Distributed Maximal Clique Computation and Management

Yanyan Xu, James Cheng, and Ada Wai-Chee Fu

Abstract—Maximal cliques are elementary substructures in a graph and instrumental in graph analysis such as the structural analysis of many complex networks, graph clustering and community detection, network hierarchy detection, emerging pattern mining, vertex importance measures, etc. However, the number of maximal cliques is also notoriously large even for many small real world graphs. This size problem gives rise to challenges in both computing and managing the set of maximal cliques. Many algorithms for computing maximal cliques have been proposed in the literature; however, most of them are sequential algorithms that cannot scale due to the high complexity of the problem, while existing parallel algorithms for computing maximal cliques are mostly immature and especially suffer from skewed workload. As for managing the set of maximal cliques, which is essential due to its large size, there is barely any efficient method for querying or updating the set of maximal cliques.

In this paper, we first propose a distributed algorithm built on a share-nothing architecture for computing the set of maximal cliques. We effectively address the problem of skewed workload distribution due to high-degree vertices, which also leads to drastically reduced worst-case time complexity for computing maximal cliques in common real-world graphs. Then, we propose a set of fundamental query operations and efficient algorithms to process the queries, to aid more efficient and effective analysis of the set of maximal cliques. Finally, we also devise algorithms to support efficient update maintenance of the set of maximal cliques when the underlying graph is updated. We verify the efficiency of our algorithms for computing, querying, and updating the set of maximal cliques with a range of real-world graphs from different application domains.

Index Terms—Distributed maximal clique enumeration, updating maximal cliques, querying maximal cliques

1 Introduction

Let $G = (V, E)$ be a simple undirected graph. A subset of vertices, $C \subseteq V$, is called a **clique** if every vertex in C is connected to every other vertex in C by an edge in G, and C is called a **maximal clique** if any proper superset of C is not a clique. The problem of **maximal clique enumeration** (**MCE**) is to compute the set of maximal cliques in G.

Maximal cliques are elementary substructures of a graph that play a vital role in graph and network analysis, and have numerous applications. MCE is a fundamental problem in graph theory and closely related to many other important graph problems, such as maximal independent sets (or minimal vertex covers), graph coloring, maximal common induced subgraphs, etc. Apart from graph theory, maximal cliques are used in a broad range of applications such as social network analysis [1], financial network analysis [2], dynamic network clustering [3], email network hierarchy detection [4], emergent pattern detection in terrorist networks [5], structural study in behavioral and cognitive networks [6], and various analytical tasks in computational biology [7].

The problem of MCE has been extensively studied. There are three main types of algorithms. The first type is sequential in-memory algorithms [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [3], [20], [21], [22], which often do not scale well for processing large graphs, mainly because of the high complexity of MCE. The second type is sequential I/O-efficient algorithms [23], [24], [25], which focus on reducing the high cost of random disk I/Os for processing graphs that cannot fit in main memory. However, addressing the I/O problem does not solve the main computational issue as MCE is a CPU-intensive task. The third type is parallel and distributed algorithms [24], [26], [27], [28], [29], which aim at reducing the elapsed running time by parallelizing the task of MCE. However, the parallel algorithms [26], [28] require a copy of the entire input graph to be resident in main memory and cannot handle unbalanced workload. The distributed algorithms [27], [29] partition a graph and distribute the subgraphs to the worker machines where MCE is processed locally on the subgraphs. However, these algorithms do not deal with unbalanced workload due to skewed degree distribution, and may also have high communication cost since many unwanted edges may be distributed. Moreover, the cliques generated may not be maximal and hence expensive postprocessing is required to remove non-maximal cliques [27]. Recently, another algorithm was proposed to recursively split a graph into smaller subgraphs and distribute the subgraphs to worker machines for MCE [24]. Their algorithm is also not work-efficient and may also have skewed workload due to high-degree vertices, while the subgraph splitting process can be

• *Yanyan Xu, James Cheng and Ada Wai-Chee Fu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, HK.*
E-mail: {yyxu,jcheng,adafu}@cse.cuhk.edu.hk

expensive.

Apart from the computational challenges, the number of maximal cliques is notoriously known to be large even for some small graphs. The sheer number of maximal cliques severely thwarts the applications of maximal cliques since managing and analyzing such a large set is often impractical. In addition, the underlying graph may be updated from time to time and any small update (e.g., an edge insertion/deletion) to the graph can cause a considerable amount of updates to the set of maximal cliques. Such updates are not only costly but also difficult without any efficient methods to store and search the maximal cliques.

In this paper, we study two main problems: *computing the set of maximal cliques* and *managing the set of maximal cliques*. We highlight the main ideas of our algorithms and the main contributions of our work as follows.

For computing the set of maximal cliques, we examine the computational bottlenecks that hinder the performance of computing the set of maximal cliques as well as parallelizing MCE, and propose a new parallel algorithm based on the share-nothing architecture to overcome these bottlenecks. Specifically, we significantly reduce the cost of a frequent and most costly operation in the process of MCE, as well as use specific vertex orderings that can achieve $O(\sum_{i=1}^{d} n_i i 3^{i/3})$ worst-case time complexity for processing most common real-world graphs (e.g., d-degenerate graphs, power-law graphs, and sparse graphs), where d is the maximum core number of a graph (which is generally small for real-world graphs, see Table 1) and n_i is the number of vertices with core number i [30]. Note that $\sum_{i=1}^{d} n_i = |V|$. This is tremendously smaller the optimal worst-case time complexity of MCE for processing general graphs, which is $O(3^{|V|/3})$ [20]. We give detailed analysis of our algorithm and also show that our algorithm achieves balanced workload and the total amount of work performed by parallelizing the MCE task is asymptotically the same as that by sequentially executing the task.

For managing the set of maximal cliques, we focus on the following two main issues, query processing and update maintenance, neither of which has been well studied in the past (we are only aware of a method that updates a data structure that can generate only a partial set of maximal cliques [23]). First, we propose a set of fundamental query operations, and efficient algorithms to process the queries, to allow users more efficiently and effectively analyze the set of maximal cliques. Second, we propose efficient algorithms to incrementally update the set of maximal cliques when the underlying graph is updated.

We evaluate the performance of our algorithms on a set of real world graphs from different domains. Our results show that our parallel MCE algorithm is significantly more efficient than an existing MapReduce algorithm for MCE [29]. The experimental results also show that our algorithms for processing all the queries are efficient under various settings. In addition, we also demonstrate the high efficiency of our algorithms for update maintenance of the set of maximal cliques.

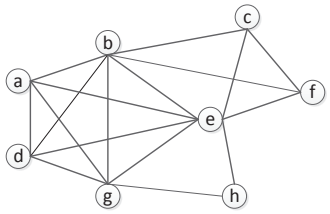


Fig. 1: A graph G

The remainder of the paper is organized as follows. Section 2 gives the basic notations and defines the problem. Sections 3, 4, and 5 present the algorithms for computing, querying, and updating the set of maximal cliques, respectively. Section 6 reports the experimental results. Section 7 discusses the related work and Section 8 gives our concluding remarks.

2 Notations and Notions

We study the problem of computing and managing maximal cliques in a simple undirected graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges of G. We keep G in its adjacency list representation. Each vertex $v \in V$ is assigned a unique **vertex ID**, denoted by $ID(v)$, where the vertex ID ranges from 1 to $|V|$. Given any two vertices u and v, we use $ID(u) < ID(v)$ or equivalently $ID(v) > ID(u)$ to denote that u is ordered before v according to the order of their IDs. In the adjacency list representation of a graph, vertices are ordered in ascending order of their IDs.

We define the set of **adjacent vertices** of a vertex $v \in V$ as $adj(v) = \{u : (u, v) \in E\}$. We further define $adj(< v) = \{u : u \in adj(v), ID(u) < ID(v)\}$ and $adj(> v) = \{u : u \in adj(v), ID(u) > ID(v)\}$.

A set of vertices, C, where $C \subseteq V$, is a **clique** in G if every $v \in C$ is adjacent to all other vertices in C, i.e., $v \in adj(u)$ for all $u \in (C \setminus \{v\})$. If $\nexists C' \supset C$ such that C' is a clique in G, then C is a **maximal clique**.

We use $\mathcal{M}(G)$ to denote the set of maximal cliques in G. We also use $\mathcal{M}_v$ to denote the set of maximal cliques **starting with** v, i.e., $\mathcal{M}_v = \{C : C \in \mathcal{M}(G), v = argmin_{u \in C} ID(u)\}$, where "$v = argmin_{u \in C} ID(u)$" means "$v \in C$ such that $ID(v) = \min\{ID(u) : u \in C\}$".

The following example illustrates the concepts.

Example 1: Figure 1 shows a graph G with 8 vertices. If we assign the vertex ID in ascending order of the vertex degree, where ties are broken arbitrarily, then we have $ID(h) = 1$, $ID(c) = 2$, $ID(f) = 3$, $ID(a) = 4$, $ID(d) = 5$, $ID(g) = 6$, $ID(b) = 7$, and $ID(e) = 8$. According to this ID assignment and ID ordering, we have $adj(< h) = \emptyset$, $adj(> h) = \{g, e\}$, $adj(< g) = \{h, a, d\}$ and $adj(> g) = \{b, e\}$. There are 3 maximal cliques in G, i.e., $\mathcal{M}(G) = \{\{a, b, d, e, g\}, \{b, c, e, f\}, \{e, g, h\}\}$.

