Information Sciences 400–401 (2017) 105–128

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

I/O-efficient calculation of *H*-group closeness centrality over disk-resident graphs

Junzhou Zhao[a], Pinghui Wang[b,*], John C.S. Lui[a], Don Towsley[c], Xiaohong Guan[b]

[a] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin N.T. Hong Kong
[b] MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, 710049, China
[c] Computer Science Department, University of Massachusetts at Amherst, MA, 01003, USA

ARTICLE INFO

Article history:
Received 22 May 2016
Revised 6 March 2017
Accepted 11 March 2017
Available online 12 March 2017

Keywords:
Online social network
Group centrality
Disk-resident graph
Submodularity
Greedy algorithm

ABSTRACT

We introduce *H*-group closeness centrality in this work. *H*-group closeness centrality of a group of nodes measures how close this node group is to other nodes in a graph, and can be used in numerous applications such as measuring the importance and influence of a group of users in a social network. When a large graph contains billions of edges that cannot reside entirely in a computer's main memory, computing and maximizing *H*-group closeness centrality both become challenging. In this work, we present a systematic solution for efficiently computing and maximizing *H*-group closeness centrality in large disk-resident graphs, only using a common PC. Our solution leverages a probabilistic counting method to efficiently estimate *H*-group closeness with high accuracy, rather than exhaustively computing it in an exact fashion. Furthermore, we design an I/O-efficient greedy algorithm to find a node group that approximately maximizes *H*-group closeness centrality in a graph. Our algorithm exploits several appealing properties of the defined *H*-group closeness measure to reduce the computational cost of handling a disk-resident graph. Extensive evaluation on large real-world and synthetic graphs demonstrates the efficiency of our proposed method. For example, our proposed I/O-efficient greedy algorithm is about 300 times faster than a simple multi-pass method on the Twitter graph with 1.4 billion edges. This reduces the running time of identifying one group member from nearly an hour to less than 20 s on average.

1. Introduction

Node centrality [8,22,24] captures the structural importance of a node in a network (or says *a graph*), and is a fundamental concept in analyzing complex networks, e.g., rank Web pages in the WWW [39], measure the influence of a user in an online social network (OSN) [10], analyze the load of a server in a communication network [47], and so forth. Many centrality measures such as degree [2], closeness [14,37,45], betweenness [43,50], and their variants [42,47], have been proposed to measure the importance of *an individual node* in a network. While these measures apply usefully in finding the *K* most important (top-*K*) nodes in a network, they are actually *not* suitable for finding *a set of nodes* of size *K* such that these *K* nodes form the most important *node group* in the network.

* Corresponding author.
E-mail address: phwang@mail.xjtu.edu.cn (P. Wang).

http://dx.doi.org/10.1016/j.ins.2017.03.017
0020-0255/

Indeed, such a problem widely exists in scores of applications. For example, in an OSN, product retailers may want to locate K people to promote their products so as to maximize the number of potentially influenced customers [25]; however, due to the *overlap* of people's friend circles, simply returning the top-K *most influential* people (say, measured by degree) in the network is unlikely to be optimal. As another example, considering the fast development of electric vehicles in recent years, it urges urban planers to build charging stations at the proper places of a city to facilitate drivers to find a nearest one [31]. If we consider this problem as finding K nodes in the urban traffic network to minimize the average distance that other nodes can reach one of these K nodes, then choosing the top-K *most central* nodes (say, measured by closeness) in the network is obviously not satisfactory. Many data summarization tasks in machine learning (e.g., extractive summarization of documents, image collections, videos etc. [48]) can also be abstracted as identifying the most important node group from some properly defined network (e.g., a network formed by words and their correlations [1,3]).

These examples motivate researchers to develop new metrics that are suitable for *measuring the importance of a group of nodes* rather than an individual node in a network. To this end, Everett and Borgatti, in their seminal work [18], extended the idea of individual node centrality to *group centrality*. They conceptually illustrated the definitions of *group degree, group closeness* and so on (which we will elaborate in Section 2) for two graphs containing 14 and 20 nodes respectively. Despite its conceptual novelty and potential usefulness in addressing practical problems, group centrality *lacks efficient calculation methods* that can scale to large graphs which contain billions of edges such as Twitter and Facebook. Such large graphs call for efficient group centrality calculation methods and tools.

Designing scalable algorithms for handling large real-world graphs is challenging. As graphs grow larger in scale and more complex in their structure, they easily outgrow the memory capacity of a single computer and complicate the system design of distributed computing [33]. In some situations, graph compression techniques can be applied to fit a graph into a computer's main memory. For example, Boldi and Vigna [7] reported that Web graphs can be compressed by using about 3 bits per edge; however, social networks are far less compressible than Web graphs [12]. Furthermore, decompression may lead to overhead in CPU time and harm computational efficiency. On the other hand, distributed graph computing systems such as Spark and GraphLab [32] can handle billion-scale graphs; however, the cost of having and maintaining a large cluster is prohibitory for most users. In addition, it is also a difficult problem to break a large graph into smaller balanced subgraphs in order to process them across cluster nodes and minimize the communication between cluster nodes [30]. In recent years, disk-based graph computation has undergone an explosive development, such as GraphChi [28], X-Stream [44], TurboGraph [23] and VENUS [11]. Such systems use clever graph storage methods and do not need to load a complete graph into main memory, and can process large graphs on just a single PC with computational efficiency as competitive as distributed computing systems. However, these existing systems do not yet support calculating group centrality.

Present work. In this work, we study how to efficiently calculate group centrality (more specifically, the *H-group closeness centrality*) over large disk-resident graphs on a single PC equipped with moderate sized main memory, e.g., $4 \sim 8$GB. Our main result is a disk-based group centrality computation system/tool that can handle billion-scale graphs within hours of time. For example, finding a node group to (approximately) maximize the group closeness centrality on a Twitter graph of 1.4 billion edges, our method needs less than 2 h, while a baseline method that relies on sequentially scanning the data on the disk multiply times needs several days.

We first introduce the group degree/closeness centrality defined by Everett and Borgatti in Section 2, and then show that they can be extended to a *H-group closeness centrality*, which captures the features of both group degree and group closeness. We thus focus on calculating the H-group closeness in a large disk-resident graph. Here, the calculation of H-group closeness includes solving two basic problems: (1) given a node group, how to efficiently compute its H-group closeness value, i.e., the *computation problem*; and (2) how to find a node group containing at most K nodes to maximize the H-group closeness, i.e., the *maximization problem*. These two problems arise from two typical real-world applications. For example, given two user groups of an OSN, which one is more influential and hence more suitable for advertising targeting? Given two charging station placement plans, which one should the urban planer choose? These questions can be answered by solving the computation problem. The solution to the maximization problem can be applied to find optimal solutions, e.g., find a user group with the maximum influence for advertising targeting; find out the best charging station placement strategy for a city. Throughout the paper, the *proposition of H-group closeness centrality* and the *solutions to these two problems* constitute **our main contributions**.

Challenges. For large graphs that cannot entirely fit in a computer's main memory, both the computation problem and the maximization problem become challenging.

- In the computation problem, we need to solve the single-source shortest path (SSSP) problem for a group of nodes, which is known to be a high computational complexity task in a large graph (either use a breadth-first-search (BFS) method or Dijkstra's algorithm).
- In the maximization problem, we need to solve the all-pairwise shortest path (APSP) problem, which has even larger time complexity than solving the SSSP problem. Even worse, the maximization problem is NP-hard. Although a greedy algorithm can find an approximate solution, it generates too many random accesses to disk when handling disk-resident graphs which results in poor computational efficiency.

Solutions. We develop efficient and novel algorithms to solve these problems in this work.

- We propose an efficient method based on a probabilistic counting method to estimate H-group closeness centrality with high accuracy, rather than exhaustively computing it in an exact fashion. The proposed method has time complexity $O(m)$ using $O(n\log\log n)$ extra storage, where n and m are the number of nodes and number of edges respectively.
- We design a novel I/O-efficient greedy algorithm, that exploits properties of the H-group closeness centrality to handle a disk-resident graph efficiently, and improves the computational efficiency a lot. For example, our method is about 300 times faster than a baseline method on the Twitter graph with 1.4 billion edges.

We validate the efficiency of our techniques by conducting extensive experiments in both synthetic and real-world graphs, which cover a large variety of networks, including citation networks, collaboration networks, social networks, etc. The empirical results demonstrate that our solution performs well, both in computational accuracy and efficiency.

The remainder of the paper will proceed as follows. In Section 2, we give the definitions of group degree, group closeness, and H-group closeness. In Section 3, we formulate the group closeness computation/maximization problem. We then elaborate the methodologies for solving these two problems in Section 4, and validate our proposed methods in Section 5. We also show some interesting observations in Section 6. Finally, Section 7 summarizes some related work, and Section 8 concludes. Proofs of main results can be found in Appendix.

2. Group centrality definitions

In this section, we first review two definitions of group centrality: *group degree centrality* and *group closeness centrality*. Then we extend them to a *H-group closeness centrality*.

2.1. Group degree and group closeness

Let $G(V, E)$ be a directed graph, where $V = \{0, \ldots, n-1\}$ is a set of n nodes, and $E \subseteq V \times V$ is a set of edges. We only consider unweighted graphs in this work. Given a subset of nodes $S \subseteq V$, referred to as a *node group*, Everett and Borgatti [18] defined its *group degree centrality* as the number of non-group members that are connected to group members in the graph.

Definition 1 (Group degree centrality)**.** In a directed graph $G(V, E)$, the group degree centrality of a node group $S \subseteq V$ is

$$C_{\text{deg}}(S) \triangleq |\{v : u \in S \wedge v \notin S \wedge (u, v) \in E\}|. \tag{1}$$

Intuitively, if we think of a directed edge $(u, v) \in E$ as user u influencing user v, group degree centrality $C_{\text{deg}}(S)$ is in fact the number of users that are directly influenced by S. Note that group degree only considers *one-hop* neighbors of each group member, and can be extended to *group closeness centrality*, which considers the distances to all other nodes in the graph. If a node group can reach to other nodes in the graph with smaller average distance, group closeness gives higher score to this node group. The original definition of group closeness given by Everett and Borgatti can be formally described as follows.

Definition 2 (Group closeness centrality)**.** In directed graph $G(V, E)$, the group closeness centrality of a node group $S \subseteq V$ is the reverse of the average distance from S to other nodes, i.e.,

$$C_{\text{close}}(S) \triangleq \frac{|V \backslash S|}{\sum_{v \in V \backslash S} d_{S,v}}. \tag{2}$$

Here, $d_{S,v}$ measures the distance from node group S to a node v in the graph, and is defined as

$$d_{S,v} \triangleq \begin{cases} \infty, & S = \emptyset, \\ \min_{u \in S} dist_{uv}, & S \neq \emptyset, \end{cases} \tag{3}$$

where $dist_{uv}$ is the shortest path distance from u to v, $dist_{uv} = \infty$ if u cannot reach v and $dist_{uu} = 0$.

Here, we should notice that Everett and Borgatti implicitly assume that graph G is *strongly connected*. Otherwise, the above group closeness centrality is not well defined: If there exists a node $v \in V \backslash S$ that is not reachable from S, i.e., $d_{S,v} = \infty$, then $C_{\text{close}}(S)$ is always zero.

2.2. H-group closeness centrality

Group degree centrality in Definition 1 and group closeness centrality in Definition 2 can both be considered as distance scores measuring how close a node group is to other nodes in the graph. Group degree considers only one-hop neighbors, while group closeness considers the average distance to all other nodes. We combine these two measures and introduce a H-group closeness centrality.

Definition 3 (*H*-group closeness centrality)**.** In a directed graph $G(V, E)$, let $H \geq 1$ be an integer, g_H: $[0, \infty) \mapsto [0, \infty)$ be a non-increasing function truncated at H, i.e., $g_H(x) = 0$ for $x > H$. The *H*-group closeness of node group $S \subseteq V$ is

$$C_H(S) \triangleq \sum_{v \in V} g_H(d_{S,v}) - f(S), \tag{4}$$

where $d_{S,v}$ is defined by (3), $f(S)$ is an additive cost function such that $f(A) + f(B) = f(A \cup B)$ for every disjoint sets $A, B \subset V$.

Remarks. (1) *H*-group closeness weights heavier to nodes that are more easily reachable from S than nodes are not, and this coincides with our intuition that in OSNs, the influence of a group of users decays as distance increases.

(2) To guarantee that *H*-group closeness is non-negative, we further require cost function f satisfying $f(\emptyset) = 0$ and $f(V) \leq g_H(0)|V|$. In experiments, we will use $f(S) = \alpha g_H(0)|S|$, where $\alpha \in [0, 1]$ is an additional parameter.

(3) *H*-group closeness captures features of both group degree and group closeness by choosing proper g_H and f. For example, let $g_1(h) \equiv 1$ and $f(S) = |S|$, then $C_1(S) = C_{\text{deg}}(S)$; let H be the diameter of the graph, then $C_H(S)$ captures how close S is to all other nodes in graph G, and similar to $C_{\text{close}}(S)$.

