Sorbonne Université

Projet STL

Compression de très grands graphes de terrain

Rapport

Auteurs:
Hichem Rami AIT EL HARA
Adel EL AMRAOUI
Anas YAHYAOUI

Superviseur:
Dr. Maximilien DANISCH

Abstract

Automatic detection of relevant groups of nodes in large real-world graphs, i.e. community detection, has applications in many fields and has received a lot of attention these last twenty years. The most popular method designed to find overlapping communities (where a node can belong to several communities) is perhaps the clique percolation method (CPM). This method formalizes the notion of community as a maximal union of k-cliques that can be reached from each other through a series of adjacent k-cliques, where two cliques are adjacent iff they overlap on $k-1$ nodes. Despite much effort CPM has not been scalable to large graphs for medium values of k.

Recent work has shown that it is possible to efficiently list all k-cliques in very large real-world graphs for medium values of k. We build on top of this work and significantly scale up CPM. In particular, our algorithm is several orders of magnitude faster than state-of-the-art methods and we show that we can compute all 8-clique percolated communities in real-world graphs containing billions of edges within an hour of computation on a commodity machine. Our algorithm needs a single pass over the k-cliques, without the need of storing them.

Contents

1 Introduction

...

Our contribution is twofold.

1. We suggest an algorithm to significantly scale up CPM (which has been an extensively studied method) as shown by our extensive experimental evaluation. We also give a detailed theoretical analysis of our algorithm showing that our algorithm has a good running time when the input graph is sparse.

2. We carry a case study using CPM with unprecedentedly considered sizes of k-cliques in large graphs...

The rest of the paper is organized as follows. In Section 3, we present the related work. In Section 2 we explain the definitions and notations used throughout the paper. In Section **??**, we present our algorithms and prove its theoretical guarantees in Section **??**. We then evaluate the performance of our algorithm against the state-of-the-art in Section 7. Finally we carry a case study in Section 8.

2 Definitions and Notations

2.1 Useful definitions

Two k-cliques are *neighbours* if they share $k-1$ nodes.

For a given graph G, and a k-clique c_k, let $\mathcal{N}(c_k)$ be the the of k-cliques neighbours of c_k, that is to say the set of k-cliques of G which share $k-1$ nodes with c_k.

A *path* between two k-cliques c_0 and c_n is a set of k-cliques $c_1, ..., c_{n-1}$ such that for $1 \leq i \leq n$, c_i and c_{i-1} are neighbours, that is they share $k-1$ nodes.

A k-clique community is a maximal set of k-cliques such that there is a path between each pair of k-clique.

In an recursive point of view, we can define a k-clique community $\mathcal{C}$ as : $\mathcal{C} = \bigcup_{\substack{k-clique \\ c_k \in \mathcal{C}}} \mathcal{N}(c_k)$.

It gives a naïve way to build a community : starting from a k-clique and extending to its neighbours until the community stops growing.

2.2 The clique percolation method (CPM) problem

The problem in which we worked during this internship is the *k-Clique Percolation Method (CPM)* problem. This problem outputs a way to group the nodes which are "close" to each other, in a way we define just below.

The goal of this work is to partition the k-cliques of a given graph into communities. The input is a graph G (a list of edges), and the output is a list of set of nodes, each one corresponding to a k-clique community of the graph.

The notion of k-clique communities allows to group nodes into overlapping sets which be analysed to show there different characteristics. Figure 1 and Figure 2 are from Palla [REF] and Kumpula [REF] papers, and they show illustrations of graph k-clique communities and its interesting overlapping.

