Quasi-Clique Mining for Graph Summarization

Antoine Castillon, Julien Baste, Hamida Seba, Mohammed Haddad

Quasi-Clique Mining for Graph Summarization (preprint)

Antoine Castillon[1,2], Julien Baste[1], Hamida Seba[2], Mohammed Haddad[2]
[1]Univ. Lille, CNRS, Centrale Lille,
UMR 9189 CRIStAL, F-59000 Lille, France.
[2]Univ Lyon, UCBL, CNRS, INSA Lyon, LIRIS,
UMR5205, F-69622 Villeurbanne, France.
antoine.castillon.etu@univ-lille.fr
julien.baste@univ-lille.fr
hamida.seba@univ-lyon1.fr
mohammed.haddad@univ-lyon1.fr

Abstract

Several graph summarization approaches aggregate dense sub-graphs into super-nodes leading to a compact summary of the input graph. The main issue for these approaches is how to achieve a high compression rate while retaining as much information as possible on the original graph structure, within the summary, without having to decompress it. These approaches necessarily involve an algorithm to mine dense structures in the graph such as quasi-clique enumeration algorithms. In this paper, we focus on improving this mining algorithms for the specific task of graph summarization. We first introduce a new pre-processing technique to speed up this mining step. Then, we adapt existing clique enumeration algorithms to dense sub-graph mining and apply them to graph summarization. Our extensive experimental study, on both synthetic and real-world graphs, show that our pre-processing technique allows an important speed up of the mining process, making it very useful for quasi-clique enumeration in general. Also, when applied to graph summarization, our approach improves significantly the execution time and the compression rate while maintaining a good retention of the original graph information especially those related to dense subgraphs.

1 Introduction

Graphs are a powerful modelling tool allowing to focus on the interactions between various entities. Today, several large-scale systems such as social networks or the link structure of the World Wide Web involve graphs with millions and even billions of nodes and edges [5]. The analysis of such graphs can prove to be very difficult given the huge amount of information contained in them. We note that a graph $G = (V, E)$ is represented by a set of vertices V and a set of edges E that link the vertices.

The goal of *graph summarizing* (or *graph compression*) is to provide, given an input graph, a smaller summary [10]. Depending of the situation, this summary can assume different roles. Either it exists uniquely to save storage space and can be decompressed to obtain the original graph with or without loss of information, or it can be used directly to obtain information on the input graph such as paths, neighbourhoods and clusters. When summarizing a graph, one can focus on the optimisation of the size of the summary introducing a size threshold [1, 7] or an information loss threshold [11]. Others adapt the summary so as to optimise some kind of queries [1, 4].

Even if these compression methods perform well in term of execution time and compression rate, they do not retain all the information available in the input graph, especially the dense components. Indeed, with these methods, such information is usually not accessible in the summary and requires a decompression in order to be retrieved. However, it appears that these dense components are essential in the analysis of many real life networks, such as social networks or protein connection networks [2, 13, 14]. Hence, in this paper, we focus on how well a summary retain information about the dense subgraphs of its original graph. This property can be measured by the concept of visibility introduced in [15] and defined in Section 3. In [16], the authors propose the Dense Subgraph Summarization method (DSS), a lossless compression scheme which addresses this issue, allowing a direct access to the dense subgraphs in the summary. However, this method requires the computation of some dense components of the Graph. In [16], the dense components are formally defined as quasi-cliques (cf. Definition 1) and the computation is done by solving the Quasi-Clique Enumeration Problem (cf. Definition 1).

[γ quasi-clique] Given $G = (V, E)$, and $\gamma \in [0, 1]$, a subset of vertices $Q \subseteq V$ is called a γ quasi-clique if for all $v \in V$, $|N(v) \cap Q| \geq \gamma(|Q| - 1)$, where $N(v)$ represents the neighbours of v.

[Quasi-Clique Enumeration Problem] Given $G = (V, E)$, a density threshold $\gamma \in [0, 1]$ and a size threshold m, the *Quasi-Clique Enumeration Problem (QCE)* consists of finding all maximal γ quasi-cliques in G of size greater than m.

DSS [16] uses the Quick algorithm [9] to solve the QCE problem. This algorithm performs a Depth-First Search on the solution space tree along with several pruning techniques.

In this paper we introduce new pruning techniques which, when added to the Quick algorithm, allow an important speed up. Also we describe how to adapt the search for quasi-cliques to this summarization approach by solving best suited relaxations of the Quasi-Clique Enumeration Problem. When compared to the DSS method alone the adding of our new techniques allows an important improvement of the performances both in time and compression rates and more importantly it retains most of the information regarding dense subgraphs within the summary.

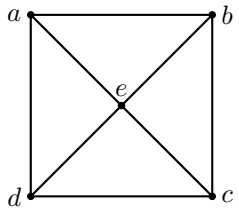
The remainder of this document is organised as follows. Section 2 describes the DSS compression method and the Quick algorithm. Section 3 introduces the new pruning techniques and presents a relaxation of the quasi-clique enumeration problem. Section 4 compares the performances of the different quasi-clique mining schemes and their application to dense subgraph based compression.

2 Background and Motivation

2.1 Dense Subgraph Summarization

Unlike other compression methods [11], the DSS method [16] is not designed to optimise the size of the summary. Thus it is, generally speaking, not the most efficient in terms of compression rates. The purpose of this method is to have in the summary a direct access to most of the original graph information, especially its dense subgraphs. We will discuss in Section 3.2 how to measure the retention of the dense subgraphs in the summary using the notion of visibility.

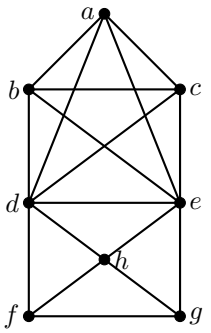
The DSS compression scheme works as follows: a graph G is compressed into a supergraph SG in several steps. First the dense subgraphs of G are computed using a quasi-clique enumeration algorithm (detailed in Section 2.2). Second the dense subgraphs are compressed into supernodes: each supernode $\texttt{Sn}_i$ records all the nodes of G it contains in the set $\texttt{node\_set}_i$. Also to ensure a lossless compression, $\texttt{Sn}_i$ keeps track of all non-edges inside $\texttt{node\_set}_i$ in the set $\texttt{loss\_edges}_i$ (an example of the supernodes transformation is given in Figure 1).

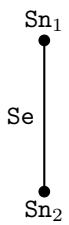


$\texttt{Sn}_1 = (\texttt{node\_set}_1, \texttt{loss\_edges}_1)$
$\texttt{node\_set}_1 = \{a, b, c, d, e\}$
$\texttt{loss\_edges}_1 = \{\{a, c\}, \{b, d\}\}$

Figure 1: A dense node set and the corresponding supernode

Third, a superedge is added between two supernodes with overlapping node sets (as shown in Figure 2). A superedge records its two endpoints along with the overlapping part of the two node sets in a set `connect_set`. Also, to ensure a lossless compression scheme, a set `remaining_edges` is added to store the edges that are not already covered by the supernodes.





