Fast Maximal Quasi-clique Enumeration: A Pruning and Branching Co-Design Approach

KAIQIANG YU, Nanyang Technological University, Singapore
CHENG LONG*, Nanyang Technological University, Singapore

Mining cohesive subgraphs from a graph is a fundamental problem in graph data analysis. One notable cohesive structure is γ-quasi-clique (QC), where each vertex connects at least a fraction γ of the other vertices inside. Enumerating maximal γ-quasi-cliques (MQCs) of a graph has been widely studied and used for many applications such as community detection and significant biomolecule structure discovery. One common practice of finding all MQCs is to (1) find a set of QCs containing all MQCs and then (2) filter out non-maximal QCs. While quite a few algorithms have been developed (which are branch-and-bound algorithms) for finding a set of QCs that contains all MQCs, all focus on sharpening the pruning techniques and devote little effort to improving the branching part. As a result, they provide no guarantee on pruning branches and all have the worst-case time complexity of $O^*(2^n)$, where O^* suppresses the polynomials and n is the number of vertices in the graph. In this paper, we focus on the problem of finding a set of QCs containing all MQCs but deviate from further sharpening the pruning techniques as existing methods do. We pay attention to both the pruning and branching parts and develop new pruning techniques and branching methods that would suit each other better towards pruning more branches both theoretically and practically. Specifically, we develop a new branch-and-bound algorithm called `FastQC` based on newly developed pruning techniques and branching methods, which improves the worst-case time complexity to $O^*(\alpha_k^n)$, where α_k is a positive real number strictly *smaller* than 2. Furthermore, we develop a divide-and-conquer strategy for boosting the performance of `FastQC`. Finally, we conduct extensive experiments on both real and synthetic datasets, and the results show that our algorithms are up to two orders of magnitude faster than the state-of-the-art on real datasets.

CCS Concepts: • **Mathematics of computing → Graph algorithms**.

Additional Key Words and Phrases: cohesive subgraph enumeration; quasi-clique; branch-and-bound

ACM Reference Format:
Kaiqiang Yu and Cheng Long. 2023. Fast Maximal Quasi-clique Enumeration: A Pruning and Branching Co-Design Approach. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 211 (September 2023), 26 pages. https://doi.org/10.1145/3617331

1 INTRODUCTION

Cohesive subgraph mining is a fundamental problem in graph data analysis. For a given graph, it aims to find *dense/cohesive* subgraphs that carry interesting information for solving practical problems [17]. One notable cohesive structure is γ-quasi-clique (QC) [19, 23, 24, 28, 32, 35, 51], which is a natural generalization of clique [6, 8, 13, 15, 26, 39]. Specifically, QC requires that each vertex connects at least a fraction γ of the other vertices inside, where γ is a fraction between 0 and 1. One of the fundamental QC mining problems, which we call MQCE, is to enumerate all large

*Corresponding author (c.long@ntu.edu.sg)

Authors' addresses: Kaiqiang Yu, kaiqiang002@e.ntu.edu.sg, Nanyang Technological University, Singapore; Cheng Long, c.long@ntu.edu.sg, Nanyang Technological University, Singapore.

2836-6573/2023/9-ART211
https://doi.org/10.1145/3617331

Maximal Quasi-Cliques (MQCs) with the number of vertices inside at least a threshold θ for a given graph [19, 23, 24, 28, 32, 51].

The MQCE problem has been widely studied in the past [19, 23, 24, 28, 32, 51] and used for various applications such as discovering biologically relevant functional groups [2, 5, 7, 21], finding social communities [16, 20], detecting anomaly [38, 41, 46], etc. For example, authors in [32] conduct a case study that finds biologically relevant functional groups by mining large MQCs which have the size at least a threshold and appear in each graph from a set of protein-protein interaction and gene-gene interaction graphs. The rationale is that for a functional group of proteins, each of them interacts with most of the rest, which would form a QC likely [32]. Another example is that the authors in [20] conduct a case study that finds meaningful communities by mining large MQCs from graphs built on publication data.

Challenges and Existing Methods. The MQCE problem is challenging, which is evidenced by several facts. First, this problem is NP-hard [30]. Second, the problem of checking whether a QC is a maximal one is also NP-hard [35]. Third, QCs do not satisfy the *hereditary property* (since a subgraph of a QC is not always a QC). As a result, many advanced techniques that have been developed for enumerating subgraphs that satisfy the hereditary property (e.g., k-plexes, s-defective cliques, etc.) cannot be utilized for this problem [10, 48, 52]. One common practice of solving the MQCE problem involves two steps: (1) it finds a set of QCs that contains all MQCs, which may involve non-maximal QCs; (2) it filters out non-maximal QCs from those QCs found in the first step [19, 20, 28]. This is mainly because checking whether a QC is maximal directly is NP-hard [35]. Therefore, we decompose the MQCE problem into two sub-problems, namely MQCE-S1 and MQCE-S2, each for a step involved in solving the MQCE problem. Existing studies [19, 20, 28] usually focus on the MQCE-S1 problem since the MQCE-S2 problem can be solved efficiently with existing techniques for the *set containment query*, for which there exists a rich literature [4, 9, 33, 36, 37].

Quite a few algorithms have been developed for the MQCE-S1 problem, including `Crochet` [23, 32], `Cocain` [51], `Quick` [28] and `Quick+` [19, 24]. They all correspond to *branch-and-bound* (BB) algorithms. Specifically, they recursively partition the search space (i.e., the set of all possible vertex sets) to multiple sub-spaces with a *branching* method - each sub-space corresponds to a *branch* and develop techniques for pruning some branches that hold no MQCs. These algorithms share the branching method, which is the one behind a classic *set-enumeration* (SE) tree and thus we call it the *SE branching* method. They differ in their pruning techniques. One insufficiency that is suffered by all existing methods [19, 23, 24, 28, 32, 51] is that they devote little effort to improving the branching part, i.e., they uniformly adopt the SE branching method, and focus *solely* on sharpening the pruning techniques. As a result, the pruning part and the branching part are often not well optimized *jointly* towards the goal of pruning as many branches as possible. In fact, none of these methods can provide theoretical guarantee on pruning branches. This is reflected by the fact all of them have the worst-case time complexity of $O^*(2^n)$, where O^* suppresses the polynomials and n denotes the number of vertices of the graph.

New Methods. In this paper, we focus on the MQCE-S1 problem but deviate from the direction of sharpening pruning techniques further while adopting the SE branching method as existing studies all pursue [19, 23, 24, 28, 32, 51]. We aim to develop new pruning techniques and branching methods that would suit each other better towards pruning more branches *both theoretically and practically*. Specifically, we first develop a pruning technique, which is based on a necessary condition for a branch to hold QCs (i.e., if a branch does not satisfy the condition, we can prune the branch). One nice property of the pruning technique is that if a branch with a *partial set* S can be pruned, then all other branches with the partial sets as *supersets* of S can also be pruned. Here, a partial set of a branch means the set of vertices that are included in all vertex sets under this branch. To fully unleash the power of this new pruning technique, we adopt a branching method that is *symmetric* to the SE branching method. We call this new branching method *Sym-SE branching*. Given a current branch, Sym-SE branching would create a series of branches such that the following branches have their partial sets as *supersets* of those of the preceding branches. Therefore, once we find that a branch can be pruned by our new pruning technique, all branches that follow this branch in the series can be pruned as well. We further observe that SE branching and Sym-SE branching can be jointly applied in certain cases so that more branches can be pruned. We call the resulting branching the *Hybrid-SE branching* method. We show that a BB algorithm that is based on our newly developed pruning technique and branching methods, which we call `FastQC`, would have a worst-case time complexity of $O(n \cdot d \cdot \alpha_k^n)$ (i.e., $O^*(\alpha_k^n)$) where d is the maximum degree of a vertex and α_k is strictly smaller than 2 and depends on the value of k, e.g., $\alpha_k = 1.769$ when $k = 2$.

In addition, we adapt a *divide-and-conquer (DC)* strategy for boosting the efficiency and scalability of `FastQC`. Basically, it divides the whole graph into multiple smaller ones and then runs `FastQC` on each of them. Furthermore, we develop some new pruning techniques to shrink the constructed smaller graphs for better efficiency. In summary, the resulting algorithm called `DCFastQC` would invoke `FastQC` multiple times, each on a smaller graph (compared with the original graph), and thus the scalability is improved. We note that this DC strategy has been widely used for enumerating subgraphs [19, 24, 48, 52]. Our technique differs from existing ones in (1) the way of how a graph is divided [19, 24]; and/or (2) the techniques for shrinking the smaller graphs [19, 24, 48, 52].

Contributions. Our contributions are summarized below.

- We propose a new BB algorithm called `FastQC` for the MQCE-S1 problem, i.e., the problem of finding a set of QCs containing all MQCs, which is based on our newly developed pruning technique and branching methods. `FastQC` has the worst-case time complexity of $O(n \cdot d \cdot \alpha_k^n)$ with $\alpha_k < 2$, which breaks the long-standing bottleneck time complexity of $O^*(2^n)$ [1]. (Section 4)
- We further introduce a divide-and-conquer strategy, called `DC`, for boosting the performance of `FastQC`. When applying `DC` to `FastQC`, the worst-case time complexity becomes $O(n \cdot \omega d^2 \cdot \alpha_k^{\omega d})$ where ω is the degeneracy of the given graph. This is better than that of `FastQC` on certain real-world graphs (e.g., those sparse graphs with $\omega << n$ or $d << n$). (Section 5)
- We conduct extensive experiments on both real and synthetic datasets to evaluate the efficiency and scalability of our algorithms, e.g., `DCFastQC` is up to two orders of magnitude faster than the state-of-the-art `Quick+` on real datasets. (Section 6)

For the rest of the paper, we review the problem in Section 2, review the state-of-the-art algorithm `Quick+` in Section 3, review the related work in Section 7 and conclude the paper in Section 8.

2 PROBLEMS

2.1 Problem Definition

We consider an undirected and unweighted graph $G = (V, E)$, where V and E are sets of vertices and edges respectively. Let n be the number of vertices, i.e., $n = |V|$. Given $H \subseteq V$, we use $G[H]$ to denote the subgraph of G induced by H, i.e., $G[H]$ includes the set of vertices H and the set of edges $\{(u, v) \in E \mid u, v \in H\}$. All subgraphs considered in this paper are *induced* subgraphs.

Given $v \in V$, we let $\Gamma(v, V)$ (resp. $\overline{\Gamma}(v, V)$) denote the set of neighbours (resp. non-neighbours) of v in V, i.e., $\Gamma(v, V) = \{u \in V \mid (u, v) \in E\}$ (resp. $\overline{\Gamma}(v, V) = \{u \in V \mid (u, v) \notin E\}$). We further define $\delta(v, V) = |\Gamma(v, V)|$ and $\overline{\delta}(v, V) = |\overline{\Gamma}(v, V)|$. We denote by d the maximum degree of a vertex in G.

[1]We note that there has been some recent progress of improving the time complexity for enumerating some subgraphs that satisfy the hereditary property (e.g., k-plex and s-defective clique) [10, 52], but they cannot be used for our problem since QCs do not satisfy the property. To our best knowledge, this is the first breakthrough of worst-case time complexity of enumerating subgraphs that do not satisfy the hereditary property.

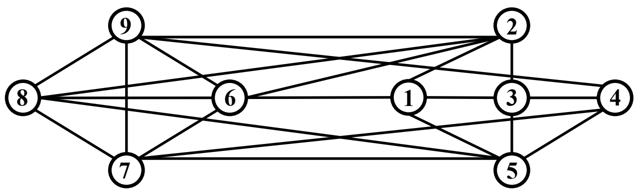


Fig. 1. Input graph used throughout the paper (number i represents vertex v_i)

We then revisit the definition of γ-quasi-clique [23, 24, 28, 32, 51].

Definition 1 (γ-quasi-clique [32]). *Given $H \subseteq V$ and a fraction $0 \le \gamma \le 1$, an induced subgraph $G[H]$ is said to be a γ-quasi-clique if and only if (1) $G[H]$ is connected and (2) for any vertex $v \in H$, it connects at least a fraction γ of the vertices in H (excluding v), i.e., $\delta(v, H) \ge \lceil \gamma \cdot (|H| - 1) \rceil$.*

In particular, a 1-quasi-clique would reduce to a clique. Besides, γ-quasi-clique has the following two properties.

Property 1 (Non-hereditary). *For a γ-quasi-clique $G[H]$, a subgraph of $G[H]$ might not be a γ-quasi-clique.*

This can be easily verified by an example in Figure 1 where $G[\{v_1, v_3, v_4, v_5\}]$ is a 0.6-QC since each vertex connects at least 2 out of 3 other vertices whereas a subgraph $G[\{v_1, v_3, v_4\}]$ is not.