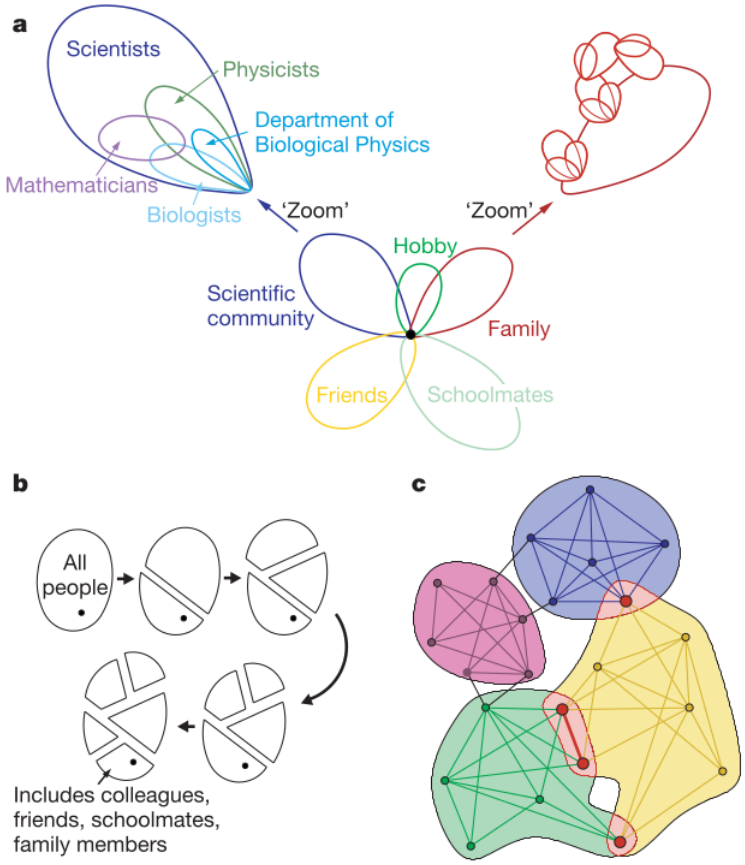

Figure 1: Illustration of the concept of overlapping communities, from Palla's paper [REF].

We generalize the term of *k-clique community*. In this report, we talk about community without necesarily having the maximality property. Each set of k-clique in which there is a path between any k-clique to any other, with the definition given in section 2.2, is called a community. Because in our algorithms communities are build little by little, it allows us to name communities in the algorithm, before they are maximal at the end of the algorithm.

3 Related Work

Existing algorithms to compute all the k-clique communities in an input graph can be split in two categories.

1. Algorithms that compute all maximal cliques of size k or more and then compute all k-clique communities out of them. Indeed, two maximal cliques that overlap on $k-1$ nodes or more belong to the same k-clique communities. Most state-of-the-art approaches ([8], [9] and [5]) fall in this category.

2. Algorithms that compute all k-cliques and then compute k-clique communities out of them strictly following the definitions of a k-clique community. [6] is the only method falling in this category.

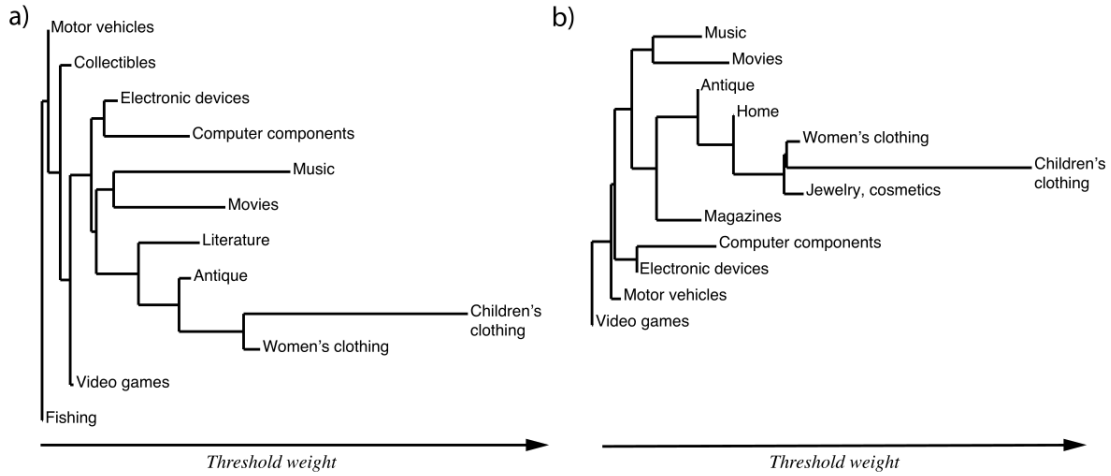

Figure 2: Dendogram visualization of k-clique communities to show the overlapping interest. Dendograms are from Kumpula paper [REF].

The differences between the algorithms falling in category (1) are in the method used to find which maximal cliques are adjacent (overlap on $(k-1)$ nodes or more). For instance, in [9] a bloom filter is used in order to reduce the number of clique overlaps computation, while in [5] massive parallelization is used. But the first step: "computing all maximal cliques" is always the same and is done sequentially even in [5]. While this problem is NP-Hard, algorithms scalable to relatively large sparse real-world graphs exist [3, 4]. They are based on the Brown-Kerbosch algorithm [1].

In Section 7, when comparing the efficiency in practice of our approach to the state-of-the-art, we simply compare the running time of our algorithm to the running time of the most efficient method to list all maximal cliques [4]. Note that this time is a lower-bound to the running time of any approach falling in category (1). While this gives a clear disadvantage to our approach (especially as the time bottleneck of the approaches in category (1) is actually the second step) we will see that our approach is still much faster in most settings.

Our approach falls in category (2), except that we do not store the k-cliques (or the $(k-1)$-cliques contrarily to what is done in [6]), but carry computations on the fly.