(4) *H*-group closeness is better defined than $C_{\text{close}}(S)$, and does not require graph G to be well-connected. In the following discussion, we will also refer *H*-group closeness as group closeness for simplicity.

3. *H*-group closeness computation and maximization

In this section, we formulate the *H*-group closeness computation/maximization problem, and analyze the hardness of solving the maximization problem.

3.1. Two basic problems

The proposed *H*-group closeness centrality is closely related to two basic problems, i.e., the *computation problem* and the *maximization problem*, which are formally stated in the following.

Definition 4 (*H*-group closeness computation problem)**.** Given graph $G(V, E)$, and node group $S \subseteq V$, calculate the value of *H*-group closeness $C_H(S)$.

Definition 5 (*H*-group closeness maximization problem)**.** Given graph $G(V, E)$, find a node group $S \subseteq V$ containing at most K nodes to maximize $C_H(S)$.

As explained in Section 1, the above two problems are from real-world applications. For example, comparing the influence of two user groups in an OSN needs to solve the computation problem, while finding an optimal charging station placement needs to solve the maximization problem.

We assign a cardinality constraint $|S| \leq K$ for the maximization problem because many real-world problems have *resource limitations*, e.g., a company can afford to choose at most K users for advertising targeting; a city has a financial budget for building at most K charging stations.

3.2. The maximization problem and submodularity

The maximization problem deserves an in-depth analysis before we present our solution. First, we can show that the maximization problem is computationally intractable to solve optimally (by a straightforward reduction from the max K-vertex cover problem which is NP-hard).

Theorem 1. *H-group closeness maximization problem is NP-hard.*

Since it is computationally difficult to find an optimal solution, we aim to find an approximate solution with solution quality guaranteed. Before we introduce such a method, we first recap a well-studied *submodularity theory* [26].

Background on submodularity. A set function $F : 2^V \mapsto \mathbb{R}$ is *submodular* if for every $S \subseteq T \subseteq V$ and $s \in V \backslash T$, it holds that $F(S \cup \{s\}) - F(S) \geq F(T \cup \{s\}) - F(T)$. To understand its meaning, let us denote by $\delta_s(S) \triangleq F(S \cup \{s\}) - F(S)$ the *marginal gain* of adding element s to set S. Then submodular functions actually capture that the marginal gain never increases as more nodes are added to S, a.k.a. the phenomenon of *diminishing returns*, which arises naturally in many areas such as economics and machine learning. Furthermore, a set function $F : 2^V \mapsto \mathbb{R}$ is *monotone* if for every $S \subseteq T \subseteq V$, $F(S) \leq F(T)$. The classical result of [36] states that a polynomial time *greedy algorithm* can provide an approximation of $1 - e^{-1}$ for maximizing a monotone submodular function with a cardinality constraint, and the bound is tight [36]. The greedy algorithm starts with an empty set $S_0 = \emptyset$, and iteratively, in step k, adds an element s_k which maximizes the marginal gain, i.e., $s_k = \arg\max_{s \in V \backslash S_{k-1}} \delta_s(S_{k-1})$. The algorithm stops once it has selected K elements (where K is the budget), or the marginal gain becomes zero.

However, if the submodular function is non-monotone, the simple greedy algorithm can perform arbitrarily poorly (an example is given in [41]) because the simple greedy algorithm may be trapped in regions where only have local maximums.

Algorithm 1: Random greedy [9].

$S_0 \leftarrow \emptyset$;
for $k = 1, \ldots, K$ **do**
  $T_k \leftarrow \arg\max_{S \subseteq V \setminus S_{k-1} \wedge |S| = K} \sum_{s \in S} \delta_s(S_{k-1})$;
  $s_k \leftarrow$ a uniformly random sample from T_k;
  $S_k \leftarrow S_{k-1} \cup \{s_k\}$;
Return S_K;

To help "jumping out of" such traps, Buchbinder et al. [9] proposed a *random greedy algorithm* (Algorithm 1): at each step, instead of selecting an element with the largest marginal gain, it randomly selects an element from a set T_k, which contains the elements with the K largest gains. Random greedy provides an approximation of e^{-1} in expectation for maximizing a general non-monotone submodular function with a cardinality constraint.

We find that H-group closeness indeed satisfies submodularity, but the monotonicity depends on f.

Theorem 2. *H-group closeness is a submodular set function, which is monotone when* $\max_{v \in V} f(\{v\}) \leq g_H(0) - g_H(1)$ *and non-monotone otherwise.*

Hence we can use the simple greedy and random greedy algorithms to solve the H-group closeness maximization problem. Although the two greedy algorithms both have $O(Kn)$ oracle calls (i.e., number of times of calculating marginal gains), they are still too expensive when handling large graphs due to the additional cost in calculating marginal gains in each step. Furthermore, the two greedy algorithms require the data to fit entirely in a computer's main memory, and they are actually not efficient for handling graphs residing on disk. We will explain the difficulties of directly applying these two memory-based algorithms in detail in the next section, and develop new techniques that enable us to use a single PC with small memory capacity to efficiently find quality guaranteed solutions in a large graph.

4. Handling disk-resident graphs

Before we elaborate our methods for solving the H-group closeness computation problem and maximization problem, it is necessary to explain why the two problems become challenging when the graph cannot fit entirely in a computer's main memory.

4.1. Challenges of handling disk-resident graphs

If we want to *exactly* calculate the group closeness of a node group, we have to solve the single-source shortest path (SSSP) problem for a group of nodes. Meanwhile, when solving the maximization problem, in order to calculate the marginal gain for *each node* in a step of the greedy algorithm, we are faced to solve the all-pairwise shortest path (APSP) problem. Solving the SSSP and APSP problems in a large graph is known to be a high complexity task and a bottleneck in many graph applications. For example, if we use the BFS method to solve the APSP problem in an unweighted and directed graph, the time complexity is $O(n^2 + nm)$. This is obviously expensive when handling a large graph with large number of nodes n and large number of edges m. Furthermore, when solving the SSSP and APSP problems, these algorithms require the computer's main memory is large enough to store the entire graph, and they are actually not suitable for handling graphs residing on disk.

When a graph cannot entirely fit in a computer's main memory, i.e., has to be stored on disk, another challenge is that *greedy algorithms generate many random disk accesses* and therefore harms computational efficiency significantly. To illustrate this issue of greedy algorithm clearly, let us consider that we already use an efficient implementation of the greedy algorithm where nodes are maintained in a priority queue ordered in decreasing marginal gain. Each time when a node is added to the selected node group S, we need to update the marginal gain of the affected nodes in the queue, and we can use an inverted index for quickly looking up these affected nodes. An inverted index can be thought of as a hash map that maps a node u to a list of nodes $L_u = \{v_1, v_2, \cdots\}$, and the inclusion of node u to set S will affect the marginal gain of node $v \in L_u$. In other words, when node u is selected, we need to look up L_u and update the marginal gain for $v \in L_u$ in the queue. If the index mapping (i.e., $\{u \mapsto L_u\}_{u \in V}$) is small, it can fit in a computer's main memory. Unfortunately, the inverted lists $\{L_u\}_{u \in V}$ are usually extremely large (even larger than the graph) and have to be stored on disk. Since u is unlikely to be selected locally during the iterations in the greedy algorithm, this will cause many random disk accesses, which harms the computational efficiency significantly.

4.2. H-group closeness estimation method

We solve the H-group closeness computation problem in this subsection. Since the exact algorithms that need to solve the SSSP and APSP problems do not scale, we turn to a different way that *estimates C_H efficiently*, and avoid solving the complex SSSP or APSP problem in a large graph.

The estimation method is based on an equivalent form of the definition of H-group closeness. Since $d_{S,v}$ takes integer values and we care values in range $[0, H]$, it is easy to see that H-group closeness defined by (4) is equivalent to

$$C_H(S) = \sum_{h=1}^{H} g_H(h)[N_h(S) - N_{h-1}(S)] + g_H(0)|S| - f(S),$$

where $N_h(S)$ is the number of nodes that are reachable from S within h hops, i.e.,

$$N_h(S) \triangleq \begin{cases} |S|, & h = 0, \\ |\{v : d_{S,v} \le h\}|, & h \ge 1. \end{cases}$$

The key is that we can efficiently estimate $N_h(S)$ based on a method by Palmer et al. [40], which used a clever Flajolet-Martin (FM) sketch method [21] to estimate an *individual neighborhood function* $N_h(u) \triangleq |\{v: dist_{uv} \le h\}| \equiv N_h(\{u\})$ for a node u. In that work, the estimation of $N_h(u)$ is thought of as *counting* the number of nodes within h hops from node u, which can be efficiently implemented using well-studied *probabilistic counting methods*[21], e.g., FM-sketch. Palmer et al.'s method is based on the following observation: Let $\mathrm{Nbr}_h(u)$ denote the set of nodes that are reachable from u within h hops, $\mathrm{Nbr}_h(u)$ then satisfies

$$\mathrm{Nbr}_h(u) = \mathrm{Nbr}_{h-1}(u) \cup \left[\bigcup_{u \to v} \mathrm{Nbr}_{h-1}(v)\right]. \tag{5}$$

In FM-sketch, we can use a bit-string $M_h(u)$ of length $\log_2 n$ bits (n is the number of nodes) to "encode" u's neighborhood information $\mathrm{Nbr}_h(u)$, and replace the set-union operation by bitwise-or operation. $M_h(u)$ is calculated by iteratively performing the bitwise-or operations among u's out-neighbors, i.e.,

$$M_h(u) = M_{h-1}(u)\ \mathrm{OR}\left[\underset{u \to v}{\mathrm{OR}} M_{h-1}(v)\right] \tag{6}$$

where $M_0(u)$, $u \in V$ is initialized by a uniformly distributed random number, and bit i of $M_0(u)$ is set if the random number has i tailing zeros. $N_h(u)$ can be "decoded" from $M_h(u)$ by $N_h(u) \approx 2^r/0.77351$, where r is the index of the leftmost '1' bit in bit-string $M_h(u)$ plus one (e.g., $r(00\underline{1}0) = 2, r(0\underline{1}10) = 3$).

Extending to a node group. We extend the above method to estimating $N_h(S)$ for node group S, based on the following observation: Let $\mathrm{Nbr}_h(S)$ denote the set of nodes that are reachable from node group S within h hops, then

$$\mathrm{Nbr}_h(S) = \bigcup_{u \in S} \mathrm{Nbr}_h(u).$$

That is, if we have obtained bit-string $M_h(u)$ for each node $u \in V$, then bit-string $M_h(S)$ for node group S can be obtained by performing the bitwise-or operations among all its group members, i.e.,

$$M_h(S) = \underset{u \in S}{\mathrm{OR}} M_h(u).$$

And finally, $N_h(S)$ can be decoded from $M_h(S)$ in a similar way as $N_h(u)$.

Enhancement. However, when we use the above method to estimate H-group closeness C_H in practice, we observe that FM-sketch sometimes incurs large estimation variance (details are discussed in Section 5). We therefore make several enhancement to improve the estimation accuracy of the above estimation framework. First, instead of using the FM-sketch method, we use a method that combines the HyperLogLog counting [20] and linear counting [49] (HLLC-LC). HyperLogLog counting is the state-of-the-art probabilistic counting method which is also adopted by Boldi et al. [6] to improve the accuracy of estimating an individual neighborhood function. The combination with a linear counting can provide more accurate estimates for small cardinalities. This is important in improving the final estimation accuracy of C_H. Because for small h, $N_h(S)$ is typically small, but they have large weights in the calculation of C_H; hence if we improve the estimation accuracy of $N_h(S)$ for small h, we can improve the estimation accuracy of C_H significantly. Second, we store N bit-strings per node per hop, and use the arithmetic mean of N estimates to further reduce the estimation error of C_H, since the variance of an arithmetic mean decreases with rate $1/N$.

Computing marginal gains. Notice that such an estimation framework also benefits the calculation of marginal gain $\delta_s(S)$ of adding s to set S. Because $M_h(S \cup \{s\}) = M_h(S)\,\mathrm{OR}\,M_h(s)$, $C_H(S \cup \{s\})$ is easily obtained after decoding and hence so is $\delta_s(S) = C_H(S \cup \{s\}) - C_H(S)$.

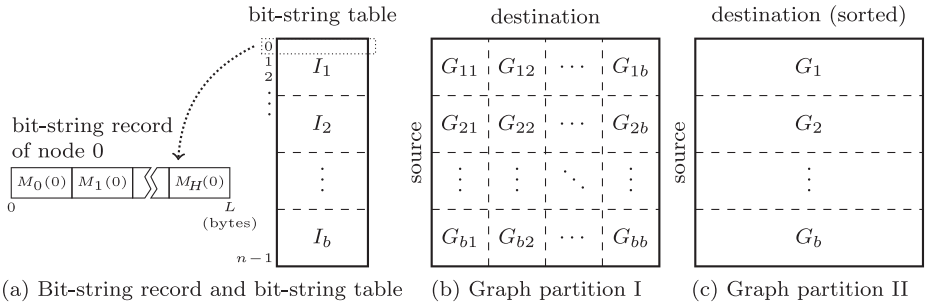


Fig. 1. Bit-string calculation method.