Algorithm 1: Parallel MCE

1 **Data distribution:**
Input : $\langle v; adj(v)\rangle$ for each $v \in V$
2 **begin**
3 **foreach** *vertex* $v \in V$ **do**
4 output $\langle ID(v); (v, adj(v))\rangle$;
5 **foreach** *vertex* $u \in adj(< v)$ **do**
6 output $\langle ID(u); (v, adj(v))\rangle$;
7 **Maximal clique enumeration (MCE):**
Input : $\langle ID(v); (v, adj(v), \{(u, adj(u)) : u \in adj(> v)\}\rangle$ for each $v \in V$
8 **begin**
9 **foreach** $u \in adj(> v)$ **do**
10 $ADJ_{>v}[u] \leftarrow adj(u) \cap adj(> v)$;
11 $ADJ_v[u] \leftarrow adj(u) \cap adj(v)$;
12 LocalMCE($\{v\}, adj(> v), adj(< v), ADJ_{>v}, ADJ_v$);

Algorithm 2: LocalMCE($C, cand, prev, ADJ_{>v}, ADJ_v$)

1 **if** $cand = \emptyset$ *and* $prev = \emptyset$ **then**
2 output C as a maximal clique;
3 **else if** $cand \neq \emptyset$ **then**
4 let u_p be the vertex in $cand$ that maximizes $|cand \cap ADJ_{>v}[u_p]|$;
5 $U \leftarrow cand \setminus ADJ_{>v}[u_p]$;
6 sort U in descending order of $|ADJ_{>v}[u]|$ for all $u \in U$;
7 **foreach** $u \in U$ **do**
8 $cand \leftarrow cand \setminus \{u\}$;
9 $cand' \leftarrow cand \cap ADJ_{>v}[u]$;
10 **foreach** $w \in cand'$ **do**
11 $ADJ'_{>v}[w] \leftarrow ADJ_{>v}[w] \cap cand'$;
12 $ADJ'_v[w] \leftarrow ADJ_v[w] \cap prev$;
13 LocalMCE($C \cup \{u\}, cand', prev \cap ADJ_v[u], ADJ'_{>v}, ADJ'_v$);
14 $prev \leftarrow prev \cup \{u\}$;

Hence, we have $\mathcal{M}_h = \{e, g, h\}$, $\mathcal{M}_c = \{b, c, e, f\}$, $\mathcal{M}_a = \{a, b, d, e, g\}$, and $\mathcal{M}_v = \emptyset$ for $v \in \{b, d, e, f, g\}$.

Problem definition. Given a graph $G = (V, E)$, this paper proposes efficient algorithms for:

- *Computing the set of maximal cliques, i.e., computing* $\mathcal{M}(G)$;
- *Managing the set of maximal cliques*, which include
  - *proposing a set of fundamental query operations on* $\mathcal{M}(G)$, and
  - *update maintenance of* $\mathcal{M}(G)$ *when* G *is updated*.

3 Computing Maximal Cliques

We first present a parallel algorithm for computing the set of maximal cliques on a *shared-nothing* architecture. The algorithm consists of two phases: *data distribution* and *maximal clique enumeration* (**MCE**), which can be easily implemented in one round of Map and Reduce [31]. We first discuss these two phases, and then we also show how different orderings of vertices can reduce the complexity of MCE in common real-world graphs.

3.1 Phase I: Data Distribution

The data distribution phase is shown in Lines 1-6 of Algorithm 1. Given a simple undirected graph $G = (V, E)$, the algorithm divides the task of MCE into many subtasks to be computed in parallel. The data necessary for MCE at each worker machine is to be distributed according to the following lemma.

LEMMA *1:* Computing $\mathcal{M}_v$ requires $ADJ = \{(u, adj(u)) : u \in adj(> v)\} \cup \{(v, adj(v)\}$.

Proof: For each $C \in \mathcal{M}_v$ and each $u \in C \setminus \{v\}$, we have $u \in adj(> v)$. To compute C, we need to know whether $(u, w) \in E$ for all $u, w \in C$, i.e., whether $w \in adj(u)$. Thus, we need $adj(u)$ for each $u \in adj(> v)$. Note that there may be some $w \in adj(u)$ and $w \in adj(v)$, $ID(w) < ID(v)$, but we still require w (i.e., the whole $adj(u)$ and $adj(v)$) because w is used to check maximality (e.g., if the only superset of $\{v, u\}$ that is a clique is $\{w, v, u\}$, then without w the algorithm will report $\{v, u\}$ as a maximal clique). □

Lemma 1 implies that for each $v \in V$, $(v, adj(v))$ is only needed to compute $\mathcal{M}_u$ for each $u \in (adj(< v) \cup \{v\})$. Thus, we only need to output the key-value pair $\langle ID(u); (v, adj(v))\rangle$ for each $u \in (adj(< v) \cup \{v\})$ instead of $u \in (adj(v) \cup \{v\})$, which will prove to achieve a tremendous reduction in the complexity of MCE for processing common real-world graphs (to be analyzed in Section 3.3).

3.2 Phase II: Maximal Clique Enumeration

The second phase, i.e., MCE, is shown in Lines 7-12 of Algorithm 1. The data for computing $\mathcal{M}_v$ is distributed to an active worker. Note that for each $C \in \mathcal{M}_v$, $(C \setminus \{v\}) \subseteq adj(> v)$. Thus, to enumerate the maximal cliques in $\mathcal{M}_v$, we only need $adj(u) \cap adj(> v)$, denoted by $ADJ_{>v}[u]$, for each $u \in adj(> v)$. However, to check maximality of the cliques, we also need $adj(u) \cap adj(v)$, denoted by $ADJ_v[u]$, for each $u \in adj(> v)$. Then, the algorithm invokes the procedure "*LocalMCE*" to compute $\mathcal{M}_v$ locally at the worker, as shown in Algorithm 2.

We first explain some notations used in Algorithms 2. We use C to denote the clique currently being enumerated, ***cand*** to denote the set of candidate vertices that can be used to expand or form a clique, and ***prev*** to denote a set of vertices that are in some other maximal cliques (either enumerated previously by the same worker or enumerated by another worker) so that C is maximal only if $prev = \emptyset$. We also use $ADJ_{>v}$ and ADJ_v to denote the sets $\{ADJ_{>v}[u] : u \in adj(> v)\}$ and $\{ADJ_v[u] : u \in adj(> v)\}$, respectively.

The LocalMCE algorithm starts from a set C initially consisting of a single vertex, and repeats the process

"find a candidate vertex $u \in cand$ that is a common neighbor of all vertices in the current C and then add u to C'' until there exists no common neighbor of the current C, in which case $cand = \emptyset$, and C is returned as a maximal clique if $prev = \emptyset$. When we grow the current clique C to $C' = (C \cup \{u\})$, we refine $cand$ by intersecting it with $ADJ_{>v}[u]$ because any candidate vertex that can grow C' must be in $ADJ_{>v}[u]$. We also refine $prev$ by intersecting it with $ADJ_v[u]$ because if another maximal clique C'' exists such that C' cannot be grown into a maximal clique in the end, then $(C'' \setminus C')$ must be a subset of $ADJ_v[u]$. For the same reasons, we also refine $ADJ_{>v}[w]$ and $ADJ_v[w]$ for each new candidate vertex $w \in cand'$, by intersecting them with $cand'$ and $prev$, respectively. Then, LocalMCE is invoked recursively to further grow C'.

The following lemma shows that computing $\mathcal{M}_v$ for each $v \in V$ gives the complete set of maximal cliques, $\mathcal{M}$, and no redundant maximal clique is generated.

LEMMA 2: $\mathcal{M}(G) = \bigcup_{v \in V} \mathcal{M}_v$, and $\mathcal{M}_u \cap \mathcal{M}_v = \emptyset$ for all $u, v \in V$ and $u \neq v$.

Proof: For any $C \in \mathcal{M}(G)$, let $v = argmin_{u \in C} ID(u)$, then $C \in \mathcal{M}_v$ by the definition of $\mathcal{M}_v$. Thus, $\mathcal{M}(G) = \bigcup_{v \in V} \mathcal{M}_v$.

For any $C_u \in \mathcal{M}_u$ and $C_v \in \mathcal{M}_v$, we have $u = argmin_{w \in C_u} ID(w)$ and $v = argmin_{w \in C_v} ID(w)$. If $u \neq v$, then $C_u \neq C_v$, and thus $\mathcal{M}_u \cap \mathcal{M}_v = \emptyset$ for all $u, v \in V$ and $u \neq v$. □

A comparison between LocalMCE and classic MCE algorithms. The LocalMCE algorithm is similar to the classic MCE algorithms that apply pruning by pivot vertex (i.e., u_p in Line 4 of Algorithm 2) [9], [10], [20]. Compared with the classic algorithm, our algorithm makes the following improvement.

When growing the current clique C to a new clique $(C \cup \{u\})$, the classic algorithm refines $cand$ and $prev$ by intersecting each of them with $adj(u)$. During the processing of MCE, these set intersections are the most costly operations to be processed, especially because $adj(u)$ can be very large for those high-degree vertices in a power-law graph. Since the number of cliques enumerated (i.e., those 'C's in the intermediate steps of MCE) can be significantly larger than the number of maximal cliques and high-degree vertices are contained in many cliques, we can tremendously reduce the running time of MCE if we can reduce the cost of the set intersections.

