information Networks*, 2011.

[19] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *KDD*, 2002.

[20] A. E. Sariyüce, E. Saule, K. kaya, and U. V. Catalyürek. Streamer : A distributed framework for incremental closeness centrality computation. In *IEEE Cluster 13 Conference*, 2013.

[21] Y. sup Lim, B. Ribeiro, D. S. Menasche, P. Basu, and D. Towsley. Online estimating the top k nodes of a network. In *IEEE NSW*, 2011.