J. Parallel Distrib. Comput. 106 (2017) 79–91

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Parallelizing maximal clique and k-plex enumeration over graph data

Zhuo Wang [a], Qun Chen [a,*], Boyi Hou [a], Bo Suo [a], Zhanhuai Li [a], Wei Pan [a], Zachary G. Ives [b]

[a] School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, 710072, PR China
[b] Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA 19104-6389, USA

ARTICLE INFO

Article history:
Received 5 April 2016
Received in revised form
24 December 2016
Accepted 5 March 2017
Available online 22 March 2017

Keywords:
Maximal clique enumeration
Maximal k-plex enumeration
Parallel graph processing
MapReduce

ABSTRACT

In a wide variety of emerging data-intensive applications, such as social network analysis, Web document clustering, entity resolution, and detection of consistently co-expressed genes in systems biology, the detection of *dense subgraphs* cliques and k-plex is an essential component. Unfortunately, these problems are NP-Complete and thus computationally intensive at scale — hence there is a need to come up with techniques for distributing the computation across multiple machines such that the computation, which is too time-consuming on a single machine, can be efficiently performed on a machine cluster given that it is large enough.

In this paper, we first propose a new approach for maximal clique and k-plex enumeration, which identifies dense subgraphs by binary graph partitioning. Given a connected graph $G = (V, E)$, it has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\mu(G))$, where $\mu(G)$ represents the number of different cliques (k-plexes) existing in G. It recursively divides a graph until each task is sufficiently small to be processed in parallel. We then develop parallel solutions and demonstrate how graph partitioning can enable effective load balancing. Finally, we evaluate the performance of the proposed approach on real and synthetic graph data and show that it performs considerably better than existing approaches in both centralized and parallel settings. In the parallel setting, it can achieve the speedups of up to 10x over existing approaches on large graphs. Our parallel algorithms are primarily implemented and evaluated on MapReduce, a popular shared-nothing parallel framework, but can easily generalize to other shared-nothing or shared-memory parallel frameworks. The work presented in this paper is an extension of our preliminary work on the approach of binary graph partitioning for maximal clique enumeration. In this work, we extend the proposed approach to handle maximal k-plex detection as well.

1. Introduction

A variety of emerging applications are focused on computations over data modeled as a graph: examples include finding groups of actors or communities in social networks [22,26], Web mining [23], entity resolution [31], graph mining [45,50], and detection of consistently co-expressed gene groups in systems biology [32]. For the problems just cited, as well as a number of others, a critical component of the analysis is the detection of cliques (fully connected components), and in some cases highly connected components or k-plexes, in the structure of the network graph. For instance, for entity resolution, each clique or k-plex may represent a block of entities that might be merged.

Maximal clique and k-plex enumeration is NP-Complete. Hence a great deal of effort has been spent on efficient search algorithms [5,42,6,1,40,17]. Most of existing algorithms for maximal clique and k-plex enumeration are based on the classical algorithm proposed by Bron and Kerbosch (*BK*) [5], which uses a backtracking technique to explore search space and limits the size of its search space by remembering the search paths it has already visited. A variant [42] of the *BK* algorithm also provides a worst-case-optimal solution. In practice, the *BK* algorithm has been widely reported as being faster than its alternatives [7,18].

Data-intensive applications usually require clique and k-plex detection to be operated over large graphs, hence there is a need to parallelize it on a large machine cluster. We note that there have been a variety of proposals that divide the graph into smaller subcomponents and exploit parallelism to improve performance [47,27,49,37,12]. They have been empirically shown to speed computation in massive networks. However, built on the BK algorithm,

* Corresponding author.
E-mail addresses: wzhuo918@mail.nwpu.edu.cn (Z. Wang), chenbenben@nwpu.edu.cn (Q. Chen), houbo@mail.nwpu.edu.cn (B. Hou), caitou@mail.nwpu.edu.cn (B. Suo), lizhh@nwpu.edu.cn (Z. Li), panwei@nwpu.edu.cn (W. Pan), zives@cis.upenn.edu (Z.G. Ives).

http://dx.doi.org/10.1016/j.jpdc.2017.03.003
0743-7315/

their performance may be limited by the efficiency of *BK* search and how evenly the graph is partitioned. In fact, as we show in experimental evaluation of Section 5, their performance is quite sensitive to particular graph characteristics.

This paper presents a new approach for maximal clique and k-plex enumeration. It computes maximal dense subgraphs by recursive binary graph partitioning. Versus prior work in this area, its key insight is to exploit iterative decomposition during the computation. It recursively divides a graph until each task is sufficiently small to be processed in parallel. As a result, its computation can be effectively parallelized across a machine cluster such that the computation, which may be too time-consuming on a single machine, can be efficiently performed in parallel.

Our proposed approach is based on iterative data processing and can work with existing popular shared-nothing or shared-memory platforms. In this paper, we choose MapReduce for parallel evaluation due to its friendly programming interface and the maturity and wide availability of its implementations. However, the implementation can easily generalize to other parallel platforms. A preliminary version of our work on maximal clique enumeration has been published in [8]. In this paper, we extend the proposed approach to handle maximal k-plex enumeration as well. Our major contributions are summarized as follows:

1. We propose a novel and efficient approach for maximal clique enumeration. Given a connected graph $G = (V, E)$, it has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\mu(G))$, where $\mu(G)$ represents the number of distinct cliques in G.
2. We develop a parallel solution to maximal clique enumeration by parallelizing the proposed algorithms and implementing the corresponding parallel algorithms based on MapReduce. By using graph partitioning to divide the tasks, the proposed solution can effectively parallelize maximal clique computation with improved load balancing.
3. We extend our techniques to also support maximal k-plex enumeration and achieve similar theoretical and practical results.
4. We experimentally evaluate the performance of our proposed approach over a wide variety of open-source graph data. Our extensive experiments demonstrate that it performs considerably better than existing techniques in both centralized and parallel settings. In the parallel setting, our approach achieves the speedups of up to 10x over existing approaches on large graphs.

The rest of this paper is organized as follows: Section 2 provides the background information and the description of the existing techniques. Section 3 presents our new sequential algorithms for maximal clique and k-plex enumeration. Section 4 presents our parallel solutions to maximal clique and k-plex enumeration and their MapReduce implementations. Section 5 empirically evaluates the performance of our approach on real and synthetic datasets. Section 6 discusses related work. Finally, Section 7 concludes this paper with some thoughts on future work.

2. Preliminaries

2.1. Definition: Cliques and k-plex

A clique is a subgraph in which every pair of vertices is connected by an edge. A quasi-clique usually refers to a dense subgraph in which every vertex is directly connected to most of the other vertices. In this paper, we focus on a type of quasi-clique called *k-plex*, whose formal definition is stated as follows:

Definition 1. An induced subgraph G_i consisting of a set of vertices V_i in G is a k-plex if $\forall v \in V_i$, $deg(v) \geq (|V_i| - k)$, in which $deg(v)$ represents the degree of vertex v in G_i.

Obviously a 1-plex corresponds to a clique. The k-plexes with low values of k (e.g., $k = 2$ or 3) provide good relaxations of clique that closely resemble the cohesive subgroups existing in real networks. The definitions of a *maximal* clique (k-plex) are as follows:

Definition 2. A maximal clique (k-plex) in a graph G is a clique (k-plex) not contained by any other clique (k-plex) in G.

The problems of maximal clique and k-plex enumeration refer to identifying all the maximal cliques and k-plexes in a given graph G. Since each connected component in G can be processed independently, we assume that G is a connected graph in this paper.

2.2. Classical sequential algorithms

Algorithm 1: `enumerateCliqueBK(anchor,cand,not)`

```
1  if (cand = ∅) then
2      if (not = ∅) then
3          Output anchor;
4  else
5      fix_v ← the vertex in cand that is connected to the greatest
       number of other vertices in cand;
6      cur_v ← fix_v;
7      while (cur_v ≠ NULL) do
8          new_not ← all the vertices in not that are connected to
           cur_v;
9          new_cand ← all the vertices in cand that are connected to
           cur_v;
10         new_anchor ← sub + {cur_v};
11         enumerateCliqueBK(new_anchor,new_cand,new_not);
12         not ← not + {cur_v};
13         cand ← cand - {cur_v};
14         if (there is a vertex v in cand that is not connected to fix_v)
           then
15             cur_v ← v;
16         else
17             cur_v ← NULL;
```

For maximal clique enumeration, the BK algorithm [5] has been widely reported as being faster in practice than its alternatives [18,37]. It is in essence a depth-first search, augmented with pruning tricks. Given a current vertex v and a set of candidate vertices S, it iteratively chooses a vertex u in S such that $N(u)$ has the biggest intersection set with S, in which $N(u)$ represents the set of u's neighboring vertices in S. When the candidate set S becomes empty, the algorithm outputs corresponding cliques and backtracks. It recursively traverses a search tree, performing the operations of vertex selection, set update, clique generation and backtracking.

The BK algorithm can be sketched by Alg. 1. It uses three vertex sets to represent a search subtree: the set `anchor` records the list of vertices in the current search path, the set `cand` records the list of candidate vertices that are not in `sub` but connected to every vertex in `sub`, and the set `not` records the list of vertices that are connected to every vertex in `sub` but could not produce new maximal cliques if combined with the vertices in the `sub` set.

The existing sequential algorithms for maximal k-plex enumeration are usually extensions of the classical algorithms for maximal clique enumeration. They also use a depth-first search strategy and similar pruning methods to reduce redundant traversals. Readers can refer to the literature [29,3] for detailed algorithmic details.

2.3. Existing parallel solutions on MapReduce

In this subsection, we briefly describe the typical parallel approach [47,27,15] for maximal clique enumeration based on

MapReduce. The parallel approach for maximal k-plex enumeration [46] is similar except that it invokes centralized k-plex search instead of clique search.

