Clique percolation method: memory efficient almost exact communities

Alexis Baudin[1], Maximilien Danisch[1], Sergey Kirgizov[2], Clémence Magnien[1], and Marwan Ghanem[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.lastname@lip6.fr
[2] LIB, Université de Bourgogne Franche-Comté
B.P. 47 870, 21078 Dijon Cedex France
sergey.kirgizov@u-bourgogne.fr

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 in the 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 if and only if 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 scale up CPM. In cases where this first algorithm faces memory limitations, we propose another algorithm, CPMZ, that provides a solution close to the exact one, using more time but less memory.

Keywords: Graphs, Graph mining, Social networks, Community detection, k-clique percolation

1 Introduction

The problem of detecting communities in real networks has received a lot of attention in recent years. Many definitions of communities have been proposed, corresponding to different requirements on the type of communities that are to be detected and/or on some properties of the studied graph: some definitions depend on global or local properties of the graph, nodes can belong to several communities or to a single one, links may have weights, communities may have a hierarchical structure, *etc.* In practice, algorithms also have to be designed to extract the communities from large graphs. Many definitions of communities proposed with their corresponding algorithm already exist [5]. Most real networks are characterized by communities that may overlap, i.e. a node may belong to several communities. In the context of social networks for instance, each person belongs to several communities such as colleagues, family, leisure activities, etc.

One of the most popular methods designed to find overlapping communities is the clique percolation method (CPM) which produces *k-clique communities* [14].

Definition 1 (k-clique). *A k-clique c_k is a fully connected set of k nodes, i.e. every pair of its vertices is connected by a link in the graph.*

For example, in Figure 2, the set $\{1, 3, 4, 6\}$ is a 4-clique.

Definition 2 (k-cliques adjacency). *Two k-cliques are said to be adjacent if and only if they share $k - 1$ nodes.*

For example, in Figure 2, the two 4-cliques $\{1, 3, 4, 6\}$ and $\{1, 3, 6, 9\}$ are adjacent.

Definition 3 (CPM community). *A k-clique community (or CPM community) is the set of the vertices belonging to a maximal set of k-cliques that can be reached from each other through a series of adjacent k-cliques.*

Though the corresponance between the obtained communities and real-world ones are hard to charactarize in a general manner, the advantages of this definition are well known [8]: it is formally well defined, totally deterministic, does not use any heuristics or function optimizations that are hard to interpret, allows communities to overlap, and each community is defined locally[3].

Despite much effort CPM has not been scalable to large graphs for medium values of k, i.e, values between 5 and 10. We therefore seek in this work to extend the computation of CPM to larger graphs. A bottleneck for most previous contributions is the memory required. Indeed, exact methods need to store in memory either all k-cliques, or all maximal cliques (cliques which are not included in any other clique), which is prohibitive in many cases.

Our contribution is twofold:

1. we improve on the state of the art concerning the computation of CPM communities, by leveraging an existing algorithm able to list k-cliques in a very efficient manner;
2. in cases where this first algorithm faces memory limitations, we propose another algorithm that provides a solution close to the exact one, using more time but less memory.

We will show that these algorithms allow to compute exact solutions in cases where this was not possible before, and to compute a close result of good quality in cases where the graph is so large that our exact method does not work due to memory limitations.

The rest of the paper is organized as follows. In Section 2, we present the related work. In Section 3, we present our exact and relaxed algorithms. We discuss time and memory requirements in Section 4. We then evaluate the performance of our exact algorithm against the state of the art in Section 5, and we compare the results and performances of our exact and relaxed algorithms. We conclude in Section 6, and present some perspectives for future work.

[3] Notice that if a node does not belong to at least one k-clique, it doesn't belong to any community.

2 Related Work

There are many algorithms for computing overlapping communities as shown in a dedicated survey [17]. The focus of our paper is on the computation of the k-clique communities. Existing algorithms to compute k-clique communities in a graph can be split in two categories:

(1) algorithms that compute all maximal cliques of size k or more and use them to compute all k-clique communities. Indeed, two maximal cliques that overlap on $k-1$ nodes or more belong to the same k-clique community. Most state-of-the-art approaches [8,14,15] belong to this category;

(2) Kumpula *et al.* [9] compute all k-cliques and then compute k-clique communities from them strictly following the definitions of a k-clique community, i.e. detect which k-cliques are adjacent.

Algorithms of the first category differ in the method used to find which maximal cliques are adjacent. However, the first step which consists in computing all maximal cliques is always the same and is done sequentially. While this problem is NP-hard, there exist algorithms scalable to relatively large sparse real-world graphs, based on the Bron-Kerbosch algorithm [1,3,4].