- $\texttt{Sn}_1 = (\{a, ..., e\}, \emptyset)$
- $\texttt{Sn}_2 = (\{d, ..., h\}, \{\{d, g\}, \{e, f\}\})$
- $\texttt{Se} = (\{\texttt{Sn}_1, \texttt{Sn}_2\}, \texttt{connect\_set} = \{d, e\})$
- $\texttt{remaining\_edges} = \emptyset$

Figure 2: A graph G and the associated supergraph with supernode set $\{\texttt{Sn}_1, \texttt{Sn}_2\}$ and superedge set $\{\texttt{Se}\}$.

However, this process can create redundant superedges for instance the supernodes $\{1, 2, 3\}, \{1, 2, 4\}$ and $\{1, 2, 5\}$ are connected with three superedges but, since they share the same overlapping part, only two

superedges are required to represent the connection between them. In order to process these redundant edges a tracker is added. The tracker prevents the creation of a superedge between two supernodes already connected to superedges with the same `connect_set`.

2.2 Quasi-Clique Enumeration

As explained in the previous section, the DSS method requires a dense sub-graph mining usually done by solving QCE problem (cf. Definition 1). This problem is well known to be difficult (the associated decision problem being NP-hard [3]), and have already been studied a lot trough several aspects: exhaustive enumerations [9, 18], or only top-k enumeration [12, 17]. We focus on the exhaustive enumeration and its most classic method: the Quick algorithm [9]. Since there is no known polynomial time algorithm, the method to tackle this problem is to review every subsets of V. Given a graph G, a density threshold γ and a size threshold m, the Quick algorithm performs a depth-first exploration of the solution space tree (as in Figure 3).

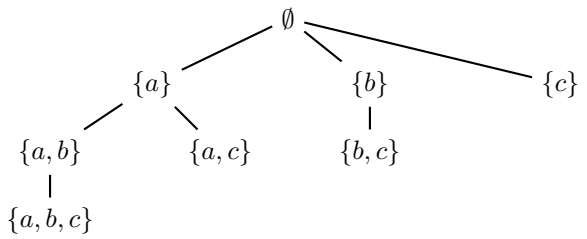


Figure 3: The solution space tree for $V = \{a, b, c\}$

In order to avoid the exponential time complexity and to speed up the computation, the algorithm prunes the tree: branches which cannot lead to a new solution are skipped. During the search, the algorithm keeps in memory X the current subset and a set `cand` formed by vertices outside X which are the next candidates to add to X. The pruning occurs according to these sets and several criteria, and actually corresponds to the removal of vertices from `cand`. Some of the most efficient criteria used in Quick are listed here.

1. Diameter:

   If $\gamma \geq 0.5$, then the diameter[1] of a quasi-clique is at most 2. So, during the execution one can restrict `cand` to the nodes that are at most two hops apart of all nodes in X.

2. Degree threshold:

   All vertices in a quasi-clique Y have at least $\lceil \gamma(|Y| - 1) \rceil$ neighbours in Y. So, during the execution one can abort if $\exists u \in X$ such that $d_{X \cup \texttt{cand}}(u) < \gamma(|X| - 1)$, and restrict `cand` to the u such that $d_{X \cup \texttt{cand}}(u) \geq \gamma(|X| - 1)$.

3. Upper/Lower bounds:

   Given X and `cand` one can compute L_X (resp. U_X) a lower (resp. upper) bound on the number of vertices of `cand` that must be added to X concurrently to form a quasi-clique. Given these numbers one can prune more efficiently[2].

4. Critical vertices:

   If there is $v \in X$ such that $d_{X \cup \texttt{cand}}(v) = \lceil \gamma(|X| + L_X - 1) \rceil$ then we can add all neighbours of v in X.

3 Our Contributions

3.1 New Pruning Techniques

While the Quick algorithm is efficient in the general case, it is not hard to point out cases where it is not. One can, for instance, take the example depicted in Figure 4. The reason that explains why Quick in not efficient on this example is because all pruning criteria of the algorithm are based only on the vertices and their degree. In the case where each vertex is contained in a quasi-clique, the pruning techniques cannot be properly applied and the algorithm proceeds to the exploration of almost the whole solution space tree.

[1] The diameter of a graph is defined as the length of the longest induced path.

[2] For instance in the second pruning technique $\gamma(|X| - 1)$ can be replaced by $\gamma(|X| + L_X - 1)$.

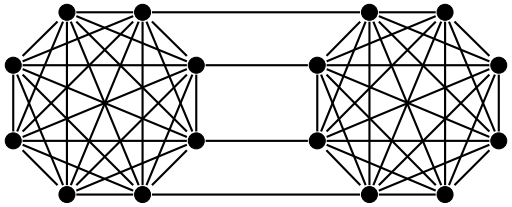


Figure 4: Example of the inefficiency of the Quick algorithm.

In order to handle these cases we introduce a new technique based on sets of vertices. According to Definition 1, each vertex v in a quasi-clique Y have at least $\lceil\gamma(|Y|-1)\rceil$ neighbours in Y. This property is extended to sets of vertices in the following lemma.

Given a graph G, $\gamma \in [0,1]$, Y a γ quasi-clique it holds that:

$$\forall X \subseteq Y : \left|\bigcap_{u\in X} N_Y(u)\right| \geq |X|\lceil\gamma(|Y|-1)\rceil - (|X|-1)|Y|.$$

We provide a proof of the lemma by induction on $|X|$.

- If $|X| = 1$: it is the definition of a γ quasi-clique.
- For $X \subseteq Y$ such that $|X| < |Y|$. Let $v \in Y \setminus X$,

$$N_Y(v) \cap \bigcap_{u\in X} N_Y(u) = N_Y(v) \setminus \left(V \setminus \bigcap_{u\in X} N_Y(u)\right),$$

  so by induction on X, $\left|\bigcap_{u\in X\cup\{v\}} N_Y(u)\right| \geq \lceil\gamma(|Y|-1)\rceil - |Y| + |X|\lceil\gamma(|Y|-1)\rceil - (|X|-1)|Y|$
  $\geq (|X|+1)\lceil\gamma(|Y|-1)\rceil - |X||Y|$.
- The results holds for all $X \subseteq Y$ and the lemma follows.

Using Lemma 3.1, we propose the following rule: given the density and size threshold γ and m, if $\{u,v\} \in E$ is such that u and v share strictly less than $2\lceil\gamma(m-1)\rceil - m$ common neighbours then we can remove $\{u,v\}$ from E. Applying this rule does not change the quasi-cliques of G because, thanks to Lemma 3.1 such edges are not contained in any quasi-cliques. This pre-processing is depicted in Algorithm 1.

Algorithm 1 Pre_processing(G,γ,m)

repeat
  delete all $u \in V$ s.t. $d(u) < \lceil\gamma(m-1)\rceil$
  delete all $\{u,v\} \in E$ s.t. $|N(u) \cap N(v)| < 2\lceil\gamma(m-1)\rceil - m$
until G has not been modified during the last loop

Actually, this rule does not use the lemma at its full potential. Indeed the lemma provides a much more general result which could be used to add a new pruning technique based on forbidden sets of vertices. For instance, when parsing the solution space tree, if the label set X of the current node does not verify the lemma then the branch starting from this node can be pruned right away. However, such pruning technique is very hard to adapt in practice since it would require to test at each node X all the subsets of X. Also, as discussed in Section 4, the performances of the pre-processing alone are already very promising. Hence, we limit ourselves to the pre-processing algorithm, and a pruning technique making a full use of the lemma could be the object of further research.