The parallel approach consists of two steps. It first retrieves the relevant induced subgraph of every vertex and then computes their maximal cliques in parallel. For the computation on an individual vertex, it simply adopts the classical sequential algorithms. Typically, enumerating the maximal cliques containing a vertex is supposed to be performed on a single machine. In case that the computation on a vertex is extremely time-consuming due to the large number of maximal cliques (as we will show in Section 5), it becomes a parallel performance bottleneck. The technique proposed in [37] can parallelize maximal clique enumeration on an individual vertex. It uses candidate path data structures to record the search progress such that any search subtree can be traversed independently. It achieves better load balancing by allowing a computing node to steal some tasks from others when becoming idle. The proposed technique was implemented by MPI, but can easily generalize to other shared-nothing parallel frameworks such as MapReduce. However, as we will show in Section 5, its parallel efficiency may still be limited by unevenness of search subtree sizes.

3. Sequential algorithms

In this section, we propose novel sequential algorithms for maximal clique and k-plex enumeration, prove their complexity bounds and describe their efficient implementation.

3.1. Idea: graph partitioning

We illustrate the idea behind the new sequential algorithms by an example of maximal clique detection. As shown in Fig. 1(1), the graph G consists of the vertices, $\{v_1, v_2, v_3, v_4, v_5\}$. We randomly choose a vertex in G (e.g. v_1) as the partitioning anchor and partition G into two subgraphs G_1^+ and G_1^-. G_1^+ denotes the induced subgraph consisting of v_1 and its neighboring vertices in G, $\{v_1, v_2, v_3\}$. G_1^- denotes the induced subgraph of G consisting of all the vertices not in G_1^+, $\{v_4, v_5\}$, and their neighboring vertices in G, $\{v_2, v_3\}$. The subgraphs G_1^+ and G_1^- are shown in Fig. 1(2) and (3) respectively. We observe that any maximal clique of G is an induced subgraph of either G_1^+ or G_1^-.

Generally, we have the following theorem:

Theorem 1. *Given a graph G, we partition G into two subgraphs, G_v^+ and G_v^-, in which v denotes a partitioning anchor, G_v^+ denotes the induced subgraph consisting of vertex v and its neighboring vertices in G, and G_v^- denotes the induced subgraph consisting of all the vertices not in G_v^+ and their neighboring vertices in G. Then, any maximal clique of G is an induced subgraph of either G_v^+ or G_v^-.*

Proof. If a maximal clique contains the vertex v, it should be an induced subgraph of G_v^+. Otherwise, it should contain at least one vertex not in G_v^+. Suppose that it is the vertex u. As a result, the maximal clique is an induced subgraph of G_u, which consists of vertex u and its neighboring vertices. According to the definition of G_v^-, G_u is obviously an induced subgraph of G_v^-. Therefore, the maximal clique is an induced subgraph of G_v^-. □

According to Theorem 1, maximal clique detection in G can be performed by searching for the maximal cliques in G_v^+ and G_v^- independently. The partitioning operation can be recursively invoked until all the resulting subgraphs become cliques. Obviously, all the maximal cliques in G are contained in the set of the resulting cliques. Unfortunately, a resulting clique generated by the above process cannot be guaranteed to be maximal. Therefore, enumeration algorithms should filter out those which are not maximal.

Algorithm 2: `enumerateClique(anchor,cand,not)`

```
1  if (G(cand) is a clique) then
2      Output the clique G(anchor ∪ cand);
3  else
4      while (G(cand) is NOT a clique) do
5          Choose a vertex v with the smallest degree in G(cand);
6          anchor+ ← anchor ∪ {v};
7          cand+ ← cand ∩ N(v);
8          not+ ← not ∩ N(v);
9          if (∄u ∈ not+:u is connected to all the vertices in cand+)
           then
10             enumerateClique(anchor+,cand+,not+);
11         cand ← cand − {v};
12         not ← not ∪ {v};
13     if (∄u ∈ not:u is connected to all the vertices in cand) then
14         Output the clique G(anchor ∪ cand);
```

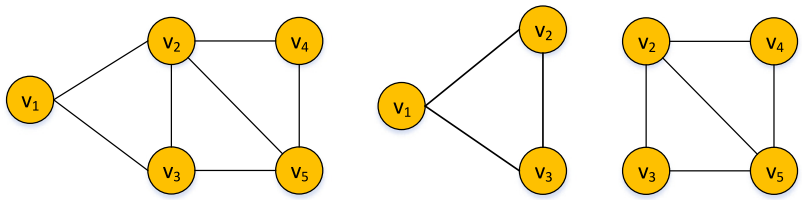
3.2. Maximal clique enumeration

3.2.1. A general algorithm

The algorithm iteratively partitions a graph until it becomes cliques. To reduce search space, it always chooses the vertex v with the smallest degree in a graph as the partitioning anchor. It can be observed that this strategy would usually result in a relatively small graph and a larger one. Generally, the small graph would be partitioned into cliques after only a few iterations, while the size of the larger one could be effectively reduced as a result of iterative partitioning. Unlike the BK algorithm, which recursively extracts the induced subgraph consisting of the vertex with the largest degree and its neighbors, our approach instead recursively performs binary partitioning by choosing the partitioning anchor with the smallest degree.

The algorithm is sketched in Alg. 2. Similar to the BK algorithm as shown in Alg. 1, it employs three sets of vertices (`anchor`, `cand` and `not`) to record the partitioning progress and prune the subtrees that cannot generate maximal cliques. The recursive function first checks whether the resulting subgraph is a clique (Line 1). If yes, it simply outputs the subgraph. Otherwise, it chooses a partitioning anchor v with the smallest degree in `cand` and partitions G(`cand`) into G(`cand`$^+$) and G(`cand`$^-$). G(`cand`$^+$) consists of v and its neighboring vertices in G(`cand`) (Lines 6–8). G(`cand`$^-$) consists of all the vertices in G(`cand`) except v (Lines 11–12). The algorithm recursively processes the subgraph G(`cand`$^+$) (Lines 9–10). Note that before the recursive function is invoked, the algorithm prunes the search space by inspecting whether there exists a vertex in the `not`$^+$ set that is connected to all the vertices in the `cand`$^+$ set (Line 9). Updating G(`cand`) with G(`cand`$^-$) (Lines 11–12), it then iteratively invokes the partition operation to search for the maximal cliques in G(`cand`$^-$) until G(`cand`$^-$) becomes a clique (Lines 4–12). After G(`cand`$^-$) becomes a clique, the algorithm checks whether it is maximal (Lines 13–14).

Given an input graph $G = (V, E)$, the algorithm can be set in motion by setting `anchor` $= \emptyset$, `not` $= \emptyset$ and `cand` $= V$. Suppose that we are running Alg. 2 on the example graph as shown in Fig. 1. Originally, `anchor` $= \emptyset$, `not` $= \emptyset$ and `cand` $= \{v_1, v_2, v_3, v_4, v_5\}$. The vertex v_1 has the smallest degree of 2, is thus chosen as the partitioning anchor. G is then partitioned into G_1^+ and G_1^-. G_1^+ consists of v_1 and its neighboring vertices, $\{v_1, v_2, v_3\}$. G_1^- consists of the vertices, $\{v_2, v_3, v_4, v_5\}$. For G_1^+, `anchor` $= \{v_1\}$, `not` $= \emptyset$ and `cand` $= \{v_2, v_3\}$. For G_1^-, `anchor` $= \emptyset$, `not` $= \{v_1\}$ and `cand` $= \{v_2, v_3, v_4, v_5\}$. It can be observed that G_1^- is not a clique and v_3 has the smallest degree of 2 in G_1^-. G_1^- would then be partitioned into two subgraphs consisting of $\{v_2, v_3, v_5\}$ and



(1) G. (2) G_1^+. (3) G_1^-.

Fig. 1. A graph partitioning example.

$\{v_2, v_4, v_5\}$ respectively, which are both clique. Therefore, maximal cliques of G can be computed with two partitioning operations.

On the correctness of Alg. 2, we have Theorem 2. Here we only provide with its proof sketch. More details can be found in our technical report [9].

Theorem 2. *Alg. 2 exactly returns all the maximal cliques in G.*

Proof. Firstly, consider a variant of Alg. 2, denoted by Alg. 2*, which is the same as Alg. 2 except that it does not have the pruning operations as specified at Lines 9 and 15 of Alg. 2. It can be observed that all the maximal cliques in G are contained in the set of cliques returned by Alg. 2*. Secondly, if there exists a vertex in the current `not` set that is connected to all the vertices in the current `cand` set, the function cannot generate any new maximal clique. Finally, any clique returned by Alg. 2 is maximal. □

In practical implementation, the algorithm iteratively partitions an input graph in a depth-first manner. After partitioning G into $G(\texttt{cand}^-)$ and $G(\texttt{cand}^+)$, it always processes the $G(\texttt{cand}^-)$ subgraph before $G(\texttt{cand}^+)$, and pushes the resulting $G(\texttt{cand}^+)$ into a stack for later processing. Whenever a $G(\texttt{cand}^-)$ subgraph becomes a clique, it pops a $G(\texttt{cand}^+)$ subgraph from the stack and repeats the iterative partitioning operation. On the space and time complexity of Alg. 2, we have the following theorem:

Theorem 3. *Given a connected graph $G = (V, E)$, Alg. 2 has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\mu(G))$, in which $\mu(G)$ represents the number of different cliques in G.*

Proof. We first analyzes its space complexity. It iteratively partitions the $G(\texttt{cand}^-)$ branch until $G(\texttt{cand}^-)$ becomes a clique. Besides the $G(\texttt{cand}^-)$ graph, it also has to store the resulting $G(\texttt{cand}^+)$ subgraphs in a stack S. Each $G(\texttt{cand}^+)$ results from a partitioning operation with a vertex v_i as anchor. Note that the first-in-last-out operation order of stack ensures that each $G(\texttt{cand}^+)$ subgraph in the stack S has a distinct partitioning anchor. Since each vertex in the $\texttt{anchor}^+$, $\texttt{cand}^+$ and $\texttt{not}^+$ sets of $G(\texttt{cand}^+)$ (except the vertex v_i itself) should be connected to v_i, the required space to store $G(\texttt{cand}^+)$ is bound by $O(|E_i|)$, in which E_i represents the set of edges with v_i as one of its end points. As a result, the required space to store all $G(\texttt{cand}^+)$ branches is bound by $O(|E|)$. It follows that the space complexity of Alg. 2 is $O(|E|)$.

