A Parallel Algorithm for Enumerating All the Maximal *k*-Plexes*

Bin Wu and Xin Pei

Beijing Key Laboratory of Intelligent Telecommunications Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing 100876, China
wubin@bupt.edu.cn, peixin11@gmail.com

Abstract. Finding and enumerating subgraphs of different structures in a graph or a network is one of the fundamental problems in combinatorics. One of the earliest subgraph models is clique. However, the clique approach has been criticized for its overly restrictive nature. *k*-plex is one of the models which are introduced by weakening the requirement of clique. The problem to enumerate all the maximal *k*-plexes is NP complete. We consider this problem and propose an algorithm *Pemp* (Parallel Enumeration of all Maximal *k*-Plexes) for enumerating all the maximal *k*-plexes. We also propose a strategy to accelerate the pruning. A diameter pruning strategy is proposed. This strategy reduces the number of small maximal *k*-plexes and improves the performance greatly. We also state the parallel edition of our algorithm to analysis large networks and a load balancing strategy is given. In addition, we evaluate the performance of *Pemp* on random graphs.

1 Introduction

Graph-based data mining (GDM) becomes a hot-spot research of KDD (Knowledge Discovery in Databases). Finding and enumerating subgraphs of different structures in a graph or a network is one of the fundamental problems in combinatorics, which has many important applications in GDM.

One kind of structures is termed community or cohesive subgroups. Given a graph, a community is a subgraph, whose nodes are tightly connected, i.e. cohesive. Members of a cohesive subgroup tend to share information, have homogeneity of thought, identity, behavior, even food habits. Modeling a cohesive subgroup mathematically has long been a subject of interest in social network analysis. The earliest graph models used for studying cohesive subgroups was the clique model. However, the clique approach has been criticized for its overly restrictive nature [5]. In real world, there is a lot of noise. Thus a strict definition of clique is not very helpful for real applications. Researchers extend the definition of clique by weakening the requirements of different aspects. The models *k-clique* and *k-clan* are defined by weakening the requirement of reachability (the diameter of clique is 1), and the

* This work is supported by the National Science Foundation of China under grant number 60402011and the co-sponsored project of Beijing Committee of Education SYS100130422.

models *k-core*, *quasi-clique* and *k-plex* are defined by weakening the requirement of closeness(all pairs of vertices are connected).

k-plex is introduced in [1] and *k*-plexes with low *k* values ($k = 2, 3$) provide good relaxations of clique that closely resemble the cohesive subgroups that can be found in real life social networks. There are few works about finding all the maximal *k*-plexes in a graph as we have known.

The main contribution of this paper is to present a parallel algorithm for enumerating all the maximal *k*-plexes in graphs and describe the strategy to reduce the number of small maximal *k*-plexes. The rest of this paper is organized as follows. Basic definitions, notations and related works are presented in Section 2. Section 3 presents our algorithm, the optimization methods and the parallel edition of *Pemp*. Finally the paper is concluded with a summary and directions for future work in Section 5.

2 Definitions, Notations and Related Works

Let $G=(V,E)$ be a simple undirected graph with vertices $V=\{v_1,v_2,...v_n\}$ and edges $E=\{e_1,e_2,...e_n\}$. $|G|$ denotes the number of vertices in G; $d_G(u,v)$ denotes the length of the shortest path between vertices *u*, *v*; $\deg_G(v)$ denotes the degree of vertex *v* in graph *G* and $diam(G)=\max_{u,v\in V} d_G(u,v)$ denotes the diameter of *G*. Denote by $G[S]=(S,E\cap S\times S)$, the subgraph induced by *S*.

A subset of vertices $S\subseteq V$ is a clique if all pairs of its vertices are adjacent. A clique is maximal if it is not a subset of another clique. The problem to enumerate all the maximal cliques is termed “Maximal Clique Problem”. The base algorithm about Maximal Clique Problem called BK [4] was first published in 1973.