Time and storage complexity. If we use the HyperLogLog counting method, the bit-string $M_h(u)$ for a node u has length $O(\log\log n)$. Thus, storing bit-strings of all the nodes in a graph has storage approximately $O(n\log\log n)$ bytes. Computing the bit-strings for all the nodes in the graph requires traversing the edge list H times, thus the time complexity of computing bit-strings is $O(m)$. Once we have obtained the bit-strings for every node, the time complexity of calculating C_H is constant because the time complexity of decoding $M_h(u)$ is constant. Note that when the bit-strings are stored on disk, their will be additional I/O cost on reading and writing bit-strings from/to disk; this additional I/O cost will be analyzed in the next subsection.

In conclusion, we see that the estimation approach avoids solving the SSSP or APSP problem in a large graph and therefore reduces the computational complexity. The estimation method relies on bit-strings $M_h(u)$ for every node $u \in V$ at each hop $h = 1, \ldots, H$. In the following, we present a bit-string calculation method that can handle large disk-resident graphs.

4.3. Bit-string calculation method

In this subsection, we design an efficient bit-string calculation method, and we generally assume that the bit-strings are stored on disk.

Given a directed graph $G(V, E)$, we associate each node $u \in V$ with a collection of bit-strings, i.e., $\{M_h(u)\}_{h=0}^{H}$, called *a bit-string record*, where $M_0(u)$ is initialized by a uniformly distributed random number and $\{M_h(u)\}_{h=1}^{H}$ is calculated later. Bit-string records are all fixed length, and the length of a record is assumed to be L bytes. We arrange these n bit-string records in ascending order of their associated node IDs, and concatenate them to form the *bit-string table*, as illustrated in Fig. 1(a). Thus, for an arbitrary node $u \in V$, its bit-string record can be located at position $L \cdot u$ of the bit-string table in time $O(1)$. The bit-string table is further divided into b blocks (logically), denoted by I_i, $1 \le i \le b$, to ensure that each block can be entirely loaded into the computer's main memory.

To calculate $M_h(u)$ on a large disk-resident graph, a direct method is to divide graph G into b^2 blocks according to the division of nodes in the bit-string table, as shown in Fig. 1(b), and then handle one graph block at a time [40]. For example, graph block G_{ij} consists of source nodes in I_i and destination nodes in I_j; when handling graph block G_{ij}, bit-string records in I_i and I_j are loaded, and source bit-string records in I_i are updated by performing bitwise-or operations with their out-neighbors in I_j according to (6). This method is described in detail in Algorithm 2.

Algorithm 2: Bit-string calculation using partition I.

```
1  for h = 1, ..., H do
2      for i = 1, ..., b do
3          Load bit-strings in interval I_i into memory;
4          foreach node u in interval I_i do
5              M_h(u) ← M_{h-1}(u);
6          for j = 1, ..., b do
7              Load bit-strings in interval I_j into memory;
8              foreach edge (u, v) in G_ij do
9                  M_h(u) ← M_h(u) OR M_{h-1}(v)
10         Flush bit-strings in I_i to disk;
```

Calculating bit-strings for all the nodes at each hop, Algorithm 2 needs to read $nL + bnL = (b+1)nL$ bytes from disk (Lines 3 and 7), and write nL bytes to disk (Line 10). One possible issue in Algorithm 2 is that, real graphs are usually very sparse, meaning that many graph blocks in Fig. 1(b) may contain only a few edges, and therefore Algorithm 2 will generate

many wasted disk reads. A way to solve this issue is to leverage graph clustering methods to reorder the nodes and then remove those empty graph blocks. However, the cost of this preprocessing step is expensive.

To alleviate the wasted disk reads problem, we design another bit-string calculation method based on a different graph partition, as illustrated in Fig. 1(c). In Fig. 1(c), we partition graph G only by its source nodes, and edges in the same graph block are sorted by destination. For example, graph block G_i consists of edges with the source nodes in I_i. When a graph block is handled, say G_i, bit-strings in I_i are loaded into memory, and bit-strings of edges' destinations are loaded *on demand*. Because destination nodes are sorted, we only need to sequentially scan the bit-string table on disk in a *streaming fashion*, and generate very few random accesses. This method is described in detail in Algorithm 3.

Algorithm 3: Bit-string calculation using partition II.

```
1  for h = 1, ..., H do
2      for i = 1, ..., b do
3          Load bit-strings in interval I_i into memory;
4          foreach node u in I_i do
5              M_h(u) ← M_{h-1}(u);
6          foreach edge (u, v) in G_i do
7              if v is not cached then
8                  Pin node v and its bit-strings into cache;
9              M_h(u) ← M_h(u) OR M_{h-1}(v);
10         Flush bit-strings in I_i to disk;
```

Assume that each graph block contains an average fraction of $a \in [0, 1]$ distinct destination nodes of the n total nodes. Then Algorithm 3 needs to read $nL + banL = (ab + 1)nL$ bytes from disk (Lines 3 and 8), and write nL bytes to disk (Line 10) to calculate bit-strings for all the nodes at each hop. Hence, Algorithm 3 requires fewer disk reads than Algorithm 2. Notice that Algorithm 2 needs to load two bit-string blocks into memory at a time, while Algorithm 3 only needs to load one. Hence, Algorithm 3 can use larger bit-string blocks than Algorithm 2, which also reduces the number of disk reads as b becomes smaller. The I/O cost comparison between Algorithms 2 and 3 is summarized in Table 1.

4.4. An i/O-efficient greedy algorithm

In this subsection, we solve the H-group closeness maximization problem using the greedy algorithms mentioned in Section 3.2. The main difficulty of directly applying these algorithms is that they generate many random disk accesses and are not suitable for handling disk-resident graphs as explained in Section 4.1. We hence design a novel I/O-efficient greedy algorithm by exploiting the submodularity of C_H. We first introduce these properties of submodularity.

4.4.1. Properties of submodularity

The first property of submodularity comes from its definition, which we state again as follows.

Property 1. *Let F be a submodular function, then for every $S \subseteq T \subset V$ and $s \in V \backslash T$, it holds that $\delta_s(S) \geq \delta_s(T)$.*

This property implies that submodular functions exhibit the diminishing return property, i.e., the marginal gain of every element never increases as more elements are selected. This property can reduce the number of times of updating marginal gains in each step of the simple/random greedy algorithm. In particular, if the recently updated node already has the largest marginal gain, there is no need to update marginal gains for the other nodes. Because their gains will only become smaller after being updated. This useful trick is also called *lazy evaluation* [35].

Next, we state that submodularity allows us to slightly weaken the conditions required by the simple/random greedy algorithm when selecting elements at each step, and we can still obtain quality guaranteed solutions.

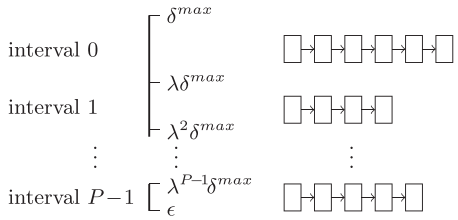
Property 2. *Let F be a monotone submodular function. In each step k of the simple greedy algorithm, instead of choosing an element s_k^* with the largest marginal gain, we choose an element s_k whose marginal gain is at least a fraction λ of the largest*

Table 1
Disk I/O cost comparison between Algorithms 2 and 3.

	Bytes of disk reads	Bytes of disk writes	Number of blocks b
Algorithm 2	$(b+1)nL$	nL	$\approx \lceil 2nL/\text{mem.} \rceil$
Algorithm 3	$(ab+1)nL, a \in [0, 1]$	nL	$\approx \lceil nL/\text{mem.} \rceil$

v	δ	t	bit-strs
23	93.1	2	011...00
8	87.3	1	001...11
13	89.3	2	101...10
⋮	⋮	⋮	⋮
39	51.0	3	110...01

(a) block structure illustration



(b) interval partition in the algorithm

Fig. 2. Block structure and interval partition in the I/O-efficient greedy algorithm.

marginal gain, i.e., $\delta_{s_k} \geq \lambda\delta_{s_k^*}$. *Then* $S_K = \{s_1, \ldots, s_K\}$ *satisfies*

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

where OPT is the optimal solution maximizing F.

Property 3. *Let F be a submodular function. In each step k of the random greedy algorithm, instead of constructing* T_k *with K elements having the largest gains, we construct* T_k' *with K elements such that* $\sum_{s\in T_k'} \delta_s(S_{k-1}) \geq \lambda \sum_{s\in T_k} \delta_s(S_{k-1})$. *We randomly sample an element* s_k *from* T_k' *at step k, and obtain* $S_K = \{s_1, \ldots, s_K\}$ *after K steps. Then* S_K *satisfies*

$$\mathbb{E}F(S_K) \geq \lambda e^{-1}F(OPT)$$

where OPT is the optimal solution maximizing F.

Properties 2 and 3 use λ as a parameter to trade off solution quality against computational efficiency. Instead of exhaustively searching for a node (or K nodes) with the largest gain at each step, we now search for a node (or K nodes) for which the gain is at least a fraction λ of the largest gain, and the final solution quality is still bounded. More importantly, these two properties enable us to organize nodes in a special manner, which benefits handling a disk-resident graph. We elaborate this technique in the following.

4.4.2. Designing an i/O-efficient greedy algorithm

We first describe the data structure used in the I/O-efficient greedy algorithm.

Data structure. In our algorithm, nodes and their associated data are organized in blocks, and a block can be entirely loaded into main memory. A block, as illustrated in Fig. 2(a), maintains the following fields for each node:

- $v \in \{0, \ldots, n-1\}$ denotes the node ID;
- δ denotes the marginal gain of a node;
- $t \in \{0, 1, \ldots\}$ identifies the iteration, in which iteration marginal gain δ is updated;
- "bit-strs" represents the bit-string record associated with the node.

In practice, a block is stored as a single file, and blocks are further organized into *intervals* to optimize disk reads and writes during the greedy steps. More specifically, let $\delta^{max} \triangleq \max_{v\in V}\delta_v(\emptyset)$ denote the largest marginal gain when no node is selected (δ^{max} is obtained by traversing the bit-string table once). Given constant $\lambda \in (0, 1)$, we split the interval $(\epsilon, \delta^{max}]$ into P sub-intervals, where the i-th sub-interval is $(\lambda^{i+1}\delta^{max}, \lambda^i\delta^{max}]$, for $i = 0, \ldots, P-1$. The number of intervals P is chosen properly to ensure that $\epsilon \triangleq \lambda^P\delta^{max}$ is sufficiently small. Given these P sub-intervals, we associate a block k to a sub-interval i such that, for each node v in the block, $\delta_v \in (\lambda^{i+1}\delta^{max}, \lambda^i\delta^{max}]$. All the blocks in the same interval are further organized as a queue, as is illustrated in Fig. 2(b). Note that, for each queue, we only maintain Block IDs belonging to the corresponding interval in the queue in main memory.

I/O-efficient greedy algorithm. Based on this data structure, we design an I/O-efficient greedy algorithm in Algorithm 4 . In general, Algorithm 4 runs for K iterations, and in each iteration, it selects a node (or K nodes) whose marginal gain satisfies the conditions of Property 2 (or 3), and finally outputs a solution that is a $(1 - e^{-\lambda})$-approximation (or λe^{-1}-approximation) of the optimal value for maximizing C_H.

In each iteration, we process blocks in interval i where i is the smallest, so this interval is most likely to contain nodes with the largest marginal gains (Lines 4–6). We update the marginal gain of nodes in the loaded blocks (Lines 8–10). At the same time, we also update the interval ID that the updated node belongs to, i.e., find j such that $\delta_v \in (\lambda^{j+1}\delta^{max}, \lambda^j\delta^{max}]$ (Line 11). If the updated node still belongs to the same interval, i.e., $j = i$, we are sure that this node has marginal gain at least a fraction λ of the current largest marginal gain. Hence, we select this node (Lines 12–20). Otherwise ($j > i$), we save the node and its associate data into a block belonging to interval j (Line 22). Note that we may need to create a new block if the tail block in interval j exceeds the capacity limit. Then we create a new block and push its block ID into the interval's queue.

Algorithm 4: I/O-efficient greedy algorithm.

```
S ← ∅, T ← ∅, t ← 0;
while |S| < K, do
    if Queue[i] is empty for all i, then break;
    i ← min{i′ : Queue[i′] is non-empty};
    BlockID ← Queue[i].Pop();
    Load the block specified by BlockID into memory;
    foreach node v in the block do
        if t_v < t, then
            δ_v ← C_H(S ∪ {v}) − C_H(S), t_v ← t;
            if δ_v < ϵ then continue;
        j ← the interval ID that node v belongs to;
        if j = i, then
            if C_H is monotone then
                S ← S ∪ {v}, t ← t + 1;
            else
                T ← T ∪ {v};
                if |T| = K then
                    s ← sample a node from T;
                    Save nodes in T\{s} to interval j;
                    S ← S ∪ {s}, T ← ∅, t ← t + 1;
        else
            Save node v to interval j;
return S;
```

Remarks. (1) In the case that C_H is non-monotone, Algorithm 4 may lead to a situation where there are not enough nodes to construct set T. This technique problem can be solved by adding enough dummy nodes to T, and a dummy node always has zero marginal gain. Obviously, these dummy nodes do not affect the solution quality and can be removed from the final solution S.