On the time complexity, consider a variant of Alg. 2 without the pruning operation specified on Line 9. Its time complexity, $O(|E|\mu(G))$, is an upper bound on the time complexity of Alg. 2. □

3.2.2. Implementation notes

The program reads a graph G into memory, and then iteratively computes the maximal cliques of every vertex in G. We store the vertices in the original graph G in an array and their adjacency lists as hash sets. Similarly, all the `cand` sets are maintained by hash sets. As a result, the intersection of two vertex sets can be performed by hash look-ups. Clique verification is achieved by checking vertex degrees. The degree of a vertex v_i in $\texttt{cand}^+$ of G_v^+ is computed by intersecting the adjacency set of v_i with the $\texttt{cand}^+$ set. For the vertices in the $\texttt{cand}^-$ set of G_v^-, only those connected to v need to decrease their degrees by 1. Selecting a partitioning anchor with the minimal degree in `cand` however requires $O(|\texttt{cand}|)$ time because it has to sequentially scan all the vertices in the hash set. To enable more efficient anchor selection, we also maintain a degree map, in which the vertex degrees of `cand` are stored as a sorted linked list and each entry in the degree list has a corresponding vertex list consisting of all the vertices with the specified degree. The degree map of the G_v^- subgraph is inherited from that of its parent with corresponding updates while the degree map of G_v^+ is constructed from scratch. With the degree map, selecting a partitioning anchor in `cand` only involves picking up a vertex in the vertex list of the first entry in the degree list. It takes only constant time.

3.3. Maximal k-plex enumeration

K-plex detection is more costly than clique detection but allows for relaxed matching criteria, and hence is needed in some applications. In this section, we generalize the sequential algorithm for maximal clique enumeration to handle maximal k-plex ($k \geq 2$) enumeration.

3.3.1. A general algorithm

Algorithm 3: `enumerateKplex(anchor,cand,not)`

```
1  if (G(anchor ∪ cand) is a k-plex) then
2      if (∄u ∈ not:G(anchor ∪ cand ∪ {u}) is a k-plex) then
3          Output G(anchor ∪ cand);
4  else
5      while (G(anchor ∪ cand) is NOT a k-plex) do
6          Choose a vertex v with the smallest degree in G(cand);
7          anchor+ ← anchor ∪ {v};
8          cand+ ← {u|u ∈ cand ∧ G(anchor+ ∪ {u}) is a k-plex};
9          not+ ← {u|u ∈ not ∧ G(anchor+ ∪ {u}) is a k-plex};
10         if (∄u ∈ not+:u is connected to all the vertices in anchor+
           and cand+) then
11             enumerateKplex(anchor+,cand+,not+);
12         cand ← cand − {v};
13         not ← not ∪ {v};
14     if (∄u ∈ not:G(anchor ∪ cand ∪ {u}) is a k-plex) then
15         Output G(anchor ∪ cand);
```

Similar to the case of maximal clique enumeration, the algorithm for maximal k-plex enumeration selects the vertex v with the smallest degree as the partitioning anchor in a graph G and partitions it into two subgraphs G_v^+ and G_v^-. The G_v^+ branch corresponds to the case that maximal k-plexes contain the vertex v while the G_v^- branch corresponds to the other case that maximal k-plexes do not contain v. It thereafter proceeds to iteratively partition G_v^- until it becomes a k-plex. All the generated G_v^+s are recursively partitioned.

The recursive function is sketched in Alg. 3. The anchor, cand and not vertex sets serve the purposes similar to those explained in Alg. 2. Lines 7–9 compute the three vertex sets for G_v^+. Line 10 specifies a pruning condition. If there exists a vertex u in $\texttt{not}^+$ such that u is connected to all the vertices in $\texttt{anchor}^+$ and $\texttt{cand}^+$, then any induced k-plex of G_v^+ is not maximal. As a result, the G_v^+ branch can be pruned. However, this condition does not guarantee to filter out all the non-maximal k-plexes. Therefore, Line 2 checks whether adding any vertex in not to a candidate k-plex would result in a bigger k-plex. If yes, the candidate k-plex is not maximal.

Similar to Alg. 2, given an input graph $G = (V, E)$, Alg. 3 can be set in motion by setting $\texttt{anchor} = \emptyset$, $\texttt{not} = \emptyset$ and $\texttt{cand} = V$. On algorithmic correctness, we have Theorem 4, whose proof is similar to the proof of Theorem 2.

Theorem 4. *Alg. 3 exactly returns all the maximal k-plexes in G.*

Proof. Firstly, consider a variant of Alg. 3, denoted by Alg. 3*, which is the same as Alg. 3 except that it does not have the pruning operations as specified at Lines 10 and 14 of Alg. 3. It can be observed that all the maximal k-plexes in G are contained in the set of k-plexes returned by Alg. 3*. Secondly, if there exists a vertex in the current not set that is connected to all the vertices in the current anchor and cand sets, the recursive function cannot generate any new maximal k-plex. Finally, any k-plex returned by Alg. 3 is maximal. □

On the space and time complexity, we have Theorem 5.

Theorem 5. *Given a connected graph $G = (V, E)$, Alg. 3 has the space complexity of $O(|V|^2)$ and the time complexity of $O(|E|\mu_k(G))$, in which $\mu_k(G)$ represents the number of different k- plexes in G.*

Proof. Unlike the case of maximal clique enumeration, Alg. 3 cannot guarantee that the vertices in the $\texttt{anchor}^+$, $\texttt{cand}^+$ and $\texttt{not}^+$ sets of G_v^+ are connected to the partitioning anchor v. As a result, it requires $O(|V|^2)$ space to store intermediate results in the worst case. On time complexity, each line of statement in Alg. 3 can be performed in $O(|E|)$ time. The size of the traversal tree generated by the recursive function is bound by $O(\mu_k(G))$. □

3.3.2. A space optimal algorithm

Note that the space complexity of Alg. 3, $O(|V|^2)$, is not asymptotically optimal. In this subsection, we present a variant of Alg. 3 (as shown in Alg. 4), which has the same time complexity $O(|E|\mu_k(G))$ but achieves the optimal $O(|E|)$ space complexity. Besides three vertex sets, anchor, cand and not, it also uses a *stack* data structure S to track the traversal progress of the partitioning search tree. The stack S maintains a series of vertices, each of which is marked as *inclusive* or *exclusive*. The entry of an *inclusive* vertex v in S, denoted by v^+, corresponds to the partitioning branch G_v^+, in which the searched subgraphs should contain v. In contrast, the entry of an *exclusive* vertex v in S, denoted by v^-, corresponds to the other partitioning branch G_v^-, in which the searched subgraphs do not contain v.

Similar to Alg. 3, Alg. 4 executes a depth-first traversal. It first traverses along the G_v^+ branch (Lines 10–13). The anchor v is added to anchor. The *inclusive* vertex v, v^+, is pushed into S. cand is updated in the same way as in Alg. 3. Whenever it reaches a leaf node, it backtracks to the last *inclusive* branch (Lines 15–17), which corresponds to the latest *inclusive* vertex pushed into S. Then it continues to traverse along the G_v^- branch (Lines 18–23). The *exclusive* vertex v, v^-, is pushed into S. The vertex v is also removed from anchor. Because the algorithm maintains only one cand during the partitioning process, the cand set of G_v^- has to be constructed from the current anchor and not sets (Line 23). It is worthy to point out that when traversing along the G_v^+ branch, the algorithm does not update the not set. This modification is to facilitate constructing the not set of G_v^- branch.

Algorithm 4: A Space Optimal Algorithm for Maximal K-plex Enumeration on $G = (V, E)$

```
1  Stack S ← ∅;
2  anchor ← ∅;
3  cand ← V;
4  not ← ∅; while (G(anchor ∪ cand) is not a k-plex) or (S ≠ ∅) do
5      if G(anchor ∪ cand) can not be pruned then
6          if G(anchor ∪ cand) is a k-plex then
7              if (∄u ∈ not:G(anchor ∪ cand ∪ {u}) is a k-plex)
               then
8                  Output G(anchor ∪ cand);
9          else
10             Choose a vertex v with the smallest degree in G(cand);
11             S.push(v+);
12             anchor=anchor ∪{v};
13             cand={u|u ∈ (cand-{v})∧ G(anchor ∪ {u}) is a
               k-plex};
14     if (G(anchor ∪ cand) is pruned) or (G(anchor ∪ cand) is a
       k-plex) then
15         while S.top() == v− do
16             S.pop();
17             not=not-{v};
18         if S.top() == v+ then
19             S.pop();
20             S.push(v−);
21             anchor=anchor-{v};
22             not=not∪{v};
23             cand={u|u ∈ (V − not − anchor) ∧ G(anchor ∪ {u})
               is a k-plex};
```

Alg. 4 only needs to maintain three vertex sets, anchor, cand and not, a stack recording traversal progress, as well as the adjacency lists of vertices. Also note that each line of statement in Alg. 4 can be performed in $O(|E|)$ time. Therefore, we have the following theorem:

Theorem 6. *Given a connected graph $G = (V, E)$, Alg. 4 has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\mu_k(G))$.*

3.3.3. Implementation notes

Even though Alg. 4 achieves the same time complexity as Alg. 3 with the optimal space complexity, it does not remember the G_v^- subgraphs while performing graph partitioning operations. Neither does it filter out the unnecessary vertices in the not sets while traversing along the G_v^+ branches. These properties make it substantially less efficient than Alg. 3 in practical implementation.