Cheng et al. [24] proposed to reduce the cost of set intersection by extracting subgraphs and then performing MCE in the subgraphs. Apparently, the smaller the subgraphs, the smaller are the sizes of the sets (i.e., the adjacency lists of the vertices within the subgraphs). But they also show that the overall search space will be increased when smaller subgraphs are used, in addition to the cost of subgraph extraction. Thus, they use a cost model to find a balance point. However, computing the optimal point by the cost model is NP-hard.

We propose a much simpler but effective mechanism, which is also more suitable for parallel MCE. We observe that by ordering the vertices we only need $ADJ_{>v}[u]$ for growing the current clique and $ADJ_v[u]$ for checking maximality. Thus, we always refine $cand$ by intersecting it with $ADJ_{>v}[u]$ instead of with the whole set $adj(u)$, and we will show in Section 3.3 that $ADJ_{>v}[u]$ is small even for high-degree vertices in common real-world graphs. Furthermore, whenever we add a new candidate vertex u to C, we refine $ADJ_{>v}[w]$ and $ADJ_v[w]$ for each new candidate vertex w, so that in the subsequent recursive steps we can intersect with the smaller $ADJ_{>v}[w]$ and $ADJ_v[w]$ instead of with $adj(w)$.

3.3 Ordered MCE and Complexity

The time complexity of the classic algorithm for MCE [9], [10], [20] is $O(3^{|V|/3})$, which is proved to be optimal for processing general graphs [20]. However, we show how different orderings of vertices can reduce the complexity of MCE in many graphs such as power-law graphs and d-degenerate graphs, which are prevalent in real world [32], [13], [33]. We consider the following types of ordering: (1) ordering by vertex degree, (2) ordering by degeneracy number [13], and (3) ordering by the core number of the vertices.

Degeneracy ordering. An undirected graph G is *k-degenerate* if for every subgraph G' of G, there exists some vertex in G' that has k or fewer neighbors within G'. The degeneracy of G is the smallest value of k for which G is k-degenerate. If the degeneracy of G is d, then G has a degeneracy ordering such that if we assign $ID(v)$ according to the degeneracy ordering (i.e., $ID(v) = i$ if v is at the i-th position by the degeneracy ordering), then $|adj(> v)| \leq d$ for all $v \in V$.

Eppstein et al. [13] prove the following complexity of MCE for a d-degenerate graph.

THEOREM 1: Let $G = (V, E)$ be a graph where the degeneracy of G is d. When the vertices in G are ordered by degeneracy ordering, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(|V| d 3^{d/3})$ time.

Note that the above complexity is proved in [13] based on the classic MCE algorithms [9], [10], [20], but applying Algorithm 2 can only be more efficient as discussed in Section 3.2. The analysis [13] is also for sequential algorithms, while in our case for parallel computation, each worker uses $O(d 3^{d/3})$ time for computing $\mathcal{M}_v$.

Since the degeneracy d is quite small for most real-world graphs, especially for sparse graphs and power-law graphs, the above complexity is a significant reduction to the $O(3^{|V|/3})$ complexity for processing general graphs.

Degree ordering. For each $v \in V$, we assign $ID(v) = i$ if v is at the i-th position when the vertices in V are ordered in ascending order of their degree, where ties

are broken arbitrarily. Let h be the maximum value of h such that there are h vertices with degree at least h. We give the following complexity analysis.

THEOREM 2: Given a graph $G = (V, E)$, when the vertices are ordered by their degree, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(|V|h3^{h/3})$ time.

Proof: We first show that $|adj(> v)| \leq h$ for all $v \in V$. Suppose on the contrary that there exists a vertex $v \in V$ such that $|adj(> v)| > h$. Since $|adj(> v)| > h$, there are at least $(h+1)$ vertices that are ordered after v, i.e., they have degree at least as large as v. Since $|adj(v)| \geq |adj(> v)| > h$, v has degree at least $(h+1)$ and hence each $u \in adj(> v)$ has degree at least $(h+1)$. This means that there are at least $(h+1)$ vertices that have degree at least $(h+1)$, which contradicts to the fact that h is the maximum value of h such that there are h vertices with degree at least h. Thus, $|adj(> v)| \leq h$ for any $v \in V$.

If $|adj(> v)| \leq h$, then following a similar analysis to Theorem 2 of [13] we can show that computing $\mathcal{M}_v$ by Algorithm 2 uses at most $O(h3^{h/3})$ time. □

For a typical power-law graph, $h \leq |V|^{0.4}$ [23]. Thus, for processing large power-law graphs, our algorithm is a significant improvement over the classic algorithms [9], [10], [20].

Core number ordering. Another interesting vertex ordering we can use is based on *k-core* [30]. The k-core of a graph G is the largest subgraph $C_k = (V_{C_k}, E_{C_k})$ of G such that $\forall v \in V_{C_k}$, the degree of v in C_k is at least k. The *core number* of a vertex $v \in V$, denoted by $c(v)$, is defined as the largest k such that v is in C_k.

Let d be the maximum core number of a vertex in G. Note that the degeneracy of G is equal to d. However, we can give an ordering of the vertices by their core number and show that this ordering achieves better time complexity than that by degeneracy ordering given in Theorem 1. We present the details in Lemma 3, Theorems 3 and 4, which are due to Fu [34].

LEMMA 3: There exists an ordering of the vertices in G such that (1) for all $v \in V$, $ID(v) = i$ if v is at the i-th position by the ordering, (2) for all $u, v \in V$, if $ID(u) < ID(v)$, then $c(u) \leq c(v)$, and (3) for all $v \in V_i$, $|adj(> v)| \leq i$.

Proof: We obtain such an ordering as follows. We repeatedly delete from G a vertex v that has the smallest degree, where tires are broken arbitrarily, and assign $ID(v) = i$ if v is the i-th vertex to be deleted. Note that when we delete v, we also delete all edges incident to v and this changes the degree of other vertices (e.g., deleting (u, v) decreases the degree of u by 1).

For all $u, v \in V$, if $ID(u) < ID(v)$, then at the time u is deleted from G, the degree of u is not larger than that of v, which implies that $c(u) \leq c(v)$.

For all $v \in V_i$, at the time when v is deleted from G, the degree of v is at most i, which implies that there are at most i neighbors of v that are deleted later than v, and thus $|adj(> v)| \leq i$. □

By the ordering given in the proof of Lemma 3, we have the following new complexity.

Let V_i be the set of vertices with core number i and $n_i = |V_i|$. Note that $n_i = |V_{C_i} \setminus V_{C_{i+1}}|$ for $1 \leq i < d$ and $n_d = |V_{C_d}|$.

THEOREM 3: Given a graph $G = (V, E)$, when the vertices are ordered by core number ordering as given in Lemma 3, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(\sum_{i=1}^{d} n_i i 3^{i/3})$ time.

Proof: The proof is similar to that of Theorem 2 except that we partition the vertex set into $\{V_1, V_2, \ldots, V_d\}$ according to the core number of the vertices. For all $v \in V_i$, since $|adj(> v)| \leq i$ according to Lemma 3, computing $\mathcal{M}_v$ for all $v \in V_i$ uses $O(n_i i 3^{i/3})$ time. □

Since in most real-world graphs, the number of vertices with core number i decreases rapidly when i increases, the complexity using core number ordering can be much smaller than that using degeneracy ordering. The following theorem further shows that the complexity of MCE given in Theorem 3 is close to the lower bound.

THEOREM 4: The maximum possible number of maximal cliques in a graph G, where the degeneracy of G is d, is given by $\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3}$, and such a graph exists.

Proof: It is shown that the maximum possible number of maximal cliques in a graph with n vertices is bounded by $3^{n/3}$ [35]. For all $v \in V_i$, $|adj(> v)| \leq i$ according to Lemma 3, which means that the subgraph of G induced by $adj(> v)$ gives at most $3^{i/3}$ maximal cliques. Since any maximal clique in $\mathcal{M}_v$ is a subset of $(\{v\} \cup adj(> v))$, v forms at most $3^{i/3}$ maximal cliques with vertices in $adj(> v)$ and hence $|\mathcal{M}_v| \leq 3^{i/3}$. Thus, $\sum_{v \in V_i} |\mathcal{M}_v| \leq n_i 3^{i/3}$.

When $i = d$, the subgraph induced by V_d, i.e., C_d, has degeneracy d. By Theorem 3 of [13], the maximum possible number of maximal cliques in a graph with degeneracy d is given by $(n_d - d)3^{d/3}$.