The algorithm detailed in [6], only algorithm in category (2), proceeds as follows. It starts with the empty graph and then insert edges of the input graph in an arbitrary order. It keeps track of the k-cliques formed when adding an edge (u, v), these newly formed k-cliques correspond to $(k-2)$-cliques in the subgraph (of the current graph) induced by $\Delta(u) \cap \Delta(v)$. Note that these $(k-2)$-cliques are simply nodes or edges when $k = 3$ or $k = 4$ respectively. The algorithm then tries to merge the found k-cliques into k-clique communities using a Union-Find datastructure fill with all $(k-1)$-cliques contained in a found k-clique. Note that this second step is similar to the algorithm we describe in Algorithm 3 which is a preliminary version of our final algorithm described in Algorithm 4.

As we will see, both our algorithms are orders of magnitude faster than this approach. This is due to the fact that (i) we use the full graph for listing k-cliques (using an efficient algorithm for that first step), rather than inserting iteratively all edges and updating the k-cliques and (ii) use a more efficient union-find data structure, especially in the case of

Algorithm 4 where we do not explicitly store the $(k-1)$-cliques.

Note that if the input graph has a large clique, say a clique of size 1000, and that we are interested in 10-clique communities then the algorithm in category (1) seems more interesting. Indeed, there are at least $\binom{1000}{10} > 10^{23}$ 10-cliques in the input graph which is prohibitively large. Furthermore, this 1000-clique is actually included in a single 10-clique community and can be found directly and efficiently using a maximal clique listing algorithm. This is the main reason why there are more methods following the approach of listing maximal cliques (category (1)) rather than listing k-cliques (category (2)). However, this argument does not hold for smaller values of k (e.g. listing all 3, 4 or 5-cliques seems more tractable than listing all maximal cliques). Furthermore it has been found that most real-world graphs actually do not contain very large cliques and that listing k-cliques for medium values of k is a scalable problem in practice []. This makes algorithms in category (2) more interesting for practical scenarios contrarily to what was previously thought.

There are many more algorithms for computing overlapping communities as shown in the dedicated survey [10]. The main focus of our paper is on the computation of the k-clique communities and we defer comparisons to other overlapping community definitions and algorithms to future work. However, we want to stress that k-clique communities are particularly interesting, indeed as noted in [5]:

- k-cliques communities are based on topological properties and uses neither heuristics nor function optimizations, in particular they do not depend on the execution of an algorithm;
- the definition allows communities to overlap and
- each community exists independently of the other ones.

4 Our Clique Percolation Method algorithm methodology

4.1 General pseudo-code

The algorithms we developed to solve the CPM problem are all based on the same principle. All the k-cliques of the graph are streamed, and we group them to form the communities as they arrive. The main point is not to store the k-cliques in memory, to be able to work on huge graph, with several billions of edges.

A community is a list of k-clique, but compressed. So we cannot tell which are the k-clique of the community. We do not need this information, what we are interested in is to know the nodes forming the community.

All the algorithms we developed are dealing with k-clique arriving one by one. The communities are built gradually, by adding one by one its k-clique as they arrive. To do so, each algorithm has its way to store a community, and a test to list the current communities the arriving k-clique has to be added to. Then, all the communities of the arriving k-clique have to be merged, and the k-clique has to be added to it.

At the end of the k-clique listing, all the current communities form the final list of communities, and it can be returned. The global pseudo code is given in Algorithm 1.

This code is not the way we implement the different algorithms, but it is a guidelines of how we build the communities.

Algorithm 1 Building communities with one pass over k-cliques of graph G

```
1: commus ← [ ]
2: for each k-clique c_k ∈ G do
3:     l ← ListCommunitiesOf(c_k,commus)
4:     if l == ∅ then                          ▷ c_k starts a new community
5:         commus ← Add(c_k, commus)
6:     else                                    ▷ c_k has to be added to existing community
7:         C ← Merge(l)
8:         C ← C ∪ {c_k}
9:         for com ∈ l do
10:            commus ← Remove(com, commus)
11:        commus ← Add(C, commus)
```

4.2 An efficient k-clique listing

First of all, before testing and building communities, we have to be able to list efficiently all the k-cliques of the graph. To do so, we use the method developed in [REF], which is the more efficient method existing in the litterature to list k-cliques, and it is scaling well for huge graphs. This method is based on using directed graph $\vec{G}$ from a graph G, where $v \to u$ if $\eta(v) < \eta(u)$, with η an ordering of the nodes. Then, this directed graph is browsed so that each k-clique is seen only once. A pseudocode for this procedure is shown in Algorithm 2. $\vec{G}[\Delta_{\vec{G}}(u)]$ represents G restricted to the out neighbours of u.