As in the implementation of maximal clique enumeration, we maintain the structure of degree map for efficient anchor selection. To facilitate efficient k-plex and pruning verification, for each vertex u in anchor, cand and not, we also maintain two degrees $deg_a(u)$ and $deg_c(u)$, in which $deg_a(u)$ denotes the number of vertices in anchor connected to u and $deg_c(u)$ denotes the number of vertices in cand connected to u. With these two degrees, k-plex and pruning verifications as specified in Alg. 3 can be efficiently processed. The details on how to efficiently maintain the values of $deg_a(u)$ and $deg_c(u)$ and use them for efficient k-plex and pruning verification can be found in our technical report [9].

4. Parallel solutions

In this section, we present the parallel algorithms and describe their corresponding MapReduce implementations.

4.1. Parallel algorithms

The algorithms essentially compute the maximal cliques or k-plexes of vertices in parallel. They perform the computation on every vertex by Algs. 2 and 3. Unfortunately, computational cost on individual vertices may be unbalanced. The computations on some vertices may be more expensive than on others because they have larger search traversal trees. In case that the computation on a vertex is too time-consuming, it becomes a parallel performance bottleneck. A good property of our proposed approach is that it can effectively transform an expensive computation at a vertex into many sufficiently small computations by only a few iterations. In practice, as demonstrated in our experiments in Section 5.2, it usually takes no more than 5 iterations to transform a big graph into many sufficiently small subgraphs. With sufficiently small tasks, effective load balancing can be achieved by sending some tasks on the computing nodes with heavy workload to others with lighter one.

In general, the parallel algorithm for maximal clique (or k-plex) detection consists of the phases of *subgraph retrieval* and *iterative computation*. In the first phase, for every vertex v in the graph G, the induced subgraph of G whose vertices are relevant to the computation of v's maximal cliques or k-plexes is retrieved. Subgraph retrieval should be distributed across multiple workers. The second phase is performed by iteratively invoking the *Compute-Shuffle* cycle. At the *Compute* step, each worker computes maximal cliques (k-plexes) of the graphs assigned to it; at the *Shuffle* step, all the unfinished graphs at the workers are reshuffled such that every worker receives roughly the same number of them. The workload limit of the *Compute* step can be quantified by the number of partitioning operations executed or CPU time consumed.

4.2. MapReduce implementation

In this subsection, we describe the MapReduce implementations of subgraph retrieval and iterative computation.

4.2.1. Subgraph retrieval

Given a connected graph $G = (V, E)$, let $d(u, v)$ denote the number of edges in the shortest path between two vertices u and v in G. The diameter of G, denoted by $diam(G)$, is defined as $diam(G) = max_{u,v \in V} d(u, v)$. A connected graph G is γ-quasi-complete if every vertex in G has a degree at least $\gamma \cdot (|V| - 1)$. According to [33], the relationship between the diameter of a γ-quasi-complete graph and γ can be established by Theorem 7.

Theorem 7. *Let G be a γ-quasi-complete graph such that $n = |V| \geq 1$. If $\frac{1}{2} \leq \gamma \leq \frac{n-2}{n-1}$, then $diam(G) \leq 2$.*

The proof of Theorem 7 can be found in [33]. According to Theorem 7, with low k values($k \leq (|V| - \lceil \frac{|V|-1}{2} \rceil)$), any maximal k-plex of v consists of v and its 2-hop neighbors. Therefore, subgraph retrieval for maximal clique and k-plex enumerations can be built on 2-hop retrieval.

Noting that non-trivial cliques consist of triangles, we propose to use the technique of triangle enumeration proposed in [14] to implement the process of subgraph retrieval for maximal clique enumeration. Compared with 2-hop retrieval, triangle enumeration usually generates less intermediate data, thus can achieve better performance. There also exists more recent work (e.g. [16]) that can optimize the process of subgraph retrieval. More discussions on this perspective are however beyond the scope of this paper because it focuses on the performance of enumerating maximal cliques and k-plexes.

As in [6], we represent each triangle with a vertex as its key and the other two vertices as its value. For instance, the triangle consisting of the vertices {1, 2, 3} is represented by the key–value pair of (1, {2, 3}). At end of subgraph retrieval, each vertex v has a corresponding triangle unit, which consists of all the key–value pairs with v as their keys.

Algorithm 5: Maximal Clique Computation in *Reducer*

```
   Input: A queue of unfinished subgraphs Q_c
1  while (Q_c is not empty) and (workload limit is not reached) do
2      Dequeue a subgraph G_u from Q_c;
3      while (G_u is not a clique) and (workload limit is not reached) do
4          Choose the vertex w with the minimal degree in G_u as the
           anchor;
5          Partition G_u into G_w^+ and G_w^-;
6          if |cand(G_w^+)| ≤ k then
7              Recursively partition G_w^+ using Alg. 2 to the end;
8          else
9              if G_w^+ can not be pruned then
10                 if G_w^+ is a clique then
11                     Output G_w^+;
12                 else
13                     Enqueue G_w^+ into Q_c;
14         G_u=G_w^-;
15     if G_u can not be pruned then
16         if G_u is a clique then
17             Output G_u;
18         else
19             Enqueue G_u into Q_c;
```

4.2.2. Iterative computation

We describe the program for iterative computation of maximal clique enumeration in this subsection. The program for maximal k-plex enumeration is the same except that it invokes Alg. 3 instead of Alg. 2 in the implementation.

The program consists of a series of Map-Reduce cycles. In the Map phase, the mappers read the unfinished subgraphs and randomly map them to reducers such that each reducer receives roughly the same number of subgraphs. In the Reduce phase, the reducers enumerate the maximal cliques of their assigned subgraphs by Alg. 2. The Map-Reduce cycle is iteratively invoked until no unfinished subgraph is left.

The algorithm of the computation at a reducer is sketched in Alg. 5. Maintaining the subgraphs by a queue Q_c, it iteratively dequeues a subgraph G_u from the queue for graph partitioning. If the resulting G_w^+ has a small size, which means that its maximal clique computation can be finished in short time, it is recursively partitioned to the end (Lines 6–7). Otherwise, it is temporarily enqueued into Q_c if it is not a clique (Line 13). It then iteratively partitions G_w^- in the same manner as G_u (Line 17). The operations of subgraph dequeue and graph partitioning are iteratively performed until the queue becomes empty or a predefined workload limit is reached.

For each vertex v in the original graph G, the program maintains the vertices and their adjacency lists in G_v by a hash table, which is constructed from v's triangle unit. It represents a subgraph in the queue by three vertex sets (`anchor`, `cand` and `not`). Note that partitioning an induced subgraph G_u of G_v requires the hash table of G_v. Therefore, between MapReduce cycles, besides the vertex sets of subgraphs, the program also transfers the hash tables of their corresponding G_v to the next cycle. However, at most one copy of the hash table of G_v is needed for each *reducer*.

5. Experimental evaluation

In this section, we empirically evaluate the performance of our new approach by a comparative study. In the centralized setting

Table 1
Sequential maximal clique enumeration on real datasets.

Dataset	$\|V\|$	$\|E\|$	Search Space (R)		Runtime (s)	
			GP	BK	GP	BK
Social Networks						
wikivote	7,115	100,762	2.11	2.40	2.83	2.36
epinions	75,879	405,740	2.27	2.66	13.39	13.27
Slashdot0902	82,168	504,230	1.87	3.02	7.56	4.50
Gowalla_edges	196,591	950,327	2.30	2.96	8.44	12.45
youtube	1,134,890	2,987,624	2.28	3.87	19.24	33.13
Pokec	1,632,803	22,301,964	2.10	3.38	157.98	114.41
WikiTalk	2,394,385	4,659,565	2.23	2.75	907.732	9258.45
Web Graphs						
uk2005	129,632	11,744,049	152.30	308.74	16.95	27.92
it2004	509,338	7,178,413	4.26	9.06	12.71	17.67
BerkStan	685,230	6,649,470	1.69	2.01	17.31	32.92
WebGoogle	875,713	4,322,051	2.41	3.98	15.41	19.15
WikiComm	1,928,669	3,494,674	2.34	3.48	48.71	403.75
wikipedia2009	1,864,433	4,507,315	2.00	5.30	20.65	16.28
Miscellaneous Networks						
HepPh	34,546	420,877	2.17	28.24	3.55	4.24
EuAll	265,009	364,481	2.12	6.71	1.77	2.43
dblp2012	317,080	1,049,866	2.84	6.10	3.75	3.05
skitter	1,696,415	11,095,298	1.99	3.15	705.9	1745.94

(Section 5.1), for maximal clique enumeration, we compare our approach with the state-of-the-art implementation of the BK algorithm [41]. For maximal k-plex enumeration, we compare our approach with a variant of the BK search algorithm proposed in [46] as well as a more recent algorithm proposed in [4]. In the parallel setting of Hadoop (Section 5.2), we compare our approach with the typical BK approach, which confines the computation on an individual vertex to a worker, as well as an improved BK approach enhanced with dynamic load balancing (denoted by BK-L) as proposed in [37]. The improved BK approach was originally implemented by MPI. We have instead implemented a similar MapReduce version. We have also implemented our parallel approach by MPI and compared it with the state-of-the-art MPI implementation of parallel maximal clique enumeration [37] (Section 5.3). Finally, we evaluate parallelizability of our approach in Section 5.4. All our implementations have been made open-source. They can be downloaded at [19].