Property 2 (2-diameter [32]). *For $\gamma \ge 0.5$, the diameter of a γ-quasi-clique is at most 2.*

Following [32], we focus on those γ-quasi-cliques with $\gamma \ge 0.5$ only in this paper. This is because for a smaller value of γ, there exist numerous γ-quasi-cliques yet the majority of them are of small size and not cohesive [19, 24, 32]. Moreover, prior studies [19, 24, 28, 51] often focus on a compact representation of the set of γ-quasi-cliques, namely the set of *maximal* γ-quasi-cliques.

Definition 2 (Maximal γ-quasi-clique). *A γ-quasi-clique $G[H]$ is said to be maximal if and only if there is no other γ-quasi-clique $G[H']$ containing $G[H]$, i.e., $H \subseteq H'$.*

In this paper, we use QC (resp. MQC) as a shorthand of γ-quasi-clique (resp. maximal γ-quasi-clique) when the context is clear. Following [19, 24, 28, 32], we consider a size threshold θ for each MQC $G[H]$ to be enumerated, namely $|H| \ge \theta$, since small MQCs are numerous and not statistically significant for real applications [19, 24, 32]. Finally, we formalize the problem of enumerating MQCs with the size at least a threshold, which we call *large* MQCs.

Problem 1 (Maximal γ-quasi-clique Enumeration [19, 24, 28, 32]). *Given a graph $G = (V, E)$, a fraction threshold $\gamma \in [0.5, 1]$ and a positive integer size threshold θ, the Maximal γ-Quasi-Clique Enumeration (MQCE) Problem aims to find all MQCs $G[H]$ with $|H| \ge \theta$.*

NP-hardness. The MQCE problem is NP-hard since the optimization problem of finding the MQC with the largest number of vertices is NP-hard [30]. Note that the optimization problem can be solved by enumerating all MQCs and returning the largest one. Furthermore, determining whether a QC is maximal is NP-hard [35]. In contrast, determining if a structure that satisfies the hereditary property, e.g., a clique [6], is maximal can usually be done in polynomial.

2.2 Problem Decomposition

One common practice of enumerating large maximal QCs is to (1) find a set of QCs that contains all maximal QCs, which may involve non-maximal QCs, and then (2) filter out non-maximal QCs with a post-processing procedure [19, 20, 28]. This is mainly because checking whether a QC is maximal directly is NP-hard [35]. Therefore, we decompose the MQCE problem into two sub-problems, namely MQCE-S1 and MQCE-S2, each for a step involved in solving the MQCE problem, as follows.

Sub-Problem 1 (MQCE-S1). *Given a graph $G = (V, E)$, a fraction threshold $\gamma \in [0.5, 1]$ and a positive integer size threshold θ, the MQCE-S1 problem is to find a set of QCs that contains all MQCs $G[H]$ with $|H| \geq \theta$.*

Sub-Problem 2 (MQCE-S2). *Given a set $\mathcal{S}$ of QCs, the MQCE-S2 problem is to filter out those that are subsets of others in $\mathcal{S}$ and then return the remaining QCs.*

The MQCE-S1 problem is *NP-hard* since the problem of finding the largest MQC (which is NP-hard [30]) can be solved by finding a set of QCs containing all MQCs and returning the largest one. For the MQCE-S2 problem, we note that it is different from the problem of determining whether a QC is maximal (which is NP-hard [35]). For the former, the input is a set of QCs only. For the latter, the inputs include a QC and an input graph and the problem is to check whether there exists a superset of the QC in the input graph, which is also a QC. In fact, the MQCE-S2 problem can be solved in *polynomial time* with respect to the size of input, which we explain as follow.

The MQCE-S2 problem is closely related to the *set containment query*, which is a fundamental problem in both database systems and theory of computer science [4, 9, 33, 36, 37]. Given a set of sets $\mathcal{S}$ and a query set H of symbols from some alphabet, one type of set containment query called `GetAllSubsets` is to find all subsets of H from $\mathcal{S}$. The state-of-the-art algorithm for `GetAllSubsets` can answer the query in $O(\min\{|\mathcal{S}| \cdot |H|, 2^{|H|}\})$ time with a set-trie data structure that can be built in $O(|\mathcal{S}| \cdot |H_{max}|)$ time, where H_{max} is the largest set in $\mathcal{S}$ [37].

Specifically, the MQCE-S2 problem can be solved by iteratively issuing a `GetAllSubsets` query for a QC H in $\mathcal{S}$ and removing the found QCs from $\mathcal{S}$, which has been adopted by existing studies of enumerating MQCs [19, 24, 28]. Consequently, the time complexity of this method is $O(\min\{|\mathcal{S}|^2 \cdot \omega,\ |\mathcal{S}| \cdot 2^{2\omega}\})$ (which is polynomial wrt $|\mathcal{S}|$), where ω is the degeneracy of the input graph G. Note that the size of a γ-QC H, i.e., $|H|$, is at most $2\omega + 1$ for $\gamma \geq 0.5$ [47] and the cost of constructing the set-trie structure is dominated by that of issuing the `GetAllSubsets` query $O(|\mathcal{S}|)$ times.

We note that the time cost for solving the MQCE-S2 problem with the aforementioned method is typically small in practice due to the following reasons: (1) we are usually interested in large MQCs only and there are usually not many large QCs, i.e., $|\mathcal{S}|$ is usually small (see the experimental results in Table 1); and (2) we usually have $\omega << n$ for the most real datasets which are sparse (see the experimental results in Table 1). For example, the time cost of solving MQCE-S2 is within 0.1s for the majority of datasets and within 10s on all datasets we have used (as shown in the technical report [47]). Therefore, in this paper, we focus on the MQCE-S1 problem, i.e., the one of finding a set of QCs containing all maximal QCs, as existing studies [19, 24, 28] did.

3 THE STATE-OF-THE-ART BRANCH-AND-BOUND ALGORITHM: `QUICK+`

In this part, we establish necessary background of branch-and-bound (BB) algorithms for the MQCE-S1 problem, i.e., the one of finding the set containing all MQCs by reviewing the state-of-the-art BB algorithm, namely `Quick+`. Specifically, `Quick+` recursively partitions the search space (i.e., the set of possible vertex sets) to multiple sub-spaces via *branching*. Each sub-space, which corresponds to a *branch*, is represented by a triple of three vertex sets (S, C, D) explained as follows.

- **Partial set** S. Set of vertices that *must* be included in every vertex set within the branch.
- **Candidate set** C. Set of vertices that *may* be included in S in order to form larger vertex sets within the branch.
- **Exclusion set** D. Set of vertices that *must not* be included in any vertex set within the branch.

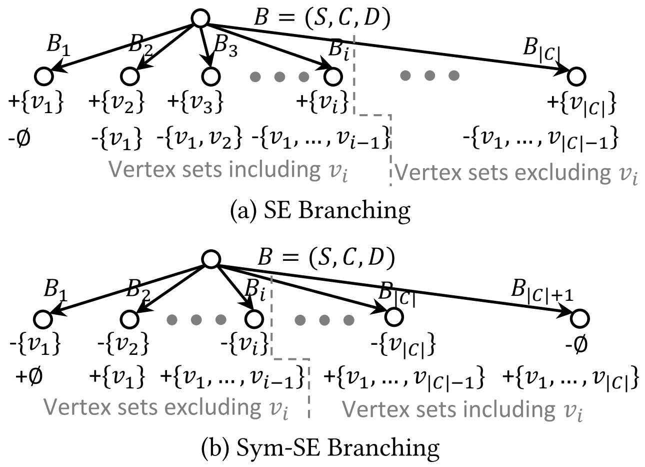

$B = (S, C, D)$
B_1
B_2
B_3
B_i
$B_{|C|}$
$+\{v_1\}$ $-\emptyset$
$+\{v_2\}$ $-\{v_1\}$
$+\{v_3\}$ $-\{v_1, v_2\}$
$+\{v_i\}$ $-\{v_1, ..., v_{i-1}\}$
$+\{v_{|C|}\}$ $-\{v_1, ..., v_{|C|-1}\}$
Vertex sets including v_i
Vertex sets excluding v_i
(a) SE Branching
$B = (S, C, D)$
B_1
B_2
B_i
$B_{|C|}$
$B_{|C|+1}$
$-\{v_1\}$ $+\emptyset$
$-\{v_2\}$ $+\{v_1\}$
$-\{v_i\}$ $+\{v_1, ..., v_{i-1}\}$
$-\{v_{|C|}\}$ $+\{v_1, ..., v_{|C|-1}\}$
$-\emptyset$ $+\{v_1, ..., v_{|C|}\}$
Vertex sets excluding v_i
Vertex sets including v_i
(b) Sym-SE Branching

Fig. 2. Illustration of SE branching and Sym-SE branching ("+" means to *include* a set of vertices, i.e., the set S is expanded with these vertices; "-" means to *exclude* a set of vertices, i.e., the set D is expanded with these vertices)

That is, each branch (S, C, D) covers all those vertex sets that (1) include S and (2) are subgraphs of $G[S \cup C]$.

Specifically, Quick+ starts from the universal search space (S, C, D) with $S = \emptyset$, $C = V$, and $D = \emptyset$, and recursively creates branches as follows. Consider a current branch $B = (S, C, D)$ with $C = \{v_1, v_2, ..., v_{|C|}\}$. It creates $|C|$ branches, denoted by $B_i = (S_i, C_i, D_i)$ for $1 \leq i \leq |C|$, from branch B. Branch B_i covers all vertex sets that include $S \cup \{v_i\}$ and exclude $D \cup \{v_1, ..., v_{i-1}\}$. Formally, for $1 \leq i \leq |C|$, we have

$$S_i = S \cup \{v_i\};\ D_i = D \cup \{v_1, v_2, ..., v_{i-1}\};\ C_i = C - \{v_1, v_2, ..., v_i\} \tag{1}$$

We call this branching the *SE branching*, as illustrated in Figure 2(a).

We note that SE branching and the existing branching strategy adopted by the Bron-Kerbosch (BK) algorithm [6], which we call *BK branching*, share the way of forming the branches. The difference is that BK branching is used for enumerating maximal subgraph structures that satisfy the hereditary property (e.g., cliques). Specifically, it would further prune some of the formed branches based on the hereditary property. In contrast, SE branching does not require the subgraphs to be enumerated to satisfy the hereditary property - this is why it is adopted by Quick+ for enumerating MQCs, and it cannot prune some formed branches as BK branching does.

During the recursive branching process, Quick+ applies two types of pruning techniques, namely Type I pruning rules and Type II pruning rules. Intuitively, Type I pruning rules are conducted on C and aim to refine C by removing those vertices that satisfy certain conditions; Type II pruning rules are conducted on S and aim to prune those branches where vertices in S satisfy certain conditions. The rationale behind is that if a vertex v satisfies certain conditions, each MQC covered by this branch does not include this vertex, and thus we can either remove v from C for this branch, i.e., Type I pruning rules apply (if $v \in C$), or prune the entire branch, i.e., Type II pruning rules apply (if $v \in S$). For simplicity, we omit the details of these pruning techniques and refer them to [24].

We finally summarize Quick+ in Algorithm 1. Specifically, it starts from the branch $(B, C, D) = (\emptyset, V, \emptyset)$ by calling a recursive procedure called Quick-Rec (line 1), recursively creates branches via SE branching (line 7), and conducts the aforementioned pruning operations (line 9-10). In particular, it terminates the branch once $C = \emptyset$, and outputs the partial set $G[S]$ only if $G[S]$ is a QC and

Algorithm 1: An existing branch-and-bound algorithm: Quick+ [24]

```
   Input: A graph $G = (V, E)$, $0.5 \le \gamma \le 1$, and $\theta > 0$
   Output: A set of QCs that includes all MQCs
1  Quick-Rec($\emptyset, V, \emptyset$);
2  Procedure Quick-Rec($S, C, D$)
       /* Termination                                              */
3      if $C = \emptyset$ then
4          if $G[S]$ is a QC then
5              Output $G[S]$ if $|S| \ge \theta$; return true;
6          return false;
       /* SE Branching                                             */
7      Create $|C|$ branches $B_i = (S_i, C_i, D_i)$ based on Equation (1);
8      for each branch $B_i$ do
           /* Pruning before the next recursion                    */
9          $C_i' \leftarrow$ Type I pruning rules on $C_i$;
10         if any of Type II pruning on $S_i$ is triggered then continue;
11         $\mathcal{T}_i \leftarrow$ Quick-Rec($S_i, C_i', D_i$);
       /* Additional step: output $G[S]$ if necessary              */
12     if all of $\mathcal{T}_i$ are false then
13         if $G[S]$ is a QC then
14             Output $G[S]$ if $|S| \ge \theta$; return true;
15         return false;
16     return true;
```

$|S| \ge \theta$ (line 4-5). We remark that Quick+ does not check whether an output QC is maximal or not (mainly due to its NP-hardness). Therefore, it would return a superset of all MQCs which inevitably contains some non-maximal QCs, i.e., it solves the MQCE-S1 problem, but not the MQCE problem.