Algorithm 2 Algorithm for listing k-cliques

```
1: Let η be a total ordering on the nodes of the input graph G
2: G⃗ ← directed version of G, where v → u if η(v) < η(u)
3: listing(k,G⃗,∅)
4: function listing(l,G⃗,C)
5:     if l = 2 then
6:         for each edge (u, v) of G⃗ do
7:             output k-clique C ∪ {u, v}
8:     else
9:         for each node u ∈ V(G⃗) do
10:            listing(l − 1,G⃗[Δ_G⃗(u)],C ∪ {u})
```

4.3 A Julia implementation

After working out the algorithms, I worked on the Julia language. We decided to learn this language because we aim to work on the guest graph possible. If it is well used, Julia has a running efficiency comparable to C, but it is much more easy to write. The syntax is

similar to the Python syntax, and the efficiency is similar to the C efficiency. A part of my work was to learn this language and its advantages. In addition to the other algorithms, I implemented the one which list kclist, and I obtained similar computation time as the C program. It is also a language easily parallelisable, which is very interesting for working with huge data.

The final objective of this work, which will be ended after the internship, is to have an efficient Julia program for solving the CPM problem. We are optimistic for the efficiency of the converging algorithm described in [REF], and we can at leat give upper and lower bound very efficiently.

5 Exact algorithms

During the internship, we developed two exact algorithms. The first one is time efficient but has a huge memory cost, and the second one is memory efficient but is quite time consuming.

5.1 A time efficient exact algorithm

We first show a preliminary version of our algorithm: Algorithm 3 which works by doing a single pass over the k-cliques and by storing all $(k-1)$-cliques that are contained in a k-clique. It relies on the fact that the output of CPM is a partition of the k-cliques of the input graph such that two k-cliques are in a same partition's cluster if they share $(k-1)$-nodes: a partition's cluster is thus a connected component in the graph where nodes are k-cliques and two k-cliques are connected if they share $(k-1)$ nodes.

Thus to compute all k-clique communities it suffice to list all k-cliques while merging the communities of two k-cliques when they share a $(k-1)$-clique.

Implementation issues. Algorithm 3 can be implemented efficiently by listing all k-cliques in the input graph using efficient algorithm such as [2]; storing all $(k-1)$-cliques (contained in a k-clique) and using a Union-Find data structure[1]. Given a Union-Find data structure `UF` containing n elements, the following two operations are allowed.

- `UF.Find`(a) returns the partition's cluster ID of element a in $O(\alpha(n))$ time where α is the inverse Ackermann function which is essentially constant.
- `UF.Union`(p, q) merges the two clusters p and q in $O(\alpha(n))$ time and returns the ID of the resulting cluster. As p and q are not elements, they are clusters IDs, so should Union complexity be $O(1)$?
- `UF.MakeSet`(a) creates a new cluster and assigns a to it while returning the cluster ID in constant time.

In order to have a more concise pseudocode, we also define `UF.FindOrCreate`(a) which returns the partition's cluster ID of element a and if a is not in `UF` it creates a new cluster and assigns a to it while returning the cluster ID.

[1] `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`.

Algorithm 3 One pass over k-cliques, storing $(k-1)$-cliques

```
1: UF ← Union-Find datastructure
2: for each k-clique c_k ∈ G do
3:     p ← NULL
4:     for each (k−1)-clique c_{k−1} in c_k do
5:         q ← UF.FindOrCreate(c_{k−1})
6:         p ← UF.Union(p, q)                ▷ Returns q if p = NULL
```

Algorithm 3 needs to store all $(k-1)$-cliques (contained in a k-clique) of the input graph. As the number of such $(k-1)$-cliques can be very large, this algorithm is problematic in some cases as it uses too much memory.

We next show how to avoid storing all these $(k-1)$-cliques. In particular, instead of storing the communities each k-clique belongs to, we store the communities each edge belongs to and we use a new datastructure that we call *Overlapping Union-Find* (OUF). This new datastruture builds on top of Union-Find and allows to do the same operations considering overlapping sets rather than a partition (i.e. non-overlapping sets).

We first detail how our datastructure works and then depict two pseudocodes:

- Algorithm 4 is very similar to Algorithm 3, but is using an OUF instead of a UF. This which ? Algo1 ? algorithm creates a variable for each $(k-1)$-cliques found and is thus not efficient in terms of memory like Algorithm 4. We show it for illustration purpose only.
- Algorithm 5 is our final algorithm also using an OUF. This algorithm does not create a variable for each $(k-1)$-cliques and is much more efficient than our two first algorithms.

As we will see in Section **??** and in Section 7, in addition of being orders of magnitude faster than the state-of-the-art, our final algorithm (Algorithm 5) is very efficient in terms of memory consumption. This allows our program to solve the problem for unprecedented values of k and sizes of graphs using a commodity machine.