3.2 Redundancy Aware Maximal Quasi-Cliques

While the Quick algorithm is an effective method to tackle the QCE problem, it is important to note that this problem is not perfectly suited to the DSS method. Indeed, having access to all quasi-cliques can create redundancy in the summary and impact the performances. Figure 5 gives such an example: the graph G has four very similar 0.85 quasi-cliques, so, in the summary the fact that $[\![1,8]\!]$ forms a clique is stored four times (one per supernodes). This redundancy does not appear with only one quasi-clique which leads to a better compression.

In order to detect and avoid redundant quasi-cliques, [15] introduce the notion of visibility[3]. The visibility of a maximal quasi-clique Q with respect to a set of maximal quasi-cliques S is a value between

[3]The results of [15] hold for cliques but can be easily adapt to quasi-cliques

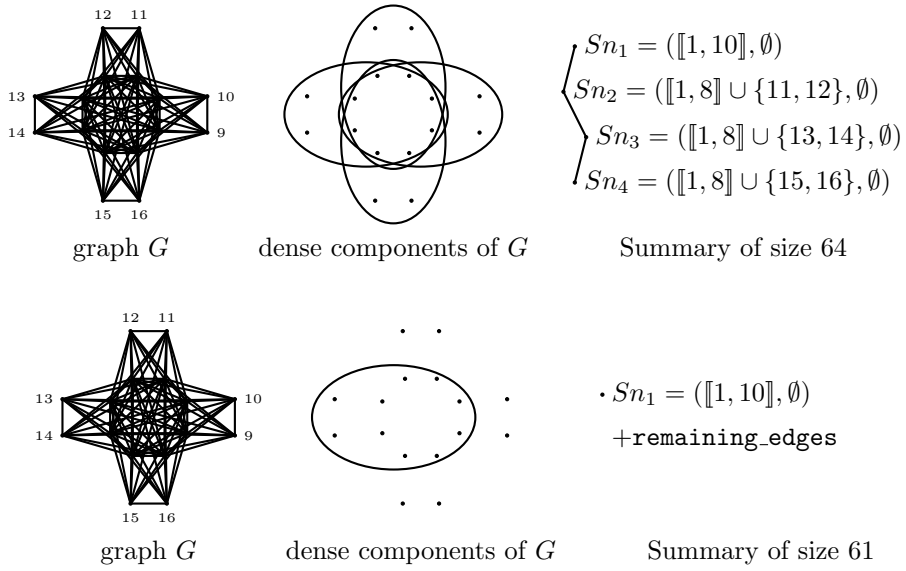


Figure 5: Comparison of two summarizations of G, one with all the dense components and the other with only one of them.

0 and 1 which describes how well Q is represented by the quasi-cliques of S. If the visibility is 1 then Q is in S, on the contrary if the visibility is 0 then Q does not overlap with any quasi-clique of S. [15] also formally introduce a variant of the QCE problem (Definition 3.2) where instead of finding all the quasi-cliques one only have to ensure that all maximal quasi-cliques have a sufficient visibility.

[Visibility] Given a graph G, $\gamma \in [0, 1]$, $Q \in \mathcal{M}_\gamma(G)$ and $\mathcal{S} \subseteq \mathcal{M}_\gamma(G)$ (the set of all maximal γ quasi-cliques of G), the $\mathcal{S}$-**visibility** of Q: $\mathcal{V}_\mathcal{S}(Q)$, is defined as:

$$\mathcal{V}_\mathcal{S}(Q) = \max_{Q' \in \mathcal{S}} \frac{|Q \cap Q'|}{|Q|}.$$

[τ-visible QCE]

Given G, $\gamma \in [0, 1]$ and $\tau \in [0, 1]$, the *τ-visible Quasi-Clique Enumeration Problem* is to compute $\mathcal{S} \subseteq \mathcal{M}_\gamma(G)$ such that $\forall Q \in \mathcal{M}_\gamma(G)$, $\mathcal{V}_\mathcal{S}(Q) \geq \tau$.

Note that this problem is well suited to the DSS method for a few reasons. First, the purpose of the DSS method is to have a direct access to the dense subgraphs in the summary. A summary obtained by the DSS method with S a solution of the τ-visible QCE problem as input, provides a direct access to all the elements of S which represent well all maximal quasi-cliques. Second, avoiding redundant quasi-cliques improves not only compression rates but, if done during the search, also allows to speed up the computation, pruning branches corresponding to redundant quasi-cliques.

Due to the similarity between this problem and the QCE problem, our approach to solve it is to adapt the Quick algorithm, adding techniques allowing to skip redundant quasi-cliques during the search. The first algorithm we use is Algorithm 2, introduced in [15]. This algorithm adds to an enumeration algorithm, a pruning technique based on visibility. This pruning occurs randomly during the exploration of the tree depending on the visibility of the set of the current vertices, $X \cup \texttt{cand}$, with respect to the last quasi-clique found. Due to the structure of the solution space tree, similar sets are usually close to each other in the tree. Thus, redundant quasi-cliques are often skipped. This approach has several advantages. First, as expected the skipping allows this algorithm to run significantly faster than Quick and gives better compression rates (performances are detailed in Section 4). Also, as proven in [15], the set $\mathcal{S}$ obtained by this algorithm verifies that for all $Q \in \mathcal{M}_\gamma(G)$, the expected value of $\mathcal{V}_\mathcal{S}(Q)$ is greater than τ which ensures a good representation of all quasi-cliques of G. Finally, the parameter τ represents a trade-off between the runtime and the visibility, allowing the user to adapt the algorithm according to its preferences. With lower values of τ the algorithm runs faster but ensures less visibility. On the contrary higher values of τ ensure better visibility but slow down the runtime.

Regarding our second approach, we note that given a graph G and two quasi-cliques Q, Q'. If Q and Q' are very similar ($\mathcal{V}_{\{Q\}}(Q')$ is close to 1) then most of the edges in $G[Q']$ are already covered in $G[Q]$. On the contrary if Q and Q' are very different then $G[Q]$ and $G[Q']$ do not share many edges. Hence, to avoid redundancy, after finding the quasi-clique Q one can remove from G all the edges inside $G[Q]$, thus the quasi-cliques of G similar to Q become sparse and are skipped by the pruning techniques of the Quick algorithm. Removing such edges usually does not impact other different quasi-cliques since these

[4] $s : r \in [0, 1] \mapsto \frac{(1-r)(2-\tau)}{(2-r-\tau)}$, note that s is decreasing, $s(0) = 1$ and $s(1) = 0$.