Any large clique, with more than k vertices, will be included in a single k-clique community, and there is no need to list all k-cliques of this large clique. 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, it has been found that most real-world graphs actually do not contain very large cliques and that listing k-cliques for small and medium values of k is a scalable problem in practice [2,12], in many cases it is more tractable that listing all maximal cliques. This makes algorithms in the category (2) more interesting for practical scenarios.

The algorithm of [9] proposes a method to list all k-cliques then merges the found k-cliques into k-clique communities using a *Union-Find* [7], a very efficient data structure [6,16] which we describe briefly in Section 3.1. In the context of [9] the Union-Find contains all $(k-1)$-cliques (as elements) and each k-clique c_k triggers the union of the subsets that contain at least one $(k-1)$-clique of c_k.

Our first contribution builds on the same idea. We first propose to use an efficient algorithm for listing k-cliques [2], which improves the overall performance. Going further, in order to provide an approximation of the community structure for graphs for which it is not possible to obtain the exact result due to memory limits, we propose to perform union of sets of z-cliques, $2 \leq z < k-1$, instead of $(k-1)$-cliques. This construction is discussed in details in the next section.

3 Algorithm

A graph $G = (V, E)$ consists of its vertex set V and its edge set E. In the following c_k will always denote a k-clique, from the context it will be clear which one exactly.

3.1 Union-Find structure

The algorithms we will present rely on the Union-Find structure, also known in the literature as a *disjoint-set data structure*. It stores a collection of disjoint sets, allowing very efficient union operations between them. The structure is a forest, whose trees correspond to disjoint subsets, and nodes correspond to the elements. The operations on the nodes are the following:

- `Find`(p): returns the root of the tree containing a Union-Find node p.
- `Union`($r_1, ..., r_l$): performs the union of trees represented by their roots r_i by making one root the parent of all others;
- `MakeSet`(): creates a new tree with one node p, corresponding to a new empty set, and returns p.

3.2 Exact CPM algorithm

First we build on the idea introduced in [9]. A CPM community is represented as the set of all the $(k-1)$-cliques it contains. These communities are represented by a Union-Find structure whose nodes are $(k-1)$-cliques. The algorithm then iterates over all k-cliques and tests if the current k-clique belongs to a community, by testing whether it has a $(k-1)$-clique in common with it.

Algorithm 1 Exact CPM algorithm

```
1: UF ← Union-Find data structure
2: Dict ← Empty Dictionary
3: for each k-clique c_k ∈ G do
4:     S ← ∅                                   ▷ communities of c_k to merge
5:     for each (k − 1)-clique c_{k−1} ⊂ c_k do
6:         if c_{k−1} ∈ Dict.keys() then
7:             p ← UF.Find(Dict[c_{k−1}])
8:         else
9:             p ← UF.MakeSet()
10:            Dict[c_{k−1}] ← p
11:        S ← S ∪ {p}
12:    UF.Union(S)
```

Algorithm 1 considers all k-cliques one by one. For every k-clique it iterates over its $(k-1)$-cliques $c^1_{k-1}, c^2_{k-1}, \dots c^k_{k-1}$. For every $c^i_{k-1}, i \in [1, k]$, it identifies the set p_i to which it belongs in the Union-Find. Then, it performs the union of all sets p_i. Several algorithms exist for efficiently listing k-cliques [12]. We substitute one the best [2] to the one proposed by the authors of [9].

As the number of $(k-1)$-cliques can be very large, this approach is problematic as in some cases it is not possible to store them all in memory. This leads us to a new algorithm which requires less memory but in rare cases incorrectly merges some CPM communities together.

3.3 Memory efficient CPM approximation

For relatively small values of k, there are far fewer z-cliques than $(k-1)$-cliques in real-world graphs. To get an intuition for this, consider the case of a large clique of size c. It contains $\binom{c}{z}$ z-cliques and this number increases with z for $z < c/2$. Therefore storing all z-cliques is feasible in cases where it is not possible to store all the $(k-1)$-cliques. We use this idea to propose an algorithm computing relaxed communities.

Definition 4 (CPMZ community). *An agglomerated k-clique community (or* CPMZ *community) is the union of one or more* CPM *communities.*

Our memory efficient method, called CPMZ *algorithm*, given a graph G, the size of k-cliques and an integer $z \in [2, k-1)$, returns a set of agglomerated k-clique communities, such that each CPM community is included in one and only one CPMZ community (see Theorem 1).

In the CPM algorithm, a community is represented as a set of $(k-1)$-cliques, and communities correspond to *disjoint* sets of $(k-1)$-cliques. In the following, a community is represented as a set of z-cliques, and CPMZ communities are represented as *non-disjoint* sets of z-cliques.

The main idea of our CPMZ algorithm is to identify each $(k-1)$-clique to the set of its containing z-cliques. The algorithm is very similar to CPM. For each k-clique, all $(k-1)$-cliques are considered. Since we consider that a $(k-1)$-clique is represented by the set of its z-cliques, the community of a $(k-1)$-clique is the one that contains all its z-cliques.