Our experiments are conducted on both real and synthetic graph datasets. The evaluation on real datasets can show the efficiency of the proposed algorithms in real applications while the evaluation on synthetic datasets can clearly demonstrate their sensitivity to varying graph characteristics. The real datasets, which are from [34,35], represent the graphs in many application domains including social networks, Web graphs and Wiki communication networks. The synthetic datasets are generated by three popular generators, SSCA#2 [20], R-MAT [2] and BTER [39]. A SSCA#2 graph is made up of random-sized cliques, with a hierarchical inter-clique distribution of edges based on a distance metric. We vary the values of the *TotVertices* and *MaxCliqueSize* parameters, which specify the number of vertices and the size of the maximum clique respectively. The R-MAT generator applies the Recursive Matrix graph model to produce the graphs with power-law degree distributions and small-world characteristics, which are common in many real-life graphs. We vary two parameter values, the number of vertices and the number of edges. The collections of the BTER graphs are community-structured and scale-free. Its generator uses two parameters, *rho_init* and *rho_decay*, to control vertex degree distribution.

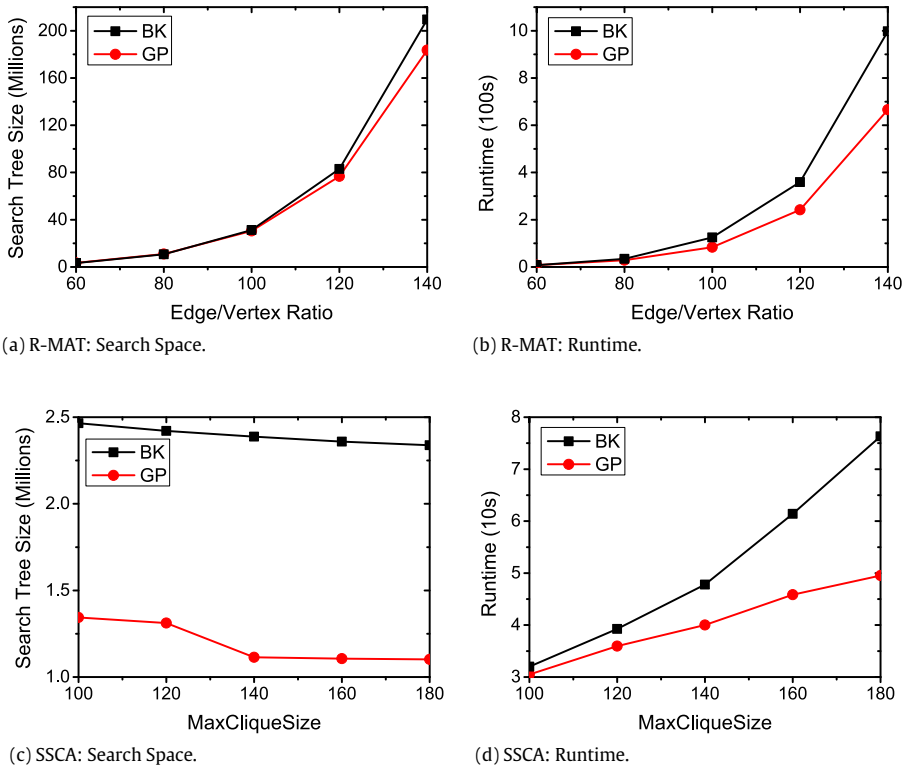
Centralized evaluation was conducted on a desktop with memory size of 16G and 6 Intel Core i7 CPU with the frequency of 3.3 GHz. Parallel evaluation was conducted on a 13-machine cluster. Each machine runs the Ubuntu Linux (version 10.04), has a memory size of 16G, disk storage of 160G and 16 Intel Xeon E5502 CPUs with the frequency of 1.87 GHz. The parallel approach based on MapReduce was implemented on Hadoop (version 0.20.2) [21]. Each experiment was run three times and its running time averaged. We observed that the time difference between runnings does not exceed 10% of the total consumed time.

5.1. Centralized evaluation

We evaluate the efficiency of different approaches on two metrics: search tree size and runtime. Both BK and GP are search algorithms. The metric of search tree size measures the number of unit operations performed by BK and GP. In the BK approach, search tree size corresponds to the total number of extracted subgraphs, whose set of vertices should be computed; in the GP approach, it corresponds to the total number of extracted G_v^+ subgraphs and G_v^- cliques. On search space, we report the ratio of search tree size to the total number of maximal cliques (k-plexes). The runtime includes both the cost of reading a graph into memory and search cost. Note that while the runtime of an algorithm may depend on its implementation details, search tree size accurately measures search space and is independent of algorithmic implementations. Besides efficiency, we also report memory requirement of the BK and GP approaches for maximal clique enumeration. The memory results for maximal k-plex enumeration are similar, thus omitted.

5.1.1. Maximal clique enumeration

The evaluation results on the real graphs are presented in Table 1. Note that on some of the real datasets, both BK and GP run very fast (less than 1 s). Their results are therefore not presented here. Some datasets will be used for parallel evaluation because maximal clique detection over them cannot be completed within reasonable time on a desktop. It can be observed that on search space, GP consistently outperforms BK by considerable margins. This observation is due to the fact that GP uses a more aggressive filtering strategy than BK. As shown in Alg. 1, BK iteratively selects a vertex with the largest degree in the current graph and partitions it into multiple subgraphs until the set `cand` becomes empty. In contrast, GP, as shown in Alg. 2, iteratively partitions a graph until it becomes a clique. Moreover, at Line 9 of Alg. 2, GP uses the set of `not` to filter out the subgraphs that cannot generate new maximal cliques. However, GP's filtering strategy comes with a cost because it has to check the filtering conditions as specified at



(a) R-MAT: Search Space. (b) R-MAT: Runtime.

(c) SSCA: Search Space. (d) SSCA: Runtime.

Fig. 2. Evaluation of sequential maximal clique enumeration on R-MAT and SSCA.

Table 2
Enumerating large maximal cliques (size $\geq$ 10) on real graphs.

Runtime (s)	Youtube	Pokec	WikiTalk	WikiComm	Skitter
GP	12.20	117.65	890.48	28.26	679.533
BK	16.45	89.85	8325.03	186.91	1579.92

Lines 9 and 12 in Alg. 2. Additionally, GP has to update $G(\texttt{cand})$ and transform it into $G(\texttt{cand}^{-})$. Therefore, GP usually takes more time per traversal than BK. It can be observed that on runtime, GP still achieves an overall better performance than BK but the improvement margins tend to become smaller. It is worthy to point out that the performance of BK is much more volatile than that of GP. For instance, on both WikiComm and WikiTalk, GP runs around 10 times faster than BK. In contrast, on the datasets (e.g. Pokec) where GP performs worse than BK, the performance of GP is competitive (its runtime is less than two times that of BK). These experiments demonstrate that the performance of BK can be quite sensitive to graph characteristics. In comparison, the performance of GP is more stable.

The experiments as reported in Table 1 enumerate the maximal cliques of any size. In some applications, users may only be interested in large maximal cliques, or the maximal cliques whose sizes exceed a user-specified threshold. Therefore, we also report the comparative results on enumerating large maximal cliques. The threshold of clique size is set to be 10. The detailed results on some real graphs in Table 1, which require relatively longer runtime, are presented in Table 2. It can be observed that setting size threshold can reduce runtime, but the relative performance of GP and BK remains the same as observed in Table 1.

We also evaluate their performance on synthetic graphs to investigate how their performance varies with different graph characteristics and densities. For R-MAT graphs, the number of vertices is set to be 10,000 and the edge-to-vertex ratio varies from 60 to 140. For SSCA graphs, the number of vertices is set to be 2^{20} and the size of the maximum clique varies from 100 to 180. The evaluation results on R-MAT and SSCA graphs are presented in Fig. 2. Similar to what was observed on real graphs, GP outperforms BK on both search space and runtime. It is interesting to observe that when the graphs are sparse, GP may perform worse than BK on runtime (but its performance is competitive). However, with increasing graph density, GP easily outperforms BK and its performance advantage steadily increases as well.

Their performance comparisons on the BTER datasets are also shown in Table 3. Note that unlike R-MAT and SSCA, the BTER generator does not allow users to directly specify graph density. We fixed the value of parameter *rho_decay* at the default 0.5 and varied the value of another parameter *rho_init* from 0.3 to 0.7. The generated graphs are similar in edge/vertex ratio (at around 10) and maximum clique size (between 6–15). It can be observed that BTER is not a challenging task for maximal clique detection because both BK and GP run fast. GP performs better than BK on search space. On runtime, GP however does not perform as well as BK but its performance is competitive.

Based on the experiments, we have the following observations: (1) GP achieves an overall better and much more stable performance than BK; (2) the performance advantage of GP over BK tends to increase with graph density. Since maximal clique detection on dense graphs is usually more computationally expensive than on sparse graphs, these observations bode well for GP's applications on real graphs.

Evaluation on memory requirement. We also compare memory requirement of the BK and GP approaches. The detailed results on some real graphs in Table 1, which have relatively larger sizes, are presented in Table 4. The metric of *Graph Memory* denotes the required memory for storing a graph. The metric of *Running Memory* instead denotes the maximal additional memory required by search process. It can be observed that GP requires significantly more memory than BK on *Graph Memory*. The difference results from the ways of a graph being stored in the BK and GP approaches.

Table 3
Sequential maximal clique detection on BTER datasets.

BTER datasets	$\|V\|$	$\|E\|$	Search Space (R)		Runtime (s)	
			GP	BK	GP	BK
cit-HepPh-0.3-0.5	34,256	450,975	1.838	4.739	1.647	1.286
cit-HepPh-0.5-0.5	34,295	443,769	2.157	3.144	2.603	2.019
cit-HepPh-0.7-0.5	34,395	436,592	2.472	2.664	10.817	7.821

Table 4
Memory requirement of sequential maximal clique enumeration.