Besides, we note that for a branch (S, C, D), $G[S]$ could be a MQC even if no QCs are found in the created sub-branches due to the non-hereditary property of QC. Therefore, Quick-Rec monitors whether a sub-branch of the current one would find a QC. If so, it returns true (e.g., line 5, line 14 and line 16); if not, it returns false (e.g., line 6 and line 15). In the case that a QC is found in a sub-branch, the QC should be a superset of $G[S]$, i.e., $G[S]$ cannot be a MQC, and therefore, there is no need to consider $G[S]$. In the other case that no QCs are found in any of the sub-branches (line 12), it checks if $G[S]$ is a large QC and outputs it if so (line 13-14).

Time Complexity. Quick+ would explore $O(2^n)$ branches in the worst case, though some pruning rules are applied to boost its performance in practice. Hence, the worst-case time complexity is $O^*(2^n)$, where O^* suppresses the polynomials [24].

4 A NEW BRANCH-AND-BOUND ALGORITHM: FASTQC

In this section, we introduce our new branch-and-bound (BB) algorithm called FastQC. First, we develop a new pruning technique, which is based on a necessary condition for a branch to hold QCs, i.e., a vertex set within the branch corresponds to a QC (Section 4.1), and introduce a method to apply the pruning technique in a *progressive* fashion by refining a branch and re-checking the necessary condition iteratively (Section 4.2). Second, we observe that the pruning technique has a nice property that if a branch with a partial set S can be pruned, then any branch with the partial set

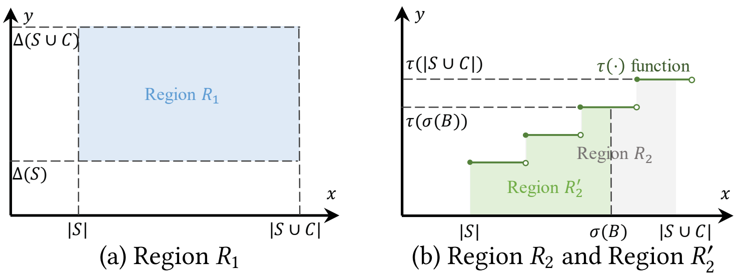


Fig. 3. Illustration of Condition C1 (Region R_1) and Condition C2 (Region R_2 and Region R_2') in the SD space

as a *superset* of S can be pruned as well. To better utilize this property, we adopt a new branching method that is *symmetric* to SE branching, which we call *Sym-SE branching* (Section 4.3). The rationale is that Sym-SE branching would produce a series of branches such that the following branches have their partial sets as *supersets* of those of preceding ones. As a result, if we find a branch that can be pruned, we can prune all branches following this branch in the series safely. Third, we observe that in certain cases, SE branching and Sym-SE branching can be *jointly* applied so that more branches can be pruned. We call the resulting branching method the *Hybrid-SE branching* and present it in Section 4.4. Finally, we summarize the `FastQC` algorithm, which is a BB algorithm based on the newly developed pruning techniques and branching methods and analyze its time complexity in Section 4.5. In particular, `FastQC` has the worst-case time $O(n \cdot d \cdot \alpha_k^n)$ with $\alpha_k < 2$.

4.1 A Novel Necessary Condition for a Branch To Hold QCs

Consider a branch $B = (S, C, D)$. We aim to find an easily tractable *necessary condition for a current branch to hold QCs* [2]. Then, for those branches that violate the condition, they hold no QCs and thus can be pruned safely. We will show that with this pruning technique employed (and some branching method designed accordingly), we would need to explore strictly fewer than $O(2^n)$ branches in theory. Below, we give the details of the condition.

Let H be a set of vertices. We define $\Delta(H)$ to be the maximum number of disconnections of a vertex in H within $G[H]$. Formally, we have

$$\Delta(H) = \max_{v \in H} \overline{\delta}(v, H). \tag{2}$$

Consider a subset H_{sub} and a superset H_{sup} of H, we have

$$\Delta(H_{sub}) \le \Delta(H) \le \Delta(H_{sup}) \text{ for } H_{sub} \subseteq H \subseteq H_{sup}. \tag{3}$$

This can be verified by the fact that the set of disconnections within a subgraph $G[H_{sub}]$ (resp. a supergraph $G[H_{sup}]$) is always a subset (resp. a superset) of that within $G[H]$.

Given a graph $G[H]$, we map it to a 2-dimensional space at the point $(|H|, \Delta(H))$. We call this space the size disconnection space (SD space). We note that a point (x, y) in the SD space corresponds to a set of possible graphs $G[H]$ with $|H| = x$ and $\Delta(H) = y$. Note that we can focus on the first quadrant of the SD space, namely $x \ge 0$ and $y \ge 0$.

With the defined SD space, we proceed to introduce two necessary conditions for a branch B to hold QCs, namely C1 and C2.

[2]We note that the problem of determining whether branch B holds a QC (or formally, whether one of the partial sets of the branches under B induces a QC) is hard. In fact, we prove that this problem is NP-hard and for simplicity, we put the proof in the technical report [47] .

Condition C1. For a QC $G[H]$ under the branch B, its point in the SD space must reside in a rectangular region R_1 defined as follows.

$$\text{Region } R_1\text{: } |S| \le x \le |S \cup C| \text{ and } \Delta(S) \le y \le \Delta(S \cup C) \tag{4}$$

This is because $S \subseteq H \subseteq S \cup C$ and thus $\Delta(S) \le \Delta(H) \le \Delta(S \cup C)$ according to Equation (3). An illustration of Region R_1 is shown in Figure 3 (a) (the blue region). Correspondingly, we obtain the following necessary condition.

Condition C1: If a branch B holds a QC $G[H]$, then the point of $G[H]$ in the SD space resides in Region R_1.

Condition C2. Recall that for a QC $G[H]$, each vertex v in $G[H]$ has the number of its connections within $G[H]$ *at least* $\lceil \gamma \cdot (|H| - 1) \rceil$ by definition, i.e.,

$$\forall v,\ \delta(v, H) \ge \lceil \gamma \cdot (|H| - 1) \rceil \tag{5}$$

Equivalently, each vertex v has the number of its disconnections within $G[H]$, which is equal to $|H| - \delta(v, H)$, *at most* $|H| - \lceil \gamma \cdot (|H| - 1) \rceil$, i.e.,

$$\forall v,\ \overline{\delta}(v, H) \le |H| - \lceil \gamma \cdot (|H| - 1) \rceil = \lfloor (1 - \gamma) \cdot |H| + \gamma \rfloor \tag{6}$$

Equation (6) implies that the maximum number of disconnections of a vertex in $G[H]$ is also *at most* $\lfloor (1 - \gamma) \cdot |H| + \gamma \rfloor$, i.e.,

$$\Delta(H) \le \lfloor (1 - \gamma) \cdot |H| + \gamma \rfloor \tag{7}$$

Equation (7) essentially says that the point of any QC $G[H]$ in the SD space is *below* the curve representing the following function $\tau(x)$ inclusively.

$$\tau(x) := \lfloor (1 - \gamma) \cdot x + \gamma \rfloor \tag{8}$$

We note that $\tau(x)$ corresponds to a piece-wise and non-decreasing function and each piece is a horizontal line segment with the left endpoint being included and the right endpoint being excluded. An illustration of $\tau(x)$ is in Figure 3(b). Based on Equation (7) and Equation (8), we deduce the following lemma.

Lemma 1. *A graph $G[H]$ is a QC iff $\Delta(H) \le \tau(|H|)$.*

Based on Lemma 1 and the fact that all partial sets within B have the size between $|S|$ and $|S \cup C|$, we deduce that the points of those QCs within B (if any) must reside in a region R_2 as defined below.

$$\text{Region } R_2\text{: } |S| \le x \le |S \cup C| \text{ and } 0 \le y \le \tau(x). \tag{9}$$

An illustration of Region R_2 is shown in Figure 3(b) (the grey and green region).

We note that the upper bound $|S \cup C|$ of the size of a QC under branch B can often be loose. We therefore tighten it to be $\sigma(B)$, which is defined as follows.

$$\sigma(B) = \begin{cases} |S \cup C| & S = \emptyset \\ \min\{|S \cup C|, d_{min}(B)/\gamma + 1\} & S \ne \emptyset \end{cases} \tag{10}$$

where $d_{min}(B)$ is the minimum degree of a vertex in S within $G[S \cup C]$. That is,

$$d_{min}(B) = \min_{v \in S} \delta(v, S \cup C) \tag{11}$$

We verify that for a QC $G[H]$ under branch B (if any), we have $|H| \le \sigma(B)$, which we formally present in the following lemma (the proof is put in the technical report [47] for simplicity).

Lemma 2. *For any QC $G[H]$ under branch B, we have $|H| \le \sigma(B)$.*

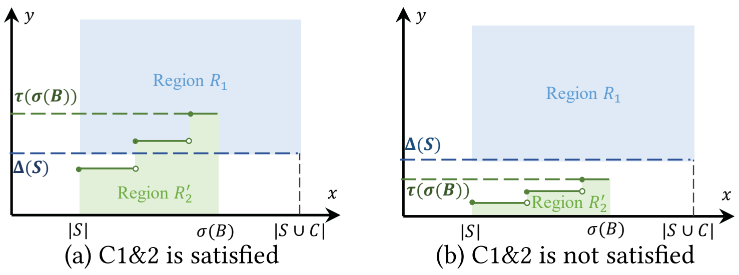


Fig. 4. Illustration of the necessary condition in the SD space

Based on Lemma 2, we obtain a region R_2' as defined below, which covers all possible QCs under branch B and is narrower than R_2.

$$\text{Region } R_2'\text{: } |S| \leq x \leq \sigma(B) \text{ and } 0 \leq y \leq \tau(x). \tag{12}$$

An illustration of Region R_2' is shown in Figure 3(b) (the green region). Correspondingly, we obtain the following necessary condition.

Condition C2: If a branch B holds a QC $G[H]$, then the point of $G[H]$ in the SD space resides in Region R_2'.

We note that in the case that $\sigma(B) < |S|$, it means that Region R_2' is empty, which implies that Condition C2 is not satisfied and there exist no QCs under the branch B.

Necessary Condition (Summary). In summary, if a branch B holds a QC $G[H]$, then the point of $G[H]$ in the SD space must reside in both Region R_1 and Region R_2'. It further implies that the intersection of the two regions, which we denote by $R_{1\&2} = R_1 \cap R_2'$, is non-empty. We use this as the necessary condition for branch B to hold a QC and shall show that it can be verified efficiently in $O(d)$ time. Specifically, we have the following necessary condition.

Condition C1&2: If a branch B holds a QC, then $R_{1\&2} = R_1 \cap R_2'$ is non-empty.

For illustration, we show the case where the necessary condition C1&2 is satisfied in Figure 4(a) and the case where C1&2 is not satisfied in Figure 4(b).

We notice that the necessary condition C1&2, i.e., Region $R_{1\&2}$ is non-empty, is equivalent to that $\Delta(S) \leq \tau(\sigma(B))$. This is because when $\Delta(S) \leq \tau(\sigma(B))$, regions R_1 and R_2' would intersect and when $\Delta(S) > \tau(\sigma(B))$, the regions would not intersect, and vice versa, as illustrated in Figure 4. Correspondingly, we have the following equivalent necessary condition.

Condition C1&2 (equivalent): If a branch B holds a QC, then $\Delta(S) \leq \tau(\sigma(B))$.

Time Complexity of Checking the Condition C1&2. The cost is dominated by the that of computing $\Delta(S)$ and $d_{min}(B)$. First, we can maintain two arrays to record the degree of each vertex v within $G[S]$ (i.e., $\delta(v, S)$) and that within $G[S \cup C]$ (i.e., $\delta(v, S \cup C)$). The maintenance cost is $O(d)$ since when forming a branch B_i by including a vertex v_i to S (recall Equation (1)), we only need to update $\delta(\cdot, S)$ and $\delta(\cdot, S \cup C)$ for v_i's neighbours and there are $O(d)$ neighbors). Second, we can compute $\Delta(S)$ by scanning the vertices in S and their degrees within $G[S]$ (which have been maintained), and the cost is $O(|S|)$. Similarly, we can compute $d_{min}(B)$ by scanning the vertices in S and their degrees within $G[S \cup C]$ (which have been maintained), and the cost is $O(|S|)$. Third, for a non-empty S, we have $|S| \leq \sigma(B)$ based on Lemma 2 and $\sigma(B) \leq d/\gamma + 1 \leq 2d + 1$ based on Equation (10) and $\gamma \geq 0.5$. In summary, the cost of checking C1&2 is $O(d) + O(|S|) + O(|S|) = O(d)$.

4.2 Progressively Refining a Branch and Re-Checking the Necessary Condition

Consider a branch $B = (S, C, D)$. We first check if the necessary condition C1&2, i.e., $\Delta(S) \leq \tau(\sigma(B))$, is satisfied. If no, we can prune B directly; If yes, while we cannot prune B immediately, we may be able to *refine* B with the information $\tau(\sigma(B))$ by removing some vertices from C. We then *re-check* the condition for the refined branch, which we denote by B', and prune it if the condition is not satisfied. The rationale is that with some vertices removed from the candidate set C, the necessary condition would become less likely to be satisfied and correspondingly the branch can be pruned more likely, as will be explained later. In fact, we can repeat this process until either (1) the branch is pruned; or (2) the branch cannot be refined further. We provide the details as follows.