The CPMZ algorithm uses two principal data structures. `UF` is an Union-Find data structure, whose nodes are identifiers of z-clique sets. We will call these *Union-Find nodes*. The operations defined on this structure are presented in Section 3.1. Each z-clique can belong to several Union-Find nodes. This is recorded in the `Setz` dictionary, which associates to each z-clique the set of Union-Find nodes to which it belongs. See Figure 1 for an example.

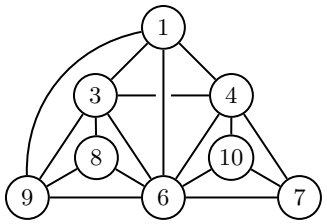
More formally, `Setz` is a dictionary with z-cliques as keys. For a z-clique c_z:

- `Setz`$[c_z]$ is a set of Union-Find nodes;
- `Setz`$[c_z]$`.add(`q`)` adds the Union-Find node q to the set of Union-Find nodes of c_z. It can also be seen as the action of adding c_z to the set identified by q.

At the end of the algorithm, every tree corresponds to a CPMZ community represented as a union of sets containing z-cliques.

Note that during the execution of the algorithm, the same z-clique c_z can belong to several Union-Find nodes of the same Union-Find set, which creates redundancies in `Setz` $[c_z]$. This is the case for instance in the example of Figure 1 in which `Setz[(3,6)]` contains both a and b which belong to the same Union-Find set. This situation can occur if a z-clique belongs to two different Union-Find sets which are merged later.

In our CPMZ algorithm (presented below) we eliminate these redundancies when we detect them (see Line 8).



$\texttt{Setz}[(1,3)] = \{a\}$, $\texttt{Setz}[(3,8)] = \{b\}$,
$\texttt{Setz}[(1,4)] = \{a\}$, $\texttt{Setz}[(3,9)] = \{a,b\}$,
$\texttt{Setz}[(1,6)] = \{a\}$, $\texttt{Setz}[(6,8)] = \{b\}$,
$\texttt{Setz}[(3,4)] = \{a\}$, $\texttt{Setz}[(6,9)] = \{a,b\}$,
$\texttt{Setz}[(3,6)] = \{a,b\}$, $\texttt{Setz}[(8,9)] = \{b\}$,
$\texttt{Setz}[(4,6)] = \{a,c\}$, $\texttt{Setz}[(1,9)] = \{a\}$,
$\texttt{Setz}[(4,7)] = \{c\}$, $\texttt{Setz}[(6,10)] = \{c\}$,
$\texttt{Setz}[(4,10)] = \{c\}$, $\texttt{Setz}[(7,10)] = \{c\}$,
$\texttt{Setz}[(6,7)] = \{c\}$.

Union-Find forest:

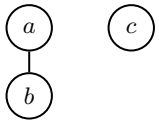


Fig. 1. Example of a graph and the corresponding data structures of the CPMZ algorithm. In this example, we have $k = 4, z = 2$. There are 17 z-cliques belonging to 4 k-cliques, namely {1,3,4,6}, {3,6,8,9}, {4,7,6,10} and {1,3,6,9}. The nodes of the Union-Find structure are represented using letters a, b and c. Each z-clique is associated to one or more Union-Find nodes, as shown in the `Setz` information on the top-right. The Union-Find structure represents two sets because there are two different root nodes: a and c. The first set contains all 2-cliques associated with a or b, the second contains 2-cliques associated with c.

Algorithm 2 CPMZ pseudocode

```
 1: UF ← Empty Union-Find data structure
 2: Setz ← Empty Dictionary
 3: for each k-clique c_k ∈ G do
 4:     S ← ∅                                          ▷ Sets of z-cliques to merge
 5:     for each (k − 1)-clique c_{k−1} ⊂ c_k do
 6:         P ← ∅
 7:         for each z-clique c_z ⊂ c_{k−1} do
 8:             Setz[c_z] ← {UF.Find(p) for p ∈ Setz[c_z]}
 9:             if P == ∅ then
10:                 P ← Setz[c_z]
11:             else
12:                 P ← P ∩ Setz[c_z]
13:         S ← S ∪ P
14:     q ← NULL                                       ▷ Identifier of the resulting set of z-cliques
15:     if S == ∅ then
16:         q ← UF.MakeSet()
17:     else
18:         q ← UF.Union(S)
19:     for each z-clique c_z ⊂ c_k do
20:         Setz[c_z].add(q)
```

Algorithm 2 is the pseudo-code of CPMZ. The for loop on Line 3 iterates over each k-clique c_k of a graph G. As in the CPM algorithm, the idea is to identify the communities of each $(k-1)$-clique of c_k and perform their union. As explained above, the communities of a $(k-1)$-clique are the ones that contain all its z-cliques, which is why we compute their intersection in the set P in Line 12. The set P is computed for all $(k-1)$-cliques in Lines 4-13 and their union is computed in set S. Then all the sets in S are merged in Line 18.