(2) The I/O-efficient greedy algorithm is superior to the simple/random greedy algorithm in following aspects: (a) it only sequentially scans data from interval to interval, hence generates few random access; (b) it does not exhaustively search for a node with the largest gain, but can stop search earlier once it finds an updated node still belonging to the same interval. Hence, I/O-efficient greedy algorithm is more efficient than the simple/random greedy algorithm in handling disk-resident graphs.

5. Experimental validation

In this section, we conduct experiments on a large variety of graphs (including citation networks, collaboration networks, social networks, etc.) with different scales (from thousands of nodes/edges to billions of nodes/edges) to validate our proposed techniques. In particular, we want to demonstrate that our proposed group closeness estimation method can well approximate the real group closeness; and our proposed I/O-efficient greedy algorithms outperform the original greedy algorithms on the trade-off between computational efficiency and accuracy when handling large graphs.

5.1. Validation datasets

We first briefly introduce the graphs that will be used in the following experiments and applications. The basic statistics of these graphs are summarized in Table 2. In the last column of Table 2, we also show the required memory capacity to load the complete graph into RAM using the graph structure in SNAP (snap.standord.edu). In general, these graphs can be classified into the following five categories.

- *Citation networks:* The citation networks are built from paper (cit-HepTh) or patent (cit-Patents) citations. In a citation network, a node represents a paper (or patent), and a directed edge (i, j) represents that paper/patent i cites paper/patent j. These datasets are publicly available from SNAP.
- *Collaboration networks:* Collaboration network ca-DBLP is built from the DBLP computer science bibliography (dblp.uni-trier.de/db). In this network, a node represents an author, and an undirected edge represents the co-authorship between two authors. This dataset is publicly available from SNAP.

Table 2
Graph statistics.

Graph	Type	# nodes	# edges	Mem.
cit-HepTh	directed	27K	352K	8MB
com-Synthetic	directed	100K	906K	24MB
ca-DBLP	undirected	317K	1.0M	80MB
com-Orkut	undirected	3.1M	117M	1.7GB
cit-Patents	directed	3.8M	16.5M	0.6GB
soc-LiveJournal	directed	5.2M	77.4M	1.8GB
ba-Synthetic-10m	directed	10M	105M	2.6GB
ba-Synthetic-20m	directed	20M	213M	5.2GB
ba-Synthetic-30m	directed	30M	322M	> 8GB
soc-Twitter	directed	41.7M	1.4B	> 8GB
soc-Friendster	undirected	65.6M	1.8B	> 8GB

- *Social networks:* soc-LiveJournal, soc-Twitter, and soc-Friendster are three social networks that are collected from LiveJournal, Twitter, and Friendster, respectively. In these networks, a node represents a user, and an edge represents the relationship between two users. A relationship can be undirected, e.g., the friendship in Friendster, or directed, e.g., the following relationship in LiveJournal and Twitter. These datasets are publicly available from SNAP.
- *Barabási-Albert (BA) graphs:* We use the Barabási-Albert model to generate three BA graphs with 10M to 30M nodes. These graphs have a power-law degree distribution.
- *Network with community structures:* We use two networks with ground-truth communities (com-Orkut and com-Synthetic) to study the characteristics of node groups identified by our method in Section 6. Network com-Orkut is built from the Orkut social network. In Orkut, beside the friend relationship between users, we also know the communities that each user belongs to. This dataset is publicly available from SNAP. The synthetic network com-Synthetic is a benchmark graph used to test community detection algorithms, and we generate it using the method described in [29].

5.2. Experiment settings

All the experiments are performed on a PC running 64-bit Ubuntu 14.04.3 LTS, with an octa-core 2.66 GHz Intel Processor, 8GB of main memory, and a 450GB 15000RPM hard disk.

When calculating the H-group closeness, we will use both the exponential weighting function $g_H(h) = e^{-h}$ and power weighting function $g_H(h) = 1/h$ (where $g_H(0) \triangleq 1$). The cost function is $f(S) = \alpha|S|, \alpha \in [0, 1]$.

5.3. Validating the estimation method

The H-group closeness estimation framework developed in Section 4.2 can be equipped with different probabilistic counting methods. Here we list some of them, and their associated parameters:

- *Flajolet-Martin (FM) sketch [21]:* It is a basic probabilistic counting method that we have described in Section 4.2. The main parameter is the number of bit-strings stored for each node at each hop, denoted by N.
- *LogLog counting (LLC) [16]:* LLC divides the counting space into m buckets, and uses arithmetic mean of the '1'-bit positions in these m buckets to estimate cardinality. Parameters are bucket number m and bit-string number N.
- *HyperLogLog counting (HLLC) [20]:* HLLC is an improved version of LLC, and uses harmonic mean for estimation. Parameters are bucket number m and bit-string number N.
- *LogLog counting combined with linear counting (LLC-LC):* We use linear counting instead of LLC when the empty buckets ratio is too high, which happens when cardinality is small.
- *HyperLogLog counting combined with linear counting (HLLC-LC):* We use linear counting instead of HLLC when empty buckets ratio is too high.

5.3.1. Estimation variance of using different probabilistic counting methods

In the first experiment, we compare the H-group closeness estimation variance of using different probabilistic counting methods in terms of the normalized root mean square error (NRMSE). NRMSE of an estimator $\hat{C}_H$ is defined as $\text{NRMSE}[\hat{C}_H] = \sqrt{\mathbb{E}[\hat{C}_H - C_H]^2}/C_H$. In our experiment, we generate node group of different size, e.g., $|S| = 10, 30,$ by randomly sampling nodes from a graph. Figs. 3 and 4 show the NRMSE of estimating C_5 for 100 different node groups in cit-HepTh and ca-DBLP, respectively.

In general, FM-sketch has the largest variance in both datasets. LLC and HLLC are much better than FM-sketch. The state-of-the-art probabilistic counting method HLLC is better than LLC, and if we use HLLC-LC to improve the estimation accuracy of small cardinalities, the NRMSE is further reduced. The experiment indicates that HLLC-LC should be our best choice in estimating C_H.

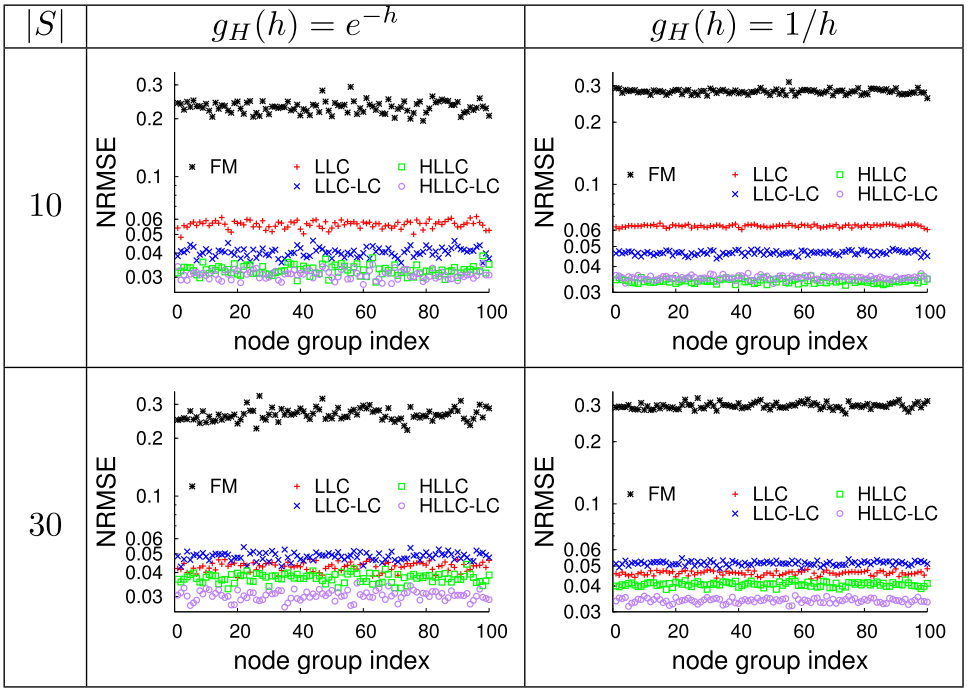


Fig. 3. NRMSE comparison in cit-HepTh, $N = 8, m = 64, H = 5, \alpha = 0$.

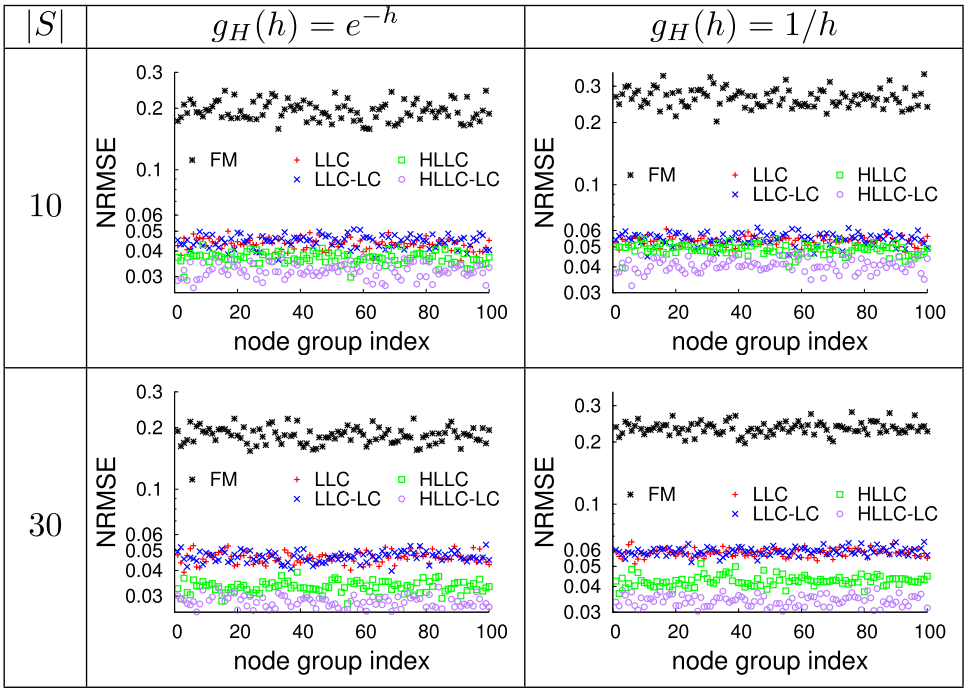


Fig. 4. NRMSE comparison in ca-DBLP, $N = 8, m = 64, H = 5, \alpha = 0$.

5.3.2. Estimation accuracy of using different parameters

Next, we use HLLC-LC to estimate C_5 and compare the estimates with the ground truth. Here the ground truth is obtained by using a breadth-first-search method in a graph. We generate 100 node groups of size $|S| = 10, 30, 50$ respectively, and show the estimated value of H-group closeness with respect to the real value in scatter plots in Fig. 5 (and the results for ca-DBLP is shown in Table 3). Both the estimated value and real value are normalized to range [0, 1] for ease of comparison.