Refining a Branch. With the information of $\tau(\sigma(B))$, we can possibly refine B by removing from C some vertices as follows.

- **Refinement Rule 1.** Remove from C those vertices v with $\Delta(S \cup \{v\}) > \tau(\sigma(B))$
- **Refinement Rule 2.** Remove from C those vertices v with $\delta(v, S \cup C) < \theta - \tau(\sigma(B))$

For Rule (1), it is because any QC $G[H]$ under branch B cannot hold vertex v since otherwise we deduce that $\Delta(S \cup \{v\}) \leq \Delta(H) \leq \tau(|H|) \leq \tau(\sigma(B))$, which contradicts to $\Delta(S \cup \{v\}) > \tau(\sigma(B))$. Here, $\Delta(S \cup \{v\}) \leq \Delta(H)$ is because $S \cup \{v\} \subseteq H$ (by assumption), $\Delta(H) \leq \tau(|H|)$ is because $G[H]$ is a QC, and $\tau(|H|) \leq \tau(\sigma(B))$ is because $|H| \leq \sigma(B)$ (based on Lemma 2) and $\tau(\cdot)$ is non-decreasing. For Rule (2), it is because only those QCs with the size at least θ are to be found. For each vertex v in such a large QC $G[H]$ under branch B, we have $\delta(v, S \cup C) \geq \delta(v, H) = |H| - \overline{\delta}(v, H) \geq |H| - \Delta(H) \geq |H| - \tau(|H|) \geq |H| - \tau(\sigma(B)) \geq \theta - \tau(\sigma(B))$.

Re-checking the Necessary Condition. Suppose the branch $B = (S, C, D)$ has been refined to $B' = (S, C', D)$ with $C' \subset C$. We note that $\tau(\sigma(B'))$ would be possibly smaller than $\tau(\sigma(B))$. This is because $C' \subset C$ implies $d_{min}(B') \leq d_{min}(B)$, which further implies $\sigma(B') \leq \sigma(B)$, and $\tau(\cdot)$ is non-decreasing. Therefore, we can re-check the necessary condition for branch B', i.e., $\Delta(S) \leq \tau(\sigma(B'))$, which is less likely to be satisfied than that for branch B, i.e., $\Delta(S) \leq \tau(\sigma(B))$, given that $\tau(\sigma(B')) \leq \tau(\sigma(B))$. If the condition is not satisfied, we prune branch B'.

Repeated Process and Stopping Criterion. In the case that the refined branch B' cannot be pruned. We can repeat the process of refining B' (based on the information of $\tau(\sigma(B'))$) and re-checking the condition for the refined branch. We stop the process until either (1) a refined branch is pruned or (2) a branch cannot be refined any further (i.e., no vertices can be removed from the candidate set).

For illustration, consider the problem of finding all 0.7-MQCs in Figure 1 and a branch B with $S = \{v_1, v_3, v_4\}$, $C = \{v_2, v_5, v_6, v_7, v_8, v_9\}$ and $D = \emptyset$. First, we check necessary condition for B. Specifically, we compute $\Delta(S) = \overline{\delta}(v_1, S) = |\{v_1, v_4\}| = 2$, $\sigma(B) = \min\{9, 4/0.7 + 1\} = 6.71$ and $\tau(\sigma(B)) = \tau(6.71) = \lfloor 0.3 \times 6.71 + 0.7 \rfloor = 2$. Since $\Delta(S) = 2 \leq \tau(\sigma(B)) = 2$, i.e., the necessary condition for B is satisfied, we cannot prune B. Then, we refine B by removing vertices v_6, v_7, v_8, v_9 from C since for each such vertex v, we have $\Delta(S \cup \{v\}) > \tau(\sigma(B)) = 2$ (i.e., Refinement Rule 1 applies). We denote the refined branch as B' and re-check the necessary condition for B'. Specifically, we compute $\sigma(B') = \min\{5, 2/0.7 + 1\} = 3.85$ and $\tau(\sigma(B')) = \tau(3.85) = 1$. Since $\Delta(S) = 2 > \tau(\sigma(B')) = 1$, i.e., the necessary condition for B' is not satisfied, we can safely prune B' and stop the process.

4.3 A Symmetric Branching Strategy of SE Branching: Sym-SE Branching

Consider a branch $B = (S, C, D)$, which satisfies the necessary condition C1&2, i.e., $\Delta(S) \leq \tau(\sigma(B))$, after the the progressive process of refining a branch and re-checking the necessary condition. We need to create sub-branches from branch B via branching. One option is to adopt the SE branching,

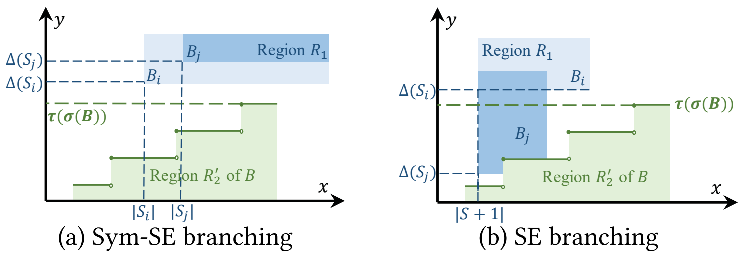


Fig. 5. Illustration of branching at B in the SD space

which is defined in Equation (1), as Quick+ does. Another option is to adopt a branching that is *symmetric* to the SE branching, called *Sym-SE branching*, which is defined as follows. It creates $(|C|+1)$ sub-branches from B, namely $B_1, B_2, ..., B_{|C|+1}$. Branch B_i $(1 \le i \le |C|+1)$ covers those vertex sets, each (1) including all vertices in $S \cup \{v_1, v_2, ..., v_{i-1}\}$; (2) excluding all vertices in $D \cup \{v_i\}$. Formally, branch B_i is defined as follows.

$$S_i = S \cup \{v_1, v_2, ..., v_{i-1}\},\ D_i = D \cup \{v_i\},\ C_i = C - \{v_1, v_2, ..., v_i\}. \tag{13}$$

Here, v_0 in branch B_1 and $v_{|C|+1}$ in branch $B_{|C|+1}$ are both fictitious. An illustration of the Sym-SE branching is shown in Figure 2(b). The two branching methods are symmetric because as shown in Figure 2, (1) for SE branching, branches B_1, ..., B_i *include* vertex v_i while the remaining branches *exclude* vertex v_i and (2) for Sym-SE branching, branches B_1, ..., B_i *exclude* vertex v_i while the remaining branches *include* vertex v_i.

We note that a symmetric branching of the BK branching, which is called *Sym-BK branching*, has also been explored for enumerating maximal subgraphs that satisfy the hereditary property [48, 52]. Sym-SE branching and Sym-BK branching share the way of forming branches. The difference is that Sym-BK branching can prune some of the formed branches based on the hereditary property. In contrast, Sym-SE does not require the subgraphs to be enumerated to satisfy the hereditary property, and correspondingly it cannot prune some formed branches as Sym-BK does.

We show that the necessary condition C1&2 defined in the SD space would work more effectively when the Sym-SE branching is used than when the SE branching is used. The reason is two-fold. First, a created branch B_i by Sym-SE branching would have a larger chance to violate the necessary condition C1&2, i.e., $\Delta(S_i) > \tau(\sigma(B))$ [3], and be pruned, when i gets larger. This is due to the fact that S_i involves $|S|+i-1$ vertices, and correspondingly we have $\Delta(S_i)$ increase with i. Second, if a branch B_i by Sym-SE branching violates the necessary condition (i.e., $\Delta(S_i) > \tau(\sigma(B))$) and can be pruned, then all other branches B_j following B_i with $j > i$ would violate the necessary condition (i.e., $\Delta(S_j) > \tau(\sigma(B))$) and can be pruned also. This is due to the fact that the partial set S_j is always a superset of S_i and thus we have $\Delta(S_j) \ge \Delta(S_i) > \tau(\sigma(B))$.

For illustration, consider Figure 5(a) for Sym-SE branching, where when a branch B_i can be pruned, any branch B_j with $j > i$ can also be pruned. Consider Figure 5(b) for SE branching, where a branch B_i can be pruned does not imply that any branch B_j with $j > i$ can be pruned.

Ordering of Vertices in C. Sym-SE branching implicitly uses an ordering of vertices in C, which we specify as follows. We find a smaller subset C' of C and put them before other vertices in the ordering such that including them to the partial set collectively would cause one sub-branch to violate the necessary condition (i.e., $\Delta(S \cup C') > \tau(\sigma(B))$) and thus this branch and also the following sub-branches can be pruned. Specifically, the ordering is defined based on a vertex $\hat{v}$ called

[3]We note that for sub-branches, we use a looser condition than the necessary condition for branch B_i, i.e., $\Delta(S_i) \le \tau(\sigma(B_i))$.

pivot, which is selected from those vertices in $S \cup C$ that have more than $\tau(\sigma(B))$ disconnections among $S \cup C$, i.e., $\overline{\delta}(\hat{v}, S \cup C) > \tau(\sigma(B))$. We define

$$a = \tau(\sigma(B)) - \overline{\delta}(\hat{v}, S) \text{ and } b = \overline{\delta}(\hat{v}, C), \tag{14}$$

where a denotes the largest possible number of vertices that can be included from $\overline{\Gamma}(\hat{v}, C)$ to S without violating the necessary condition. Note that $a < b$ since $b - a = \overline{\delta}(\hat{v}, S \cup C) - \tau(\sigma(B)) > 0$. We put those vertices in $\overline{\Gamma}(\hat{v}, C)$ before others in the ordering. There are two cases.
Case 1: $\hat{v} \in S$. We define the ordering of vertices in C as follows.

$$\langle v_1, v_2, ..., v_b, v_{b+1}, ..., v_{|C|} \rangle, \tag{15}$$

where the first b vertices, namely $v_1, v_2, ..., v_b$, are from $\overline{\Gamma}(\hat{v}, C)$ in any order, and the others are from $\Gamma(\hat{v}, C)$ in any order. Then, branch B_{a+2} would violate the necessary condition since $\Delta(S_{a+2}) \geq \overline{\delta}(\hat{v}, S_{a+2}) = \overline{\delta}(\hat{v}, S \cup \{v_1, v_2, ..., v_{a+1}\}) = \overline{\delta}(\hat{v}, S) + \overline{\delta}(\hat{v}, \{v_1, v_2, ..., v_{a+1}\}) = \overline{\delta}(\hat{v}, S) + a + 1 = \tau(\sigma(B)) + 1$. Consequently, the branches $B_{a+2}, B_{a+3}, ..., B_{|C|+1}$ violate the necessary condition and can be pruned.

For illustration, consider the example of finding all 0.6-MQCs from the graph given by Figure 1 as shown in Figure 6(a). Based on pivot v_1 in S, which disconnects 4 vertices in C, i.e., $\{v_4, v_7, v_8, v_9\}$, we define the ordering $\langle v_4, v_7, v_8, v_9, v_3, v_5, v_6 \rangle$. The branches $B_4, ..., B_9$ can be pruned since B_4 has $S_4 = \{v_1, v_2, v_4, v_7, v_8\}$ and $\Delta(S_4) = 4 > \tau(\sigma(B)) = 3$.
Case 2: $\hat{v} \in C$. We define the ordering of vertices in C as follows.

$$\langle \hat{v}, v_2, v_3, ..., v_b, v_{b+1}, ..., v_{|C|} \rangle, \tag{16}$$

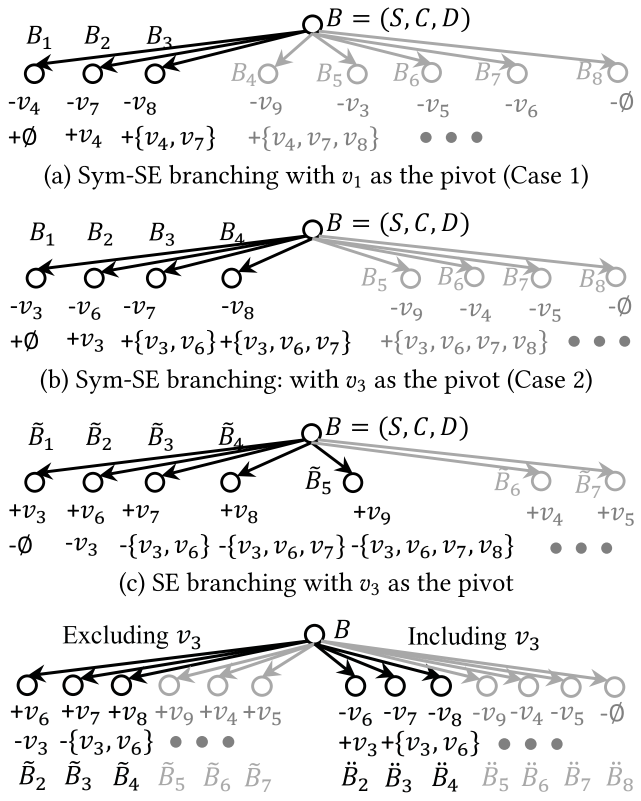
where the first $b + 1$ vertices, namely $\hat{v}, v_2, v_3, ..., v_b$, are from $\overline{\Gamma}(\hat{v}, C)$, and the others are from $\Gamma(\hat{v}, C)$ in any order. Similarly, we only need to keep the first $a + 1$ branches, since branch B_{a+2} would have $\Delta(S_{a+2}) > \tau(\sigma(B))$.