Next we show that there exists a graph G that has $\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3}$ maximal cliques. The graph G is formed by $d/3$ components (for simplicity, we assume that d is a multiple of 3) and a main component. The main component is a Turan's graph $T(d, d/3)$, forming $d/3$ independent sets (each of size 3), and with edges from each vertex in each independent set connected to all vertices in the other independent sets. The other $d/3$ components are $L_1, ..., L_{d/3}$, where L_i contains m_i vertices and there is no edge among vertices in L_i, i.e., L_i is an independent set in G (for simplicity, we assume that $m_i \geq 3$). Each vertex in L_i is connected to all vertices in i independent sets in $T(d, d/3)$. Obviously, vertices in L_i have core number $3i$, and hence $m_i = n_{3i}$

for $i < d/3$, and $n_d = m_{d/3} + d$ since the d vertices in $T(d, d/3)$ are also in the d-core. For $1 \leq i \leq d$, if $i\%d \neq 0$, then $n_i = 0$ (note that no maximal clique is formed by vertices with core number i in this case). Now consider the number of maximal cliques that can be formed by vertices in L_i for $1 \leq i < d/3$. Note that each vertex in L_i is linked to i sets of independent sets $\{I_1, ..., I_i\}$ in $T(d, d/3)$. A maximal clique can be formed by picking one vertex from L_i, and one vertex from each of $I_1, ..., I_i$. Hence the number of maximal cliques that can be formed by vertices in L_i is given by $m_i 3^i = n_{3i} 3^i$. Since $n_d = m_{d/3} + d$, the number of maximal cliques that can be formed by vertices in $L_{d/3}$ is given by $m_{d/3} 3^{d/3} = (n_d - d)3^{d/3}$. Combining all components we have a graph with $\sum_{i=1}^{d-1}(n_i)3^{i/3} + (n_d - d)3^{d/3}$ maximal cliques. □

Theorem 4 implies that the worst-case time complexity of MCE in a graph with degeneracy d is at least $O(\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3})$ since there are so many maximal cliques in the worst case. Thus, the worst-case time complexity of our algorithm for computing $\mathcal{M}_v$ for all $v \in V_i$ is only a factor of i greater than the lower bound time complexity. Importantly, i is small when n_i is large, and n_d is usually small for most real-world graphs. We further verify the efficiency of MCE by each ordering by experiments.

3.4 Work Efficiency and Workload Balancing

The vertex ordering not only gives a tremendously reduced worst-case time complexity for MCE in processing many common real-world graphs especially power-law graphs and d-degenerate graphs, but it also effectively solves the problem of skewed workload distribution due to high-degree vertices.

The following theorem shows that the total amount of work performed by all the machines is asymptotically the same as that by sequentially executing Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$, which achieves the best known time complexity (as given in Theorem 3) for computing maximal cliques in a graph with degeneracy d (the previous best complexity is given by [13] as shown in Theorem 1). In addition, it also shows that the workload at each worker machine is bounded by d instead of $|V|$ or the degree of the vertices.

THEOREM 5: The total amount of work performed by all the machines for computing the set of maximal cliques by core number ordering, as well as the total amount of data being communicated, is $O(\sum_{i=1}^{d} n_i i 3^{i/3})$. The amount of work done by any worker machine is $O((|V|/\phi)n_d d 3^{d/3})$, where ϕ is the number of worker machines.

Proof: The total amount of data distributed to all the worker machines is bounded by the total amount of data processed in the worker machines. Since $\mathcal{M}_v$, for each $v \in V$, is computed only in one machine, and no machine performs any other extra work, the total amount of work performed at all the worker machines is $O(\sum_{i=1}^{d} n_i i 3^{i/3})$.

Each worker machine may compute $\mathcal{M}_v$ for $(|V|/\phi)$ vertices. Computing each $\mathcal{M}_v$ uses $O(n_i i 3^{i/3})$ time, where $v \in V_i$. In the worst case, $v \in V_d$ and $O(n_i i 3^{i/3}) = O(n_d d 3^{d/3})$. Here, we assume $O(dn_d) = O(3^{d/3})$, otherwise d must be small and $n_d \leq n_i \leq |V|$, which implies that both $O(n_i i 3^{i/3}) = O(|V|)$ and $O(n_d d 3^{d/3}) = O(|V|)$ (which is a better bound for a task like MCE). □

Note that the same analysis can be applied if the vertices are ordered by degeneracy ordering or degree ordering, by substituting the complexity given in Theorems 1 and 2 into the analysis, respectively.

4 QUERYING MAXIMAL CLIQUES

The set of maximal cliques is often too large to manage and be used directly for analysis. Thus, we propose a set of fundamental queries on $\mathcal{M}(G)$ to help users find targeted maximal cliques for more effective and efficient analysis.

4.1 Data Structure

Before we discuss the queries, we first present the data structure that we use to store the maximal cliques. When we apply Algorithm 2 to compute $\mathcal{M}_v$, for each $v \in V$, the process naturally constructs a prefix tree, denoted by T_v, such that the root of T_v is v and each root-to-leaf path represents a maximal clique in $\mathcal{M}_v$. We use the prefix tree structure because it can effectively save storage space by sharing the common subsets among maximal cliques.

The prefix trees T_v for each $v \in V$ are stored in distributed file system. To support efficient querying of the maximal cliques, we also record the following information for each T_v:

- $level(x)$: for each node x in T_v, we record the level of x in T_v, where the level of the root is 1 and $level(x) = level(parent(x)) + 1$ ($parent(x)$ is the parent of x in T_v).
- $label(x)$: a vertex $u \in adj(> v)$ may occur in multiple maximal cliques in $\mathcal{M}_v$ and hence may be represented by multiple nodes in T_v; if u is represented by node x in T_v, then $label(x) = u$.
- $nodeList_v(u)$: since a vertex $u \in adj(> v)$ may be represented by multiple nodes in T_v, $nodeList_v(u)$ gives the list of nodes in T_v that represent u, i.e., for each $node(u) \in nodeList_v(u)$, $label(node(u)) = u$, where we use $node(u)$ to indicate an occurrence of u in T_v.
- $leafList_w$: this gives the list of leaf nodes (in the order from left to right) in T_v.

Algorithm 3: Query(v)

Input : A query vertex v, and T_u for all $u \in (adj(< v) \cup \{v\})$
Output : $\mathcal{M}(v)$

```
output all maximal cliques in T_v;
foreach u ∈ adj(< v) do
    foreach node(v) ∈ nodeList_u(v) do
        initialize an empty array C;
        C[level(node(v))] ← v;
        search-up(C, node(v), T_u);
        search-down(C, node(v), T_u);
```

Algorithm 4: search-up(C, x, T)

```
while x is not the root of T do
    x ← parent(x);
    C[level(x)] ← label(x);
```

Algorithm 5: search-down(C, x, T)

```
if x is a leaf in T then
    output C' = {C[i] : 1 ≤ i ≤ level(x)} as a maximal clique;
else
    foreach child y of x in T do
        C[level(y)] ← label(y);
        search-down(C, y, T);
```

4.2 Query(v)

The first type of queries we want to consider is to find all maximal cliques containing a given vertex, denoted by $query(v)$: given a query vertex v, $query(v)$ returns $\mathcal{M}(v) = \{C : C \in \mathcal{M}(G), v \in C\}$. Note that $\mathcal{M}(v) \supseteq \mathcal{M}_v$.

The following observation shows how $query(v)$ can be processed efficiently.

OBSERVATION *1:* For any maximal clique $C \in \mathcal{M}(v)$, C can be found in T_u for some $u \in (adj(< v) \cup \{v\})$.

Proof: Let $u' = argmin_{w \in C} ID(w)$. Then, either $u' = v$ in which case C is in T_v, or $u' \in adj(< v)$ in which case C is in $T_{u'}$. □

We give our algorithm for processing $query(v)$ in Algorithm 3, which follows Observation 1 to search T_u for all $u \in (adj(< v) \cup \{v\})$ to find all maximal cliques in $\mathcal{M}(v)$.

First, we can simply perform a depth-first search in T_v, and output the maximal clique represented by each root-to-leaf path. Then, we search T_u for each $u \in adj(< v)$ as follows. We follow the node list of v in T_u. For each $node(v) \in nodeList_u(v)$, we first search up from $node(v)$ to the root (by the "search-up" procedure, i.e., Algorithm 4), and then search down from $node(v)$ to each leaf f in the subtree rooted at $node(v)$ (by the "search-down" procedure, i.e., Algorithm 5). Each root-to-f path via $node(v)$ represents a maximal clique in $\mathcal{M}_u$ that contains v, and is thus reported as a query answer.

Algorithm 6: Query($\geq s$)

Input : A size threshold s, and T_v for all $v \in V$
Output : $\mathcal{M}_{\geq s}(G)$

```
foreach v ∈ V do
    if depth(T_v) ≥ s then
        initialize an empty array C;
        foreach f ∈ leafList_v, where level(f) ≥ s, do
            C[level(f)] ← label(f);
            search-up(C, f, T_v);
            output C' = {C[i] : 1 ≤ i ≤ level(f)} as a maximal clique;
```

4.3 Query(≥s)

The set $\mathcal{M}(G)$ is large; however, most of the maximal cliques in $\mathcal{M}(G)$ are small in size. According to [19], large maximal cliques are more useful and interesting than small maximal cliques, and the set of large maximal cliques is much smaller and hence more manageable. Thus, the second type of queries we ask is to find all large maximal cliques, denoted by $query(\geq s)$: given a size threshold s, $query(\geq s)$ returns $\mathcal{M}_{\geq s}(G) = \{C : C \in \mathcal{M}(G), |C| \geq s\}$. The term "large" depends on applications and hence we allow users the flexibility to specify a size threshold s.

We give the algorithm for processing $query(\geq s)$ in Algorithm 6, which searches T_v for all $v \in V$ if the depth of T_v is at least s. Note that the query is executed in parallel and the worker machines retrieve T_v from distributed file system where T_v is stored, and then the partial query answers from each worker machine are sent to the machine where the query is issued.

To process $query(\geq s)$, we follow the leaf list of each T_v. For each leaf node f in $leafList_v$, if the level of f is at least s, we simply output the maximal clique represented by the root-to-f path. Since there may be many leaf nodes that share the same prefix subpath, we can perform an optimization as follows. During the search-up process, when we reach the internal node x at level s, we first obtain the root-to-x prefix subpath by the "search-up" procedure (i.e., Algorithm 4). Then, we invoke the "search-down" procedure (i.e., Algorithm 5) starting at x to every leaf f' in the subtree rooted at x, and output the maximal clique represented by each root-to-f' path via x. Then, we start the process again from the next leaf node in $leafList_v$ that is not in the subtree rooted at x.