The formal definition of *k*-plex is stated as follows:

Definition 1. A subset of vertices $S\subseteq V$ is a *k*-plex if $\deg_{G[s]}(v)\geq|S|-k$ for every vertex *v* in *S*.

A *k*-plex is a maximal *k*-plex if it is not a subgraph of another *k*-plex and it is maximum if it is the largest maximal *k*-plex. “Maximal *k*-plex Problem” is the generalization of “Maximal Clique Problem” because a 1-plex is a clique. So it is more difficult for its weak restriction. [2] proves that the decision edition of Maximum *k*-plex Problem (finding the maximum *k*-plex) is NP complete. Because the Maximal *k*-plex Problem is the general edition of Maximal Clique Problem, we use the basic framework of BK algorithm to find all the maximal *k*-plexes. However, *Pemp* is more complex than BK because of the weak requirement of *k*-plex.

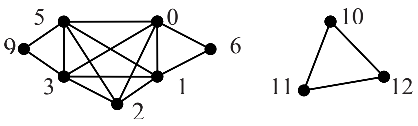

Fig. 1.

We should point out the fact that Definition 1 dose not emphasize the requirement of the connectivity. The triangles {10, 11, 12} and {0, 1, 6} in Fig.1 can also form a *k*-plex ($k \geq 4$). This kind of disconnected *k*-plex is trivial. The number of trivial *k*-plex is tremendous while it is useless to analyze these *k*-plexes. So the object of *Pemp* is just to find all the non-trivial *k*-plexes (hereinafter called *k*-plexes unless noted).

3 Algorithm Pemp

3.1 Basic Idea and Method

We use depth first search method to find all the maximal *k*-plexes. [1] states a property of *k*-plex as follows:

Theorem 1. Any vertex-induced subgraph of a *k*-plex is a *k*-plex.

According to Theorem 1, we can expand a *k*-plex until we get a maximal *k*-plex. If a vertex *v* is adjacent to at least one vertex in a *k*-plex *S*, we say that v is connected to *S* or *v* and *S* is connected.

Definition 2. Assume that a vertex *v* in graph *G* is connected to a *k*-plex *S* of *G*. *v* can expands S if the vertex-induced subgraph of *G*, $S \bigcup \{v\}$, is a *k*-plex.

Theorem 2. A vertex v can expand a *k*-plex S if v satisfies the two conditions as follows:

1. v is adjacent to at least $|S|-k+1$ vertices in S
2. Any vertex u in S is adjacent to at least $|S|-k+1$ vertices in $S \cup \{u\}$

Assume that v is associated with a counter, counting the number of vertices in S that v is not connected to. v can satisfy condition 1 if the counter of v is not greater than $k-1$. A definition of critical vertex is introduced for condition 2.

Definition 3. A vertex *u* in a *k*-plex *S* is a critical vertex if *u* is not connected to $k-1$ vertices in $\{S-u\}$.

To satisfy condition 2 the vertex to expand *S* should be selected in the neighborhood of critical vertex if there is a critical vertex in *S*. Otherwise there will be k vertices which the critical vertex *u* disconnected to in $S \cup \{u\}$. The candidate vertex used to expand S should be chosen in the intersection of neighborhoods of all the critical vertices if there are *n* critical vertices.