For illustration, consider again the example in Figure 6(b). Based on pivot v_3 in C that disconnects 5 vertices in C, i.e., $\{v_3, v_6, v_7, v_8, v_9\}$, we define the ordering $\langle v_3, v_6, v_7, v_8, v_9, v_4, v_5 \rangle$. The branches $B_5, ..., B_8$ can be pruned since B_5 has $S_5 = \{v_1, v_2, v_3, v_6, v_7, v_8\}$ and $\Delta(S_5) = 4 > \tau(\sigma(B)) = 3$.

Pivot Selection. There could be multiple vertices in $S \cup C$ with more than $\tau(\sigma(B))$ disconnections, which are qualified to be a pivot. We select from them the one with the largest number of disconnections within $S \cup C$, i.e., $\overline{\delta}(\hat{v}, S \cup C)$. We explain this strategy as follows. First, we prune $(|C| + 1) - (a + 1)$ branches, i.e., $B_{a+2}, B_{a+3}, ..., B_{|C|+1}$. Second, we also prune $b - (a + 1)$ vertices from C_{a+1}, i.e., $v_{a+2}, v_{a+3}, ..., v_b$, via Refinement Rule 1 (since including each of them, says v, to S_{a+1} would have $\Delta(S_{a+1} \cup \{v\}) \geq \overline{\delta}(\hat{v}, S_{a+1} \cup \{v\}) = \overline{\delta}(\hat{v}, S) + \overline{\delta}(\hat{v}, \{v_1, ..., v_a, v\}) = \overline{\delta}(\hat{v}, S) + (a + 1) = \tau(\sigma(B)) + 1 > \tau(\sigma(B_{a+1}))$). In summary, we would prune more branches/vertices when a vertex has a *smaller* a and/or a *larger* b. Considering $(b - a) = \overline{\delta}(\hat{v}, S \cup C) - \tau(\sigma(B))$, we select the vertex with the largest $\overline{\delta}(\hat{v}, S \cup C)$ as the pivot. Besides, there exists at least one vertex in $S \cup C$ that has more than $\tau(\sigma(B))$ disconnections (i.e., $\Delta(S \cup C) > \tau(\sigma(B))$) since otherwise the branch holds a QC $G[S \cup C]$ and the branching process can be terminated.

4.4 A Hybrid Branching Strategy: Hybrid-SE Branching

With Sym-SE branching, we can prune those branches that violate the necessary condition (i.e., they hold no QCs). We observe that for those branches that satisfy the necessary condition (i.e., they may hold QCs), some of them may hold *non-maximal* QCs only and thus they can also be pruned. Such a branch would be formed especially after *excluding* a certain set of vertices from the candidate set. Specifically, we consider a scenario, where $B = (S, C, D)$ is a branch and $\hat{v}$ is a vertex in C such that $\overline{\delta}(\hat{v}, S \cup C) > \tau(\sigma(B))$ and $\overline{\delta}(\hat{v}, S) = 0$. We have the following lemma.


$B = (S, C, D)$
B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8
-v_4 -v_7 -v_8 -v_9 -v_3 -v_5 -v_6 -$\emptyset$
+$\emptyset$ +v_4 +$\{v_4, v_7\}$ +$\{v_4, v_7, v_8\}$ ● ● ●
-v_3 -v_6 -v_7 -v_8 -v_9 -v_4 -v_5 -$\emptyset$
+$\emptyset$ +v_3 +$\{v_3, v_6\}$ +$\{v_3, v_6, v_7\}$ +$\{v_3, v_6, v_7, v_8\}$ ● ● ●
$\tilde{B}_1$ $\tilde{B}_2$ $\tilde{B}_3$ $\tilde{B}_4$ $\tilde{B}_5$ $\tilde{B}_6$ $\tilde{B}_7$
+v_3 +v_6 +v_7 +v_8 +v_9 +v_4 +v_5
-$\emptyset$ -v_3 -$\{v_3, v_6\}$ -$\{v_3, v_6, v_7\}$ -$\{v_3, v_6, v_7, v_8\}$ ● ● ●
B
Excluding v_3
Including v_3
+v_6 +v_7 +v_8 +v_9 +v_4 +v_5
-v_3 -$\{v_3, v_6\}$ ● ● ●
$\tilde{B}_2$ $\tilde{B}_3$ $\tilde{B}_4$ $\tilde{B}_5$ $\tilde{B}_6$ $\tilde{B}_7$
-v_6 -v_7 -v_8 -v_9 -v_4 -v_5 -$\emptyset$
+v_3 +$\{v_3, v_6\}$ ● ● ●
$\ddot{B}_2$ $\ddot{B}_3$ $\ddot{B}_4$ $\ddot{B}_5$ $\ddot{B}_6$ $\ddot{B}_7$ $\ddot{B}_8$

(a) Sym-SE branching with v_1 as the pivot (Case 1)

(b) Sym-SE branching: with v_3 as the pivot (Case 2)

(c) SE branching with v_3 as the pivot

(d) Hybrid-SE branching with v_3 as the pivot

Fig. 6. Illustration of Sym-SE branching ((a) and (b)), SE branching (c) and Hybrid-SE branching (d) at $B = (S, C, D)$ with $S = \{v_1, v_2\}$, $C = \{v_3, v_4, ..., v_9\}$ and $D = \emptyset$ ($\gamma = 0.6$ and $\tau(\sigma(B)) = 3$)

LEMMA 3. *For any QC $G[H]$ to be found in B that excludes all vertices in $\overline{\Gamma}(\hat{v}, C)$ (i.e., $\forall u \in \overline{\Gamma}(\hat{v}, C), u \notin H$), QC $G[H]$ is not maximal.*

We consider applying SE branching and Sym-SE branching separately by selecting $\hat{v}$ as the pivot and using the ordering of vertices as specified in Equation (16).

SE Branching. Based on Equation (1), we would create $|C|$ branches, which we denote by $\tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{|C|}$. We have three observations. First, branch $\tilde{B}_1$ *includes* vertex $\hat{v}$ and all other branches *exclude* vertex $\hat{v}$. Second, branch $\tilde{B}_{b+1}$ *excludes* $D \cup \overline{\Gamma}(\hat{v}, C)$ (recall that $\overline{\Gamma}(\hat{v}, C) = \{\hat{v}, v_2, v_3, ..., v_b\}$), and hence branch $\tilde{B}_{b+1}$ holds no maximal QCs based on Lemma 3. Therefore, we can prune $\tilde{B}_{b+1}$. Third, all branches following $\tilde{B}_{b+1}$, namely, $\tilde{B}_{b+2}, \tilde{B}_{b+3}, ..., \tilde{B}_{|C|}$, have their exclusion sets as supersets of that of branch $\tilde{B}_{b+1}$, and hence they can be pruned as well. We summarize the findings as below (branches marked in grey are those that can be pruned).

$$\text{SE Branching:} \begin{cases} \tilde{B}_1 & \text{including } \hat{v} \\ \tilde{B}_2, ..., \tilde{B}_b, \tilde{B}_{b+1}, ..., \tilde{B}_{|C|} & \text{excluding } \hat{v} \end{cases} \tag{17}$$

For illustration, consider the example in Figure 6(c). The branching is based on the pivot v_3, which disconnects 5 vertices in C, i.e., $\{v_3, v_6, v_7, v_8, v_9\}$, and the ordering $\langle v_3, v_6, v_7, v_8, v_9, v_4, v_5\rangle$. The branches B_6 and B_7 can be pruned since they both exclude all vertices in $\overline{\Gamma}(v_3, C) = \{v_3, v_6, v_7, v_8, v_9\}$.

Sym-SE Branching. Based on Equation (13), we would create $|C| + 1$ branches, which we denote by $\ddot{B}_1, \ddot{B}_2, ..., \ddot{B}_{|C|+1}$. Similarly, we know that branch $\ddot{B}_1$ *excludes* vertex $\hat{v}$ and all other branches *include* vertex $\hat{v}$. Recall that we can prune branch $\ddot{B}_{a+2}$ and all branches following $\ddot{B}_{a+2}$. We provide a summary as follows (branches marked in grey are those that can be pruned).

$$\text{Sym-SE Branching:} \begin{cases} \ddot{B}_1 & \text{excluding } \hat{v} \\ \ddot{B}_2, ..., \ddot{B}_{a+1}, \ddot{B}_{a+2}, ..., \ddot{B}_{|C|+1} & \text{including } \hat{v} \end{cases}$$

Hybrid-SE Branching. As can be noticed, with SE branching, we can prune some branches (namely, $\tilde{B}_{b+1}, ..., \tilde{B}_{|C|}$) that hold no maximal QCs. With Sym-SE branching, we can prune some branches (namely, $\ddot{B}_{a+2}, ..., \ddot{B}_{|C|+1}$) that hold no QCs. We therefore propose a hybrid branching method based on SE branching and Sym-SE branching so as to inherit the merits of both strategies. We call this hybrid branching method *Hybrid-SE branching*. Specifically, for the case of excluding the vertex $\hat{v}$, we take the branches $\tilde{B}_2, ..., \tilde{B}_b$ created by SE branching, and for the other case of including the vertex $\hat{v}$, we take the branches $\ddot{B}_2, ..., \ddot{B}_{a+1}$ created by Sym-SE branching. Clearly, all branches that have been taken cover all possible vertex sets under branch B. We provide a summary of Hybrid-SE branching as follows (branches marked in grey are those that can be pruned).

$$\text{Hybrid-SE Branching:} \begin{cases} \tilde{B}_2, ..., \tilde{B}_b, \tilde{B}_{b+1}, ..., \tilde{B}_{|C|} & \text{excluding } \hat{v} \\ \ddot{B}_2, ..., \ddot{B}_{a+1}, \ddot{B}_{a+2}, ..., \ddot{B}_{|C|+1} & \text{including } \hat{v} \end{cases} \tag{18}$$

For illustration, consider the example in Figure 6(d). The branching takes branches $\tilde{B}_2, ..., \tilde{B}_7$ created by SE branching (which exclude v_3) and branches $\ddot{B}_2, ..., \ddot{B}_8$ created by Sym-SE branching (which include v_3). The branches $\tilde{B}_6, \tilde{B}_7$ and $\ddot{B}_5, ..., \ddot{B}_8$ can be pruned.

Remark. The Hybrid-SE branching is applicable only in the case where we can find a pivot $\hat{v}$ such that $\forall u \in \overline{\Gamma}(\hat{v}, C), u \notin H$ (note that we apply Hybrid-SE branching when $b = a + 1$ or $\tau(\sigma(B)) = 1$ is also satisfied for obtaining better time complexity); otherwise, we would use the Sym-SE branching method, which is always applicable. In Section 4.5, we will show that a BB algorithm based on our new Hybrid-SE branching (if possible) and Sym-SE branching (otherwise) would achieve new state-of-the-art worst-case time complexity. Furthermore, in the case we always use Sym-SE branching, the worst-case time complexity would be slightly worse than when we use Hybrid-SE branching if possible, but still better than when we use the existing SE branching (details can be found in the technical report [47] for the sake of space).

4.5 `FastQC`: Summary and Analysis

Based on the newly proposed pruning techniques (namely the the progressive procedure of refining a branch and re-checking the necessary condition) and branching methods (namely Hybrid-SE and Sym-SE), we develop a new BB algorithm called `FastQC`. The pseudocode of `FastQC` is presented in Algorithm 2, which differs from `Quick+` in the following aspects. First, it progressively refines a branch and re-checks the necessary condition until a refined branch is pruned or a branch cannot be refined any further (line 3-7). Second, if a refined branch satisfies the necessary condition and is not pruned, we then check two termination conditions, namely T1 based on the obtained $\tau(\sigma(B))$ and T2 based on the size constraint θ and terminate the branch if any of the condition is satisfied (line 8-11).