It may turn out that S is empty after the loop of Line 5. This corresponds to the case where none of the $(k-1)$-cliques of c_k were observed before: if a $(k-1)$-clique c_{k-1} has not yet been seen in the algorithm, its z-cliques may not belong to a common Union-Find set, and therefore P computed at Line 12 is empty. If this happens for all $(k-1)$-cliques of c_k S is empty and a new set (Union-Find node) is created on Line 16. The identifier of the resulting (new or merged) set is added to the set of Union-Find nodes for every z-clique of the current k-clique (Line 20).

In some rare cases, Line 12 will consider that a $(k-1)$-clique belongs to a Union-Find set while this is not true in the CPM exact case. Figure 2 gives an example. In that case this causes an incorrect k-clique adjacency detection and results in an incorrect merge of two or more k-clique communities.

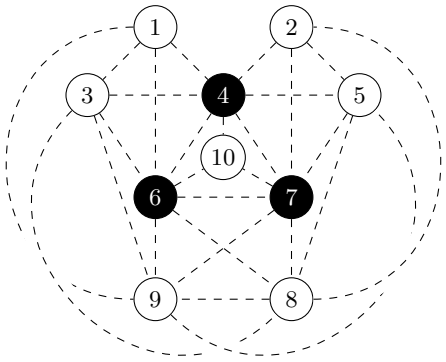


Fig. 2. In this example $k = 4$ and $z = 2$. A k-clique percolation community is formed by the nodes of the following k-cliques: $\{1, 3, 4, 6\}$, $\{1, 3, 6, 9\}$, $\{3, 6, 8, 9\}$, $\{6, 7, 8, 9\}$, $\{5, 7, 8, 9\}$, $\{2, 5, 7, 8\}$, $\{2, 4, 5, 7\}$. The middle $(k-1)$-clique with nodes $\{4, 6, 7\}$ is formed by the z-cliques (edges) of other k-cliques, whereas it is not itself a part of any k-clique given above. When a new k-clique $\{4, 6, 7, 10\}$ is observed, the CPMZ algorithm will produce one community $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, but the exact CPM algorithm gives two communities, namely $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $\{4, 6, 7, 10\}$.

Theorem 1 (CPMZ validity). *The* CPMZ *algorithm returns a set of agglomerated k-cliques communities, such that each* CPM *community is included in one and only one agglomerated community.*

Proof. If two k-cliques c_k^1 and c_k^2 are adjacent, this means they share a $(k-1)$-clique c_{k-1}. This will be correctly detected by Lines 5-12 of Algorithm 2: after the iteration on c_k^1 in the main loop, all z-cliques of c_{k-1} will belong to a common Union-Find set. The root of this set will belong to S during the iteration on c_k^2, ensuring that the Union-Find sets of both k-cliques will be merged. In other words, CPM communities are never split by CPMZ Algorithm and each CPM community belongs to a single agglomerated community. Conversely, an agglomerated community may contain more than one CPM community.

4 Analysis

We denote the number of k-cliques in graph G by n_k. For each k-clique, the CPM algorithm performs a `Find` and a `Union` operation for each of its $(k-1)$-clique (see Algorithm 1). It is well known (see for example [16]) that Union-Find data structure performs operations in $O(\alpha(n))$ amortized time, where α is the inverse Ackermann function, and n is the number of elements. It grows extremely slowly. Therefore, the number of operations is proportional in practice to $k \cdot n_k$. The space complexity of this algorithm is dominated by the tree on the $(k-1)$-cliques and the corresponding cost is proportional to $(k-1) \cdot n_{k-1}$.

CPMZ is a tradeoff between memory and time. Indeed, we will see that the CPMZ has a higher running time than the exact CPM algorithm, but requires less memory.

For the CPMZ algorithm as well, the time required by the `Union` and `Find` operations can still be considered as constant, as is the time required for the `MakeSet` and `Add` operations. The total number of operations is then dominated by the number of `Find` operations of Line 8. This line runs on each distinct triplet (c_k, c_{k-1}, c_z), where $c_{k-1} \subset c_k$ and $c_z \subset c_{k-1}$, and there are $\binom{k-1}{z} \cdot k \cdot n_k$ such triplets. Each z-clique c_z of Line 8 belongs to a number of Union-Find nodes $|\texttt{Setz}[c_z]|$ that depends on the considered clique but also varies during the execution of the algorithm: it can either increase as new Union-Find node are added to $\texttt{Setz}[c_z]$ (on Line 20) or decrease (on Line 8) if some Union-Find trees are merged (on Line 18). This number $|\texttt{Setz}[c_z]|$ is bounded by the number of k-cliques each z-clique belongs to, which can theoretically be quite high. However, we computed in practice the average number of `Find` operations performed for each z-clique, and we will see in Section 5.3 that it is very often equal to 1 or 2 and never exceeds 6 in our experiments. The main difference in the running time with respect to the exact CPM algorithm is, therefore, the extra $\binom{k-1}{z}$ factor.