Datasets	Graph Memory (MB)		Running Memory (MB)	
	GP	BK	GP	BK
Youtube	747.64	71.44	0.09	14.46
Pokec	3727.65	238.72	0.07	18.74
WikiTalk	1289.51	136.11	2.14	27.44
WikiComm	963.49	127.22	1.44	27.42
Skitter	1860.51	155.92	1.27	19.50

Table 5
Sequential k-plex detection on real datasets: GP vs BK.

Datasets	$\|V\|$	$\|E\|$	Search Space (R)		Runtime (s)	
			GP	BK	GP	BK
2-plex Evaluation						
edu	3,031	6,474	2.20	4.21	1.405	1.845
CA-GrQc	4,158	13,422	1.67	6.83	3.384	5.163
celegans	453	2,025	1.24	2.59	2.626	3.228
infecthyper	113	2,196	3.34	11.63	19.54	92.82
caida	26,475	53,381	1.29	2.65	320.19	1013.28
3-plex Evaluation						
edu			1.05	1.31	38.98	45.75
CA-GrQc			1.20	1.98	32.03	60.01
celegans			1.09	1.40	9.07	59.37
infecthyper			5.33	14.87	812.96	3935.32

BK represents the adjacency information in a graph by arrays. GP instead stores it by hash sets, which need considerably more memory than arrays in our C++ programming environment (Microsoft Visual Studio 2015). On the metric of *Running Memory*, both of them require significantly less memory compared with what is required for *Graph Memory*. It is interesting to point out that GP consumes considerably less running memory than BK. Efficient BK search requires recursive implementation. In contrast, our GP implementation only needs to simultaneously store the subgraphs corresponding to the nodes in a path from the root to a leaf in a search traversal tree. *It is important to point out that GP's disadvantage compared with BK on Graph Memory should not be a concern in practical implementation due to the fact that it can only increase linearly with graph size.*

5.1.2. Maximal k-plex enumeration

We evaluate the performance of the sequential GP algorithm for maximal k-plex enumeration with low values of k ($k = 2$ and 3). Maximal k-plex enumeration is much more costly than maximal clique enumeration. Therefore, we can only use small real datasets in centralized evaluation. The test synthetic R-MAT graphs also have smaller sizes. On 2-plex detection, the R-MAT graphs have 10,000 vertices and their edge-to-vertex ratios vary from 5 to 25. On 3-plex detection, the R-MAT graphs have 200 vertices and their edge-to-vertex ratios vary from 5 to 25. The evaluation results on the SSCA datasets were similar to what were observed on R-MAT, thus omitted here.

5.1.3. Comparison with BK

The evaluation results on the real graphs are presented in Table 5. Note that maximal 3-plex detection over the dataset of caida cannot be finished within reasonable time on desktop. Its result is therefore not presented here. The evaluation results on the R-MAT datasets for 2-plex and 3-plex are presented in Fig. 3. It can be observed that they are similar to what were reported on maximal clique enumeration except that GP outperforms BK by even larger margins. The maximal k-plexes of a graph have larger sizes than its maximal cliques. BK traversal on a maximal k-plex search tree is also less efficient than on a maximal clique search tree. As a result, the performance advantage of GP over BK becomes more considerable. On the R-MAT graphs, we observe that the performance gap between BK and GP increases with graph density. It is also worthy to point out that the performance advantage of GP over BK is more considerable on 3-plex computation than on 2-plex computation.

Table 6
Sequential 2-plex detection on real datasets: GP vs algorithm of [4].

Runtime (s)	edu	CA-GrQc	celegans	infecthyper	caida
GP	1.405	3.384	2.626	19.54	320.19
Algorithm of [4]	10.23	7.87	9.87	89.54	>1200

5.1.4. Comparison with algorithm of [4]

We compare our implementation with the code the authors of [4] kindly shared with us. The comparative results of maximal 2-plex detection on the real graphs are presented in Table 6. Note that the code we compared with is without optimization. According to the experimental results presented in [4], the performance difference measured by runtime between the codes w/o optimization is between 10% and 40%. It can be observed that GP outperforms by the margins considerably larger than 40%.

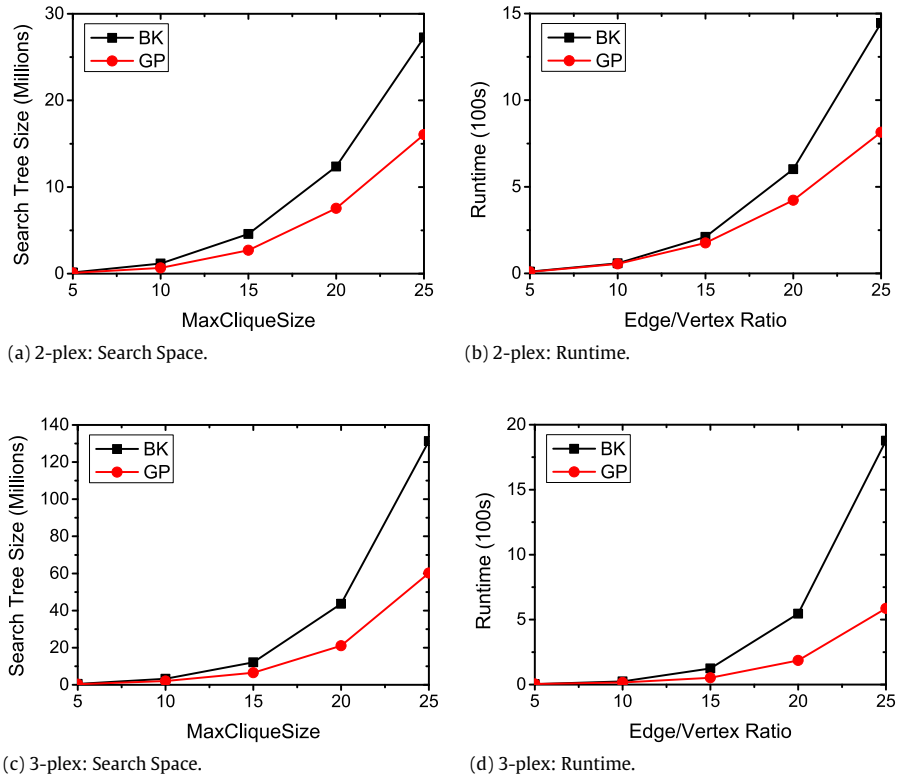
5.2. Parallel evaluation by Hadoop

In this subsection, we compare the GP approach against the BK and BK-L approaches in parallel setting. Since all the parallel approaches use the same method of subgraph retrieval, we exclude its cost from our comparative study. However, we do report the consumed time of subgraph retrieval and show that its cost is not substantial compared with the cost of iterative search. We use triangle enumeration and 2-hop retrieval to implement subgraph retrieval for maximal clique and k-plex enumerations respectively.

5.2.1. Maximal clique enumeration

For the GP approach, we specify the parameter k in Alg. 5 by the number of vertices contained by a graph. It is set to be 50. The maximal execution time per *Reduce* phase is set to be 300 s. The workload limit of execution time per *Reduce* phase is similarly set for the BK-L approach.

Note that the real graphs as shown in Table 1 can be efficiently processed on a single worker. Therefore, we used the Twitter dataset, whose details are shown in Table 8 for parallel clique detection. On R-MAT and SSCA graphs, even the BK approach manages to evenly distribute the workload across workers. The parallel evaluation results on them are similar to what were observed in sequential evaluation, thus omitted here. Also note that processing the entire graph of the Twitter graph is too time-consuming on our cluster. Therefore, we randomly choose some vertices with large degrees on them and enumerate the maximal cliques



(a) 2-plex: Search Space. (b) 2-plex: Runtime.

(c) 3-plex: Search Space. (d) 3-plex: Runtime.

Fig. 3. Evaluation of sequential maximal 2-plex and 3-plex enumeration on R-MAT.

Table 7
Runtime of subgraph retrieval for clique enumeration on Twitter.

	D_T^1	D_T^2	D_T^3	D_T^4	D_T^5
Runtime (s)	137	123	135	147	167

Table 8
Parallel maximal clique detection based on Hadoop on Twitter.

Twitter	$\|V\|$: 11,316,811			$\|E\|$: 85,331,846		
	Map-Reduce Cycles			Runtime (s)		
	BK	BK-L	GP	BK	BK-L	GP
D_T^1	1	2	1	989	465	55
D_T^2	1	3	2	2379	775	441
D_T^3	1	4	3	>7200	1136	768
D_T^4	1	4	3	>7200	1407	707
D_T^5	1	16	5	>7200	5455	711

containing them over the entire graphs. The computations on the vertices with large degrees can be supposed to be representative and challenging. With the maximal degree of vertices on Twitter at the scale of 10^7, we choose the vertices with the degrees of at least 10^5. Totally ten vertices are chosen for each task.

We first report the consumed time of subgraph retrieval in Table 7. Subgraph retrieval takes between 100(s) and 200(s) on all the test tasks. It can be observed that the retrieval cost is much less than the cost of iterative search, which takes time between 500(s) and 2000(s) as reported in Table 8.

Then, we present the comparative results on iterative search in Table 8. It can be observed that BK-L performs better than BK and GP outperforms both of them by considerable margins. Our experiments showed that the typical BK approach performs very poorly in many cases (e.g., D_T^4 and D_T^5). It cannot finish computation within the 2-hour runtime limit. Closer scrutiny reveals that there exist some vertices, whose maximal clique computations are extremely expensive, in these tasks. The computations on these vertices thus become parallel performance bottleneck if without dynamic load balancing. The experiments show that both the BK-L and GP approaches can effectively break the performance bottleneck by redistributing the computation on an individual vertex across multiple workers. However, it can be observed that the GP approach achieves overall better parallel performance than the BK-L approach. Compared with BK-L, GP usually generates much smaller traversal trees and is able to partition big graphs into sufficiently small subgraphs with less iterations.

Table 9
Runtime of subgraph retrieval for 2-plex enumeration on real graphs.