OUF is a known data structure ? If not we should explain in more details, especially prove stuff about the complexity. **Overlapping Union-Find.** Given an Overlapping Union-Find data structure `OUF`, we denote by n the number of overlapping clusters in `OUF` and given an element a, n_a denotes the number of clusters a belongs to. The following operations are allowed.

- `OUF.Find`(a) returns the set of clusters that element a belongs to in time $O(n_a \cdot \alpha(n))$;
- $OUF.Union(p, q)$ merges the two clusters p and q in time $O(\alpha(n))$; As p and q are not elements, they are clusters IDs, so should Union complexity be $O(1)$?
- $OUF.MakeSet()$ creates an empty cluster and returns it in time $O(1)$ and
- $OUF.Add(a, p)$ add element a to cluster p (do nothing if a is already in p) in time $O(1)$.

These operations are allowed by storing variables representing the overlapping clusters in a Union-Find datastructure and by pointing each element to the clusters that it belongs to.

In order to have a clearer and more concise pseudocodes, we generalize the operations on sets of elements or clusters. In particular, the following operations are allowed.
Used in Algorithm 4:

- `OUF.FindOrCreate`(A) returns the intersection of the sets of clusters of the elements in A: (i) if the intersection is a singleton, it returns a single cluster; (ii) if the intersection is empty, it creates a new cluster and add all elements in A to it while returning its ID.

Used in Algorithm 5:

- `OUF.Find`(A) returns the intersection of the sets of clusters of the elements in A: (i) it returns a single cluster if the intersection is a singleton; (ii) it returns $NULL$ if the intersession is empty;

  Hi, I propose to simplify and say just that it "returns the intersection of the sets of clusters of the elements in A." replacing lines 5-7 by only one line

  $S \leftarrow S \cup$ `OUF.Find`$(E(c_{k-1}))$

- $OUF.UnionOrCreate(P)$ merges all clusters in P into a single one and returns the ID of the resulting cluster, if P is empty it creates a new empty cluster and returns its ID.

- $OUF.Add(A, p)$ add all elements in A to cluster p (do nothing for an element already in p).

Note that following this description of operations, in line 5 of Algorithm 4, `OUF.FindOrCreate`$(E(c_{k-1})$ always returns a single cluster and in line 5 of Algorithm 4, `OUF.Find`$(E(c_{k-1}))$ always returns a single cluster or $NULL$ as the set of all edges belonging to a given $(k-1)$-clique can only belong to a single community.

Algorithm 4 One pass over k-cliques, storing edges

```
1: OUF ← Overlapping Union-Find datastructure
2: for each k-clique c_k ∈ G do
3:     q ← NULL
4:     for each (k − 1)-clique c_{k−1} in c_k do
5:         p ← OUF.FindOrCreate(E(c_{k−1}))
6:         q ← OUF.Union(p, q)                ▷ Returns q if p = NULL
```

Algo 2 and 3 produce exactly the same result !

We now proceed to the theoretical analysis of our algorithms.

Algorithm 5 One pass over k-cliques, storing edges

```
1: OUF ← Overlapping Union-Find datastructure
2: for each k-clique c_k ∈ G do
3:     S ← ∅
4:     for each (k − 1)-clique c_{k−1} in c_k do
5:         p ← OUF.Find(E(c_{k−1}))
6:         if p ≠ NULL then
7:             S ← S ∪ {p}
8:     q ← OUF.UnionOrCreate(S)
9:     OUF.Add(E(c_k), q)
```

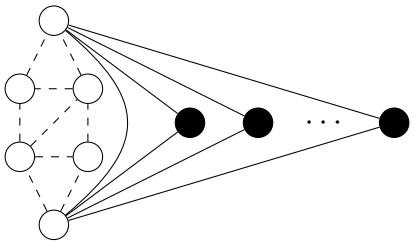

Figure 3: A problematic graph for algorithm that stores only nodes CPM0. It merges dashed and black triangle communities if any black clique is considered after all cliques of the dashed community. The number of black cliques can be arbitrary large, implying that CPM0 with a random order of clique processing will almost surely merge two communities when the number of black cliques tends to infinity. Our approximate Algorithms 4 and 5 stores links not just nodes. They are immune to this problematic case.