Concerning the space requirements, the CPMZ algorithm needs to store all z-cliques, which takes a space proportional to $z \cdot n_z$. Each z-clique c_z then belongs to a number of Union-Find nodes $|\texttt{Setz}[c_z]|$ that varies during the algorithm execution. Even if the average of this number is low, we are interested now in the maximum space taken at any time of the execution. Finally, the number of nodes in the Union-Find structure is equal to the number of `MakeSet` operations that have been performed during the execution. In theory this number can also be quite high. However, we will see in Section 5 that in practice the memory

requirements of the CPMZ algorithm are much lower than those of the CPM algorithm.

Finally, notice that both our algorithms result in the Union-Find structure whose nodes represent sets of cliques. This structure encodes the communities. In order to obtain the actual node list of each community, post-processing is needed. It consists in iterating over all cliques in the Union-Find structure. Then for each clique one must find its root node in the Union-Find and add the clique nodes to the corresponding set. We do not take into account this post-processing in the reported running time and memory usage in the next section.

5 Experimental evaluation

Machine We carried out our experiments on a Linux machine DELL PowerEdge R440, equipped with 2 processors Intel Xeon Silver 4216 with 32 cores each, and with 384Gb of RAM.

Datasets We consider several real-world graphs that we obtained from [11]. Their characteristics are presented in Table 1. We distinguish between three categories of graphs according to their number of k-cliques. For graphs with small core values all algorithms are able to run; for graphs with medium core values, the state-of-the-art algorithms take too long to complete (with the exception of DBLP discussed below) while our CPM algorithm obtains results for small values of k; for graphs with large core values even our CPM algorithm runs out of memory or time except for very small values of k, but we will show that we are able to obtain relaxed results of high quality with our CPMZ algorithm.

Table 1. Our dataset of real-world graphs, ordered by core value c. k_{min} and k_{max} represent the minimum and maximum k on which we could run our CPM algorithm. n_k is the number of k-cliques of the graph.

network	n	m	c	$k_{min} - k_{max}$	$n_{k_{min}}$	$n_{k_{max}}$
soc-pokec	1,632,803	22,031,964	47	3 - 15	32,557,458	353,958,854
loc-gowalla	196,591	950,327	51	3 - 15	2,273,138	201,454,150
Youtube	1,134,890	2,987,624	51	3 - 15	3,056,386	1,068
zhishi-baidu	2,140,198	17,014,946	78	3 - 15	25,207,196	1,080,702,188
as-skitter	1,696,415	11,095,298	111	3 - 6	28,769,868	9,759,000,981
DBLP	425,957	1,049,866	113	3 - 7	2,224,385	60,913,718,813
WikiTalk	2,394,385	4,659,565	131	3 - 7	9,203,519	5,490,986,046
Orkut	3,072,627	117,185,083	253	3 - 5	627,584,181	15,766,607,860
Friendster	124,836,180	1,806,067,135	304	3 - 4	4,173,724,124	8,963,503,236
LiveJournal	4,036,538	34,681,189	360	3 - 4	177,820,130	5,216,918,441

Implementation We implemented our CPM and CPMZ algorithm in C. The implementation is available on the following gitlab repository: `https://gitlab.lip6.fr/baudin/cpm-cpmz`. For the competitors, we used the publicly available implementations of their algorithms [8,9,14,15].

Computing domain For each graph and each algorithm, we performed the computations of CPM for all values of k from 3 to 15, unless we were not able to finish the computation for one of the following reasons:

- the memory exceeded 390 Gb of RAM,
- or the computation time exceeded 72 hours.

We ran the CPMZ algorithm for $z = 2$ and $z = 3$. It could be computed on all the cases for which CPM works, except for $z = 3$ in graphs zhishi-baidu with $k = 15$ and DBLP with $k = 7$.

The interesting point is that we manage to have results with CPMZ in cases where the computation could not be carried out by the CPM algorithm: for as-skitter $k = 7$, WikiTalk $k = 8, 9$, Orkut $k = 6$ and Friendster $k = 5, 6$.

The detail of all the calculated values, with which the following figures were generated, is available on the following gitlab repository: `https://gitlab.lip6.fr/baudin/cpm-supplementary-material`.