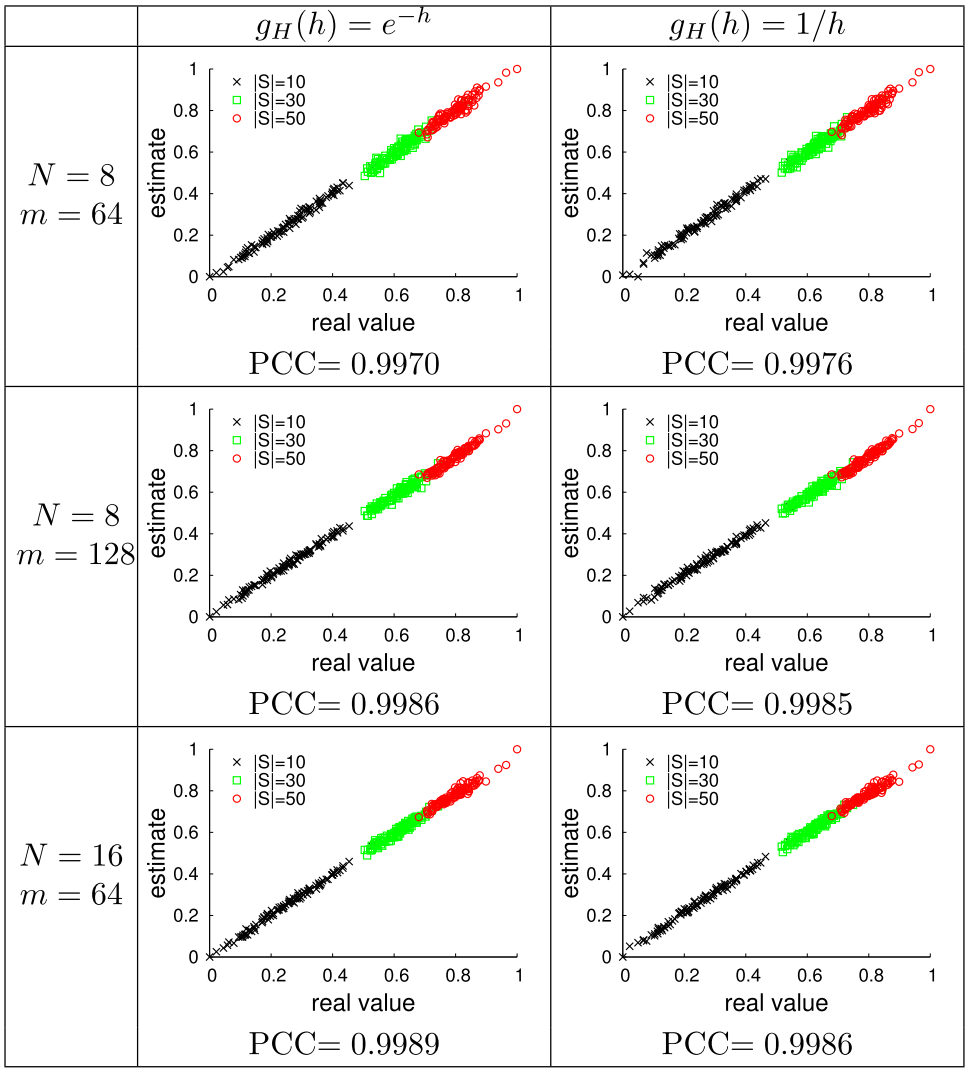


Fig. 5. Estimation accuracy of C_5 using HLLC-LC in cit-HepTh ($\alpha = 0$).

Table 3
Estimation accuracy (measured by PCC) of C_5 using HLLC-LC in ca-DBLP ($\alpha = 0$).

N	M	$g_H(h) = e^{-h}$	$g_H(h) = 1/h$
8	64	0.9983	0.9961
8	128	0.9993	0.9981
16	64	0.9993	0.9980

We can observe that the estimated values well approximate the real values for node groups of different size. The Pearson Correlation Coefficient (PCC) (which can quantify how strongly two series are linearly correlated) between the estimated values and real values is approximately 1.0. When we increase the number of bit-strings from $N = 8$ to 16, or increase the number of buckets from $m = 64$ to 128, PCC increases accordingly. This experiment indicates that the proposed estimation method can well approximate the true H-group closeness.

5.4. Validating the bit-string calculation method

The bit-strings $\{M_h(u)\}$ play an important role in solving both the computation problem and the maximization problem. In the following experiment, we verify the efficiency of our proposed bit-string calculation method.

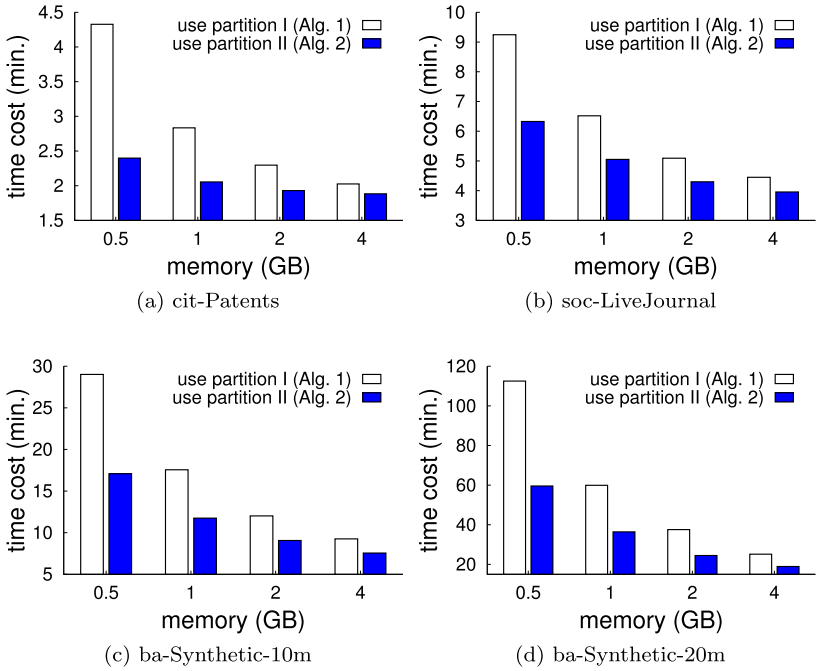


Fig. 6. Comparison of bit-string calculation methods, $N = 8, m = 64$.

Table 4
Time cost of calculating bit-strings on Synthetic-30m, Twitter and Friendster using Algorithm 3 with 7GB memory budget, $N = 8, m = 64$.

Graph	Time cost
ba-Synthetic-30m	24 min
soc-Twitter	37 min
soc-Friendster	1 h 2 min

We have two bit-string calculation methods, i.e., Algorithms 2 and 3, which are based on two different graph partitions as illustrated in Fig. 1. We show that Algorithm 3 based on graph partition II performs more efficiently than Algorithm 2 based on graph partition I. Here, we measure the average time cost of calculating bit-strings for all the nodes at one hop in different graphs, with respect to different memory budget, ranging from 0.5GB to 4GB. The results are shown in Fig. 6.

We observe that Algorithm 3 requires less time cost than Algorithm 2 on four graphs, indicating that Algorithm 3 is indeed more efficient than Algorithm 2. The reason is that Algorithm 3 needs fewer disk reads and can use larger bit-string blocks than Algorithm 2. The time cost on the other graphs is shown in Table 4.

5.5. Validating the i/O-efficient greedy algorithm

Next, we validate the effectiveness of our proposed I/O-efficient greedy algorithm in solving the H-group closeness maximization problem. We compare our method with two baseline methods:

- *Multi-pass greedy algorithm:* A simple way to enable the simple/random greedy algorithm to handle large data is to let them scan the data on disk multiply times. In each scan, it selects a node (or K nodes) having the largest gain.
- *Greedy algorithm using an inverted index:* As we mentioned earlier, another approach is that we maintain an inverted index on the disk, and each time when a node is selected, we look up this inverted index and update the nodes whose marginal gain is changed in the priority queue. After all the affected nodes are updated, we pick the node (or K nodes) with the largest gain from the queue.

We compare our I/O-efficient greedy algorithm with these two baseline methods on two smaller graphs cit-HepTh and ca-DBLP.[1] Figs. 7 and 8 show the results respectively. (Note that C_H is monotone when $\alpha = 0$ and non-monotone when

[1] The two baseline methods are not scalable on large graphs. For example, using the multi-pass greedy algorithm, a complete pass requires 59 minutes on the Twitter graph.

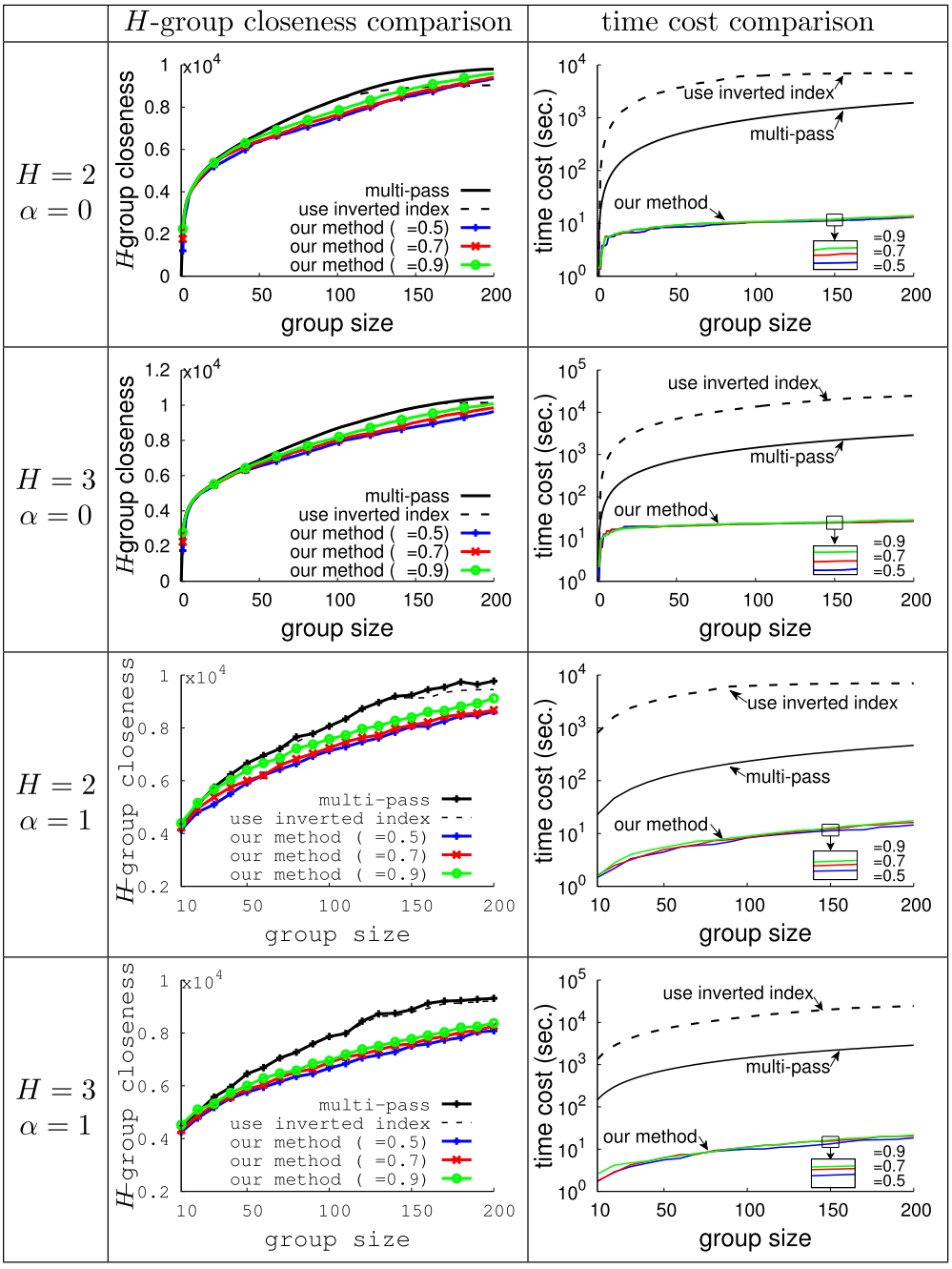


Fig. 7. Validating the I/O-efficient greedy algorithm in cit-HepTh.

$\alpha = 1$.) In the left columns of Figs. 7 and 8, we observe that the two baseline methods can obtain better solutions than our proposed method. This coincides with our analysis that the quality bound of our proposed method is worse than the simple/random greedy algorithm. The solution quality of our method can be improved by increasing λ, and as λ increases, the solution becomes better.

The main appeal of our proposed I/O-efficient greedy algorithm is its computational efficiency. This can be seen clearly from the right columns of Figs. 7 and 8. We see that the two baseline methods are much more time consuming than our proposed method. The greedy algorithm using an inverted index is actually not efficient than the multi-pass greedy algorithm in general. We also observe that λ has a trade-off effect in balancing the solution quality and computational efficiency, i.e., larger λ helps finding better solutions, but is more time consuming. This is more clearly seen in Fig. 9, which shows the results of choosing node groups containing 500 and 1000 nodes on the cit-Patents and soc-LiveJournal graphs.