	EuAll	WebGoogle	BerkStan	WikiComm	Pokec
Runtime (s)	43	85	92	72	132

5.2.2. Maximal k-plex enumeration

We evaluate the performance of different approaches in the cases of $k = 2$. Note that maximal 2-plex enumeration is computationally more expensive than maximal clique enumeration and processing the entire graphs of most real datasets listed in Table 1 takes too long on our cluster. Therefore, on the datasets of EuAll and WebGoogle, we process the entire graphs. On BerkStan, WikiComm and Pokec, as in Section 5.2.1, we instead randomly select 10 vertices with large degrees in the graphs and compute their maximal 2-plexes over the entire graphs. As in Section 5.2.1, our implementations specify that any graph with no more than 50 vertices is recursively processed to the end without being redistributed. The maximal execution time per *Reduce* phase is set to be 300 s.

The consumed time of subgraph retrieval is reported in Table 9. We report the retrieval time on all the vertices in the test graphs. It can be observed that similar to the case of clique enumeration, the retrieval cost is much less than the cost of iterative search as reported in Table 10.

Table 10
Parallel 2-plex detection based on hadoop on real graphs.

Datasets	Map-Reduce Cycles			Runtime		
	BK	BK-L	GP	BK	BK-L	GP
EuAll	1	4	1	2512	1178	286
WebGoogle	1	>20	3	>7200	>7200	751
BerkStan[10]	1	5	2	>7200	1643	587
WikiComm[10]	1	12	4	>7200	4387	1365
Pokec[10]	1	6	4	4879	2133	1268

The comparative results on iterative search are presented in Table 10. It can be observed that without load balancing, the performance of the BK approach is volatile; it performs very poorly in many cases (e.g. WebGoogle). Between BK-L and GP, GP outperforms BK-L by considerable margins. GP traverses considerably smaller search space and is able to partition big graphs into sufficiently small subgraphs with less iterations. Compared to what were observed in parallel clique computation, the performance advantage of GP is more considerable. It achieves the speedups of over 10x on WebGoogle.

5.3. Parallel evaluation by MPI

We have also implemented the GP approach on MPI, and compared its performance with the state-of-the-art MPI implementation of parallel maximal clique detection based on BK search [37], which has been made open-source. The detailed results on real datasets are presented in Table 11, in which the speedup compares the parallel runtime with the runtime with only one worker. The parallel runtime reports the performance of GP and BK running with 48 workers.

It can be observed that as it is in centralized setting, the performance of BK search is very volatile, sensitive to graph characteristics. In comparison, GP performs better and much more stably. For instance, on the datasets of Orkut, Aanon and Banon, GP runs around 10 times faster than BK. On the first three datasets, GP achieves similar speedups to BK. On Aanon and Banon, GP runs much faster than BK and it can actually achieve the speedup of around 7 with only 10 workers. Our experiments showed that increasing the work scale from 10 to 48 can only marginally improve the parallel performance of GP. Our experiments demonstrated that both MPI implementations of GP and BK can achieve balanced workload among the workers.

We also compare the performance of GP and BK on enumerating large maximal cliques by MPI. The size threshold is set to be 10. The detailed results are presented in Table 12. It can be observed that the relative performance of GP and BK is very similar to what was observed in Table 11. GP outperforms BK by comfortable margins on most test cases. The exception is on the dataset of WikiTalk: GP takes more time than BK, but its performance is very competitive.

5.4. Parallelizability evaluation

To evaluate the parallelizability of our GP implementation on MPI, we run it on the worker clusters of different sizes and track its performance variation. We set up 5 cluster configurations which have 8, 16, 24, 32 and 40 workers respectively. The performance is measured in terms of runtime.

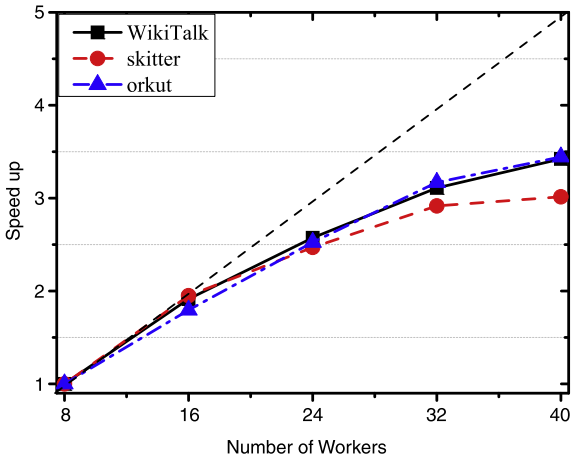


Fig. 4. Parallelizability evaluation on MapReduce.

The evaluation results of maximal clique enumeration on the three datasets (WikiTalk, skitter and orkut) are presented in Fig. 4. It can be observed that on all the tasks, increasing cluster size can effectively reduce runtime. We also observe that the achieved speedup increases with runtime. On the task with the longest runtime (orkut), GP achieves the best speedup: its performance on 40-worker cluster achieves a speedup close to 4 compared with the result on 8-worker cluster. Our experiments demonstrate that our GP implementation achieves desirable parallelizability on MPI.

6. Related work

Maximal clique detection. Maximal clique enumeration has been studied extensively in the literature [5,42,30,40,10,12,11]. Due to its NP-completeness, existing work focused on efficient search. Most of the proposed approaches were based on the classical BK algorithm [5], which has been widely reported as being faster than its alternatives [7,18]. Authors of [12] proposed an efficient algorithm, which was also based on BK search, for maximal clique enumeration with limited memory. Authors of [10,11] proposed to speed up clique detection by indexing the core structures of a special type of graph called H*-graph. Instead of BK, another approach [44,13,28] uses the strategy of *reverse search*. The key feature of this approach is that it is possible to define an upper bound on their runtime as a polynomial with respect to the number of maximal cliques in a graph. Note that focusing on centralized search, the efficient implementations of existing algorithms usually rely on global state and cannot be easily parallelized. There are also some work [36,35] studying the closely related problem of detecting maximum clique. The algorithms they used are however variants of the BK algorithm.

Maximal k-plex detection. The strictness of clique definition might limit its appeal in applications and has thus motivated the study of clique relaxations [3]. K-plex is a degree-based relaxation first proposed in [38]. Maximal k-plex detection is also a NP-completeness problem and existing work [29] focused on its combinatorial optimization. The efficient implementation for maximal

Table 11
Parallel maximal clique enumeration by MPI on real graphs.

Dataset	$\|V\|$	$\|E\|$	Runtime(s)		Speedup	
			GP	BK	GP	BK
WikiTalk	2,394,385	3,016,553	57.57	74.02	26.54	25.26
Skitter	1,696,415	10,489,784	56.66	413.96	22.15	25.32
Orkut	2,997,166	106,349,209	318.53	2673.12	22.60	28.63
Aanon	3,097,165	23,667,394	59.96	675.62	7.01	30.30
Banon	2,937,612	16,183,457	55.55	505.40	6.93	30.86

Table 12
Enumerating large maximal cliques (size $\geq$ 10) by MPI on real graphs.

Runtime (s)	WikiTalk	Skitter	Orkut	Aanon	Banon
GP	52.91	53.83	234.94	58.53	51.81
BK	33.65	292.92	688.61	174.17	355.65

k-plex detection in [46] used a variant of the BK algorithm. More recently, an alternative algorithm was proposed in [4]. It runs in polynomial delay for a constant k and incremental FPT delay when k is a parameter.

Parallel solutions. Due to the increasing popularity of the MapReduce framework, the solutions have been proposed to parallelize maximal clique [47,27,15] (k-plex [46]) detection on MapReduce. They proposed to distribute the vertices across workers and compute every vertex's maximal cliques (k-plexes) in parallel. On the core algorithm for efficient search, they however used the BK algorithm or its variants. As shown in our experimental study in Section 5.2, their parallel performance may be severely limited by the expensive computation on an individual vertex. Authors of [48] proposed a fault-tolerant parallel solution for maximum clique detection based on MapReduce. It also used the BK algorithm for efficient search. A parallel solution for maximal clique enumeration based on MPI has been proposed in [37]. It proposed a dynamic load balancing technique that enabled an idle worker to "steal" workload from another busy worker. As we showed in Section 5.3, limited by the efficiency of BK search, its performance was still quite sensitive to graph characteristics.

Detection of other dense graph structures. Orthogonal to ours, many works extended the definition of clique to dense subgraph structures other than k-plex (e.g. maximal cliques in an uncertain graph [51], cross-graph quasi-cliques [25], *k*-truss [24], and densest-subgraph [43]), and studied their applications. The existing algorithms for these problems are centralized. The search process of these dense structures is usually NP-complete, thus computationally expensive over massive real graphs. However, efficient parallelization of their search processes over a machine cluster remains an open question.

7. Conclusion and future work

In this paper, we have proposed a novel approach based on binary graph partitioning to address the problem of maximal clique and k-plex enumeration over graph data. Compared with previous approaches, it achieves smaller search space and is also inherently more parallelizable. Its better parallelizability enables more effective load balancing and ultimately results in more efficient parallel performance. Our extensive experiments have validated its efficacy.

Future work can be pursued in two directions. Firstly, many maximal cliques may be highly overlapping in real-world networks. There is a need to find diverse maximal cliques with few overlapping vertices on these graphs. Therefore, it is interesting to investigate whether the proposed approach can be extended to efficiently search for diverse maximal cliques over graph data. Secondly, as mentioned in Section 6, there are currently many proposals to define dense subgraphs by the structures other than k-plex. How to efficiently parallelize the search process of these dense subgraphs over large graphs is also an interesting challenge.

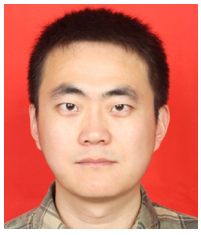
Acknowledgments

This work was supported in part by the National High Technology Research and Development Program 863 of China (2015AA015307), the Ministry of Science and Technology of China, National Key Research and Development Program (2016YFB 1000700), the Natural Science Foundation of China (Nos. 61502390, 61472321, 61332006, 61672432).