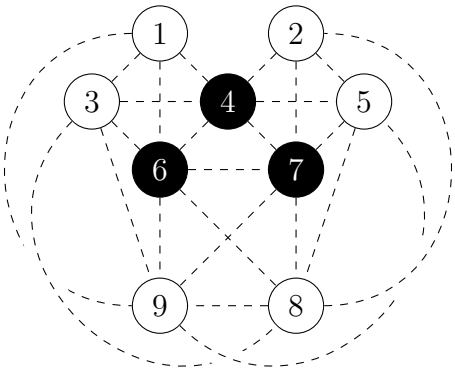

Figure 4: A $(k-1)$-clique F is called *phantom* if it is contained in a (sub)graph covered by a k-clique percolation denoted by P but there is no k-clique in P that contains F. This figure represents a minimal example of a graph covered by a k-clique percolation and containing a phantom $(k-1)$-clique, for $k = 4$. Duplicating a node from the clique one can easily generate an additional phantom clique.

5.2 A memory efficient exact algorithm

A community is a set of k-cliques. Keeping in memory all its k-cliques, or all its $(k-1)$-cliques, is not scaling well for huge graphs. A way of tackling this issue is to find a way to compress the data. In this section, the main idea is to store a community as a graph, storing only the nodes and the edges of the k-cliques added. So, we say that a k-clique is in the community if all its nodes and edges are in the community. Consequently, it can happen that a k-clique is tested beeing in the community whereas it has not be added to it. So, the community appears to have more k-clique that should be. The main issue is that if a k-clique appears to be in a community, and is also in another one, the two communities are merged, whereas it should not.

Therefore, whereas keeping the community as a list of edges allows to store communities with few memory, it creates parasites cliques that we need to compute and to store. A k-clique is added to the community if it shares a $(k-1)$-clique with it. A (-1)-clique parasite of a community is a $(k-1)$-clique that is tested to be in the community but which is not part of any k-clique added to the community. In consequence, if the $(k-1)$-clique common to a k-clique and a community is a parasite, we need to know it not to add the k-clique to the community.

In consequence, we choose to use a procedure to list all the $(k-1)$-cliques parasites of a community. Every time an edge is added to the community, we list all the $(k-1)$-cliques of the community containing this edge. Each one which is not a sub-clique of the k-clique added is then a parasite $(k-1)$-clique (RESULTAT A METTRE EN VALEUR ? + PREUVE ?). Then, we remove all the $(k-1)$-subliques of the k-clique added, so the list of $(k-1)$-cliques parasites is always up-to-date.

Algorithm 6 CPM 3 condensé

```
GraphCommus ← dict()
GhostCliques ← dict()
Fusions ← []
for each k-clique c_k ∈ G do
    commus_ck ← CommusNeighboursOf(c_k, GraphCommus, GhostCliques)
    if commus_ck == ∅ then
        CreateCommu(c_k, GraphCommus, GhostCliques)        ▷ c_k : new community
    else        ▷ Pick one neighbour community and add c_k to it, computing new ghost cliques for each new edge
        commu ← ONE community from commus_ck
        AddToCommuGraph(c_k, GraphCommus, GhostCliques)
        if —commus_ck— ¿ 1 then ▷ Record the other communities to merge them at the end
            Fusions.push(commus_ck)
ApplyFusions(Fusions, GraphCommus)
```

6 Converging algorithms

The algorithms we developed to the exact computation of k-clique communities do not scale well for huge graphs. However, starting from these algorithms, we can make them less precise but more efficient. In consequence, we are able only to have approximation of communities. However, working on an upper and on a lower approximation is very intersting if the gain of time and memory computation are important. That is what we propose here :

- From the algorithm which record the parasites : if we do not compute the parasites, and say that all k-cliques represented by the community dataset are k-cliques of the community, then there are k-cliques which appear to be in the community whereas they are not. In consequence, there are too much potential links between communities, some are merged whereas they should not. This idea gives a lower bound of communities.

- From the algorithm which stores the communities as $(k-1)$-cliques : if we do not store all the $(k-1)$-cliques, we earn memory. But some links between current communities are broken, and at the end there are more communities than it should be. There is not enough fusion during the computation. This idea gives an upper bound of communities

More details about these ideas and its implementations are given in this section. The memory and time of computation are well improved compared to exact algorithms described precedently. Besides, the precision achieved is quite encouraging.

6.1 Lower bound algorithm

6.2 Upper bound algorithm

In this section, we give an algorithm which gives an upper bound of the number of communities from the k-clique percolation. This algorithm takes as a parameter the integer z such as $1 \leq z \leq k-1$. The more z is high and the more the algorithm is precise, but with a cost in time and memory efficiency. For $z = k-1$, the value of the bound is the exact number of k-clique communities.

The more we add k-cliques to a community, the more we increase the possibility of merging it with other communities. So, this upper bound algorithm is based on not adding all the k-cliques to the communities (but enough to have a relevant bound). In consequence, some communities which normally have to be merged, are not, and the bound is higher than the real number of k-clique communities.

Let z be fixed in $\{1, ..., k-1\}$.