5.1 Comparison with the state of the art

The algorithm proposed by Palla *et al.* in the original paper introducing CPM [14] is quadratic in the number of maximal cliques. Given that we are interested in graphs with at least several million cliques, these graphs are too big to be processed by this algorithm. We do not perform experiments with this algorithm.

In addition, our tests have shown that the algorithm by Reid *et al.* [15] has a better performance than that of Gregori *et al.* [8] (sequential version) therefore we do not present the results obtained with the version of Gregori *et al.*

We observed that there are indeed linearity factors:

- in time: for each k-clique, each of its $(k-1)$-clique is processed in constant time, hence the running time of CPM is indeed linear in $k \cdot n_k$
- in memory: the memory is used to store the Union-Find structure on the $(k-1)$-cliques: one node per $(k-1)$-clique encoded on $k-1$ integers, hence the memory needed by CPM is linear in $(k-1) \cdot n_{k-1}$.

Figure 3 compares the time and the memory necessary for the computation of the CPM communities by our CPM algorithm and the remaining competitive algorithms in the state of the art, proposed by Reid *et al.* [15] and Kumpula *et al.* [9]. For each competitive algorithm, we plot its running time (resp. memory usage) divided by the running time (resp. memory usage) of our CPM algorithm. We display the results as a function of n_k, where n_k is the number of k-cliques of the input graph. In some cases, our CPM algorithm obtains results whereas one of the state-of-the-art doesn't. This can happen because this algorithm exceeds

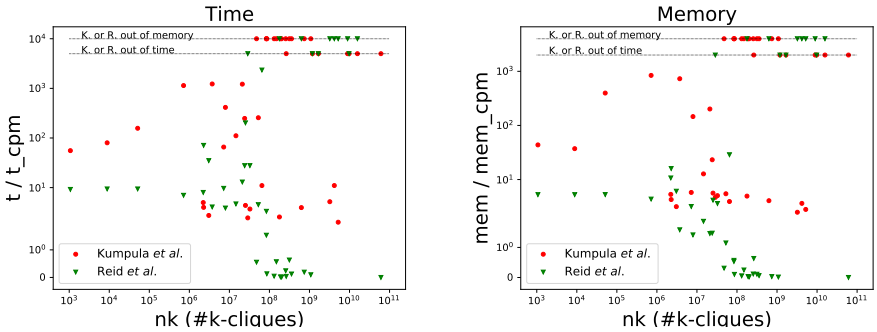


Fig. 3. Comparison between time and memory consumption of the state-of-the-art CPM methods and those from ours. For each competitive algorithm, we plot its running time (resp. memory usage) divided by the running time (resp. memory usage) of our CPM algorithm. We display the results as a function of n_k, where n_k is the number of k-cliques of the input graph. The maximum time is limited to 72h and the maximum memory to 390Gb. A marker placed at the corresponding line therefore indicates a computation that did not finish, because of either time or memory limit.

either the time or memory limit. We display this by placing a symbol on the corresponding horizontal line at the top of the figure.

First notice that the Reid *et al.* algorithm is better than ours for the four smallest graphs in our dataset (soc-pokec, loc-gowalla, Youtube, zhishi-baidu). Indeed, this algorithm begins by computing the maximum cliques, then processes them to form communities. In the case of these small graphs, the maximum cliques are easily computed. For such graphs, the memory used does not depend on k because in all cases the maximal cliques are stored; interestingly, the time computation time decreases with k as only the maximal cliques of size larger than or equal to k have to be tested for adjacency.

This algorithm is also more efficient than ours in certain configurations, when the graph is already well segmented into large cliques. This is the case with our DBLP graph, for which the Reid *et al.* algorithm manages to compute the communities in 10 seconds when we need several hours to process the large number of k-cliques.

Notice however that their algorithm does not allow to process the largest graphs of our dataset. The as-skitter intermediate graph contains too many cliques and their algorithm does not provide a result in less than 72 hours. For denser graphs (WikiTalk, Orkut, Friendster, LiveJournal), there are too many maximum cliques for RAM, and the algorithm cannot run, while ours is able to compute the result.

Finally, the algorithm of Kumpula *et al.* is systematically less efficient than ours. Our algorithm is also able to obtain results in cases where no other algorithm can provide any (see the points on the two horizontal lines on top of the figures).

5.2 Memory gain of the CPMZ algorithm

Figure 4 (right) compares the memory used by our algorithms CPM and CPMZ with $z = 2, 3$. We show the memory used by CPMZ divided by the memory used by CPM, as a function of n_k. As for the previous figure, we represent cases where CPMZ exceeds the time limit on a horizontal line on top of the figure. In addition, cases where we obtain results with CPMZ and not CPM are represented by symbols on a horizontal line at the bottom of the figure. For some small graphs, the number of $(k-1)$-cliques, which are stored by CPM, is smaller than the number of z-cliques. For these graphs CPMZ requires more memory than CPM. In most cases however, we observe a huge memory gain, and in some cases it is even possible to obtain results unachievable by our CPM algorithm.