T1: Termination condition based on $\tau(\sigma(B))$. As discussed earlier, we can terminate the branch when $\Delta(S \cup C) \leq \tau(\sigma(B))$ (line 8-10) since the branch holds a QC $G[S \cup C]$ and any other QC under this branch is a subgraph of $G[S \cup C]$. In this case, we check the following necessary condition for a QC $G[H]$ to be maximal,

$$\nexists\, v \in V - H,\ G[H \cup v] \text{ is a QC},$$

```
Algorithm 2: A new branch-and-bound algorithm: FastQC
  Input: A graph $G = (V, E)$, $0.5 \le \gamma \le 1$, and $\theta > 0$
  Output: A set of QCs that includes all MQCs
1 FastQC-Rec($\emptyset, V, \emptyset$);
2 Procedure FastQC-Rec($S, C, D$)
      /* Progressively refining&re-checking (Sec. 4.2) */
3     repeat
4         if $\Delta(S) > \tau(\sigma(B))$ (Condition C1&C2 is not satisfied) then
5             return false;
6         Refine $B$ via Refinement Rule (1) and (2);
7     until no vertices can be removed from the candidate set $C$;
      /* Termination condition based on $\tau(\sigma(B))$ (T1) */
8     if $\Delta(S \cup C) \le \tau(\sigma(B))$ then
9         if the necessary condition of maximality is satisfied and $|S \cup C| \ge \theta$ then Output $G[S \cup C]$ ;
10        return true;
      /* Termination condition based on $\theta$ (T2) */
11    if any of termination conditions is satisfied then return false;
      /* Sym-SE & Hybrid-SE branching (Sec. 4.3 & 4.4) */
12    Select a pivot $\hat{v}$ from $S \cup C$ for branching;
13    if $\hat{v} \in C$ and $\overline{\delta}(\hat{v}, S) = 0$ and ($b = a + 1$ or $\tau(\sigma(B)) = 1$) (Hybrid-SE branching) then
14        Create $\{\ddot{B}_2, ..., \ddot{B}_{a+1}, \tilde{B}_2, ..., \tilde{B}_b\}$ based on Equation (18)
15    else if $\hat{v} \in S$ (Sym-SE branching: Case 1) then
16        Create branches $\{\ddot{B}_1, \ddot{B}_2, ..., \ddot{B}_{a+1}\}$ based on Equation (13,15)
17    else
18        Create branches $\{\ddot{B}_1, \ddot{B}_2, ..., \ddot{B}_{a+1}\}$ based on Equation (13,16) // Sym-SE branching: Case 2
19    for each branch $B_i$ do
20        $\mathcal{T}_i \leftarrow$ FastQC-Rec($S_i, C_i, D_i$);
      /* Additional step: output $G[S]$ if necessary */
21    if all of $\mathcal{T}_i$ are false then
22        if $G[S]$ is a QC and satisfies the necessary condition of maximality then
23            Output $G[S]$ if $|S| \ge \theta$; return true;
24        return false;
25    return true;
```

which can be done in polynomial time (note that the problem of checking the maximality of a QC exactly is NP-hard [35]). If yes and the size of $G[S \cup C]$ is at least θ, we output $G[S \cup C]$.

T2: Termination condition based on size constraint θ. During the recursive procedure, if any of the following two conditions is satisfied, we can terminate the branch (line 11) since no MQC with size at least θ would be found.

(1) $|S \cup C| < \theta$.
(2) There exists a vertex $v \in S$ such that $\delta(v, S \cup C) < \theta - \tau(\sigma(B))$.

For simplicity, we put the proof in the technical report [47].

Third, we select a pivot from $S \cup C$ and conduct the Hybrid-SE branching (if possible) and Sym-SE branching (otherwise) for forming sub-branches (line 12-20).

Note that FastQC would also need to monitor whether a sub-branch of the current one would find a QC (line 21-25), similarly as Quick+ does. In addition, FastQC would return a superset of all MQCs which inevitably contains some non-maximal QCs, i.e., FastQC solves the MQCE-S1 problem, but not the MQCE problem, as Quick+ does.

Worst-case time complexity. The worst-case time complexity of FastQC is strictly smaller than that of Quick+. We give the details in the following theorem (with the proof provided in the technical report [47]).

Theorem 1. *Given a graph $G = (V, E)$,* FastQC *finds a set of QCs that includes all MQCs with the size at least θ, i.e., it solves the MQCE-S1 problem, in $O(n \cdot d \cdot \alpha_k^n)$ time where α_k is the largest real root of $x^{k+2} - x^{k+1} - 2x^k + 2 = 0$ when $k \geq 2$ and $k = \tau(n)$ is an upper bound of the largest $\tau(\sigma(B))$ (i.e., $\tau(\sigma(B)) \leq k$ for any branch B). α_k is strictly smaller than 2. For example, when $k = 2$ and 3, $\alpha_k =$ 1.769 and 1.899, respectively. Besides, when $k = 1$, $\alpha_k = 1.445$.*

Remark 1. We note that there exist some studies, which break the $O^*(2^n)$ worst-case time complexity for enumerating subgraphs that satisfy the hereditary property [48, 52]. We emphasize that (1) these methods cannot be directly applied to our problem of enumerating QCs which do not satisfy the hereditary property; (2) our method is the first one which breaks the $O^*(2^n)$ worst-case time complexity for enumerating QCs; and (3) the constants α_k in the time complexity of our method (e.g., our smallest constant is 1.769 when $k = 2$) are smaller than those of many existing methods [48, 52] (e.g., their smallest constant is 1.839 for $k = 2$), which is due to the newly proposed necessary condition in the SD space and the branching methods.

Remark 2. We remark that for solving the MQCE problem, the worst-case time complexity is $O^*(\alpha_k^n + \min\{|\mathcal{S}_{fast}|^2,\ |\mathcal{S}_{fast}| \cdot 2^{2\omega}\})$ (resp. $O^*(2^n + \min\{|\mathcal{S}_{quick}|^2,\ |\mathcal{S}_{quick}| \cdot 2^{2\omega}\})$) when adopting FastQC (resp. Quick+) for solving MQCE-S1 and the method in [37] for sovling MQCE-S2, where $\mathcal{S}_{fast}$ (resp. $\mathcal{S}_{quick}$) is the set of QCs returned by FastQC (resp. Quick+). We note that $|\mathcal{S}_{fast}|$ and $|\mathcal{S}_{quick}|$ can be bounded by the number of branches produced by FastQC and Quick+, i.e., $O^*(\alpha_k^n)$ and $O^*(2^n)$, respectively, since at most one QC can be returned per branch. Since ω is bounded by n, we deduce that the time complexity of solving MQCE with FastQC is $O^*(\alpha_k^{2n})$ and that with Quick+ is $O^*(2^{2n})$. Furthermore, for sparse graphs with ω bounded by $O(\log n^c)$ where c is a constant, the time complexity of solving MQCE with FastQC is $O^*(\alpha_k^n)$ and that with Quick+ is $O^*(2^n)$. In any case, the method based on FastQC has a strictly smaller theoretical time complexity than that based on Quick+. Based on our experimental results, the former runs faster than the latter by up to two orders of magnitude.

5 A DIVIDE-AND-CONQUER FRAMEWORK WITH FASTQC: DCFASTQC

While FastQC has a lower time complexity than existing methods (e.g., Quick+), it may still suffer from a scalability issue when running on big graphs. To further boost the efficiency and scalability of finding MQCs, we adopt a *divide-and-conquer* strategy, which is to divide the whole graph into multiple smaller ones and then run FastQC on each of them. We call the resulting algorithm DCFastQC, which guarantees to find all MQCs. Furthermore, we develop some pruning techniques to shrink the constructed smaller graphs for better efficiency. In summary, DCFastQC would invoke FastQC multiple times, each on a smaller graph (compared with the original graph), and thus the scalability is improved. We note that this *divide-and-conquer* strategy has been widely used for enumerating subgraphs [19, 24, 48, 52]. Our technique differs from existing ones in (1) the way of how a graph is divided [19, 24]; and/or (2) the techniques for shrinking the smaller graphs [19, 24, 48, 52].

To be specific, given an ordering $\langle v_1, v_2, ..., v_{|V|}\rangle$, it divides the whole graph G into $|V|$ subgraphs, namely $G_i = G[V_i]$ for $1 \le i \le |V|$, as follows.

$$V_i = \Gamma_2(v_i, V) - \{v_1, v_2, ..., v_{i-1}\}, \tag{19}$$

where $\Gamma_2(v_i, V)$ is the set of 2-hop neighbours of v_i in V and $|V_i|$ is thus bounded by $O(d^2)$. Then, on each subgraph G_i, it runs `FastQC` by starting with the branch $B = (S, C, D)$ with $S = \{v_i\}$, $C = V_i - \{v_i\}$ and $D = \{v_1, v_2, ..., v_{i-1}\}$. Note that all MQCs found in G_i would include v_i and exclude $\{v_1, v_2, ..., v_{i-1}\}$. It is not difficult to verify that each MQC would be found exactly once from one of above subgraphs based on Property 2 (for which we put the proof in the technical report [47] for the sake of space).

The framework can be further improved by shrinking the subgraphs formed as above, with techniques of *vertex ordering* and *pruning rules on G_i* as presented below.

Degeneracy ordering. By following some existing studies [48, 52], we adopt the degeneracy ordering of V for dividing a graph. The reason is two-fold. First, the size of each subgraph $|V_i|$ would be bounded by $O(\omega d)$ based on the property of degeneracy ordering where ω denotes the degeneracy of G [48, 52]. Second, the degeneracy ordering can be obtained by core decomposition in polynomial time $O(|E|)$ efficiently [3].

Pruning rules on G_i. We can prune the following vertices from a subgraph G_i.

- **One-hop pruning.** $u \in V_i - \{v_i\}$, $\delta(u, V_i) < \lceil \gamma \cdot (\theta - 1) \rceil$.
- **Two-hop pruning.** $u \in V_i - \{v_i\}$, (1) if $u \in \Gamma(v_i, V_i)$, $|\Gamma(v_i, V_i) \cap \Gamma(u, V_i)| < f(\theta)$ or (2) if $u \notin \Gamma(v_i, V_i)$, $|\Gamma(v_i, V_i) \cap \Gamma(u, V_i)| < f(\theta) + 2$, where $f(\theta) = \theta - \tau(\theta) - \tau(\theta + 1)$.

We put the proof of above pruning rules in the technical report [47] for the sake of space. Moreover, we can iteratively apply one-hop pruning and two-hop pruning on G_i for multiple rounds, which would boost their effectiveness. The rationale is that with some vertices excluded from G_i in a former round, the degrees of the remaining vertices would become smaller and thus they can potentially be pruned in the current round.

Algorithm 3: A divide-and-conquer framework with `FastQC`: `DCFastQC`

```
Input: A graph $G = (V, E)$, $0.5 \le \gamma \le 1$, and $\theta > 0$
Output: A set of QCs that includes all MQCs
1 Reduce $G = (V, E)$ as a $\lceil \gamma \cdot (\theta - 1) \rceil$-core of $G$;
2 Compute the degeneracy ordering $\langle v_1, v_2, ..., v_n \rangle$;
3 for each $v_i$ in $\{v_1, v_2, ..., v_n\}$ do
4     Construct $G_i = G[V_i]$ based on Equation (19);
5     for $i$ = 1, 2, ..., MAX_ROUND do
6         Refine $V_i$ by one-hop pruning and two-hop pruning;
7     Construct $S = \{v_i\}$, $C = V_i - \{v_i\}$ and $D = \{v_1, ..., v_{i-1}\}$;
8     FastQC-Rec$(S, C, D)$;
```

The `DCFastQC` Algorithm. The pseudocode of `DCFastQC` is presented in Algorithm 3. First, it reduces the graph to be the $\lceil \gamma \cdot (\theta - 1) \rceil$-core of G (line 1). This is because every QC with size at least θ is within the $\lceil \gamma \cdot (\theta - 1) \rceil$-core of G [19]. Then, it computes the degeneracy ordering (line 2). Finally, it performs n iterations (line 3 - 8). At the i^{th} iteration, it constructs a smaller graph $G_i = G[V_i]$ (line 4), prunes the vertices from V_i for MAX_ROUND rounds, where MAX_ROUND is a user parameter for controlling the trade-off between the workload and the effectiveness of the pruning techniques (line 5-6), and then runs `FastQC` on the refined graph (line 7-8).