A z-community is represented as :

- "Packs" of k-cliques added to the community. A pack is the set of $(k-1)$-cliques of the k-cliques added. Two types of packs are possible, we need to work out wich one is better :

  - a big clique standing for all its $(k-1)$-sublique

– the exhaustive list of all $(k-1)$-subliques

To simplify, we first work on the exhaustive list of all $(k-1)$-subliques added.

- List of all z-subcliques of the k-cliques added to the community

Algorithm 7 One pass over k-cliques, storing some $(k-1)$-cliques

```
1: z fixed by the user : 1 ≤ z ≤ k − 1
2: UF ← Union-Find datastructure                          ▷ For (k − 1)-cliques
3: Comz ← List{}                          ▷ Comz[p] : Set of z-cliques of community p
4: for each k-clique c_k ∈ G do
5:     p ← NULL
6:     for each (k − 1)-clique c_{k−1} in c_k do
7:         q ← UF.Find(c_{k−1})
8:         if p ≠ NULL and p ≠ q then
9:             UF.Union(p, q)
10:            Comz[q] = Comz[p] ∪ Comz[q]
11:        p ← q
12:    add_ck ← False
13:    for each z-clique c_z in c_k do
14:        if c_z ∉ Comz[p] then
15:            Comz[p].add(c_z)
16:            add_ck ← True
17:    if add_ck then
18:        for each (k − 1)-clique c_{k−1} in c_k do
19:            q ← UF.Find(c_{k−1})
20:            if q == NULL then
21:                q ← UF.MakeSet(c_{k−1})
22:                UF.Union(p, q)
```

7 Experimental evaluation

In this section we compare our approach to the state-of-the-art approaches to compute k-clique communities in terms of running time and memory consumption.

7.1 Experimental Setup

We consider several real-world graphs that we obtained from [7] and present in Tables 1. These graphs are large having almost one million edges for the smallest one and almost two billion edges for the largest one. They also have ground truth communities, we will use these only in the case study (Section 8).

We carried our experiments on a Linux machine equipped with 4 processors Intel Xeon CPU E5- 2660 @ 2.60 GHz with 10 cores (a total of 40 threads) and with 64 G of RAM DDR4 2133 MHz.We evaluate both the sequential version of our algorithm, denoted as

networks	n	m	c
Amazon	334,863	925,872	7
Youtube	1,134,890	2,987,624	51
DBLP	425,957	1,049,866	113
Orkut	3,072,627	117,185,083	253
Friendster	124,836,180	1,806,067,135	304
LiveJournal	4,036,538	34,681,189	360

Table 1: Our set of real-world graphs with ground truth communities.

Algorithm 1, and the parallel version of our algorithm denoted as Algorithm n, where n denotes the number of threads. We evaluate our method using 1, 10, and 40 threads (Algorithm 1, Algorithm 10 and Algorithm 40, respectively) against the state-of-the-art for finding k-clique communities[1]. In particular, we consider the following approaches:

- **Palla et al.:** the algorithm proposed in the original paper introducing CPM [8]. We used the implementation provided by the authors on the dedicated webpage[2].
- **Kumpula et al.:** The algorithm suggested in [6] for which we used ... .
- **Reid et al.** The algorithm suggested in [9] ...
- **Gregori et al.:** The algorithm suggested in [5] ...

All the state-of-the-art methods except Kumpula et al. use a subroutine that first list all maximal cliques and then compute the k-clique communities out of the k-cliques. We thus also include the time to list all maximal cliques using the method detailed in [4] using the implementation provided by the authors. This method is, to the best of our knowledge, the fastest one for sparse real-world graphs. Note that the time bottleneck of this type of approach is actually the second step (building k-clique percolated communities out of the maximal cliques) and thus the running time to list all maximal cliques is a -rather not tight- lower bound on the running time. This method will be referred to by **Eppstein et al.**.

7.2 Approximation

Max, is the following true ?
CPM = Algorithm 1, algorithm gives exact results.

CPM0 is only on github, we store just nodes. Does CPM0 uses union find or overlapping union find structure ?

CPM1 In overlapping union find we store links, we merge two cliques if they share at least $\frac{(k-1)(k-2)}{2}$ links.

CPM2 is Algorithm 3, storing links.

[1] Our code is available at `https://github.com/maxdan94/CPM`.
[2] `http://www.cfinder.org/`

Notations starts to be complicated for me, I propose to adopt CPM0,CPM1,....,CPM(n-1)=CPM

where CPMk stores k-1 cliques notations.

==========

See also K-tree https://en.wikipedia.org/wiki/K-tree

Our algorithm, that stores links, in fact detects subgraphs having a spanning K-tree! Yes it seems to be true but they are not maximal it depends on the order.