Like BK algorithm, *Pemp* maintains three vertices sets: *compsub*, *candidate* and *not*. The vertices in *compsub* are found *k*-plex. The vertices in *candidate* are used to expand the *compsub* set. The vertices in *not* have been used in previous search and can not be used to expand *compsub* set because all the extensions of *compsub* containing any vertex in *not* have already been generated. The vertices set of *compsub* should always be a *k*-plex and the vertices in *candidate* and *not* should always be able to expand *compsub*. There are another two vertices sets are generated from previous three sets. *connected_candidate* (*connected_not*) is the subset of *candidate* (*not*) and it contains all the vertices connected to *compsub* in *candidate* (*not*). Every vertex in every set is associated to a counter. The counter of vertex *u* in *compsub* counts the number of points that *u* is disconnected to in $S-\{u\}$. The counter of vertex *u* in *candidate* and *not* counts the number of disconnected *compsub* vertices of *u*.

```
Input: A Graph
Output: All the Maximal k-plex
Method:
1: Read the Graph and Copy all the vertices to candidate set. Let compsub and not be empty;
2: while there are vertices in candidate:
3:       Call FindAllMaxKplex (compsub,candidate,not);
4:       Move the used vertex to not upon the return;
5: endwhile;
Function: FindAllMaxKplex (inputCompsub,nputCandidate, inputNot)
6: Copy inputCompsub to compsub. Copy inputCandidate to candidate. Copy inputNot to not;
7: Select a vertex v in connected_candidate of inputCandidate in lexicographic order;
8: Move v to compsub and Update the counters of the vertices in compsub;
9: if there are n critical vertices in compsub (n > 0):
10:      Compute the intersection C of the neighborhoods of the n critical vertices and Remove all the vertices in candidate and not which are not in C;
11: Update the counters of the vertices in candidate and not and Remove the vertices of candidate and not if the vertices can not expand compsub (the counter is greater than k-1);
12: Generate connected_candidate of candidate and connected_not of not;
13: if connected_candidate and connected_not are empty:
14:      compsub is a maximal k-plex, Return v;
15: if connected_candidate is empty and connected_not is not empty:
16:      Return that there are no vertices which can expand compsub;
17: if connected_candidate and connected_not is not empty:
18:      while there are vertices in candidate:
19:        Call FindAllMaxKplex(compsub,candidate,not);
20:        Move the used vertex to not upon the return;
21:      endwhile;
22: Return v;
```

Fig. 2. Algorithm *Pemp*: the basic case

The found k-plex (*compsub*) becomes larger by moving the vertices which can expand the found k-plex from *candidate* to *compsub* until *compsub* is a maximal k-plex. It is easy to find out that the combination of any m vertices in candidate is a k-plex when $m \le k$. There will be great cost and many trivial k-plexes will be found. Taking the Fig.1 as an example, there is no vertex which can expand compsub if comps*ub* is {6, 9} by Theorem 2. So vertex 6 and 9 form a trivial maximal k-plex without analysis significance. That is why we use the *connected_candidate* set. In order to prevent trivial k-plex we only use the vertices connected to found k-plex to expand *compsub*. Copy the *candidate* vertices which are connected to *compsub* into *connected_candidate*. Then we select vertices from *connected_candidate* to expand *compsub*. This method does not only prevent the algorithm from finding trivial k-plexes but also improve its performance because it will not traverse all the combinations of any m ($m \le k$) vertices in *candidate*. The basic algorithm is summarized in Fig. 2.

A necessary condition for having created a maximal k-plex is that the *connected_candidate* is empty. This condition, however, is not sufficient, because if now *not* is not empty, we know from the definition of *not* that the present configuration of *compsub* has already been contained in another configuration and is therefore not maximal. We may now state that the *compsub* is maximal k-plex if and only if *connected_candidate* and *connected_not* is empty.

3.2 Acceleration of Pruning

The pruning condition is that *connected_candidate* is empty while *connected_not* is not empty. We can bring forward the pruning if we can predict that the *connected_not* will not be empty finally at an early stage. If a vertex v in *connected_not* is connected to all the vertices in *compsub* and *candidate*, *connected_not* will not be empty. So we can prune as soon as we find such a vertex. For this object another counter is added to the vertex in *not*. The counter defined in section 3.1 is termed counter1 and the new counter is called counter2, which counts the number of vertices in both *candidate* and *compsub* that the *not* vertex disconnected to. The pruning condition is satisfied when the counter2 equals 0.