Algorithm 2 Quick_redundancy_aware(G, γ, m, τ):

Let $\mathcal{T}$ be the solution space tree of V (nodes are labelled with X and `cand`)
Let $\mathcal{S} = \emptyset$
Let $Q' = \emptyset$
Depth first search on $\mathcal{T}$
  X, `cand` = current node
  $\bar{l} = |X| + |\texttt{cand}|$
  Prune $\mathcal{T}$ with probability $1 - \sqrt[\bar{l}]{s\left(\frac{|X \cap Q'|}{\bar{l}}\right)}$ [4]
  if $\mathcal{T}$ is not pruned pursue the depth first search
    ⋮ (same as Quick)
  if X is a quasi-clique and with probability $\sqrt[|X|]{s\left(\frac{|X \cap Q'|}{|X|}\right)}$
    Add X to $\mathcal{S}$
    $Q' = X$
return $\mathcal{S}$

quasi-cliques do not share many edges with Q. This method depicted in Algorithm 3 outputs a set of quasi-cliques $\mathcal{S}$ which covers all edges contained in quasi-cliques. Similarly both the runtime and the compression rate are better with this method than with the Quick algorithm (Section 4). Even if this algorithm do not ensure any lower bound on the visibility (contrary to Algorithm 2), the experimental results presented in Section 4 show that the visibility obtained is mostly acceptable.

Algorithm 3 Quick_delete_covered_edges(G, γ, m)

run Quick(G, γ, m)
  each time a quasi-clique X is found:
    remove from G all edges of $G[X]$

4 Performances

We compare the performances of four different quasi-clique mining algorithms: the Quick algorithm presented in Section 2.2, Algorithms 2 and 3 presented in Section 3.2 and finally a greedy quasi-clique mining algorithm, i.e., Algorithm 4, which serves as a basis of comparison. These four algorithms have been implemented in Python 3 and executed on a Windows 10 computer with a 2.5 GHz Intel(R) Core (TM) i5 processor and 8 GB RAM. Each algorithm is tested with and without the pre-processing algorithm, namely Algorithm 1, on several types of graphs. Since the DSS method is especially interesting for social networks, we use two types of synthetic graphs which model well social networks: the community graph model, developed in Section 4.1, and the LFR graph model, developed in Section 4.2. In Section 4.4, we also use graphs from social networks such as Facebook, Twitter and Google+. As for the performances we want to measure not only the runtime of the algorithms but also how well they are suited to the DSS method. Hence, three aspects have been retained: the runtime, the size of the summary obtained with the DSS method and the visibility of all quasi-cliques in the input graph.

Algorithm 4 Greedy_quasi_cliques(G, γ)

`Qclqs` $= \emptyset$
`non_covered_edges` $= E$
while `non_covered_edges` $\neq \emptyset$ **do**
  Let $\{u, v\}$ in `non_covered_edges`
  $X = \{u, v\}$
  `cand` $= \{w \in V \setminus X | X \cup \{w\}$ is a quasi-clique $\}$
  while `cand` $\neq \emptyset$ **do**
    Let z in `cand`
    $X = X \cup \{z\}$
    `cand` $= \{w \in V \setminus X | X \cup \{w\}$ is a quasi-clique $\}$
  end while
  `Qclqs` $=$ `Qclqs` $\cup \{X\}$
  remove from `non_covered_edges` all edges inside X
end while

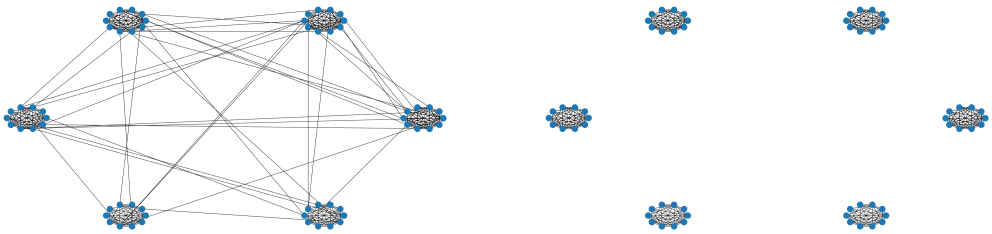


Figure 6: Left: a community graph; Right: the same graph after the execution of the pre-processing algorithm.

4.1 Community Graph Model

Algorithm 5 Community_graph(n, k, s, α, β)

$G = (\llbracket 1, n \rrbracket, \emptyset)$
partition V_G into $k+1$ sets $R, C_1, ..., C_k$ $(|C_1| = ... = |C_k| = s)$
for $i \in \llbracket 1, k \rrbracket$ **do**
  for $\{u, v\} \subseteq C_i$ **do**
    add uv with probability α
  end for
end for
for $\{u, v\} \subseteq V_G$ s.t. u and v are not both in any C_i **do**
  add u, v with probability β
end for
return G

The first model we consider is the community graph model described in Algorithm 5. As input, the user specifies n the number of vertices, k the number of communities, s the size of the communities and two parameters $\alpha, \beta \in [0, 1]$. Given these inputs the algorithm constructs first the communities and then the graph. The edges inside the communities are added to the graph with probability α, usually close to 1, and the ones outside the communities are added with probability β, usually close to 0. In order to model further social networks we add the possibility for communities to intersect one another and also to have different sizes replacing the parameter s with two parameters s_1 and s_2 respectively the lower and upper bound of the community sizes.

On this type of graphs, since the communities are mostly disjoints, the pre-processing algorithm is expected to be very effective. An instance of a community graph is given in Figure 6. Note that the pre-processing algorithm manages to delete all edges outside the communities while edges inside communities are mostly preserved. Also, the communities are separated in different connected components which corresponds to the best case scenario, the quasi-cliques of $G = (V, E)$ are computed in time $O(|V| \times |E|)$ (complexity of the pre-processing algorithm).

The performances of the algorithms on community graphs are given in Figures 7-8. In Figure 8 there is the comparison in time, compression rates and visibility (respectively, first, second and third column) of all algorithms. Note that the Quick algorithm achieves the slowest runtime, the worst compression rates and the best visibility, that is always 1. This is due to the fact that this algorithm actually enumerates all quasi-cliques (which explains the visibility). Also, as discussed in Section 3.2, this causes the algorithm to be slower than the others (which prune more the search tree) and creates redundancy in the summary and thus, leads to worst compression rates. Regarding the other algorithms, it seems there is a trade-off between runtime, compression rate and visibility: the greedy algorithm which achieves the best runtimes and the second best compression rates have the worst visibility (around 0.4 against 0.9 for the others). This trade-off explains why, the DSS method requires complex quasi-clique mining algorithms like Quick (or Quick_redundancy_aware or Quick_delete_covered_edges) and why it cannot properly work with simple greedy algorithms.

Figure 7 presents the results of the comparison between the performances with and without the use of our pre-processing algorithm, Algorithm 1. It appears that the use of this algorithm lowers significantly the runtimes, up to a factor 7 for values of β greater than 0.4, avoiding a lot of cases like Figure 4 and thus is very useful for speeding up quasi-clique mining algorithms. Also, since this algorithm does not delete edges inside quasi-cliques, the performances in compression rates and visibility are not really impacted by its use, making this algorithm also very interesting for the DSS method.