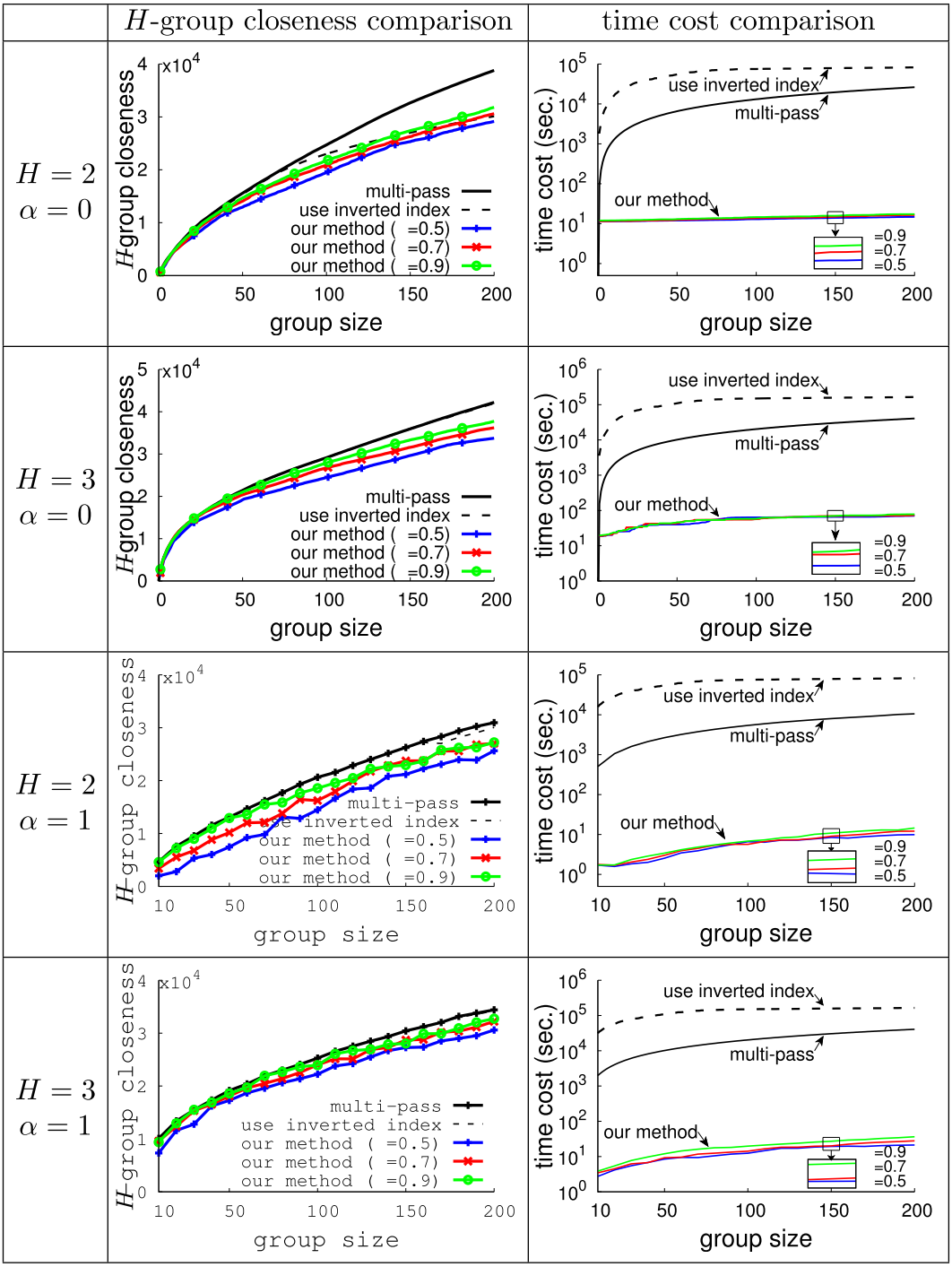
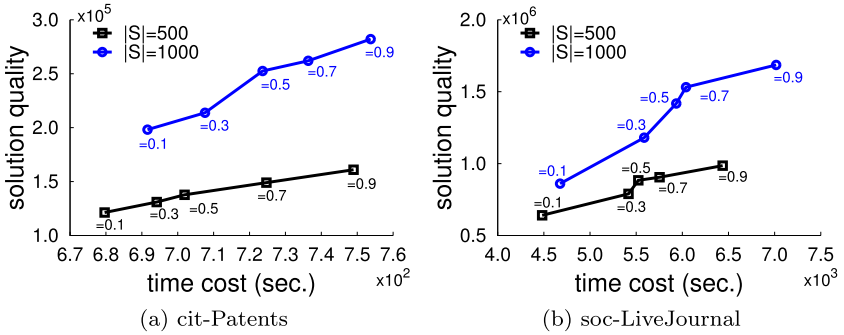


Fig. 8. Validating the I/O-efficient greedy algorithm in ca-DBLP.

When λ increases from 0.1 to 0.9, we obtain better solutions, but at the cost of longer running time. Running time on the other graphs is shown in Table 5.

6. Applications

In this section, we apply our developed techniques to study the characteristics of node groups in several real-world OSNs. We show some interesting features of the node groups identified by our algorithms.

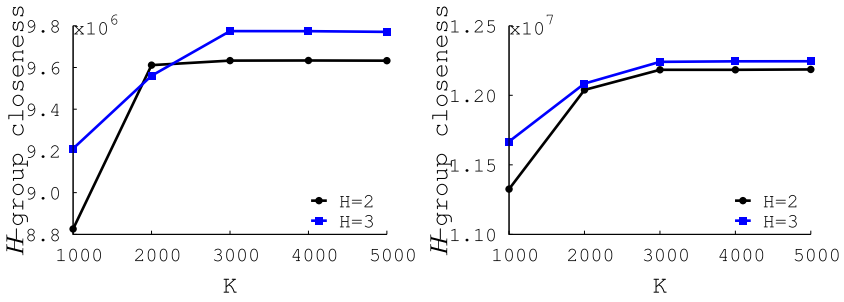


(a) cit-Patents (b) soc-LiveJournal

Fig. 9. Trade-off effect of λ in the I/O-efficient greedy algorithm ($H = 5, \alpha = 0$).

Table 5
Time cost of the I/O-efficient greedy algorithm ($K = 500, \lambda = 0.5, N = 8, m = 64$).

Graph	$H = 3, \alpha = 0$	$H = 3, \alpha = 1$
ba-Synthetic-10m	16min	12min
ba-Synthetic-20m	36min	29min
ba-Synthetic-30m	55min	43min
soc-Twitter	1 h 24min	56min
soc-Friendster	2 h 35min	2 h 12min



(a) soc-Twitter (b) soc-Friendster

Fig. 10. Node group size and H-group closeness.

6.1. Observations in large real-world OSNs

According to Theorem 2, C_H is non-monotone when α is large. Then a straightforward question to ask is: what is the minimum size of a node group maximizing C_H? We study this problem by gradually increasing budget K, and find out the minimum K after which C_H no longer increases, generally. The results are depicted in Fig. 10(a) and (b). We find that the general minimum node group size is approximately 3000 in both the soc-Twitter and soc-Friendster datasets.

For individual node centralities, several work [34,46] have suggested using degree as an alias for other node centrality metrics. For example, Lim et al. [46] find that degree is a good alias for closeness centrality. Here, we ask a similar question: are the top-K largest degree nodes good aliases for node group of size K maximizing the H-group closeness? To answer this question, we use our method to find node groups maximizing C_H, and compare the overlap with the top largest degree nodes in the same graph. Fig. 11(a) and (b) depict the results on soc-Twitter and soc-Friendster, respectively. It is surprising that the overlap is actually very low, with different H and α. When we further compare the values of H-group closeness of node groups obtained by different methods, e.g., using the greedy algorithm, top-largest degree nodes, and randomly selecting, they are indeed very different, as depicted in Fig. 12. Node groups obtained by the greedy algorithm have significantly larger H-group closeness than others. These observations indicate that the node group maximizing H-group closeness centrality in a graph cannot be simply represented by top largest degree nodes of the graph.

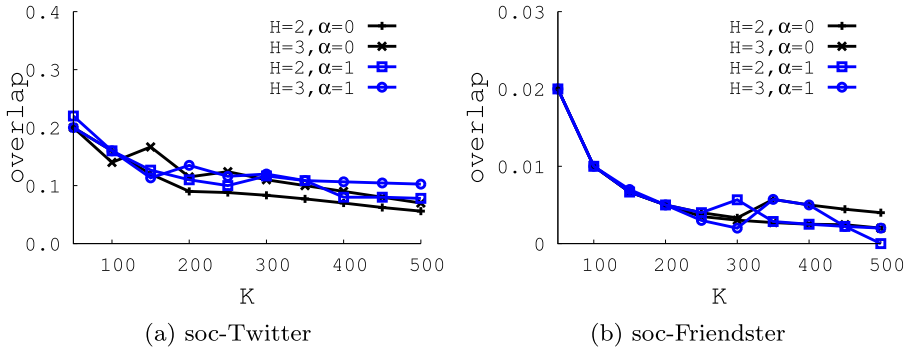


Fig. 11. Overlap between a node group by maximizing C_H and top largest degree nodes. (Overlap between two sets S and T is calculated by $|S \cap T|/K$ where $|S| = |T| = K$.).

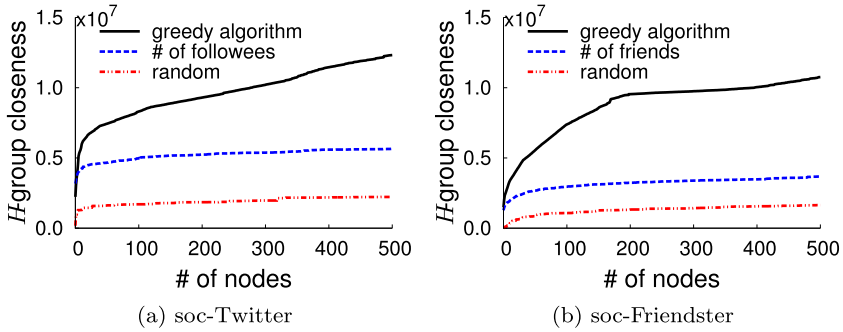


Fig. 12. Comparing the value of H-group closeness of node groups obtained by different methods.

6.2. Observations in networks with community structures

Real-world networks often exhibit community structures, where sets of nodes organize into densely connected clusters [30]. In the following discussion, we study how node groups identified by our algorithms are distributed in the network with community structures.

We conduct a series of experiments on the com-Synthetic and com-Orkut networks, and the results are depicted in Fig. 13. The top two figures compare the value of H-group closeness of node groups identified by different methods, and the results are consistent with our previous results in Fig. 12: our method can identify node groups with significantly larger H-group closeness than the other two methods.

Next, we measure how these identified large H-group closeness nodes are distributed in networks with community structures, and understand the difference with small H-group closeness nodes. Intuitively, node groups with larger H-group closeness should distribute more evenly in different communities than small H-group closeness nodes; otherwise, if high H-group closeness nodes are only in a few communities, then from these nodes we can hardly reach nodes located in other communities and this would result in small H-group closeness.

To verify our conjecture, we propose two metrics. The first metric directly measures the number of distinct communities that a group of nodes are located in, and the results are shown in the middle two figures. The second metric measures the Shannon entropy $-\sum_c p_c \log p_c$, where p_c is the proportion of selected nodes belonging to community c .[2] The Shannon entropy achieves maximum when nodes are distributed evenly in different communities. The results are shown in the bottom two figures.

We can see that the two metrics obtain consistent results: nodes identified by our method are indeed located in more communities and distributed more evenly than nodes selected by the other two methods. Although randomly selected nodes also distributed evenly, which is easy to understand, they have much smaller H-group closeness value.

7. Related work

There is a vast literature on scaling up the individual node centrality computations in large graphs [14,17,24,37,50], but little work is conducted on scaling up group centrality calculations.

[2] In the case of overlapping communities, if a node belongs to i communities simultaneously, we assume that the node contributes $1/i$ to each of its community when calculating entropy.

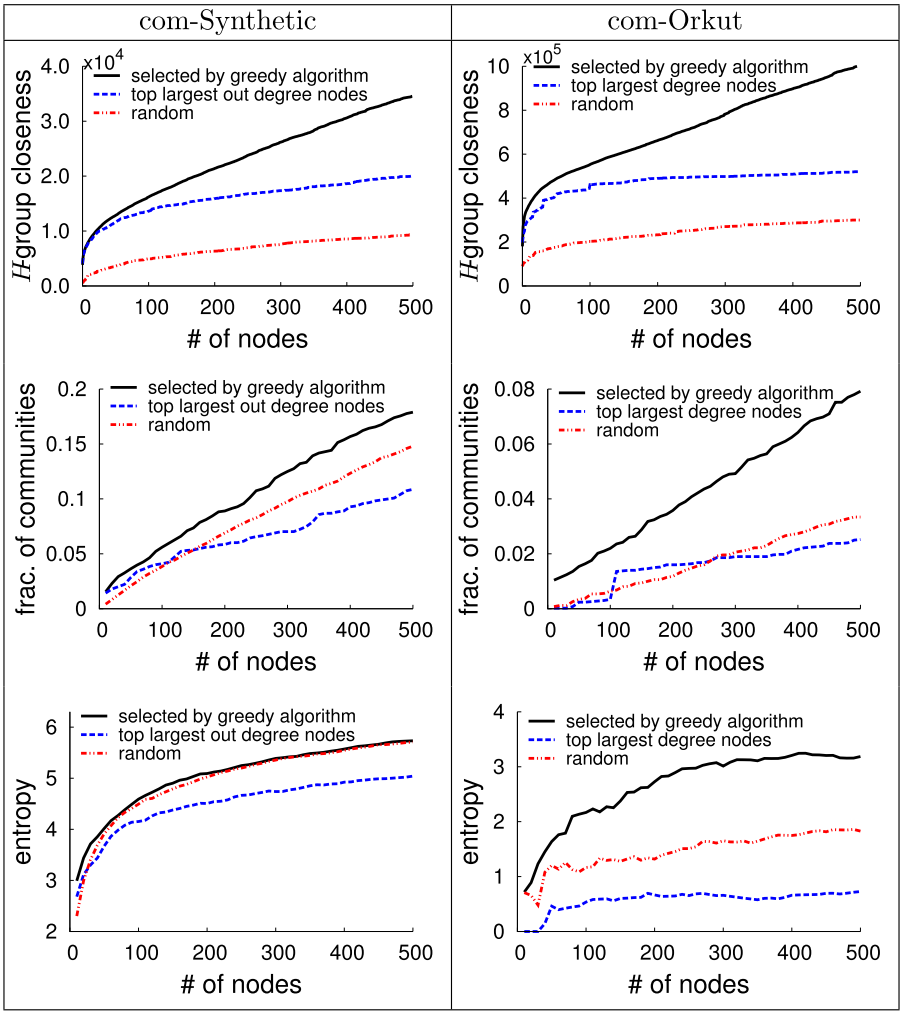


Fig. 13. Node groups in networks with community structures. ($H = 5, \alpha = 0$. The total number of communities in com-Synthetic is 2,392; in com-Orkut we discard small communities with one or two nodes and only consider the top 5,000 largest communities.)