Pruning can be accelerated if we can make the counter2 of vertices in *not* decrease faster. We introduce a vertices set termed *prunable_not*. The entire *not* vertices which connected to all the vertices in *compsub* (Because the counter2 of vertex will not be 0 if the vertex is not adjacent to all the vertices in *compsub*.) are copied to *prunable_not*. When expanding the *compsub* we will not select the next *candidate* vertex in lexicographic order but the vertex which is disconnected to the vertex with minimum counter2 in *prunable_not* set. Note that no counter2 is ever decreased by more than one when a vertex is moved to the *compsub*. Let v be the vertex which has minimum counter2 in *prunable_not*. If we keep selecting *candidate* vertices disconnected to v , the counter2 of v will be decreased by one at every repetition. Obviously no other *not* vertices's counter2 can go down more rapidly than v .

It should be pointed out that the method to accelerate the pruning can not be used during the whole process of the algorithm. Before there are critical vertices in *compsub*, *connected_candidate* may not contain the vertex which is disconnected to the vertex in *prunable_not* which has minimum counter2. So the vertex can only be selected in lexicographic order during this period. Additionally, this method can be used when *compsub* is empty. In conclusion, we can replace line 7 in Fig. 2 by calling the function in Fig. 3 to select a better vertex.

```
Function: SelectCandiate (compsub, candidate, not)
1: if compsub is not empty and there are no critical vertices in compsub:
2:       Return next vertex in connected_candidate of candidate in lexicographic order
3: endif;
4: else:
5: Copy all the vertices in connected_not of not, which are adjacent to all the vertices in compsub, to prunable_not set;
6:       if prunable_not is not empty:
7:         if the minimum counter2 of the vertices in prunable_not is 0:
8:           Return that there is no vertex can expand compsub;
9:         endif;
10:        else:
11:          Return the connected_candidate vertex which is not connected to the vertex in punable_not which has minimum counter2;
12:        endelse;
13:      endif;
14       else:
15:        Return next vertex in connected_candidate of candidate in lexicographic order;
16:      endelse;
17: endelse;
```

Fig. 3. The function to find a vertex to expand *compsub*

Although the method can bring forward the pruning, the overhead increases because of the extra counter. The additional cost may even countervail the effect of this method, so further optimization is necessary. We can consider the process of the algorithm as traversing the backtracking trees whose nodes correspond to the vertex which is selected to expand the found *k*-plex. Let d_1 be the degree of the root of the first tree to be traversed. We know that the algorithm needs to traverse $|G|-d_1+1$ trees if we want to find all the maximal *k*-plexes. Obviously the tree rooted by the vertex which has maximum degree should be traversed first.

3.3 Diameter Pruning

Clearly it is not all the vertices in the graph can form a non-trivial *k*-plex with a certain vertex *v* but only the vertices within finite hops of *v*. The size of *k*-plex has something with its diameter. We find the general relationship between the size of *k*-plex and its diameter as follows.

Theorem 3. Let *G* be a *k*-plex. If $|G|>2k-d$, then $diam(G) \le d$.

We can get the follow corollary from Theorem 3.

Corollary 1. The maximal *k*-plex, whose size is greater than m, and which contains vertex v can be found within $2k-m$ hops of v.

We only need to find all the non-trivial maximal *k*-plexes whose sizes are larger than k if we want to find all the maximal *k*-plexes because the maximal *k*-plexes whose sizes are smaller than k will be found inevitably by *Pemp*. By Corollary 1, we can find all the maximal *k*-plexes which contain vertex v within k hops of v and the vertices in *candidate* and *not* which are not in k hops of v can be removed. The next *candidate* vertex should be within k hops of both u and v when *compsub* is $\{u, v\}$. In our experiment, this method is only used when the size of *compsub* is 1. And we can also only find the maximal *k*-plexes with sufficient size by Corollary 1.