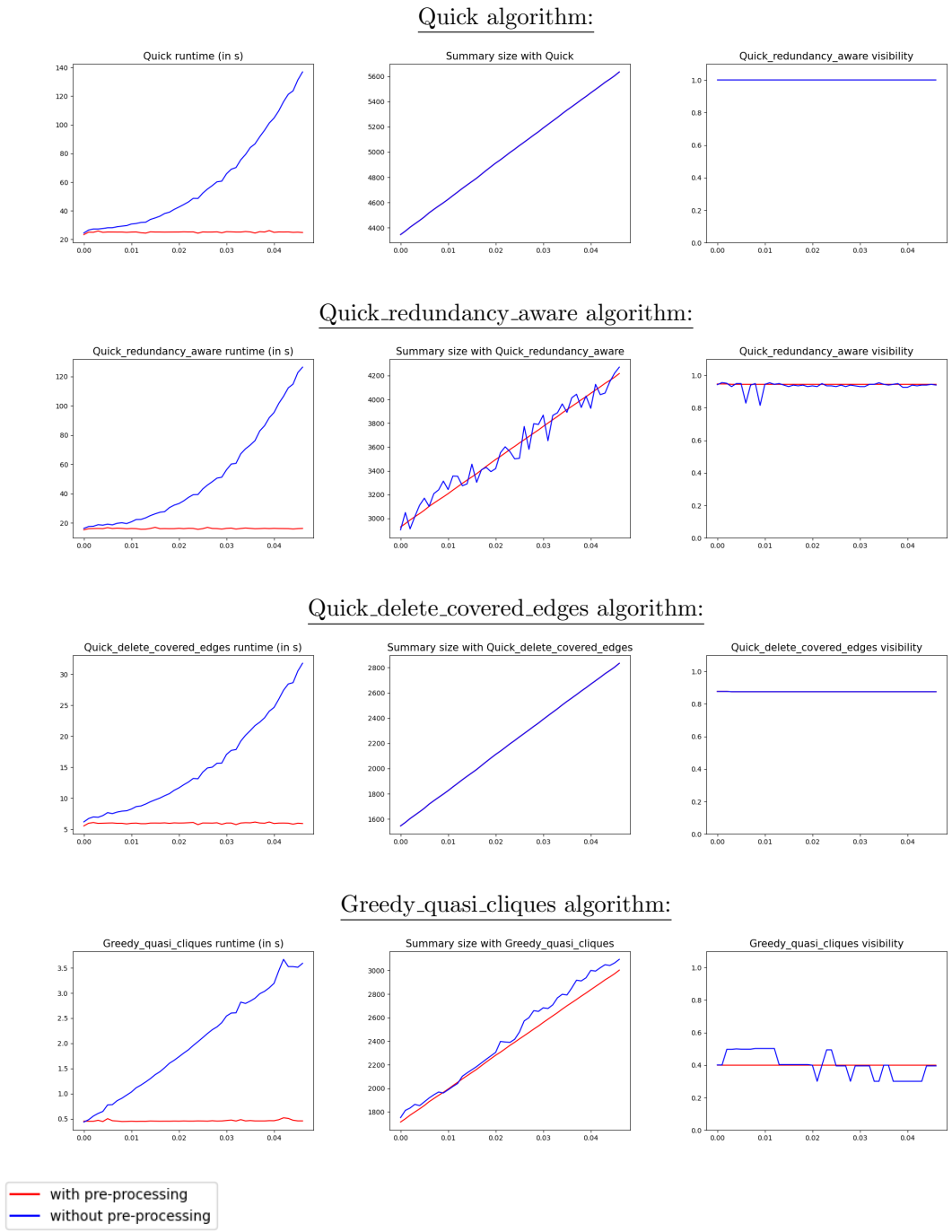


Figure 7: Comparative performances of the algorithms with and without the pre-processing on community graphs with $n = 250$ nodes, $k = 8$ communities of size 30, $\alpha = 0.9$ and $\beta \in [0, 0.05]$ corresponds to the values on the x-axis.

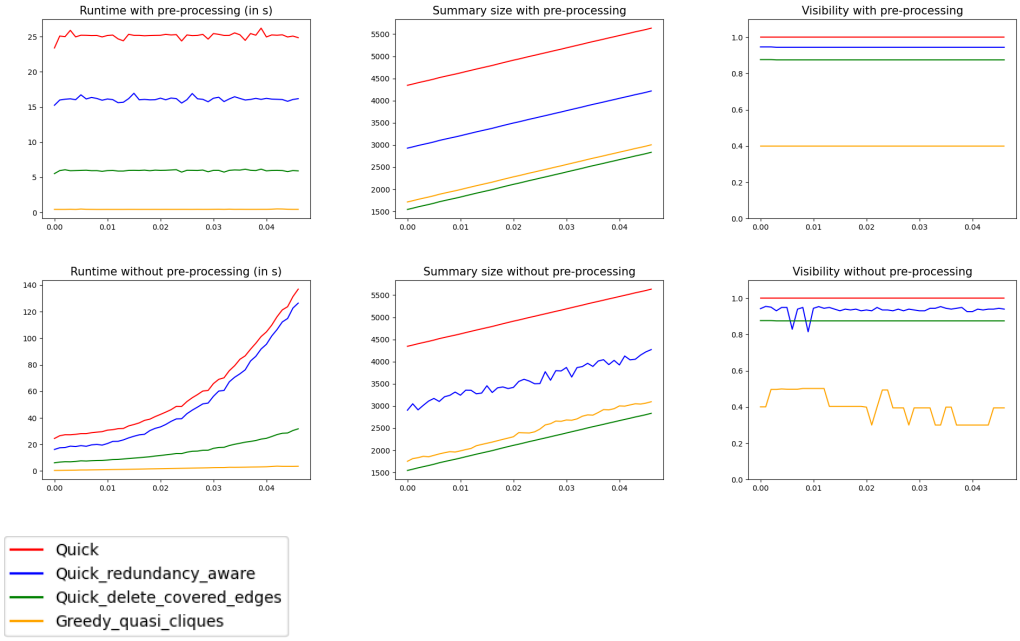


Figure 8: Performances of all algorithms on community graphs with $n = 250$ nodes, $k = 8$ communities of size 30, $\alpha = 0.9$ and $\beta \in [0, 0.05]$ corresponds to the values on the x-axis.

4.2 LFR Graph Model

Even if the results with the community graph model are promising, the model itself is very simplistic. Indeed, for instance two nodes in a community C have mostly the same number of neighbours inside and outside C ($\approx \alpha|C|$ and $\beta(n - |C|)$), also the communities can be very similar (as in Figure 6). On the contrary, real networks often have a much more heterogeneous distributions of nodes degree and community sizes [6].

The second model we consider, the LFR graph model [6], addresses these issues using power laws for the degree and the community size distributions (which models well social networks). Given as input n the number of vertices, $\tau_1, \tau_2 > 1$ the power law exponents of respectively the degree distribution and the community size distribution, $\mu \in [0, 1]$ and `average_degree` the average degree of the vertices, the LFR benchmark algorithm proceeds as follows. First, the algorithm finds a degree sequence with a power law distribution which respects the `average_degree` value. Then, the algorithm generates the sizes of each community according to another power law and each node is randomly assigned to a community according to the community sizes. Finally, the edges are computed according to the degree of each node and to the community, each node u having $(1 - \mu) \cdot d(u)$ neighbours in the same community and $\mu \cdot d(u)$ neighbours in other communities.