4.4 Query(U)

The query $query(v)$ finds all maximal cliques containing a single vertex. We now consider another type of queries that return all maximal cliques containing a set of vertices, denoted by $query(\supseteq U)$: given a query vertex set U, $query(\supseteq U)$ returns $\mathcal{M}(\supseteq U) = \{C : C \in \mathcal{M}(G), C \supseteq U\}$.

Algorithm 7: Query($\supseteq U$)

Input : A query vertex set U,
and T_w for all $w \in (adj(< v) \cup \{v\})$,
where $v = argmin_{u \in U} ID(u)$
Output : $\mathcal{M}(\supseteq U)$

```
foreach w ∈ {adj(< v) ∪ {v}} do
    if U ⊆ adj(> w) ∪ {w} then
        foreach f ∈ leafList_w, where level(f) ≥ |U|, do
            output the maximal clique C represented by
            the root-to-f path in T_w, if C ⊇ U;
```

Note that U must be a clique for this query, otherwise $\mathcal{M}(\supseteq U) = \emptyset$.

We also consider the type of queries that return all maximal cliques that are subsets of a given query vertex set, denoted by $query(\subseteq U)$: given U, $query(\subseteq U)$ returns $\mathcal{M}(\subseteq U) = \{C : C \in \mathcal{M}(G), C \subseteq U\}$.

We first discuss how we process $query(\supseteq U)$. The following observation gives the search space of processing the query.

OBSERVATION *2:* Let $v = argmin_{u \in U} ID(u)$. For any maximal clique $C \in \mathcal{M}(\supseteq U)$, C can be found in T_w for some $w \in (adj(< v) \cup \{v\})$.

Proof: Since U itself is a clique but may not be maximal, any maximal clique containing U as a subset must be either in $\mathcal{M}_v$ or in $\mathcal{M}_w$ for some $w \in adj(< v)$. □

We give the algorithm for processing $query(\supseteq U)$ in Algorithm 7. We first prune the search space by discarding any $w \in \{adj(< v) \cup \{v\}\}$ if $U \nsubseteq adj(> w) \cup \{w\}$, because in this case U cannot be contained in any maximal clique in $\mathcal{M}_w$. Then, for each T_w, we follow the leaf list of T_w. For each leaf node f in $leafList_w$, if the level of f is smaller than $|U|$, we can safely skip f. Otherwise, we invoke the "search-up" procedure to obtain the root-to-f path in T_w. During the search process, we count the number of nodes that are not in U, and denote this number by c. If $level(f) - c < |U|$, then U cannot be contained in the maximal clique represented by this root-to-f path and hence we terminate the process and continue with next $f \in leafList_w$. Otherwise, if the search reaches the root and $level(f) - c \geq |U|$, then the maximal clique C represented by the root-to-f path must contain U as a subset, and we output C.

Next, we discuss how we process $query(\subseteq U)$. The following observation gives the search space of processing the query.

OBSERVATION *3:* For any maximal clique $C \in \mathcal{M}(\subseteq U)$, C can be found in T_u for some $u \in U$.

Proof: Let $u = argmin_{w \in C} ID(w)$. First, $C \in \mathcal{M}_u$ and C can be found in T_u. Since C must be a subset of U, we have $u \in U$. □

We give the algorithm for processing $query(\subseteq U)$ in Algorithm 8. We search T_u for each $u \in U$ according to Observation 3. Since each maximal clique represented in T_u must be a subset of $(adj(> u) \cup \{u\})$ and also of U, we first obtain $cand = adj(> u) \cap U$. Then, we invoke the "search-down" procedure starting from the root of T_u. During the process, if any internal node x is not in $cand$, we can prune the whole subtree rooted at x, because the maximal clique represented by any root-to-leaf path via x cannot be a subset of U since x is not in U. If the search reaches a leaf node f, then the maximal clique C represented by any root-to-f must be a subset of U, and we output C.

Algorithm 8: Query($\subseteq U$)

Input : A query vertex set U, and T_u for all $u \in U$
Output : $\mathcal{M}(\subseteq U)$

```
foreach u ∈ U do
    cand ← adj(> u) ∩ U;
    output the maximal clique C represented by each
    root-to-leaf path in T_u, if C ⊆ (cand ∪ {u});
```

4.5 Top-k Query

A top-k query finds the k maximal cliques that have the largest size (ties are broken arbitrarily), i.e., the query returns $\mathcal{M}(k)$ where $\mathcal{M}(k) \subseteq \mathcal{M}(G)$, $|\mathcal{M}(k)| = k$, and for any $C \in \mathcal{M}(k)$ and $C' \in (\mathcal{M}(G) \setminus \mathcal{M}(k))$, $|C| \geq |C'|$. A special case is the top-1 query, which returns the maximum clique in G.

We discuss how a top-k query can be processed as follows. Let α be the size of the maximum clique in G and N_i be the number of maximal cliques with size i (note that the distribution of the sizes of the maximal cliques can be easily obtained as a by-product of the MCE process). Then, we approximate the size of the k-th largest maximal clique as β, such that $\sum_{i=\beta+1}^{\alpha} N_i \leq k \leq \sum_{i=\beta}^{\alpha} N_i$. Based on this approximation, a top-k query can be transformed into $query(\geq \beta)$ and return the top-k largest maximal cliques in the query answer.

5 UPDATING MAXIMAL CLIQUES

To manage the set of maximal cliques, we also need to consider update maintenance of the set when the underlying graph is updated. Clearly it is not practical to re-compute $\mathcal{M}(G)$ every time when G is updated. Thus, we propose efficient algorithms to support incremental update of $\mathcal{M}(G)$. We consider two update operations: edge insertion and edge deletion. Note that vertex deletion can be considered as a series of edge deletion followed by the deletion of an isolated vertex, while the deletion (and insertion) of an isolated vertex (i.e., a maximal clique of size 1) is trivial.

In the following discussion, we process the insertion and deletion of an edge (u, v), and without loss of generality we assume $ID(u) < ID(v)$. Note that after the insertion and deletion of (u, v), the vertex ordering

discussed in Section 3.3 may change; however, we do not change the ID of the vertices during the update and hence the update result does not affect the correctness of query processing discussed in Section 4.

5.1 Edge Insertion

When an edge (u, v) is inserted into G, we have two cases to handle: (1) some existing maximal cliques become non-maximal, which need to be deleted; and (2) new maximal cliques appear in G, which need to be inserted. We process the two cases as shown in Algorithm 9.

For Case (1), an existing maximal clique C becomes non-maximal only if C contains either u or v, and $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$, since now $C \cup \{v\}$ or $C \cup \{u\}$ is a new maximal clique containing C. The following observation shows where C can be found.

OBSERVATION *4:* Let C be a maximal clique before inserting (u, v), where $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$ and either $u \in C$ or $v \in C$. Then, C is represented by a root-to-leaf path in T_w, where $w \in ((adj(> u) \cap adj(< v)) \cup \{v\})$ or $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$.

Proof: Let $w = argmin_{w' \in C} ID(w')$. Then, $C \in \mathcal{M}_w$ and hence C is represented by a root-to-leaf path in T_w. Regarding w we have two cases: (1) $ID(u) < ID(w) < ID(v)$ or $w = v$, which implies $w \in ((adj(> u) \cap adj(< v)) \cup \{v\})$; or (2) $ID(w) < ID(u)$ or $w = u$, which implies $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$. □

Following Observation 4, first Lines 3, 5 and 6 of Algorithm 9 call the "DeleteNode" procedure (i.e., Algorithm 10) to delete an existing maximal clique C from T_w, if C will become non-maximal after inserting (u, v), i.e., $C \subset cand = (adj(u) \cap adj(v)) \cup \{u, v\}$ (checked in Line 3 of Algorithm 10). Since C is represented by a root-to-leaf path in T_w but a prefix subpath of C may be shared by some other maximal cliques in $\mathcal{M}_w$, we only remove the part of the path that is not shared by any other maximal cliques (Line 4 of Algorithm 10).

Now we process Case (2). The following observation shows where a new maximal clique C should be inserted.

OBSERVATION *5:* Let C be a new maximal clique after inserting (u, v) into G. Then, C should be inserted into T_w, where $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$.

Proof: Let $w = argmin_{w' \in C} ID(w')$. Then, C should be in $\mathcal{M}_w$ and hence inserted into T_w. Since $u, v \in C$ and $ID(u) < ID(v)$, we have $ID(w) < ID(u)$ or $w = u$, which implies $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$. □

Following Observation 5, we first generate all new maximal cliques that contain $\{u, v, w\}$, for each $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$, which is processed by the algorithm "LocalMCE" (i.e., Algorithm 2), and we also insert the new maximal cliques into T_w (Lines 7-12 of Algorithm 9).