Many centrality measures such as betweenness [50] and closeness [14,45] rely on calculating all-pairwise shortest path length in a graph, which is high computational complexity for large graphs. Specifically in closeness centrality, to reduce its computational complexity, Eppstein and Wang [17] proposed a sampling approach that only calculates the distances to a small number of sampled nodes and then uses these distances to estimate the closeness of each node. Okamoto et al. [37] also used this idea to rank the top-k nodes with the highest closeness centrality. However, Cohen et al. [14] argue that such a method may incur large estimation error when the distance distribution is heavy-tailed, and they propose a hybrid estimator that combines the sampling approach and a pivoting approach to obtain better estimates. These methods are helpful in efficiently estimating individual node closeness centrality, but are not suitable for computing group closeness in our setting.

On the other hand, there are increasingly many works on scaling centrality calculations by distributed computing. For example, Kang et al. [24] developed a parallel graph mining tool based on Hadoop to estimate individual node centrality. Oktay et al. [38] use the MapReduce programming framework to estimate pair-wise shortest path distance in a large graph. Sariyüce et al. [45] presented a distributed framework for calculating closeness centrality incrementally in dynamic graphs. However, developing distributed graph algorithms remains a challenging task [33], and there is still a need and potential for optimizing graph algorithms on a single machine [28].

Many NP-hard problems can leverage the greedy heuristic to obtain an approximate solution. Despite the importance of greedy algorithm, relatively little effort has focused on scaling it up for large datasets. Cormode et al. [15] presented a variation of the greedy algorithm that can solve the set cover problem on large scale 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 greedy algorithm is parallelization [5,13,27], although greedy algorithm is inherently sequential. To relax the constraint of sequen-

tially choosing elements, Berger et al. [4] proposed to use multiple processors to randomly cover sets and avoid covering the same elements redundantly in solving the set cover problem. Inspired by Berger's work, Chierichetti et al. [13] developed an algorithm for the MaxCover problem using MapReduce. These two methods still require data fitting in the computer cluster's memory (and also the work [27]) so that multiple processors can randomly access data; if data residents on disk, random access on disk will cause I/O costs and harm computational efficiency. Therefore they are not suitable in processing disk-resident data.

8. Conclusion

H-group closeness centrality is a useful measure in measuring the importance of a group of nodes in a graph. However, for a large graph containing billions of edges, that cannot entirely fit in a computer's main memory, computing the H-group closeness of a node group and finding a node group to maximize H-group closeness, both become challenging tasks. This work presents a systematic solution for efficiently computing and maximizing H-group closeness centrality over large disk-resident graphs. To solve the computation problem, we leverage a probabilistic counting method to efficiently estimate the H-group closeness rather than exhaustively computing it in an exact manner; to solve the maximization problem, we exploit properties of submodular functions to reduce the number of random disk accesses in the original simple/random greedy algorithm. We conduct experiments on many large graphs, and the results demonstrate the efficacy of our proposed solution.

Acknowledgments

The authors thank the editors and reviewers for their constructive comments and suggestions that greatly improved the quality of this paper. This work was supported by National Natural Science Foundation of China (61603290, 61103240, 61103241, 61221063, 91118005, 61221063, U1301254), Ministry of Education & China Mobile Joint Research Fund Program (MCM20150506, MCM20160311), Shenzhen Basic Research Grant (JCYJ20160229195940462), 863 High Tech Development Plan (2012AA011003), 111 International Collaboration Program of China, and Application Foundation Research Program of SuZhou (SYG201311).

Proof of Theorem 2

Proof. (1) Submodularity. We first prove that C_H is submodular. For every subsets $S \subseteq T \subseteq V$ and $s \in V \backslash T$,

$$\begin{aligned}\delta_s(S) &\triangleq C_H(S \cup \{s\}) - C_H(S)\\ &= \sum_{v \in V} [g_H(\min\{d_{S,v}, dist_{sv}\}) - g_H(d_{S,v})] - f(\{s\})\\ &\triangleq \sum_{v \in V} \left[\delta_s^{(v)}(S)\right] - f(\{s\}),\end{aligned}$$

and similarly, we write $\delta_s(T) \triangleq \sum_{v \in V}[\delta_s^{(v)}(T)] - f(\{s\})$ where $\delta_s^{(v)}(T) = g_H(\min\{d_{T,v}, dist_{sv}\}) - g_H(d_{T,v})$.

In the following, we show that $\delta_s^{(v)}(S) \geq \delta_s^{(v)}(T)$ always holds for every $v \in V$. Notice that $S \subseteq T$ implies $d_{S,v} \geq d_{T,v}$. Hence, we need to consider the following cases:

1. $dist_{sv} \geq d_{S,v} \geq d_{T,v}$, which yields $\delta_s^{(v)}(S) = \delta_s^{(v)}(T) = 0, \forall v \in V$.
2. $d_{S,v} \geq dist_{sv} \geq d_{T,v}$. In this case, $\delta_s^{(v)}(T) = 0$, and

$$\begin{aligned}\delta_s^{(v)}(S) &= g_H(dist_{sv}) - g_H(d_{S,v})\\ &= \begin{cases} 0, & \text{if } d_{S,v} \geq dist_{sv} > H \\ g_H(dist_{sv}), & \text{if } d_{S,v} > H \geq dist_{sv} \\ g_H(dist_{sv}) - g_H(d_{S,v}), & \text{if } H > d_{S,v} \geq dist_{sv} \end{cases}\\ &\geq 0 = \delta_s^{(v)}(T).\end{aligned}$$

3. $d_{S,v} \geq d_{T,v} \geq dist_{sv}$. In this case, we have

$$\begin{aligned}\delta_s^{(v)}(S) &= g_H(dist_{sv}) - g_H(d_{S,v})\\ &\geq g_H(dist_{sv}) - g_H(d_{T,v}) = \delta_s^{(v)}(T).\end{aligned}$$

We thus conclude that $\delta_s(S) \geq \delta_s(T)$, $\forall s \in V \backslash T$ and C_H is a submodular function.

(2) Monotonicity. The monotonicity of a submodular function is equivalent to saying that the marginal gain of every element is always non-negative. Hence, the inclusion of arbitrary nodes never decreases the objective; otherwise, if the marginal gain could be negative, the inclusion of an element can decrease the objective.

For H-group closeness $C_H(S)$, a node having the smallest marginal gain must be a node that is an out-neighbor of some node in S, and the smallest marginal gain is $g_H(0) - g_H(1) - f(\{s\})$. Hence, if the smallest marginal gain is always non-negative, i.e., $\max_{s \in V} f(\{s\}) \leq g_H(0) - g_H(1)$, C_H is monotone; otherwise C_H is non-monotone. □

Proof of Property 2

Proof. Since F is monotone, we have

$$\begin{aligned} F(OPT) - F(S_{t-1}) &\le F(OPT \cup S_{t-1}) - F(S_{t-1}) \\ &= F(OPT \backslash S_{t-1} \cup S_{t-1}) - F(S_{t-1}). \end{aligned}$$

Assume $OPT \backslash S_{t-1} = \{z_1, \ldots, z_m\}$, $m \le K$, and we define

$$Z_j \triangleq F(S_{t-1} \cup \{z_1, \ldots, z_j\}) - F(S_{t-1} \cup \{z_1, \ldots, z_{j-1}\})$$

for $j \le k$. Then we have

$$F(OPT) - F(S_{t-1}) \le \sum_{j=1}^{m} Z_j. \tag{A.1}$$

Since F is submodular,

$$Z_j \le F(S_{t-1} \cup \{z_j\}) - F(S_{t-1}) = \delta_{z_j} \le \delta_{s_t^*} \le \lambda^{-1} \delta_{s_t}.$$

Substituting it into (A.1), we have

$$F(OPT) - F(S_{t-1}) \le \sum_{j=1}^{m} Z_j \le K\lambda^{-1}(F(S_t) - F(S_{t-1})),$$

which yields

$$F(S_t) \ge [1 - (1 - \lambda/K)^t] F(OPT).$$

Finally, let $t = K$, and we conclude that

$$F(S_K) \ge [1 - (1 - \lambda/K)^K] F(OPT) \ge (1 - e^{-\lambda}) F(OPT).$$

□

Proof of Property 3

Proof. We extend Buchbinder et al.'s proof [9] by allowing choosing K elements T_k' such that the sum of their marginal gains is at least a fraction λ of the sum of the K largest marginal gains in each step k. First, we need the following lemma.

Lemma 1 (Lemma 2.2 [19]). *Let $R : 2^V \mapsto \mathbb{R}$ be submodular. Denote by A a random subset of V where each element appears with probability at most p (not necessarily independently). Then $\mathbb{E}R(A) \ge (1 - p)R(\emptyset)$.*

Intuitively, this lemma states that a "random enough" subset A cannot have much worse value than that of the empty set. Let $R(S) \triangleq F(S \cup OPT)$ which is submodular, and observe that in each step k of our random greedy algorithm, each element in $V \backslash S_{k-1}$ belongs to S_k with probability at most $1 - (1 - 1/K)^k$. Thus we have the following observation.

Lemma 2 (Observation 3.2 [9]). *In each step k of the random greedy algorithm, $\mathbb{E}F(OPT \cup S_k) \ge (1 - 1/K)^k F(OPT)$.*

Fix S_{k-1} and assume $OPT \backslash S_{k-1} = \{z_1, \ldots, z_m\}$, $m \le K$. Then, in step k, the marginal gain of the selected element s_k has expectation

$$\begin{aligned} \mathbb{E}\delta_{s_k}(S_{k-1}) &= K^{-1} \sum_{s \in T_k'} \delta_s(S_{k-1}) \ge \lambda K^{-1} \sum_{i=1}^{m} \delta_{z_i}(S_{k-1}) \\ &\ge \lambda K^{-1} \sum_{i=1}^{m} \delta_{z_i}(S_{k-1} \cup \{z_1, \ldots, z_{i-1}\}) \\ &= \lambda K^{-1} [F(OPT \cup S_{k-1}) - F(S_{k-1})] \end{aligned}$$

where the first inequality is due to the construction method of T_k', the second inequality is from submodularity, and the last equality is a straightforward telescoping sum. Now unfixing S_{k-1} and taking expectation over both S_{k-1} and s_k, we obtain

$$\begin{aligned} \mathbb{E}\delta_{s_k}(S_{k-1}) &\ge \lambda K^{-1} [\mathbb{E}F(OPT \cup \{S_{k-1}\}) - \mathbb{E}F(S_{k-1})] \\ &\ge \lambda K^{-1} \left[(1 - 1/K)^{k-1} F(OPT) - \mathbb{E}F(S_{k-1}) \right] \end{aligned}$$

where the second inequality is from Lemma 2. Reasoning in a similar way as the proof of Property 2 or use induction, we can prove