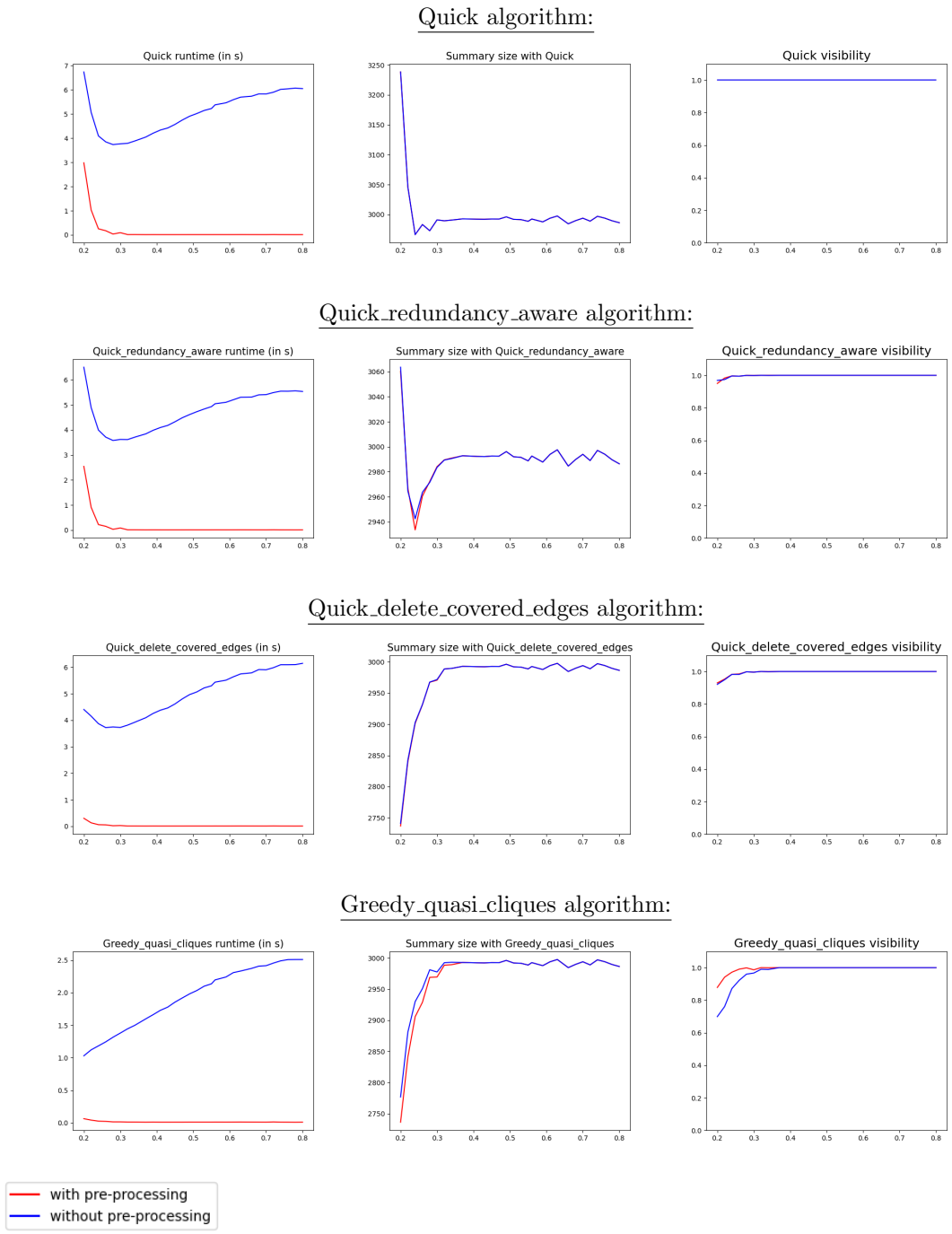


Figure 9: Comparative performances of the algorithms with and without the pre-processing on LFR graphs with $n = 300$, `average_degree` $= 20$, 16 communities of size 19 (on average) and $\mu \in [0.2, 0.8]$ corresponds to the value on the x-axis.

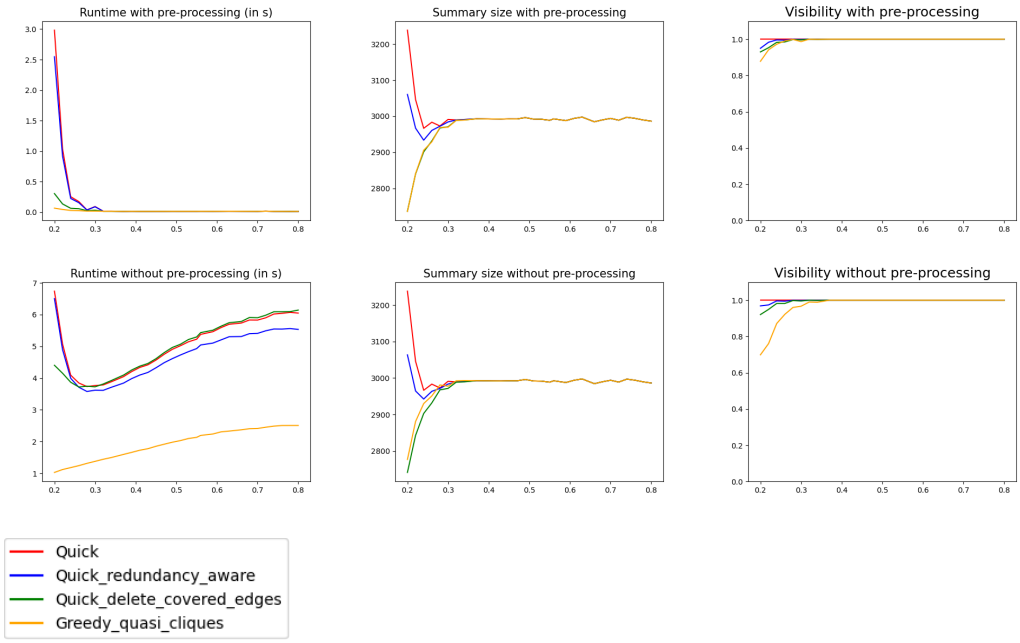


Figure 10: Performances of all algorithms on LFR graphs with $n = 300$, `average_degree` $= 20$, 16 communities of size 19 (on average) and $\mu \in [0.2, 0.8]$ corresponds to the value on the x-axis.