3.4 Parallel Model of *Pemp*

The algorithm traverses all the trees whose roots are the vertices of graph G to find all the maximal *k*-plexes. Every vertex will be moved to the *not* set after the backtracking of the tree rooted by it. The backtracking of every tree only depends on the *candidate* and *not*. If the order in which the vertices are selected is $\{v_1, v_2 ... v_n\}$ when *compsub* is empty (We can know the order before the algorithm begins.), the traverse of the tree rooted by the vertex v_m begins with *not* set $\{v_1, v_2, \ldots, v_{m-1}\}$ and *candidate* set $\{v_m, v_{m+1}, \ldots, v_n\}$. It is unnecessary to begin after the first *m*-1 trees have been traversed. Thus, we can traverse the trees collaterally. Practically, the number of processing elements of the parallel platform is often much smaller than the number of vertices in graph G, so mapping techniques are required.

Suppose that the order in which vertices are selected is $\{v_1, v_2 ... v_n\}$ when *compsub* is empty. A simple mapping scheme is to assign the n tasks to P processing elements sequentially. However, this scheme suffers from load unbalancing. Obviously the size of the tree rooted by v_1 is larger than v_n because the vertices $v_1, v_2, ..., v_{n-1}$ have been moved to *not* when the tree rooted by v_n are traversed. Let $I_p \in \{1, 2, \ldots, P\}$ denotes the index of processing elements. We can solve this problem by defining the tasks set on a single processing element I_p as follows:

$$\left\{ v_i \middle| i = I_p + j \times P \vee i = n - \left(I_p + j \times P\right), I_p + P \times j \le \frac{n+1}{2} j = 0, 1, \ldots, \left\lfloor \frac{n}{P} \right\rfloor \right\} \tag{1}$$

4 Experiment Result

To evaluate the performance of *Pemp*, we implement it in C++ and run it on random graphs. Our experiments are performed on a Cluster (84 3.2GHz processors with 2Gbytes memory on each node, Linux AS3). The number of vertices and edges of the random graphs and the runtime (in seconds) on 1 CPU (Sequential) and on 20 CPUs (Parallel) when *k*=1, 2, 3, 4 are shown in Table 1. Many other results of experiments, witch show the strategy of acceleration of Pruning and diameter pruning are very effective, can not be covered because of space constraints.

Table 1. Runtime on 1 CPU (Sequential) and on 20 CPUs (Parallel) when *k*=1, 2, 3, 4

|V|	|E|	Sequential (1 CPU)				Parallel (20 CPUs)			
		k=1	k=2	k=3	k=4	k=1	k=2	k=3	k=4
1000	14432	3.60	26.7	595	12158	0.43	2.34	47.2	977
2000	28709	10.5	70.6	1416	28601	3.15	8.18	105	2000
4000	58063	60.2	307	4896	89099	22.7	45.1	418	6568
8000	116276	436	1860	23804	-	161	269	1944	29810
16000	231622	3340	13550	-	-	1214	1976	13101	186938

5 Conclusion and Future Work

By allowing some strangers or noise in a cohesive subgroup, *k*-plex provides a more realistic alternative to model cohesive subgroups in a real network. So we propose *Pemp* algorithm and we hope most of the applications which use maximal clique can obtain better performance if clique is superseded by *k*-plex. We also propose the optimization for the pruning and the size of maximal *k*-plex and the parallel model of *Pemp* to analysis large networks. The future work mainly focuses on improving the performance. We are currently studying the strategy of pruning and load balancing to improve *Pemp*.

References

[1] Seidman, S.B., Foster, B.L.: A graph theoretic generalization of the clique concept. Journal of Mathematical Sociology 6, 139–154 (1978)

[2] Balasundaram, B., Butenko, S., Hicks, I.V., Sachdeva, S.: Clique Relaxations in Social Network Analysis: The Maximum k-plex Problem (2006), http://ie.tamu.edu/people/faculty/Hicks

[3] Hanneman, R.: Introduction to Social Network, Methods (1998), http://faculty.ucr.edu/~hanneman/networks/nettext.pdf

[4] Bron, C., Kerbosch, J.: Algorithm 457, Finding all cliques of an undirected graph. Proceedings of the ACM 16(9), 575–577 (1973)

[5] Alba, R.: A graph-theoretic definition of a sociometric clique. Journal of Mathematical Sociology 3, 113–126 (1973)