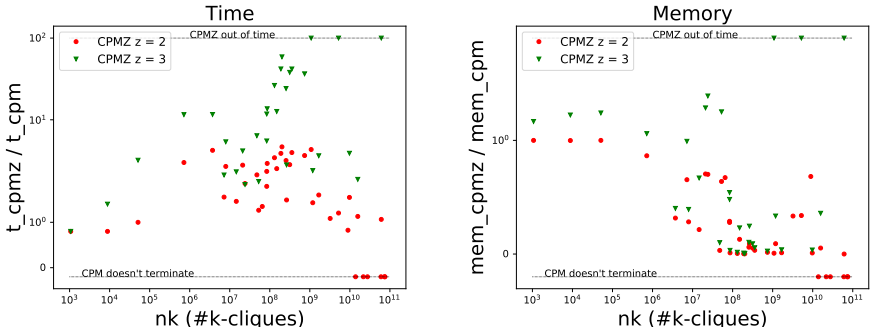


Fig. 4. Comparison between time and memory consumption of the CPMZ algorithm and our CPM. For CPMZ with $z = 2$ and $z = 3$, we plot its running time (resp. memory usage) divided by the running time (resp. memory usage) of our CPM algorithm. We display the results as a function of n_k, where n_k is the number of k-cliques of the input graph. The maximum time is limited to 72h, and the memory is not limiting in comparison with CPM. A marker placed at the top line therefore indicates a computation that did not finish, because of time limit. A marker placed at the bottom line indicates a computation that finishes for CPMZ but not for CPM.

In addition to this display and to the discussion about memory requirements in Section 4, we performed experiments to evaluate the memory associated with each z-clique in the algorithm. To do so, we computed the mean number of Union-Find nodes to which each z-clique belongs. In Algorithm 2, we therefore computed the sum the size of all `Setz[`c_z`]` every time the algorithm reaches Line 8 Then, we divided this sum by the number of times this line is browsed, which is $n_{k-1} \cdot k \cdot \binom{k-1}{z}$. We observed that this factor remains low; for all graphs and all values of k and z it is between 1 and 2, except for the small graphs of Youtube with a value around 4 for $z = 2$ and $k \in [10, 15]$, $z = 3$ and $k \in [11, 15]$, and zhishi-baidu with a value around 5 for $z = 2$ and $k = 5, 6, 7$.

Concerning the running time, as discussed in Section 4, there are two factors that cause CPMZ to be slower than CPM. The first one is the fact that a z-clique

can belong to several Union-Find nodes. As we just shown, this number is small in practice and therefore it does not play a strong role in the running time. The other factor is the extra $\binom{k-1}{z}$ factor induced by the fact that we consider all z-cliques included in a k-clique. This factor is high and therefore the computation time remains the limiting factor. Figure 4 (left) compares it to the running time of CPM.

5.3 CPMZ communities are very close to CPM communities

To measure the precision of our algorithm, we compare the agglomerated communities computed by the CPMZ algorithm with those of the CPM algorithm. To do so, we use an implementation of a Normalized Mutual Information measure for sets of overlapping clusters, called ONMI, provided by McAid *et al.* [13]. This tool measures how similar two sets of overlapping communities are.

We carried out the similarity comparisons on all the graphs of our dataset, with all the values of k for which we can compute the communities with the CPM algorithm (see Table 1). We observe the following:

- for CPMZ with $z = 2$, the average similarity is 98.6%, the median is 99.4% and all values are larger than 93.8%;
- for CPMZ with $z = 3$, the average similarity is 99.95%, the median is 100% and all values are larger than 99.5%.

This confirms that the incorrect merges between communities performed by the CPMZ algorithm have little influence on the final result: the structure of communities is barely impacted by the CPMZ algorithm.

6 Conclusion and discussions

In this paper we addressed the problem of overlapping community detection on graphs through the clique percolation method (CPM). Our contributions are twofold: first we proposed an improvement in the computation of the exact result by leveraging a state of the art k-clique listing method; then we proposed a heuristic algorithm called CPMZ that provides agglomerated communities, i.e. communities that are supersets of the exact communities; this algorithm uses much less memory than the exact algorithm, at the cost of a higher running time.

Through extensive experimentations on a large set of graphs coming from different contexts, we show that:

- our exact CPM algorithm outperforms the state of the art algorithms in many cases, and we are able to compute the CPM communities in cases where it was not possible before;
- our relaxed CPMZ algorithm uses significantly less memory than the exact algorithm; even though its running time is higher, this allows us to obtain agglomerated communities in cases where no other algorithm can run;
- finally the results provided by the CPMZ algorithm have an excellent accuracy, according to the ONMI method and obtain a score very close to 1 in the vast majority of cases.