The performances of the algorithms on LFR graphs are given in Figures 9-10. The results obtained are very similar to the results with the community graph model. They show as previously that the use of the pre-processing algorithm improves significantly the runtimes without affecting the compression rates nor the visibility. Additional remarks can be pointed out, for instance with values of μ greater than 0.4 the visibility of all algorithms is constant to 1. This is due to the fact that for these values of μ, the communities are sparse (the density of edges inside them is around $1 - \mu$) and thus, no quasi-cliques exist, and with no quasi-cliques, the visibility is always 1. Also, with the same values of μ, the compression rates of all algorithms are the same and mostly stable because with no quasi-cliques the summary graph is empty and the original graph is entirely stored in the set `remaining_edges`. Thus, all summarizations have the same size, the size of the input graph (since, the parameter μ does not affect the number of edges, the size of the input graph is mostly constant). Finally, note that with the Quick and the Quick_redundancy_aware algorithms, the compression rates first drop for values of μ from 0.2 to 0.3, increases for values of μ in $[0.3, 0.4]$ and then stabilises. This is caused again by redundancies: for small values of μ (around 0.2) the communities are dense and contain a lot of quasi-cliques which creates redundancies (note that with the Quick algorithm redundancies can cause the summary to be larger than the input graph). With larger values of μ (around 0.3), fewer quasi-cliques remain which prevents redundancy. And as explained before with μ larger than 0.4 the input graph does not contain any quasi-clique and thus no compression is achieved.

4.3 Social Networks

Since our methods and algorithms have proven to be effective on synthetic graphs and in order to test further their reliability we also evaluate their performances on real social networks. The networks used are available on the Stanford large network dataset collection [8]. The performances on social networks are given in Figures 11-12. When comparing the performances with and without the use of the pre-processing algorithm, these results prove its utility showing again a significant decrease of the execution while not impacting the compression rates nor the visibility. However, when comparing the different algorithms, they give a more mixed result. Indeed, first the redundancy aware algorithm does not seems to be much more efficient than Quick both in time and compression rates, where the gain should be. Also, even if Algorithm 3 seems efficient regarding execution time and compression rates, the visibility obtained with this algorithm is worst than the others dropping suddenly below 0.8 while the others stay close to 1. Finally, the greedy algorithm seems to perform surprisingly well on these graphs, being the most efficient both in time and compression rate while keeping a very high visibility. However, we think that this high visibility rate is due to the small number of different quasi-cliques existing in the graphs. Indeed, even with 1800 nodes the graphs only contain highly similar quasi-cliques, for instance five quasi-cliques are enough to achieve a visibility of 0.95, and with three of them the visibility stays higher than 0.93. Hence, given the other results, we think that the visibility rate of the greedy algorithm will drop significantly when adding more nodes (and thus more quasi-cliques) to the graph.

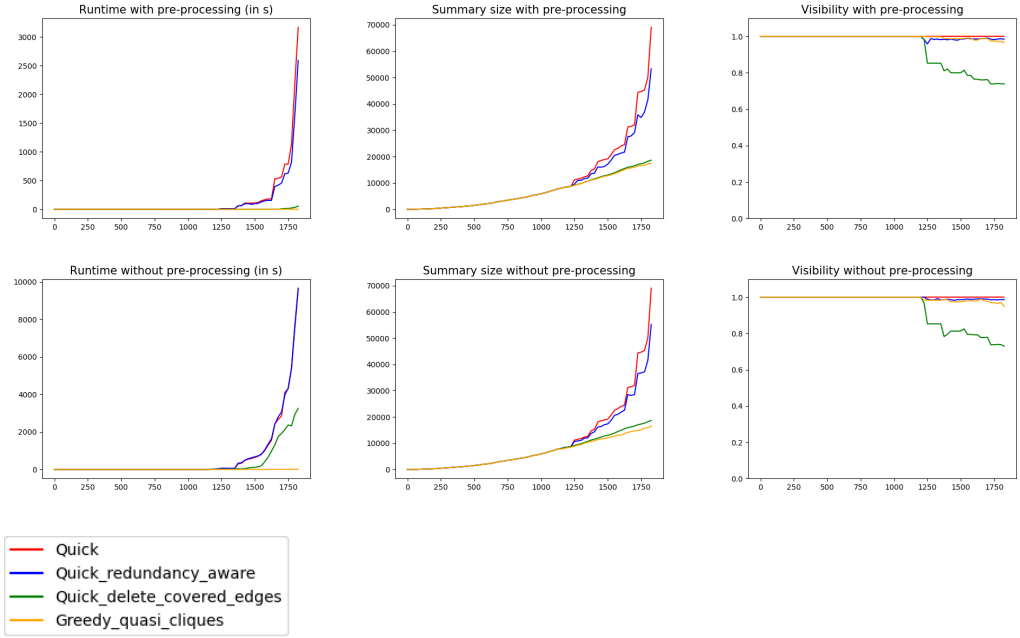


Figure 11: Performances of all algorithms on social networks with $n \in [0, 2000]$ nodes, the value of n is represented by the x-axis.

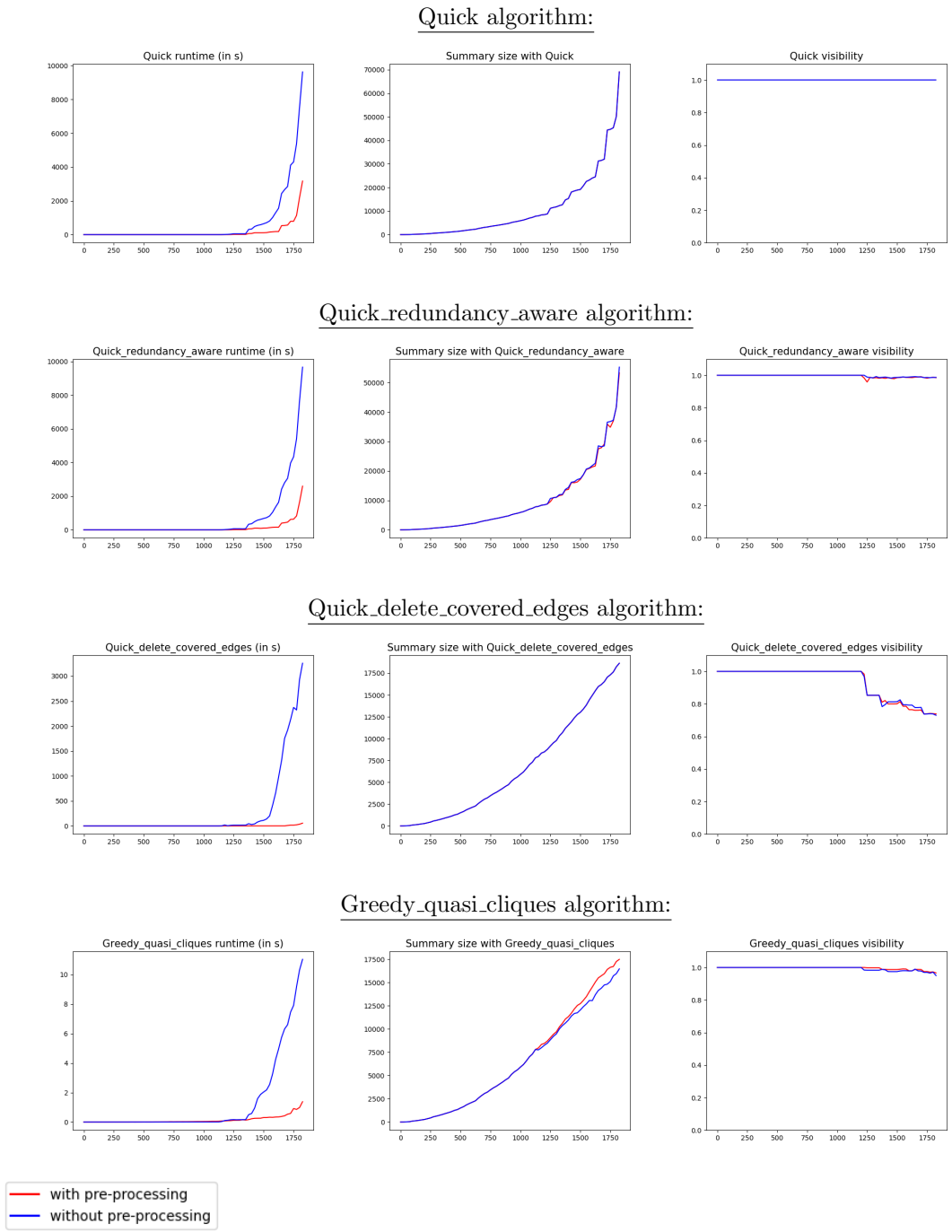


Figure 12: Comparative performances of the algorithms with and without the pre-processing on social networks with $n \in [0, 2000]$ nodes, the value of n is represented by the x-axis.