Algorithm 9: Insert(u, v)

```
1  cand ← (adj(u) ∩ adj(v)) ∪ {u, v};
2  foreach w ∈ ((adj(> u) ∩ adj(< v)) ∪ {v}) do
3      DeleteNode(v, T_w, cand);
4  foreach w ∈ ((adj(< u) ∩ adj(< v)) ∪ {u}) do
5      DeleteNode(v, T_w, cand);
6      DeleteNode(u, T_w, cand);
7      newcand ← (adj(> w) ∩ cand) \ {u, v};
8      newprev ← adj(< w) ∩ cand;
9      foreach w' ∈ newcand do
10         ADJ_{>w}[w'] ← adj(w') ∩ adj(> w);
11         ADJ_w[w'] ← adj(w') ∩ adj(w);
12     invoke LocalMCE({u, v, w}, newcand, newprev,
       ADJ_{>w}, ADJ_w) to generate new maximal cliques
       containing {u, v} and {w} and insert them into T_w;
```

Algorithm 10: DeleteNode$(x, T_w, cand)$

```
1  foreach node(x) ∈ nodeList_w(x) do
2      foreach maximal clique C represented by each root-to-leaf
       path via node(x) in T_w do
3          if C ⊂ cand then
4              starting from the leaf node and moving along
               the path up to the root, recursively delete
               from T_w any node that has no child (note that
               initially only the leaf node has no child);
```

Algorithm 11: Delete(u, v)

```
1  cand ← (adj(u) ∩ adj(v)) ∪ {u, v};
2  foreach w ∈ ((adj(< u) ∩ adj(< v)) ∪ {u}) do
3      delete unshared nodes along any path in T_w that
       represents a maximal clique containing both u and v
       by a procedure similar to Algorithm 10;
4      compute all new maximal cliques that contain
       C = {w, u} and insert them into T_w by a procedure
       similar to Lines 7-12 of Algorithm 9; and similarly for
       C = {w, v};
5  foreach w ∈ ((adj(> u) ∩ adj(< v)) ∪ {v}) do
6      compute all new maximal cliques that contain
       C = {w, v} and insert them into T_w by a procedure
       similar to Lines 7-12 of Algorithm 9;
```

5.2 Edge Deletion

The deletion of an edge (u, v) is processed in essentially the reverse way of how we process the edge insertion. As shown in Algorithm 11, we first delete all the existing maximal cliques that contain both u and v, where such maximal cliques appear in T_w, where w is defined in Observation 5. Then, we general all new maximal cliques that contain only u or v, and insert them into T_w, where w is defined in Observation 4.

6 Experimental Evaluation

We evaluate the performance of our algorithms for computing, querying, and updating the set of maximal cliques. We tested our algorithms on a cluster of 64

computing nodes, each with a 2.0GHz and 4GB RAM. The communication speed between the computing nodes in the cluster is 1Gbps.

We select five datasets from five different domains in the Stanford Large Network Dataset Collection (`http://snap.stanford.edu/data/`). The Youtube dataset comes from the community networks with ground-truth communities, where users form friendship each other. The Patents dataset is from the citation networks. The Google dataset is a Web graph from Google and it is selected from the category of Web graphs. The Skitter dataset is an Internet topology graph and it is selected from the category of Systems graphs. The Wiki dataset is a Wikipedia talk (communication) network from the Wikipedia networks. Some of the graphs are directed and we ignore the edge direction to study maximal cliques in these graphs. Table 1 gives some statistical information about the datasets, including the number of vertices ($|V|$), the number of edges ($|E|$), the maximum vertex degree (deg_{max}), the maximum core number or degeneracy of the graph (d), the size of the maximum clique (α), and the number of maximal cliques ($|\mathcal{M}(G)|$).

TABLE 1: Dataset statistics

	Youtube	Patents	Google	Skitter	Wiki
$\|V\|$	1134890	3774767	875713	1696415	2394385
$\|E\|$	2987624	16518948	4322051	11059298	4659562
deg_{max}	28754	793	6332	35455	100029
d	51	58	43	111	131
α	17	11	44	67	26
$\|\mathcal{M}(G)\|$	3265953	14787028	1417580	37322351	86333297

6.1 Results of Computing Maximal Cliques

We evaluate the performance of computing the set of maximal cliques, compared with an existing MapReduce MCE algorithm [29], denoted by **Wu et al.** in Table 3. Wu et al.'s algorithm is implemented using Hadoop. We also use the Hadoop distributed file system to store the graph data and the prefix trees that represent the set of maximal cliques, but implemented our own version of Map and Reduce phases for both data distribution (see Section 3.1) and MCE computation (see Section 3.2). We ran the algorithms on 4, 8, 16, 32, and 64 machines, respectively, and recorded the elapsed running time (in seconds).

Effect of vertex orderings. We first report the elapsed running time of our algorithm for MCE using different vertex orderings by **core number**, **degree**, and **degeneracy**, respectively, as shown in Figure 2. Note that the running time includes the time to compute the orderings. The results show that for all the datasets, when more machines are used, there is a significant decrease in the elapsed running time. On average, we record 1.60 times reduction in the elapsed running time when the number of machines is doubled for both core number ordering and degeneracy order, while for degree ordering the reduction is 1.54 times. Among the three vertex orderings, core number ordering achieves the best performance consistently in most of the cases. The result thus verifies our analysis that core number ordering gives the lowest time complexity for MCE in Section 3.3. The performance of degree ordering is comparable with that of degeneracy ordering in most cases, except for processing the Skitter dataset its performance is considerably worse when 4 to 16 machines are used.

We also measured the data distribution time of our algorithm, and found that the data distribution time is almost the same regardless of which vertex ordering is used. We thus report the data distribution time of core number ordering when different number of machines are used in Table 2. The result shows that the data distribution time does not decrease significantly when more than 16 machines are used, which may be due to the fact that the communication time also increases when more machines are used, and the amount of data to be distributed is not large. Compared with the running time shown in Figure 2, we can see that the overall data distribution time is only a small portion of the elapsed running time, especially for the Skitter and Wiki datasets. Thus, it is important to have an efficient algorithm for the main MCE process.

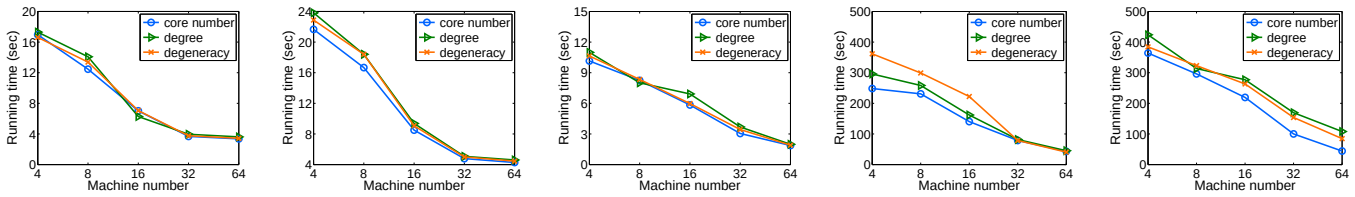
TABLE 2: Data distribution time (in seconds) for 4, 8, 16, 32, and 64 machines

	Youtube	Patents	Google	Skitter	Wiki
4	4.8286	4.3470	4.4218	4.1530	4.3232
8	3.7168	3.5472	3.6615	3.5509	3.6869
16	2.8447	2.6010	2.6174	2.9907	2.7394
32	2.4480	2.6241	2.5171	2.3189	2.4165
64	2.5729	2.3870	2.4692	2.4641	2.5176

TABLE 3: Running time (in seconds) of our algorithm and Wu et al.'s algorithm

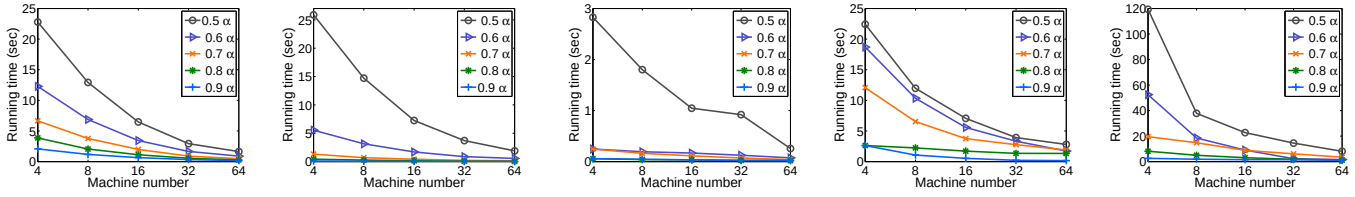
	Ours	Wu et al.
Youtube	7.0162	982
Patents	8.4981	2868
Google	5.8394	1178
Skitter	140.3460	$>$ 10000
Wiki	219.2920	$>$ 10000

Comparison with Wu et al. [29]. We now compare the performance of our algorithm (by core number ordering) with that of Wu et al.'s algorithm. We report the running time for 16 machines only due to limited space, as the running time of Wu et al.'s algorithm is orders of magnitude larger than ours in all cases while our algorithm also shows a better scalability in terms of number of machines used. As shown in Table 3, Wu et al.'s algorithm uses up to orders of magnitude more time than our algorithm, which clearly demonstrates the efficiency of our algorithm.



(a) YouTube (b) Patents (c) Google (d) Skitter (e) Wiki

Fig. 2: Running time (in seconds) of MCE with different vertex orderings



(a) Youtube (b) Patents (c) Google (d) Skitter (e) Wiki

Fig. 3: Running time (in seconds) for processing $query(\geq s)$

6.2 Results of Querying Maximal Cliques

We now evaluate the performance of our algorithms for querying the set of maximal cliques. We are not aware of any existing work on querying maximal cliques and hence we only report the results of our algorithms for different types of queries under different settings. We processed each query using 4, 8, 16, 32, and 64 machines, respectively, and recorded the elapsed running time (in seconds).