References

[1] E.A. Akkoyunlu, The enumeration of maximal cliques of large graphs, SIAM J. Comput. 2 (1) (1973) 1–6.
[2] D.A. Bader, K. Madduri, Gtgraph: A synthetic graph generator suite, 2006, pp. 1–4. http://www.cse.psu.edu/kxm85/software/GTgraph/.
[3] B. Balasundaram, S. Butenko, I.V. Hicks, Clique relaxations in social network analysis: The maximum k-plex problem, INFORMS 59 (1) (2011) 133–142.
[4] D. Berlowitz, S. Cohen, B. Kimelfeld, Efficient enumeration of maximal k-plexes, in: SIGMOD, 2015, pp. 431–444.
[5] C. Bron, J. Kerbosch, Algorithm 457: Finding all cliques of an undirected graph, Commun. ACM 16 (9) (1973) 575–577.
[6] F. Cazals, C. Karande, A note on the problem of reporting maximal cliques, Theoret. Comput. Sci. 407 (1–3) (2008) 564–568.
[7] F. Cazals, C. Karande, A note on the problem of reporting maximal cliques, Theoret. Comput. Sci. 407 (1) (2008) 564–568.
[8] Q. Chen, C. Fang, Z. Wang, B. Suo, Z. Li, Z.G. Ives, Parallelizing maximal clique enumeration over graph data, in: DASFAA, 2016, pp. 249–264.
[9] Q. Chen, W. Zuo, S. Bo, B. Hou, Z. Li, Z. G Ives, Parallelizing clique and k-plex detection over graph data, http://www.wowbigdata.net.cn/gp/clique.html.
[10] J. Cheng, Y. Ke, A.W.-C. Fu, J.X. Yu, L. Zhu, Finding maximal cliques in massive networks by h*-graph, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, 2010, pp. 447–458.
[11] J. Cheng, Y. Ke, A.W.-C. Fu, J.X. Yu, L. Zhu, Finding maximal cliques in massive networks, TODS 36 (4).
[12] J. Cheng, L. Zhu, Y.K.S. Chu, Fast algorithms for maximal clique enumeration with limited memory, in: KDD, 2012, pp. 1240–1248.
[13] N. Chiba, T. Nishizeki, Arboricity and subgraph listing algorithms, SIAM J. Comput. 14 (1) (1985) 210–223.
[14] J. Cohen, Graph twiddling in a mapreduce world, Comput. Sci. Eng. 11 (4) (2009) 29–41.
[15] N. Du, B. Wu, L.T. Xu, B. Wang, X. Pei, A parallel algorithm for enumerating all maximal cliques in complex network, in: ICDM Workshops, 2006, pp. 320–324.
[16] A. Epasto, S. Lattanzi, V. Mirrokni, I.O. Sebe, A. Taei, S. Verma, Ego-net community mining applied to friend suggestion, Proc. VLDB Endow. 9 (4) (2015) 324–335.
[17] D. Eppstein, M. Löffler, D. Strash, Listing all maximal cliques in sparse graphs in near-optimal time, in: ISAAC(1), 2010, pp. 403–414.
[18] D. Eppstein, D. Strash, Listing all maximal cliques in large sparse real-world graphs, in: 10th International Symposium on Experimental Algorithms, 2011, pp. 364–375.
[19] GP Project: Efficient maximal clique and k-plex detection over graph data, http://www.wowbigdata.cn/gp/clique.html.
[20] GTgraph: A suite of synthetic random graph generators, http://www.cse.psu.edu/~kxm85/software/GTgraph/.
[21] Hadoop: An open-source implementation of mapreduce, http://hadoop.apache.org/.
[22] R. Hanneman, Introduction to social network methods, 2005, Ch. 11: Cliques, pp. 1–18. http://www.faculty.ucr.edu/~hanneman/nettext/C11_Cliques.html.
[23] M. Haraguchi, Y. Okubo, A method for pinpoint clustering of web pages with pseudo-clique search, in: K. Jantke, A. Lunzer, N. Spyratos, Y. Tanaka (Eds.), Federation over the Web, in: Lecture Notes in Computer Science, vol. 3847, Springer, Berlin, Heidelberg, 2006, pp. 59–78.
[24] X. Huang, H. Cheng, L. Qin, W. Tian, J.X. Yu, Querying *k*-truss community in large and dynamic graphs, in: SIGMOD, 2014, pp. 1311–1322.
[25] D. Jiang, J. Pei, Mining frequent cross-graph quasi-cliques, ACM Trans. Knowl. Discov. Data 2 (4) (2009) 1–42.
[26] J. Leskovec, K.J. Lang, A. Dasgupta, M.W. Mahoney, Statistical properties of community structure in large social and information networks, in: WWW, 2008, pp. 695–704.
[27] L. Lu, Y. Gu, R. Grossman, dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution, in: IEEE International Conference on Data Mining Workshops, 2010, pp. 1320–1327.
[28] K. Makino, T. Uno, New algorithms for enumerating all maximal cliques, in: SWAT, in: Lecture Notes in Computer Science, vol. 3111, 2004, pp. 260–272.
[29] B. McClosky, I.V. Hicks, Combinatorial algorithms for the maximum k-plex problem, J. Combin. Optim. 23 (1) (2012) 29–49.
[30] N. Modani, K. Dey, Large maximal cliques enumeration in sparse graphs, in: CIKM, 2008, pp. 1377–1378.
[31] B.-W. On, E. Elmacioglu, D. Lee, J. Kang, J. Pei, Improving grouped-entity resolution using quasi-cliques, in: ICDM, 2006, pp. 1008–1015.
[32] G.A. Pavlopoulos, M. Secrier, C.N. Moschopoulos, T.G. Soldatos, S. Kossida, J. Aerts, R. Schneider, P.G. Bagos, Using graph theory to analyze biological networks, BioData Min. 4 (1) (2011) 10.
[33] J. Pei, D. Jiang, A. Zhang, On mining cross-graph quasi-cliques, in: KDD, 2005, pp. 228–238.
[34] Real graph datasets, http://snap.stanford.edu/data/.
[35] R.A. Rossi, D.F. Gleich, A.H. Gebremedhin, Parallel maximum clique algorithms with applications to network analysis, SIAM J. Sci. Comput. 37 (5) (2015) 589–616.
[36] R.A. Rossi, D.F. Gleich, A.H. Gebremedhin, M.M.A. Patwary, Fast maximum clique algorithms for large graphs, in: WWW, 2014, pp. 365–366.
[37] M.C. Schmidt, N.F. Samatova, K. Thomas, B.H. Park, A scalable, parallel algorithm for maximal clique enumeration, J. Parallel Distrib. Comput. 69 (4) (2009) 417–428.

[38] S.B. Seidman, B.L. Foster, A graph theoretic generalization of the clique concept, J. Math. Sociol. 6 (1) (1978) 139–154.
[39] C. Seshadhri, T.G. Kolda, A. Pinar, Community structure and scale-free collections of erdős–rényi graphs, Phys. Rev. E 85 (5) (2012) 056109.
[40] V. Stix, Finding all maximal cliques in dynamic graphs, Comput. Optim. Appl. 27 (2) (2004) 173–186.
[41] D. Strash, Quick cliques: Quickly compute all maximal cliques in sparse graphs, 2013. https://github.com/darrenstrash/quick-cliques.
[42] E. Tomita, A. Tanaka, H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theoret. Comput. Sci. 363 (1) (2006) 28–42.
[43] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, M. Tsiarli, Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees, in: KDD, 2013, pp. 104–112.
[44] S. Tsukiyama, M. Ide, I. Shirakawa, A new algorithm for generating all the maximal independent sets, SIAM J. Comput. 6 (3) (1977) 505–517.
[45] J. Wang, Z. Zeng, L. Zhou, Clan: An algorithm for mining closed cliques from large dense graph databases, in: ICDE, 2006, pp. 73–82.
[46] B. Wu, X. Pei, A parallel algorithm for enumerating all the maximal k-plexes, in: PAKDD, 2007, pp. 476–483.
[47] B. Wu, S. Yang, H. zhao, B. Wang, A distributed algorithm to enumerate all maximal cliques in mapreduce, in: International Conference on Frontier of Computer Science and Technology, 2009, pp. 45–51.
[48] J.G. Xiang, C. Guo, A. Aboulnaga, Scalable maximum clique computation using mapreduce, in: ICDE, 2013, pp. 74–85.
[49] S. Yang, B. Wang, H. zhao, B. Wu, Efficient dense structure mining using mapreduce, in: IEEE International Conference on Data Mining Workshops, 2009, pp. 332–337.
[50] Y. Zhang, F.N. Abu-Khzam, N.E. Baldwin, E.J. Chesler, M.A. Langston, N.F. Samatova, Genome-scale computational approaches to memory-intensive applications in systems biology, in: ACM/IEEE Supercomputing, 2005, pp. 12–12.
[51] Z.N. Zou, J.Z. Li, H. Gao, S. Zhang, Finding top-k maximal cliques in an uncertain graph, in: ICDE, 2010, pp. 649–652.



Zhuo Wang is a Ph.D. student in NWPU, China, his research interest is big data management.

Qun Chen is a professor in NWPU, China, his research interests include Big Data and Data Quality.

Boyi Hou is a Ph.D. student in NWPU, China, his research interest is big data management.

Bo Suo is a Ph.D. student in NWPU, China, his research interest is big data management.

Zhanhuai Li is a professor in NWPU, China, his research interests include Big Data and Data Quality.

Wei Pan is a professor in NWPU, China, his research interests include Big Data and In-memory Database.

Zachary G. Ives is a Professor at the University of Pennsylvania. His research interests include data integration and big data management.