Notice however that for the DBLP graph, which is of medium size, the approach proposed in [15] works better than ours. This can be explained because this graph naturally has a strong clique structure. Indeed, a link exists in this graph if two authors have written a paper together, and each paper therefore induces a clique on the set of its authors. In this case computing the maximal cliques and extracting the community out of them is more efficient than detecting adjacent k-cliques. This raises the interesting question of whether it is possible to predict which method will be more efficient on a graph by studying this graph's structure.

Several other interesting perspectives arise from our work. It should be noted that the order in which k-cliques are processed plays an important role in the incorrect community agglomerations performed by the CPMZ algorithm, that we do not yet fully understand. Our experiments show that in practice only a few merges of k-clique communities happen. This gives rise to many interesting graph theoretical questions about the characterisation of the sub-graphs that can produce an incorrect k-clique adjacency detection: how many of them are there in a typical real-world graph? We conjecture that it is possible to construct examples in which no processing order of the k-cliques will lead to the exact solution. However, in many cases including real-world graphs, it is possible that a certain processing order of k-cliques yields results of a higher quality than other orderings. This raises the question of how to design such an ordering. Another interesting possibility would be to run the CPMZ algorithm with two or more different k-cliques ordering and compare their output: since the CPMZ communities are coarser than the exact CPM communities, it is possible to compare the communities of both outputs to obtain a better result that any of the individual runs.

Finally, the linkstream formalism [10] allows to represent interactions that occur at different times, which is a natural framework to represent people meeting at different time during the week or computers exchanging IP packets on the internet. It would be interesting to investigate the community structure and its temporal aspects in such data by extending the definition of k-clique communities to linkstreams.

Acknowledgements

This work was partly supported by projects ANER ARTICO (Bourgogne-Franche-Comté region), ANR (French National Agency of Research) Limass project (under grant ANR-19-CE23-0010), ANR FiT LabCom and ANR COREGRAPHIE project (grant ANR-20-CE23-0002). We would like to greatly thank Lionel Tabourier for insightful discussions, useful comments and suggestions, and Fabrice Lecuyer for his careful proofreading.

References

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

2. Danisch, M., Balalau, O.D., Sozio, M.: Listing k-cliques in sparse real-world graphs. In: Champin, P.A., Gandon, F.L., Lalmas, M., Ipeirotis, P.G. (eds.) WWW. pp. 589–598. ACM (2018)
3. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in sparse graphs in near-optimal time. In: International Symposium on Algorithms and Computation. pp. 403–414. Springer (2010)
4. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. Experimental Algorithms pp. 364–375 (2011)
5. Fortunato, S., Castellano, C.: Community Structure in Graphs, pp. 490–512. Springer New York (2012)
6. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing. p. 345–354. STOC '89, Association for Computing Machinery (1989)
7. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM **7**(5), 301–303 (May 1964)
8. Gregori, E., Lenzini, L., Mainardi, S.: Parallel k-clique community detection on large-scale networks. IEEE Transactions on Parallel and Distributed Systems **24**(8), 1651–1660 (2013), implementation available at: `https://sourceforge.net/p/cosparallel`
9. Kumpula, J.M., Kivelä, M., Kaski, K., Saramäki, J.: Sequential algorithm for fast clique percolation. Physical Review E **78**(2), 026109 (2008), implementation available at: `https://github.com/CxAalto/scp`
10. Latapy, M., Viard, T., Magnien, C.: Stream Graphs and Link Streams for the Modeling of Interactions over Time. Social Networks Analysis and Mining **8**(1), 61:1–61:29 (Dec 2018). https://doi.org/10.1007/s13278-018-0537-7, `https://hal.archives-ouvertes.fr/hal-01665084`
11. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (Jun 2014)
12. Li, R.H., Gao, S., Qin, L., Wang, G., Yang, W., Xu Yu, J.: Ordering heuristics for k-clique listing. In: PVLDB (2020)
13. McDaid, A.F., Greene, D., Hurley, N.: Normalized mutual information to evaluate overlapping community finding algorithms (2013), `https://arxiv.org/abs/1110.2515`
14. Palla, G., Derényi, I., Farkas, I., Vicsek, T.: Uncovering the overlapping community structure of complex networks in nature and society. Nature **435**(7043), 814–818 (2005), implementation available at: `http://www.cfinder.org/`
15. Reid, F., McDaid, A., Hurley, N.: Percolation computation in complex networks. In: Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on. pp. 274–281. IEEE (2012), implementation available at: `https://sites.google.com/site/cliqueperccomp/home`
16. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (Mar 1984)
17. Xie, J., Kelley, S., Szymanski, B.K.: Overlapping community detection in networks: The state-of-the-art and comparative study. Acm computing surveys (csur) **45**(4), 43 (2013)