Performance of $query(v)$**.** We randomly select 1000 vertices from each graph and process $query(v)$ on each of the vertices. Table 4 reports the average running time. The result shows that this type of queries is fast to process. We also notice that the querying time is not significantly reduced (as compared with the time trend of MCE as shown in Figure 2) when more machines are used, especially when the number increases to 32 and 64. This can be explained as the processing of $query(v)$ is concentrated on T_v, which is processed by a single worker machine, and thus using more machines does not reduce the querying time a lot due to *the curse of the last worker*, i.e., when measuring the running time, the time taken by the last worker that completes the last querying task determines the elapsed time.

TABLE 4: Running time (in seconds) for $query(v)$

	Youtube	Patents	Google	Skitter	Wiki
4	0.0596	0.1810	0.0755	0.0862	0.1205
8	0.0384	0.1015	0.0474	0.0554	0.0648
16	0.0264	0.0600	0.0343	0.0387	0.0479
32	0.0251	0.0578	0.0231	0.0403	0.0433
64	0.0284	0.0671	0.0319	0.0296	0.0509

Performance of $query(\geq s)$**.** Since the size of the maximal cliques is a relative measure for different datasets, we set s to be relative to the size of the maximum clique, denoted by α in Table 1. We set s from 0.5α to 0.9α, and report the elapsed running time in Figure 3. The result shows that a smaller s has a longer running time, as more maximal cliques are returned as query answers. From Figure 3, we also see that, unlike $query(v)$, the running time of $query(\geq s)$ decreases significantly in many instances when more machines are used, especially for $s = 0.5\alpha$. This is because processing $query(\geq s)$ searches T_v for all $v \in V$ and hence the search time is balanced and shared by all machines. Overall, the result shows that the efficiency of processing $query(\geq s)$ depends on the number of query answers returned, and processing $query(\geq s)$ is fast when $s \geq 0.8\alpha$.

Performance of $query(\supseteq U)$**.** For $query(\supseteq U)$, since U should be a clique for this query, we test smaller sets of query vertices from $|U| = 2$ to $|U| = 32$. We randomly generate 1000 query sets for each $|U|$ and report the average running time in Tables 5 to 9. For some datasets, when there is no answer returned for the query, we indicate it by "$-$" in the tables. The running time of processing $query(\supseteq U)$ decreases when more machines are used in most cases. However, when $|U|$ increases, the running time may not increase. This is because the running time also depends on the number of maximal cliques that are supersets of U, which is generally smaller for a larger $|U|$.

Performance of $query(\subseteq U)$**.** For $query(\subseteq U)$, since query answers are maximal cliques that are subsets of U, we set $|U|$ to be larger. In fact, randomly selecting vertices for U does not lead to a meaningful query. Thus, we use the top-5 cores of each graph, i.e., $(d-i)$-core for $0 \leq i \leq 4$ (d is given in Table 1). The d-core

TABLE 5: Running time (in seconds) for processing $query(\supseteq U)$ in Youtube

	$\|U\|=2$	$\|U\|=4$	$\|U\|=8$	$\|U\|=16$	$\|U\|=32$
4	0.4040	0.5898	0.0640	–	–
8	0.2399	0.3221	0.0373	–	–
16	0.1339	0.1721	0.0239	–	–
32	0.0762	0.0961	0.1437	–	–
64	0.0689	0.0892	0.0982	–	–

TABLE 6: Running time (in seconds) for processing $query(\supseteq U)$ in Patents

	$\|U\|=2$	$\|U\|=4$	$\|U\|=8$	$\|U\|=16$	$\|U\|=32$
4	0.1827	0.1889	0.1626	–	–
8	0.1168	0.1132	0.1099	–	–
16	0.0730	0.0731	0.0690	–	–
32	0.0704	0.0721	0.0657	–	–
64	0.0615	0.0622	0.0605	–	–

TABLE 7: Running time (in seconds) for processing $query(\supseteq U)$ in Google

	$\|U\|=2$	$\|U\|=4$	$\|U\|=8$	$\|U\|=16$	$\|U\|=32$
4	0.8969	0.1068	0.1169	0.1267	0.1415
8	0.5956	0.0717	0.0728	0.0793	0.0963
16	0.4856	0.0561	0.0698	0.0743	0.0461
32	0.0536	0.0539	0.0532	0.0527	0.0490
64	0.0461	0.0329	0.0341	0.0338	0.0737

TABLE 8: Running time (in seconds) for processing $query(\supseteq U)$ in Skitter

	$\|U\|=2$	$\|U\|=4$	$\|U\|=8$	$\|U\|=16$	$\|U\|=32$
4	0.0784	0.1138	0.1139	0.1135	0.1138
8	0.2563	0.0782	0.0778	0.0779	0.0776
16	0.1218	0.0526	0.0532	0.0535	0.0526
32	0.0947	0.0492	0.0490	0.0493	0.0494
64	0.0811	0.0421	0.0421	0.0420	0.0428

TABLE 9: Running time (in seconds) for processing $query(\supseteq U)$ in Wiki

	$\|U\|=2$	$\|U\|=4$	$\|U\|=8$	$\|U\|=16$	$\|U\|=32$
4	0.1411	0.9583	0.7626	1.2850	–
8	0.1269	0.8204	0.6932	1.3464	–
16	0.0891	0.5984	0.7338	0.9074	–
32	0.0837	0.5777	0.5722	0.6455	–
64	0.0743	0.3172	0.3144	0.3162	–

is considered as the heart of a graph and the top-k cores for a small k are generally dense subgraphs with concentrated occurrences of large maximal cliques in the graph. We report the elapsed running time in Tables 10 to 14. In most cases, the running time of processing $query(\subseteq U)$ is longer for the $(d-i)$-core than for the $(d-i-1)$-core, since the $(d-i)$-core is larger in size, and the time also decreases when more machines are used.

Overall, processing $query(\supseteq U)$ and $query(\subseteq U)$ is considerably more costly than processing $query(v)$, but

TABLE 10: Running time (in seconds) for processing $query(\subseteq U)$ in Youtube

	d-core	$(d-1)$-core	$(d-2)$-core	$(d-3)$-core	$(d-4)$-core
4	0.0697	0.8444	0.1715	0.4195	0.5148
8	0.0485	0.4845	0.1144	0.2757	0.2764
16	0.0393	0.2390	0.0804	0.1555	0.1553
32	0.0378	0.0945	0.0711	0.0920	0.0857
64	0.0412	0.0909	0.0573	0.0805	0.0744

TABLE 11: Running time (in seconds) for processing $query(\subseteq U)$ in Patents

	d-core	$(d-1)$-core	$(d-2)$-core	$(d-3)$-core	$(d-4)$-core
4	0.1838	0.1826	0.1996	0.1948	0.3370
8	0.1142	0.1109	0.1105	0.1270	0.1211
16	0.0742	0.0733	0.0732	0.0726	0.0717
32	0.0714	0.0705	0.0698	0.0700	0.0702
64	0.0899	0.0835	0.0827	0.0820	0.0690

TABLE 12: Running time (in seconds) for processing $query(\subseteq U)$ in Google

	d-core	$(d-1)$-core	$(d-2)$-core	$(d-3)$-core	$(d-4)$-core
4	0.3370	0.3361	0.1786	0.2484	0.0672
8	0.3544	0.0326	0.1196	0.1765	0.0380
16	0.0359	0.3591	0.0888	0.1472	0.0288
32	0.0366	0.2905	0.0759	0.1171	0.0244
64	0.0749	0.7163	0.1247	0.1483	0.7354

TABLE 13: Running time (in seconds) for processing $query(\subseteq U)$ in Skitter

	d-core	$(d-1)$-core	$(d-2)$-core	$(d-3)$-core	$(d-4)$-core
4	1.2994	1.4566	1.5215	2.0322	2.3524
8	0.0681	0.9186	0.8719	1.1336	1.2886
16	0.0480	0.4170	0.5307	0.5303	0.7615
32	0.0473	0.3305	0.2924	0.3263	0.4626
64	0.0686	0.4632	0.2833	0.3048	0.3483

TABLE 14: Running time (in seconds) for processing $query(\subseteq U)$ in Wiki

	d-core	$(d-1)$-core	$(d-2)$-core	$(d-3)$-core	$(d-4)$-core
4	2.5353	2.7972	3.2721	3.4666	4.0481
8	1.4458	1.7369	2.3970	2.6223	2.6401
16	0.8564	1.0326	1.5980	1.7243	1.8993
32	0.4838	0.6255	0.8171	0.8207	0.9370
64	0.2524	0.2752	0.7225	0.7652	0.7755

is still quite efficient considering that we store and search the data in distributed file system and the query answers need to be sent to the machine where the query is issued.

6.3 Results of Updating Maximal Cliques

We now evaluate the performance of updating the set of maximal cliques, $\mathcal{M}(G)$. For edge insertion, we randomly generated 1000 edges that are not in G, and insert them into G. For edge deletion, we randomly selected 1000 existing edges in G to be deleted. Tables 15 and 16 report the average elapsed running time for each update operation, running on 4, 8, 16, 32, and 64 machines, respectively.