Table 1. Real datasets (“K” means a thousand and “M” means a million)

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$	d	ω	θ_d	γ_d	#{MQC}	#{DCFastQC}	#{Quick+}	$\lvert H_{min}\rvert$	$\lvert H_{max}\rvert$	$\lvert H_{avg}\rvert$
Ca-GrQC	5K	14K	2.77	81	43	10	0.9	1,665	1,725	2,232	10	46	26.56
Opsahl	2K	15K	5.33	473	28	20	0.9	34,508	35,681	263,943	21	26	21.69
CondMat	39K	175K	4.43	278	29	10	0.9	7,222	7,977	11,465	10	30	13.33
Enron	36K	183K	5.01	1383	43	23	0.9	200	212	335	23	24	23.08
Douban	154K	327K	2.11	287	15	12	0.9	26	26	26	12	12	12
WordNet	146K	656K	4.49	1008	31	14	0.9	2,515	2,691	5,231	14	32	17.29
Twitter	465K	833K	1.79	677	30	6	0.9	11	11	11	6	6	6
Hyves	1M	2M	1.98	31,883	39	23	0.9	114	117	168	23	24	23.05
Trec	1M	6M	1.98	25,609	140	50	0.96	682,736	682,862	2,659,161	51	91	54.64
Flixster	2M	7M	3.14	1,474	123	35	0.96	22,853	24,829	52,845	35	38	35.16
Pokec	1M	22M	13.66	20,518	47	32	0.9	7	7	7	32	32	32
FullUSA	23M	28M	1.20	9	3	3	0.51	35	35	35	6	6	6
Kmer	67M	69M	1.02	35	6	10	0.51	146	176	265	10	12	10.09
UK2002	18M	261M	14.16	194,955	943	450	0.96	6	27	—	475	944	651

Time complexity. The time cost is dominated by the part of invoking `FastQC` $O(n)$ times. Recall that the number of vertices in a graph G_i is bounded by $O(\omega d)$ (as analyzed earlier). Based on the time complexity of `FastQC` presented in Theorem 1, we deduce that the time complexity of `DCFastQC` is $O(n \cdot \omega d^2 \cdot \alpha_k^{\omega d})$ where α_k is the largest real root of $x^{k+2} - x^{k+1} - 2x^k + 2 = 0$ when $k \geq 2$ and $k = \lfloor \omega(1-\gamma)/\gamma + 1 \rfloor$ (the proof is put in the technical report [47] for simplicity). We remark that in practice, `DCFastQC` is faster than `FastQC` since large graphs usually have ω and d far smaller than the total number of vertices, which will be verified in our experiments.

6 EXPERIMENTAL RESULTS

Datasets. We use both real and synthetic datasets in experiments. The real datasets are collected from http://konect.cc/ and come from different domains. The statistics of the real datasets are summarized in Table 1, where the edge density of a graph $G = (V, E)$ is defined by $|E|/|V|$, d denotes the maximum degree, ω represents the graph degeneracy, θ_d and γ_d are default settings of θ and γ, respectively. The synthetic datasets are generated based on the Erdös-Réyni (ER) graph model. Specifically, we first generate a certain number of vertices and then randomly add a certain number of edges between pairs of vertices. By default, the number of vertices and edge density are set as 100k and 20 for synthetic datasets, respectively.

Statistics of large MQCs. The statistics of large MQCs in the real datasets are provided in Table 1, where #{MQC} denotes the number of large γ_d-MQCs with the size at least θ_d and $|H_{min}|$, $|H_{max}|$ and $|H_{avg}|$ denote the minimum, maximum and average size of MQCs in the datasets, respectively. We remark that the number of *large* MQCs would decrease significantly as θ grows (details can be found in the technical report [47]) and thus is far smaller than the exponential in n under our settings of θ_d. Besides, the found MQCs are usually sufficiently large to be meaningful (with at least 10 and up to 944 vertices for the most datasets). We note that the largest MQC found in Twitter ($\gamma = 0.9$) and FullUSA ($\gamma = 0.51$) contains 6 vertices since they are quite sparse and do not have any locally dense region. We use them mainly for testing the efficiency and scalability of our algorithm.

Algorithms. We compare our proposed algorithm `DCFastQC` with `Quick+` [24]. `Quick+` is the state-of-the-art algorithm as introduced in Section 3, which runs significantly faster than previous methods, including `Crochet` [23, 32], `Cocain` [51], `Quick` [28]. We also compare different branching strategies and different divide-and-conquer frameworks, including the one proposed in this paper and the one proposed in [19, 24]. Besides, we use the set containment query algorithm proposed in [37] for implementing the post-processing step for filtering out the non-maximal outputs.

Implementation and Settings. All algorithms are implemented in C++ and tested on a Linux machine with a 2.10GHz Intel CPU and 128GB memory. We use the recent implementation of `Quick+` [24]. We measure and compare the running times of the algorithms under various settings.

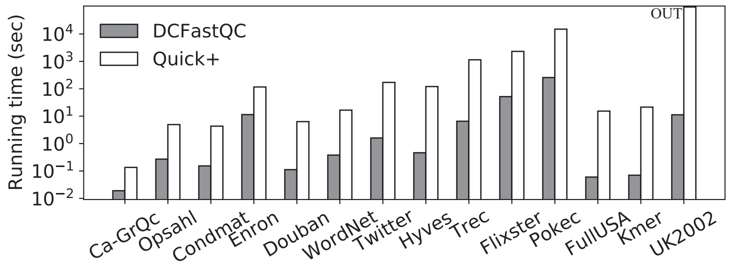


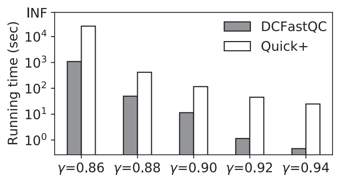
Fig. 7. Comparison on all real datasets

By following existing studies [19, 24, 28], we report the running time that excludes the time for filtering out non-maximal QCs since it can be done efficiently [24] (e.g., it can be finished within 16s for all datasets used in our experiments). We set the running time limit (INF) as 24 hours and select four datasets, namely Enron, WordNet, Hyves and Pokec, as default ones since they cover different graph sizes and edge densities). Besides, the default settings of parameter γ and θ are given in Table 1 for each dataset, which are determined based on the graph statistics. For example, Trec and Flixster have a larger default value of γ (i.e., 0.96) since the number of MQCs grows exponentially when γ decreases and there exist a significant number of 0.96-MQCs. In contrast, FullUSA has a smaller default value of γ (i.e., 0.51) since it is very sparse and thus has few MQCs for a large γ. Our code and datasets are available at https://github.com/KaiqiangYu/SIGMOD24-MQCE.

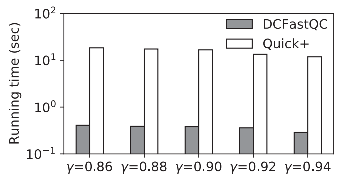
6.1 Comparison among Algorithms

All datasets (Default Settings). We compare our algorithm `DCFastQC` with the baseline `Quick+` on various datasets using default γ_d and θ_d settings as shown in Table 1. We report the running time in Figure 7 and the number of returned QCs, denoted by #{DCFastQC} and #{Quick+}, in Table 1. We observe that (1) our algorithm `DCFastQC` outperforms `Quick+` on all datasets and achieves up to 100x speedup and (2) `Quick+` runs out of the 128GB memory budget (denoted by OUT) and cannot finish on the largest dataset UK2002. This observation demonstrates the efficiency and scalability of `DCFastQC` in practice and is also compatible with the theoretical results that `DCFastQC` has the worst-case running time strictly smaller than that of `Quick+`. Besides, `DCFastQC` has the number of outputs almost the same as that of MQCs, and outputs fewer non-maximal QCs compared with `Quick+`. This is mainly because the necessary condition of maximality would prune many non-maximal outputs. For example, on the dataset Opsahl with 34k MQCs inside, `DCFastQC` returns 35k QCs while `Quick+` returns 263k QCs. Consequently, the post-processing step of `DCFastQC` runs faster than that of `Quick+` (the results are put in the technical report [47] for simplicity since it can be done quickly within 0.1 second on most datasets). Finally, we observe that `DCFastQC` would run slower on a denser graph (e.g., Enron) while running faster on a sparser graph (e.g., Douban). This is because the time cost of `DCFastQC` is $O(n \cdot \omega d^2 \cdot \alpha_k^{\omega d})$ and the values of d and ω of a denser graph tend to be larger.

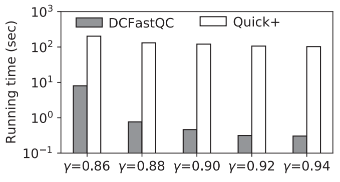
Varying γ. We report the running time in Figure 8 as γ varies. We have the following observations. First, `DCFastQC` significantly outperforms `Quick+` by achieving up to two orders of magnitude speedup. Second, the running times of all algorithms usually drop as γ increases. This is because the number of MQCs decreases exponentially as γ increases. Third, the achieved speedup increases as γ increases, which indicates that `DCFastQC` performs better for lager γ's. Possible reasons include (1) the parameter $\tau(\sigma(B))$ (with the value equal to $\min\{\lfloor |S \cup C| \cdot (1-\gamma) + \gamma \rfloor, \lfloor d_{min}(B) \cdot (1-\gamma)/\gamma + 1 \rfloor\}$) decreases as γ grows and correspondingly the pruning rules based on $\tau(\sigma(B))$ become more effective; (2) our branching strategy would produce fewer branches for larger γ's according to the



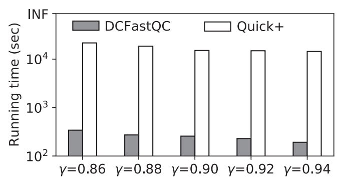
(a) Varying γ (Enron)



(b) Varying γ (WordNet)

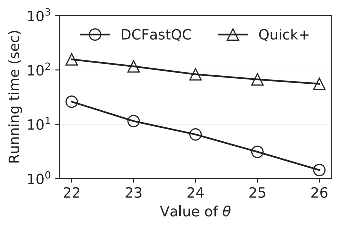


(c) Varying γ (Hyves)

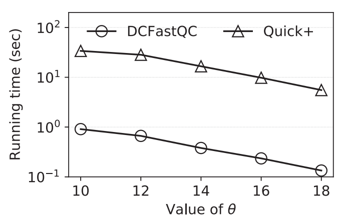


(d) Varying γ (Pokec)

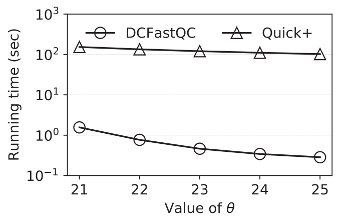
Fig. 8. Comparison by varying γ



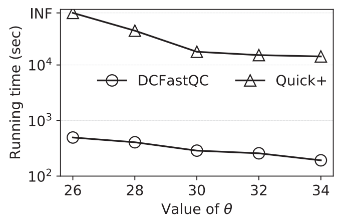
(a) Varying θ (Enron)



(b) Varying θ (WordNet)



(c) Varying θ (Hyves)



(d) Varying θ (Pokec)

Fig. 9. Comparison by varying θ

theoretical results, i.e., the number of formed branches in the worst case is bounded by $O^*(\alpha_k^{\omega d})$ (details can be found in the proof of the time complexity of `FastQC`, which is put in the technical report [47]) . Note that the parameter k (with the value of $\min\{\lfloor \omega d(1-\gamma)+\gamma \rfloor, \lfloor \omega(1-\gamma)/\gamma+1 \rfloor\}$) decreases as γ grows and α_k becomes slightly smaller.

Varying size threshold θ. We report the running time in Figure 9 as θ varies. Our algorithm `DCFastQC` outperforms `Quick+` by achieving up to two orders of magnitude speedup on various settings. In addition, the running times of all algorithms drop as θ increases. This is mainly because (1) the number of large QCs (with the size at least θ) decreases exponentially with the increase of θ; (2) the pruning techniques based on θ and the proposed `DC` framework are more effective for larger θ's.

Varying # of vertices (scalability test on synthetic datasets). We test the scalability on synthetic datasets using default settings of $\gamma_d = 0.9$ and $\theta_d = 10$ and report the running time in Figure 10(a) as the number of vertices varies. `DCFastQC` is faster than `Quick+` by achieving at least 10× speedup

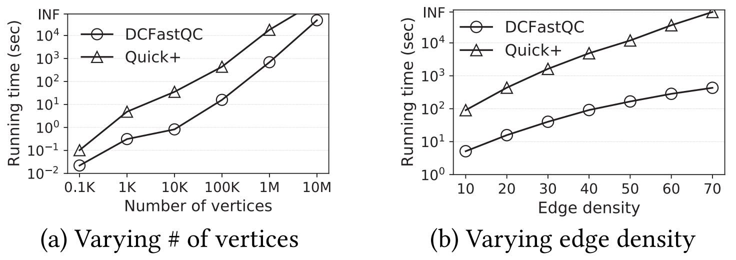


Fig. 10. Comparison on synthetic datasets

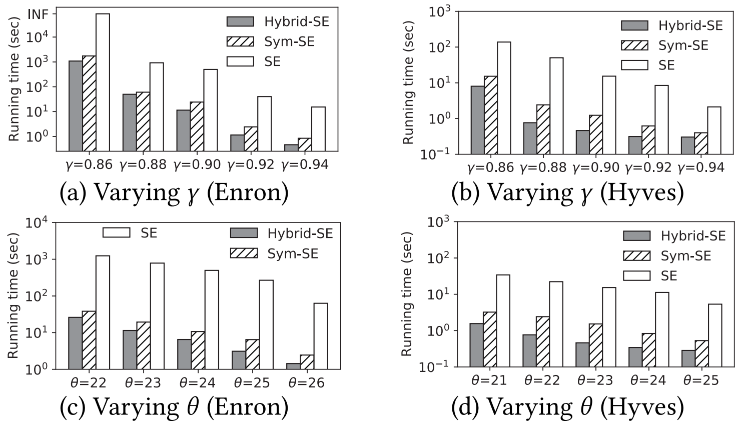


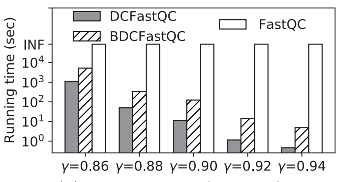
Fig. 11. Comparison among various branching strategies

and can handle the largest datasets within INF while Quick+ cannot. In addition, the running time increases as the graph scale becomes larger.