$$\mathbb{E}F(S_k) \ge \lambda k/K (1 - 1/K)^{k-1} F(OPT).$$

Letting $k = K$, yields

$$\mathbb{E}F(S_K) \geq \lambda e^{-1} F(OPT).$$

□

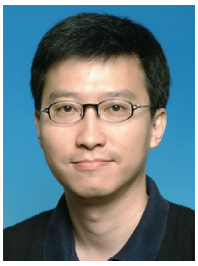
References

[1] D.R. Amancio, O.N. Oliveira Jr., L.d.F. Costa, Unveiling the relationship between complex networks metrics and word senses, Europhys. Lett. 98 (18002) (2012) 1–6.
[2] K. Avrachenkov, N. Litvak, M. Sokol, D. Towsley, Quick detection of nodes with large degrees, in: Proceedings of the ninth Workshop on Algorithms and Models for the Web Graph, 2012, pp. 54–65.
[3] E. Baralis, L. Cagliero, N. Mahoto, A. Fiori, Graphsum: discovering correlations among multiple terms for graph-based summarization, Inf. Sci. 249 (2013) 96–109.
[4] B. Berger, J. Rompel, P.W. Shor, Efficient NC algorithms for set cover with applications to learning and geometry, J. Comput. Syst. Sci. 49 (3) (1994) 454–477.
[5] G.E. Blelloch, R. Peng, K. Tangwongsan, Linear-work greedy parallel approximate set cover and variants, in: Proceedings of the 23rd annual ACM Symposium on Parallelism in Algorithms and Architectures,San Jose, California, USA, 2011, pp. 23–32.
[6] P. Boldi, M. Rosa, S. Vigna, HyperANF: Approximating the neighbourhood function of very large graphs on a budget, in: Proceedings of the 20th International World Wide Web Conference, Hyderabad, India, 2011, pp. 625–634.
[7] P. Boldi, S. Vigna, The webgraph framework I: Compression techniques, in: Proceedings of the 13th International World Wide Web Conference, New York, NY, USA, 2004, pp. 595–602.
[8] P. Boldi, S. Vigna, Axioms for centrality, Internet Math. 10 (3-4) (2014) 222–262.
[9] N. Buchbinder, M. Feldman, J.S. Naor, R. Schwartz, Submodular maximization with cardinality constraints, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, Portland, Oregon, 2014, pp. 1433–1452.
[10] M. Cha, H. Haddadi, F. Benevenuto, K.P. Gummadi, Measuring user influence in Twitter: The million follower fallacy, in: Proceedings of the fourth International AAAI Conference on Weblogs and Social Media, 2010, pp. 1–8.
[11] J. Cheng, Q. Liu, Z. Li, W. Fan, J.C. Lui, C. He, VENUS: Vertex-centric streamlined graph computation on a single PC, in: Proceedings of the 31st IEEE International Conference on Data Engineering, 2015, pp. 1131–1142.
[12] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, P. Raghavan, On compressing social networks, in: Proceedings of the 15th ACM SIGKDD International Conference on Konwledge Discovery and Data Mining, 2009, pp. 219–228. Paris, France.
[13] F. Chierichetti, R. Kumar, A. Tomkins, Max-cover in MapReduce, in: Proceedings of the 19th International World Wide Web Conference, 2010, pp. 231–240. Raleigh, North Carolina, USA.
[14] E. Cohen, D. Delling, T. Pajor, R.F. Werneck, Computing classic closeness centrality at scale, in: Procceedings of the second ACM Conference on Online Social Networks, Dublin, Ireland, 2014, pp. 37–50.
[15] G. Cormode, H. Karloff, A. Wirth, Set cover algorithms for very large datasets, in: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, Toronto, ON, Canada, 2010, pp. 479–488.
[16] M. Durand, P. Flajolet, Loglog counting of large cardinalities, in: Proceedings of the 11th Annual European Symposium on Algorithms, 2003, pp. 605–617.
[17] D. Eppstein, J. Wang, Fast approximation of centrality, in: Proceedings of the 20th annual ACM-SIAM symposium on Discrete algorithms, 2001, pp. 228–229. Washington, D.C., USA.
[18] M.G. Everett, S.P. Borgatti, The centrality of groups and classes, J. Math. Sociol. 23 (3) (1999) 181–201.
[19] U. Feige, V.S. Mirrokni, J. Vondrák, Maximizing non-monotone submodular functions, in: Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science, 2007, pp. 461–471.
[20] P. Flajolet, E. Fusy, O. Gandouet, F. Meunier, HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm, in: Proceedings of the 2007 International Conference on Analysis of Algorithms, Nancy, France, 2007, pp. 127–146.
[21] P. Flajolet, G.N. Martin, Probabilistic counting, in: Proceedings of the 24th Annual Symposium on Foundations of Computer Science, 1983, pp. 76–82.
[22] L.C. Freeman, Centrality in social networks: conceptual clarification, Soc. Netw. 1 (79) (1978) 215–239.
[23] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, H. Yu, TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC, in: Proceedings of the 19th ACM SIGKDD International Conference on Konwledge Discovery and Data Mining, Chicago, Illinois, USA, 2013, pp. 77–85.
[24] U. Kang, S. Papadimitriou, J. Sun, H. Tong, Centralities in large networks: Algorithms and observations, in: Proceedings of the 2011 SIAM International Conference on Data Mining, 2011, pp. 119–130.
[25] D. Kempe, J. Kleinberg, E. Tardos, Maximizing the spread of influence through a social network, in: Proceedings of the 9th ACM SIGKDD International Conference on Konwledge Discovery and Data Mining, Washington, D.C., USA, 2003, pp. 137–146.
[26] A. Krause, D. Golovin, Submodular function maximization, in: L. Bordeaux, Y. Hamadi, P. Kohli (Eds.), Tractability: Practical Approaches to Hard Problems, 1st ed., Cambridge University Press, 2014, pp. 71–99. chap. 3.
[27] R. Kumar, B. Moseley, S. Vassilvitskii, A. Vattani, Fast greedy algorithms in mapreduce and streaming, ACM Trans. Parallel Comput. 2 (3) (2015) 1–22.
[28] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: Large-scale graph computation on just a PC, in: Procceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, CA, USA, 2012, pp. 31–46.
[29] A. Lancichinetti, S. Fortunato, F. Radicchi, Benchmark graphs for testing community detection algorithms, Phys. Rev. E 78 (046110) (2008) 1–5.
[30] J. Leskovec, K.J. Lang, A. Dasgupta, M.W. Mahoney, Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters, Internet Math. 6 (1) (2009) 1–66.
[31] Y. Li, J. Luo, C.-Y. Chow, K.-L. Chan, Y. Ding, F. Zhang, Growing the charging station network for electrical vehicles with trajectory data analytics, in: Proceedings of the 31st IEEE International Conference on Data Engineering, 2015, pp. 1376–1387.
[32] Y. Low, D. Bickson, J. Gonzalez, Distributed GraphLab: A framework for machine learning and data mining in the cloud, in: Proceedings of the VLDB Endowment, volume 5, 2012, pp. 716–727.
[33] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (5) (2007) 1–16.
[34] A.S. Maiya, T.Y. Berger-Wolf, Online sampling of high centrality individuals in social networks, in: Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Hyderabad, India, 2010, pp. 91–98.
[35] M. Minoux, Accelerated greedy algorithms for maximizing submodular set functions, optimization techniques, Lecture Notes Control Inf. Sci. 7 (1978) 234–243.
[36] G. Nemhauser, L. Wolsey, M. Fisher, An analysis of approximations for maximizing submodular set functions - i, Math. Program. 14 (1) (1978) 265–294.
[37] K. Okamoto, W. Chen, X.-Y. Li, Ranking of closeness centrality for large-scale social networks, in: Proceedings of the second annual international workshop on Frontiers in Algorithmics, Changsha, China, 2008, pp. 186–195.
[38] H. Oktay, A.S. Balkir, I. Foster, D.D. Jensen, Distance estimation for very large networks using MapReduce and network structure indices, in: Proceedings of the Workshop on Information Networks, 2011, pp. 1–5.
[39] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank citation ranking: Bringing order to the Web, techreport, Stanford Digital Library Technologies Project, 1998.
[40] C.R. Palmer, P.B. Gibbons, C. Faloutsos, ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs, in: Proceedings of the eighth ACM SIGKDD International Conference on Konwledge Discovery and Data Mining, Edmonton, Alberta, Canada, 2002, pp. 81–90.

[41] X. Pan, S. Jegelka, J. Gonzalez, J. Bradley, M.I. Jordan, Parallel double greedy submodular maximization, in: Advances in Neural Information Processing Systems, Montreal, Canada, 2014, pp. 118–126.
[42] J. Pfeffer, K.M. Carley, k-Centralities: Local approximations of global measures based on shortest paths, in: The first International Workshop on Large Scale Network Analysis, Lyon, France, 2012, pp. 1043–1050.
[43] R. Puzis, Y. Elovici, S. Dolev, Fast algorithm for successive computation of group betweenness centrality, Phys. Rev. E 76 (056709) (2007) 1–9.
[44] A. Roy, I. Mihailovic, W. Zwaenepoel, X-Stream: Edge-centric graph processing using streaming partitions, in: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Farminton, Pennsylvania, 2013, pp. 472–488.
[45] A.E. Sariyüce, E. Saule, K. kaya, U.V. Catalyürek, STREAMER: A distributed framework for incremental closeness centrality computation, in: Proceedings of the 2013 IEEE Cluster Conference, 2013, pp. 1–8.
[46] Y.s. Lim, B. Ribeiro, D.S. Menasche, P. Basu, D. Towsley, Online estimating the top k nodes of a network, in: Proceedings of the IEEE Network Science Workshop, 2011, pp. 118–122.
[47] A. Tizghadam, A. Leon-Garcia, On traffic-aware betweenness and network criticality, in: Proceedings of the second IEEE International Workshop on Network Science for Communication Networks, 2010, pp. 1–6.
[48] S. Tschiatschek, R.K. Iyer, H. Wei, J.A. Bilmes, Learning mixtures of submodular functions for image collection summarization, in: Advances in Neural Information Processing Systems, 2014, pp. 1413–1421.
[49] K.-Y. Whang, B.T. Vander-Zanden, H.M. Taylor, A linear-time probabilistic counting algorithm for database applications, ACM Trans. Database Syst. 15 (2) (1990) 208–229.
[50] Y. Yoshida, Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches, in: Proceedings of the 20th ACM SIGKDD International Conference on Konwledge Discovery and Data Mining, 2014, pp. 1416–1425.

Junzhou Zhao received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, P.R. China. He is currently a postdoctoral researcher with the Department of Computer Science and Engineering at The Chinese University of Hong Kong, Shatin N.T. Hong Kong. His research focuses on mining and measuring massive large scale networks/graphs, with a particular interest in online social networks, a.k.a. the network science.

Pinghui Wang received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, P.R. China. From April 2012 to October 2012, he was a postdoctoral researcher with the Department of Computer Science and Engineering at The Chinese University of Hong Kong. From October 2012 to July 2013, he was a postdoctoral researcher with the School of Computer Science at McGill University, QC, Canada. He is currently an associate professor with the department of automation at Xi'an Jiaotong University. His research interests include Internet traffic measurement and modeling, traffic classification, abnormal detection, and online social network measurement.



John C.S. Lui received the Ph.D. degree in computer science from UCLA. He is currently a professor in the Department of Computer Science and Engineering at The Chinese University of Hong Kong. His current research interests include communication networks, network/system security (e.g., cloud security, mobile security, etc.), network economics, network sciences (e.g., online social networks, information spreading, etc.), cloud computing, large-scale distributed systems and performance evaluation theory. He serves in the editorial board of IEEE/ACM Transactions on Networking, IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Journal of Performance Evaluation and International Journal of Network Security. He was the chairman of the CSE Department from 2005 to 2011. He received various departmental teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award. He is also a corecipient of the IFIP WG 7.3 Performance 2005 and IEEE/IFIP NOMS 2006 Best Student Paper Awards. He is an elected member of the IFIP WG 7.3, fellow of the ACM, fellow of the IEEE, and croucher senior research fellow.

Don Towsley holds a B.A. in Physics (1971) and a Ph.D. in Computer Science (1975) from University of Texas. From 1976 to 1985 he was a member of the faculty of the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He is currently a Distinguished Professor at the University of Massachusetts in the Department of Computer Science. He has held visiting positions at IBM T.J. Watson Research Center, Yorktown Heights, NY; Laboratoire MASI, Paris, France; INRIA, Sophia-Antipolis, France; AT&T Labs-Research, Florham Park, NJ; and Microsoft Research Lab, Cambridge, UK. His research interests include networks and performance evaluation. He currently serves as Editor-in-Chief of IEEE/ACM Transactions on Networking and on the editorial boards of Journal of the ACM, and IEEE Journal on Selected Areas in Communications, and has previously served on numerous other editorial boards. He was Program Co-chair of the joint ACM SIGMETRICS and PERFORMANCE 92 conference and the Performance 2002 conference. He is a member of ACM and ORSA, and Chair of IFIP Working Group 7.3. He has received the 2007 IEEE Koji Kobayashi Award, the 2007 ACM SIGMETRICS Achievement Award, the 1998 IEEE Communications Society William Bennett Best Paper Award, and numerous best conference/workshop paper awards. Last, he has been elected Fellow of both the ACM and IEEE.

Xiaohong Guan received the B.S. and M.S. in automatic control from Tsinghua University, Beijing, China, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from the University of Connecticut, Storrs, US, in 1993. From 1993 to 1995, he was a consulting engineer at PG&E. From 1985 to 1988, he was with the Systems Engineering Institute, Xi'an Jiaotong University, Xi'an, China. From January 1999 to February 2000, he was with the Division of Engineering and Applied Science, Harvard University, Cambridge, MA. Since 1995, he has been with the Systems Engineering Institute, Xi'an Jiaotong University, and was appointed Cheung Kong Professor of Systems Engineering in 1999, and dean of the School of Electronic and Information Engineering in 2008. Since 2001 he has been the director of the Center for Intelligent and Networked Systems, Tsinghua University, and served as head of the Department of Automation, 2003–2008. He is an Editor of IEEE Transactions on Power Systems and an Associate Editor of Automata. His research interests include allocation and scheduling of complex networked resources, network security, and sensor networks. He has been elected Fellow of IEEE.