The results show that updating $\mathcal{M}(G)$ for both edge insertion and deletion is fast for all the datasets, but it is

TABLE 15: Running time (in seconds) for updating $\mathcal{M}(G)$ due to edge insertion

	Youtube	Patents	Google	Skitter	Wiki
4	0.0669	0.0623	0.0493	0.0830	0.0188
8	0.0410	0.0423	0.0333	0.0532	0.1304
16	0.0311	0.0299	0.0285	0.0436	0.0702
32	0.0269	0.0286	0.0353	0.0387	0.0727
64	0.0269	0.0254	0.0228	0.0336	0.0421

TABLE 16: Running time (in seconds) for updating $\mathcal{M}(G)$ due to edge deletion

	Youtube	Patents	Google	Skitter	Wiki
4	0.0780	0.0731	0.2082	0.0784	0.2459
8	0.0461	0.0486	0.1636	0.0539	0.2606
16	0.0283	0.0409	0.0294	0.0455	0.2375
32	0.0220	0.0247	0.0346	0.0393	0.2062
64	0.0261	0.0243	0.0226	0.0342	0.1529

not easy to see a trend when more machines are used. In general, when more machines are used, the running time decreases but the time reduction is not significant. We examined the details and found that when an edge (u, v) is inserted or deleted, most updates are operated on T_u and/or T_v. Since the updates on T_u and/or T_v take much longer time, the time taken by the worker that processes T_u and/or T_v is longer than that by other workers. Since we measure the elapsed time, the finishing time of the last worker determines the running time, and thus using more machines does not help much in this situation. However, we emphasize that the use of vertex ordering has significantly limited the size of any T_v to $O(3^{d/3})$. Without the ordering, the size of T_v is bounded by $O(3^{n/3})$ or $O(3^{|adj(v)|/3})$ in practice, where $O(3^{|adj(v)|/3})$ is still drastically larger than $O(3^{d/3})$ for high-degree vertices (see Table 1). Thus, our method has already achieved a good bounded balanced workload.

7 Related Work

The classic algorithms for MCE are the *backtracking* methods [8], [9], [10], [16], [20], which employ effective pruning by selecting good *pivots* to reduce the search space, and give an optimal worst-case time complexity of $O(3^{|V|/3})$ for processing general graphs [20]. For processing d-degenerate graphs, an extension of the algorithm that uses degeneracy ordering achieves a time complexity of $O(d|V|3^{d/3})$ [13], [14]. Various output-sensitive MCE algorithms whose processing time is proportional to the number of maximal cliques were also studied [21], [18]. Other algorithms, such as computing a k-clique by joining two $(k-1)$-cliques [17], by utilizing triangles [22], and enumerating maximal cliques of size larger than a threshold [19], were also studied. All these algorithms are sequential in-memory algorithms, which do not scale well due to the high complexity of MCE. To process graphs that are too large to fit in main memory, I/O-efficient algorithms that recursively extract a core part of the input graph for local MCE computation were proposed [23], [24], and a theoretical analysis on the I/O complexity of transforming the algorithm of [13], [14] into an I/O-efficient version was given in [25]. Recently, several parallel or distributed algorithms were proposed [24], [26], [27], [28], [29], which we have discussed in Section 1. Apart from algorithms for computing the maximal cliques, a recent work proposed a concise summary of the set of maximal cliques, which ensures that for every maximal clique C, there is a good portion of C that is represented by some maximal clique in the summary [36]. This summary, however, is not a lossless representation of the set of maximal cliques.

Note that computing maximal cliques is different from finding the **maximum clique**, which is to find the largest maximal clique. We refer readers to a comprehensive survey in [37] and a recent work on parallel maximum clique finding by MapReduce [38] (and the references therein).

Finally, we note that a preliminary version of this paper appeared in [39]. The preliminary version mainly focuses on computing maximal cliques, while this submission focuses on both computing and querying the set of maximal cliques. Among the two problems, querying maximal cliques is totally new, for which we introduce a new set of queries on maximal cliques, discuss algorithms for efficient query processing, and conduct extensive experiments to verify the efficiency of the querying algorithms. In addition, the current submission also adds significantly more technical details in the discussion of maximal clique computation.

8 Conclusions

We studied efficient algorithms for computing, querying, updating the set of maximal cliques. Existing parallel algorithms for computing maximal cliques are still immature and we showed that our parallel algorithm is orders of magnitude faster than the existing MapReduce algorithm for MCE [29]. Both querying and updating the set of maximal cliques have not been well studied in the past, and we proposed efficient algorithms that achieve high efficiency under various settings for a range of real-world datasets from different domains.

References

[1] K. Faust and S. Wasserman, "Social network analysis: Methods and applications," *Cambridge University Press*, 1995.
[2] V. Boginski, S. Butenko, and P. M. Pardalos, "Statistical analysis of financial networks," *Computational Statistics & Data Analysis*, vol. 48, no. 2, pp. 431–443, 2005.
[3] V. Stix, "Finding all maximal cliques in dynamic graphs," *Computational Optimization and applications*, vol. 27, pp. 173–186, 2004.
[4] G. Creamer, R. Rowe, S. Hershkop, and S. J. Stolfo, "Segmentation and automated social hierarchy detection through email network analysis," in *WebKDD/SNA-KDD*, 2007, pp. 40–58.
[5] N. M. Berry, T. H. Ko, T. Moy, J. Smrcka, J. Turnley, and B. Wu, "Emergent clique formation in terrorist recruitment," in *The AAAI-04 Workshop on Agent Organizations: Theory and Practice*, 2004.

[6] H. R. Bernard, P. D. Killworth, and L. Sailer, "Informant accuracy in social network data iv: a comparison of clique-level structure in behavioral and cognitive network data," *Social Networks*, vol. 2, no. 3, pp. 191–218, 1979.
[7] F. N. Abu-Khzam, N. E. Baldwin, M. A. Langston, and N. F. Samatova, "On the relative efficiency of maximal clique enumeration algorithms, with applications to high-throughput computational biology," in *International Conference on Research Trends in Science and Technology*, 2005.
[8] E. A. Akkoyunlu, "The enumeration of maximal cliques of large graphs," *SIAM J. Comput.*, vol. 2, no. 1, pp. 1–6, 1973.
[9] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.
[10] F. Cazals and C. Karande, "A note on the problem of reporting maximal cliques," *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 564–568, 2008.
[11] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[12] M. Chrobak and D. Eppstein, "Planar orientations with low out-degree and compaction of adjacency matrices," *Theor. Comput. Sci.*, vol. 86, no. 2, pp. 243–266, 1991.
[13] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *ISAAC (1)*, 2010, pp. 403–414.
[14] D. Eppstein and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," in *SEA*, 2011, pp. 364–375.
[15] K. Gouda and M. J. Zaki, "Efficiently mining maximal frequent itemsets," in *ICDM*, 2001, pp. 163–170.
[16] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theor. Comput. Sci.*, vol. 250, no. 1-2, pp. 1–30, 2001.
[17] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn, "Visualizing plant metabolomic correlation networks using clique-metabolite matrices," *Bioinformatics*, vol. 17, no. 12, pp. 1198–1208, 2001.
[18] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *SWAT*, 2004, pp. 260–272.
[19] N. Modani and K. Dey, "Large maximal cliques enumeration in large sparse graphs," in *COMAD*, 2009.
[20] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comput. Sci.*, vol. 363, no. 1, pp. 28–42, 2006.
[21] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM J. Comput.*, vol. 6, no. 3, pp. 505–517, 1977.
[22] L. Wan, B. Wu, N. Du, Q. Ye, and P. Chen, "A new algorithm for enumerating all maximal cliques in complex network," in *ADMA*, 2006, pp. 606–617.
[23] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks by h*-graph," in *SIGMOD Conference*, 2010, pp. 447–458.
[24] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *KDD*, 2012, pp. 1240–1248.
[25] M. T. Goodrich and P. Pszona, "External-memory network analysis algorithms for naturally sparse graphs," in *ESA*, 2011, pp. 664–676.
[26] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin, "Parallel algorithm for enumerating maximal cliques in complex network," in *Mining Complex Data*, 2009, pp. 207–221.
[27] L. Lu, Y. Gu, and R. Grossman, "dMaximalCliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution," in *Proceedings of the IEEE International Conference on Data Mining Workshops*, 2010, pp. 1320–1327.
[28] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park, "A scalable, parallel algorithm for maximal clique enumeration," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 417–428, 2009.
[29] B. Wu, S. Yang, H. Zhao, and B. Wang, "A distributed algorithm to enumerate all maximal cliques in mapreduce," in *Proceedings of the International Conference on Frontier of Computer Science and Technology*, 2009, pp. 45–51.
[30] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.
[31] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[32] S. N. Dorogovtsev and J. F. F. Mendesand, "Evolution of networks: From biological nets to the internet and www," *Oxford University Press*, 2003.
[33] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, pp. 167–256, 2003.
[34] A. W.-C. Fu, "Vertex ordering for maximal clique enumeration," *manuscript*, Janurary, 2013.
[35] J. Moon and L. Moser, "On cliques in graphs," *Israel J. Math*, vol. 3(1), pp. 23–28, 1965.
[36] J. Wang, J. Cheng, and A. W.-C. Fu, "Redundancy-aware maximal cliques," in *KDD*, 2013, pp. 122–130.
[37] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, "The maximum clique problem," in *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, 1999, pp. 1–74.
[38] J. Xiang, C. Guo, and A. Aboulnaga, "Scalable maximum clique computation using mapreduce," in *ICDE*, 2013, pp. 74–85.
[39] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu, "Distributed maximal clique computation," in *IEEE International Congress on Big Data*, 2014.