In practice we detect something that is between a true k-clique percolation and a maximal subgraph having spanning k-tree.

It can link and motivate our paper from graph-theoretical perspective in addition to existing real-world clique-related motivations. And it will be useful to mention such combinatorial construction in the introduction.

See a graph is connected iff it has a spanning 1-tree.

Our algorithm (that stores links) detects graph components that are connected in spanning k-tree sense.

Indeed, it is not a clique percolation problem, but k-tree spaning decomposition problem. We even can define this problem, solve it exactly, than discuss application to clique percolation.

=========

Network	k	CPM	CPM1	CPM02	CPM2
Amazon	4	23,134	23,111	23,134	23,134
	5	10,942	10,942	10,942	10,942
	6	2621	2621	2621	2621
	7	30	30	30	30
Youtube	4	7433	6840	7369	7417
	5	2606	2222	2524	2575
	6	1147	872	1044	1107
	7	671	470	591	642
	8	346	218	293	322
	9	235	138	192	213
	10	126	96	113	121
	11	71	52	58	67
	12	52	35	47	49
	13	27	22	26	27
DBLP	4	47,307	47,025	47,305	47,306
	5	27,075	27,043	27,074	27,075
	6	14,112		14,112	14,112
	7	-		7388	7388
Orkut	4	306,755	115,527	288,209	298,676
	5	-		227,274	240,328
	6	-		167,610	179,059
LiveJournal	4	173,213	136,591	170,188	171,917
	5	-		123,470	125,476

Table 2: .

8 Case Study

In this section we study the results obtained by our algorithm:

- Number of k-clique communities and k-clique as a function of k
- Overlap between the k-clique communities
- Accuracy compared to the ground truth communities as a function of k.

9 conclusion and discussions

It works!

However, in some rare cases our approximate Algorithms 4 and 5 merge two different communities into one. As we can deduce from real-world experiences, whose results are presented in Table 2, it happens only for some rare cases. One of such problematic graphs is presented in Figure **??**.

VERIFY THIS STATEMENT! Algorithms 4 and 5 produce exact results when $k = 3$: because by construction they store all $k-1$-cliques, i.e. edges int the case of $k = 3$. In order to get around the problematic case illustrated on Figure **??** we could store all triangles. It suggest the creation of a family of algorithms, the pseudo-code will remain almost the same, but we replace $E(c_{k-1})$ (resp. $E(c_k)$) by $\mathcal{E}_\ell(c_{k-1})$ (resp. $\mathcal{E}_\ell(c_k)$) where $\mathcal{E}_\ell(c)$ enumerates all cliques of size ℓ inside a current clique c. In this regard, our results could be viewed as ℓ-clique relaxation of the original clique percolation problem. The quality augments as ℓ approaches $k-1$, when $\ell = k-1$ algorithm produces only exact results. We could also imagine a sequence of passes over the k-cliques considering the difference between numbers of detected communities, when this difference is less than ϵ we say "Ok, it is good enough".

It should be noted, that the order of k-clique processing plays an important yet not fully understood role in the problematic cases. It give rise to many interesting graph theoretical questions about characterisation of problematic sub-graphs: how many of them are presented in a typical real-world graph ? does there exists an order of k-clique processing that completely eliminates all approximation errors ?

MAX!!! See, maybe we should turn the paper a way around and present it as a sequence of approximate algorithms (starting from cpm0) discussing in more details the sequence of counter-examples and giving some intuitions grown under the light of extensive experiments ? Do rare problematic configurations happen less and less proportionately frequently when we add more and more value to ℓ ?

Doing this we could probably cover the current lack of serious analysis, by providing simple formulae based on natural parameters but extended to ℓ-clique approximation case. (A you dare enough to express the complexity using parameters that depends on output?)

We even could add the last Fantomas example from my mail to illustrate how algo CPM0 goes to be bad, and why cpm1 is better.

En fait dans nos communautés il y a des trous... des zones qui doivent être interdites à ajouter les nouveau cliques... si jamais on arrive a maintenir la liste de ces zones de le début ce serait top! on va gagner ! Car vos expériences montrent que ils sont quand même assez rares.

References

[1] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[2] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[3] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation*, pages 403–414. Springer, 2010.

[4] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *Experimental Algorithms*, pages 364–375, 2011.

[5] E. Gregori, L. Lenzini, and S. Mainardi. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1651–1660, 2013.

[6] J. M. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki. Sequential algorithm for fast clique percolation. *Physical Review E*, 78(2):026109, 2008.

[7] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[8] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[9] F. Reid, A. McDaid, and N. Hurley. Percolation computation in complex networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, pages 274–281. IEEE, 2012.

[10] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)*, 45(4):43, 2013.