4.4 Discussion

While the previous results have shown that our pre-processing algorithm and our Quick variants are very effective for the DSS method some social networks present patterns unsolvable with the Quick algorithm (and its variants). For instance, in Figure 13 there is a large sparse quasi-clique which is very problematic for the Quick algorithm and its variants. Even if the density of the quasi-clique is quite low, the large number of vertices prevents the pruning, making all techniques (including the pre-processing) useless. Hence, the Quick algorithm proceeds to the review of every subsets, thus running indefinitely without finding any quasi-clique. Since, the redundancy aware and delete covered edges algorithms run similarly as Quick as long as no quasi-clique is found, the variants have exactly the same issue. Finally, this case cannot be solve by a better management of the arbitrary parameters γ and m. Indeed, as shown in [9] the Quick algorithm is always faster with larger values of γ, also only values of $\gamma > 0.5$ achieve interesting compression rates with the DSS method (with $\gamma < 0.5$ there are more lost edges than compressed edges). Also, in this example, with a size threshold m greater than 121, there is no quasi-clique and thus no compression and with a size threshold m lower than 120 the issue occurs.

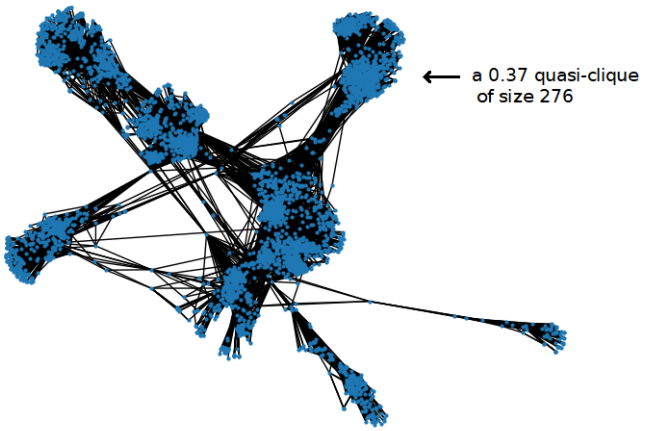


Figure 13: A social network from Facebook with 4039 nodes and 88234 edges.

5 Conclusion

In this paper, we focused on graph compression using dense subgraphs and more precisely quasi-cliques, trough the example of the DSS method. This approach requiring a quasi-clique mining algorithm, we show how to improve the already existing quasi-clique enumeration algorithms introducing a very effective pre-processing technique. Also, we adapt and introduce other quasi-clique mining algorithms designed to avoid redundancy (without loosing the purpose of the DSS method) improving performances both in runtime and compression rate. The experimental results show that if our algorithms are very promising, some specific patterns still cannot be handled and require new techniques. This can be the subject of future work.

Acknowledgement:

This work is funded by Agence Nationale de la Recherche (ANR) under grant ANR-20-CE23-0002.

References

[1] Muhammad Ahmad, Maham Anwar Beg, Imdadullah Khan, Arif Zaman, and Muhammad Asad Khan. Ssag: Summarization and sparsification of attributed graphs. 2021. http://arxiv.org/abs/2109.15111 arXiv:2109.15111.

[2] Gary Bader and Christopher Hogue. An automated method for finding molecular complexes in large protein interaction networks. 02 2003. https://doi.org/10.1186/1471-2105-4-2 doi:10.1186/1471-2105-4-2.

[3] Ambroise Baril, Riccardo Dondi, and Mohammad Mehdi Hosseinzadeh. Hardness and tractability of the γ-complete subgraph problem. 2021. https://doi.org/10.1016/j.ipl.2021.106105 doi:10.1016/j.ipl.2021.106105.

[4] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. 2003.

[5] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. 2004. https://doi.org/10.1145/988672.988752 doi:10.1145/988672.988752.

[6] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. Oct 2008. https://doi.org/10.1103/PhysRevE.78.046110 doi:10.1103/PhysRevE.78.046110.

[7] Kyuhan Lee, Hyeonsoo Jo, Jihoon Ko, Sungsu Lim, and Kijung Shin. Ssumm: Sparse summarization of massive graphs. 2020. http://arxiv.org/abs/2006.01060 arXiv:2006.01060.

[8] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[9] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. 09 2008. https://doi.org/10.1007/978-3-540-87481-2$_3$*doi* : 10.1007/978 − 3 − 540 − 87481 − 2$_3$.

[10] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. 2018. https://doi.org/10.1145/3186727 doi:10.1145/3186727.

[11] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. 2008. https://doi.org/10.1145/1376616.1376661 doi:10.1145/1376616.1376661.

[12] Seyed-Vahid Sanei-Mehri, Apurba Das, and Srikanta Tirthapura. Enumerating top-k quasi-cliques. Dec 2018. https://doi.org/10.1109/bigdata.2018.8622352 doi:10.1109/bigdata.2018.8622352.

[13] Quoc-Dinh Truong, Taoufiq Dkaki, and Quoc-Bao Truong. Graph methods for social network analysis. 03 2016. https://doi.org/10.1007/978-3-319-46909-6$_2$5*doi* : 10.1007/978 − 3 − 319 − 46909 − 6$_2$5.

[14] Duygu Ucar, Sitaram Asur, Umit Catalyurek, and Srinivasan Parthasarathy. Improving functional modularity in protein-protein interactions graphs using hub-induced subgraphs. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Knowledge Discovery in Databases: PKDD 2006*, 2006.

[15] Jia Wang, James Cheng, and Ada Wai-Chee Fu. Redundancy-aware maximal cliques. 2013. https://doi.org/10.1145/2487575.2487689 doi:10.1145/2487575.2487689.

[16] Ling Wang, Yu Lu, Bo Jiang, Kai Tai Gao, and Tie Hua Zhou. Dense subgraphs summarization: An efficient way to summarize large scale graphs by super nodes. 2020.

[17] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Diversified top-k clique search. 04 2015. https://doi.org/10.1109/ICDE.2015.7113300 doi:10.1109/ICDE.2015.7113300.

[18] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. 2007. https://doi.org/10.1145/1242524.1242530 doi:10.1145/1242524.1242530.