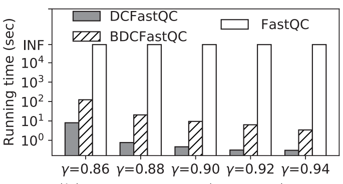
Varying edge density. We use default settings of $\gamma_d = 0.9$ and $\theta_d = 10$ and report the running time in Figure 10(b) as the edge density varies. We have the following observations. First, DCFastQC runs faster than Quick+ by achieving up to 1000× speedup and can handle the densest datasets with the edge density $|E|/|V|$ up to 70 while Quick+ cannot. Second, the running time clearly rises as the graph becomes denser. The reason is two-fold: (1) the number of MQCs increases as the edge density grows and (2) those pruning rules based on the degree of vertices are less effective for denser graphs since the vertices have the degree increase as the graph becomes denser and thus are hard to be pruned. Third, DCFastQC achieves higher speed-ups as the graph becomes denser.

6.2 Performance Study

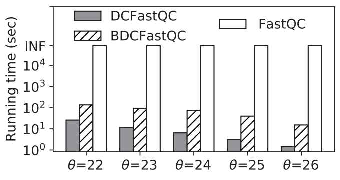
Comparison among various branching strategies. We study the effects of various branching strategies by comparing three different versions of DCFastQC, namely (1) Hybrid-SE: DCFastQC with the Hybrid-SE branching (if applicable) and Sym-SE branching (otherwise), (2) Sym-SE: DCFastQC with the Sym-SE branching only and (3) SE: DCFastQC with the SE branching only. The results are shown in Figure 11(a) and (b) for varying γ and (c) and (d) for varying θ. First, both Hybrid-SE and Sym-SE outperform SE with up to 100× speedup. Moreover, the achieved speedup decreases as θ (resp. γ) grows since the search space (i.e., the number of QCs with the size at least θ) narrows with the increase of θ (resp. γ). Second, Hybrid-SE performs the best and achieves



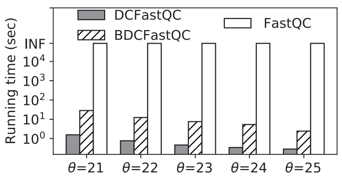
(a) Varying γ (Enron)



(b) Varying γ (Hyves)



(c) Varying θ (Enron)



(d) Varying θ (Hyves)

Fig. 12. Comparison among DC frameworks

around 1 - 5× speedup compared with Sym-SE. This is well aligned with the theoretical results and demonstrates the efficiency of the Hybrid-SE branching.

Comparison among various DC frameworks. We study the effects of DC frameworks by comparing three different versions, namely, (1) FastQC: without any divide-and-conquer framework, (2) BDCFastQC: with a *basic* divide-and-conquer framework proposed in [19, 24], (3) DCFastQC: with the DC framework proposed in Section 5. The results are shown in Figure 12(a) and (b) for varying γ and (c) and (d) for varying θ. First, DCFastQC and BDCFastQC run significantly faster than FastQC and the achieved speedup increases as θ or γ grows. This is well aligned with the theoretical results, i.e., the worst-case running time of FastQC is exponential wrt n. Second, DCFastQC outperforms BDCFastQC by achieving at least 10× speedup. This is because our DC framework with the additional two-hop pruning would produce smaller refined graphs G_i compared with those in [19, 24].

Other experiments. We conduct some additional experiments and put the details in the technical report [47]. (1) We show that the methods that replace the SE branching with our proposed branching methods perform similarly as Quick+ does and significantly worse than DCFastQC, which implies that our proposed pruning techniques suit our proposed branching methods better than those in Quick+; (2) We study the effect of DC on reducing graph size and find that the reduced graph G_i produced by DC is around 0.01% of the original graph; (3) We study the effect of MAX_ROUND on DC and find that when MAX_ROUND= 2, 3, 4, they would achieve similar performance but better than when MAX_ROUND= 1. We therefore adopt MAX_ROUND = 2 by default.

7 RELATED WORK

Maximal quasi-clique enumeration. In the literature, existing studies [19, 23, 24, 28, 32, 51] all adopt a branch-and-bound (BB) framework for enumerating MQCs. They mainly aim to design effective pruning rules to refine the search space. Specifically, Crochet [23, 32] and Cocain [51] are the earliest BB algorithms proposed for mining MQCs. They are then combined as a new algorithm Quick [28] which integrates all previous pruning rules and employs new effective ones. Authors in [19, 24] further improve some pruning rules in Quick and address a few boundary cases that were not properly handled before, which leads to the state-of-the-art algorithm Quick+. To scale Quick+ to big graphs, a distributed solution [19] on top of G-thinker [43] and a (single-machine) parallel solution [24] on top of T-thinker [42] are developed. We note that (1) all these BB algorithms employ the SE branching method and thus (2) they all have the worst-case time complexity of

$O^*(2^{|V|})$. In this paper, we develop a new BB algorithm `DCFastQC`, which employs new pruning techniques and branching methods and achieves a better time complexity.

Other variants of quasi-clique mining. There are many variants of QC mining which consider various problem settings [11, 12, 25, 34, 35], different types of graphs [20, 22, 27, 29, 44], and different definitions of QC [1, 12, 31]. In the sequel, we review these studies. First, some studies aim to only find those QCs that contain a particular vertex [11, 12] or a set of query vertices [25]. They also adopt a BB framework while developing some pruning rules based on the query set. Some other studies aim to find the (top-k) largest QC $G[H]$ such that $|H|$ is maximized [34, 35]. In particular, they use a kernel-expansion-based framework. Specifically, to find top-k γ-QCs, they first find some γ'-QCs ($\gamma' > \gamma$) as "kernels" by using `Quick`, which are faster to find since $\gamma' > \gamma$. The top-k γ-QCs are then generated by expanding these kernels. This approach has been shown more efficient than directly mining from the input graph. We note that (1) it still needs to find some γ'-QCs in the first step by using `Quick` and (2) it only finds top-k γ-QCs that contain the kernels. Therefore, it is hard to adapt these algorithms to improve existing methods for finding all MQCs. Second, QC has also been introduced to bipartite graphs [22, 29], temporal graphs [27, 44] and directed graphs [20]. Specifically, authors in [22, 29] define quasi-biclique which is a counterpart of QC in bipartite graphs. Besides, temporal quasi-clique is defined on temporal graphs by considering the time interval that a QC spans over [27, 44]. Authors in [20] introduce directed quasi-clique to directed graphs by considering both the in-degree and out-degree of each vertex. We note that most of these algorithms are adapted from `Quick` or `Quick+` and incorporate additional pruning rules based on specific graph types. Hence, these algorithms do not work better than `Quick+` on general graphs, which are targeted in this paper. Third, authors of [1, 12, 31] study edge-based QCs, which are different from the degree-based QCs studied in this paper. Specifically, given a fraction $0 \leq \gamma \leq 1$, an edge-based γ-QC is a subgraph $G[H]$ with the number of edges inside at least $\gamma \cdot |V|(|V|-1)/2$. It has been shown that degree-based QC is denser than edge-based QC [12]. Therefore, we focus on degree-based QC in this paper. Moreover, those algorithms for mining edge-based QCs cannot be adapted to find degree-based QCs since these two types of QCs are different. In addition, there are some other cohesive subgraphs which tolerate some disconnections inside, which include k-plex [14, 40, 52], k-biplex [48–50], and s-defective clique [10, 18, 45]. However, they all satisfy the hereditary property while QCs do not, and thus their corresponding solutions cannot be adapted to our problem of finding MQCs.

8 CONCLUSION

In this paper, we propose a new branch-and-bound algorithm `FastQC` for finding a set of QCs that includes all maximal QCs. `FastQC` is based on our developed pruning techniques and branching methods and achieves a smaller worst-case time complexity than the state-of-the-art `Quick+`. We further develop a divide-and-conquer strategy to boost the performance of `FastQC`. Extensive experiments on real and synthetic datasets validate the superiority of our method. In the future, we will develop efficient parallel implementations of our algorithms and explore possibilities of extending our algorithm to other cohesive subgraph mining problems.

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Award MOE-T2EP20221-0013 and Tier 1 Award (RG77/21)). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. The authors would like to thank the anonymous reviewers for providing constructive feedback and valuable suggestions.

REFERENCES

[1] James Abello, Mauricio GC Resende, and Sandra Sudarsky. 2002. Massive quasi-clique detection. In *Latin American symposium on theoretical informatics*. Springer, 598–612.

[2] Gary D Bader and Christopher WV Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics* 4, 1 (2003), 1–27.

[3] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[4] Stas Bevc and Iztok Savnik. 2009. Using tries for subset and superset queries. In *Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces*. IEEE, 147–152.

[5] Malay Bhattacharyya and Sanghamitra Bandyopadhyay. 2009. Mining the largest quasi-clique in human protein interactome. In *2009 International conference on adaptive and intelligent systems*. IEEE, 194–199.

[6] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[7] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. 2003. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research* 31, 9 (2003), 2443–2450.

[8] Lijun Chang. 2019. Efficient maximum clique computation over large sparse graphs. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*. 529–538.

[9] Moses Charikar, Piotr Indyk, and Rina Panigrahy. 2002. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Automata, Languages and Programming: 29th International Colloquium, ICALP 2002 Málaga, Spain, July 8–13, 2002 Proceedings*. Springer, 451–462.

[10] Xiaoyu Chen, Yi Zhou, Jin-Kao Hao, and Mingyu Xiao. 2021. Computing maximum k-defective cliques in massive graphs. *Computers & Operations Research* 127 (2021), 105131.

[11] Yuan Heng Chou, En Tzu Wang, and Arbee LP Chen. 2015. Finding Maximal Quasi-cliques Containing a Target Vertex in a Graph.. In *DATA*. 5–15.

[12] Patricia Conde-Cespedes, Blaise Ngonmang, and Emmanuel Viennet. 2018. An efficient method for mining the maximal α-quasi-clique-community of a given node in complex networks. *Social Network Analysis and Mining* 8, 1 (2018), 1–18.

[13] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. 2020. Sublinear-space and bounded-delay algorithms for maximal clique enumeration in graphs. *Algorithmica* 82, 6 (2020), 1547–1573.

[14] Qiangqiang Dai, Rong-Hua Li, Hongchao Qin, Meihao Liao, and Guoren Wang. 2022. Scaling Up Maximal k-plex Enumeration. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 345–354.

[15] David Eppstein and Darren Strash. 2011. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*. Springer, 364–375.

[16] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29, 1 (2020), 353–392.

[17] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2021. Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions. In *Proceedings of the 2021 International Conference on Management of Data*. 2829–2838.

[18] Jian Gao, Zhenghang Xu, Ruizhi Li, and Minghao Yin. 2022. An Exact Algorithm with New Upper Bounds for the Maximum k-Defective Clique Problem in Massive Sparse Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 10174–10183.

[19] Guimu Guo, Da Yan, M Tamer Özsu, Zhe Jiang, and Jalal Khalil. 2020. Scalable mining of maximal quasi-cliques: an algorithm-system codesign approach. *Proc. VLDB Endow.* 14, 4 (2020), 573–585.

[20] Guimu Guo, Da Yan, Lyuheng Yuan, Jalal Khalil, Cheng Long, Zhe Jiang, and Yang Zhou. 2022. Maximal directed quasi-clique mining. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1900–1913.

[21] Eric Harley, Anthony Bonner, and Nathan Goodman. 2001. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics* 17, 6 (2001), 487–494.

[22] Dmitry I Ignatov, Polina Ivanova, Albina Zamaletdinova, and Oleg Prokopyev. 2019. Preliminary Results on Mixed Integer Programming for Searching Maximum Quasi-Bicliques and Large Dense Biclusters.. In *ICFCA*. 28–32.

[23] Daxin Jiang and Jian Pei. 2009. Mining frequent cross-graph quasi-cliques. *ACM Trans. Knowl. Discov. Data (TKDD)* 2, 4 (2009), 1–42.

[24] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. 2022. Parallel mining of large maximal quasi-cliques. *The VLDB Journal* 31, 4 (2022), 649–674.

[25] Pei Lee and Laks VS Lakshmanan. 2016. Query-driven maximum quasi-clique search. In *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 522–530.

[26] Xiaofan Li, Rui Zhou, Lu Chen, Chengfei Liu, Qiang He, and Yun Yang. 2022. One set to cover all maximal cliques approximately. In *Proceedings of the 2022 International Conference on Management of Data*. 2006–2019.

[27] Longlong Lin, Pingpeng Yuan, Rong-Hua Li, Jifei Wang, Ling Liu, and Hai Jin. 2021. Mining stable quasi-cliques on temporal networks. *IEEE Trans. Syst. Man Cybern. Syst.* 52, 6 (2021), 3731–3745.

[28] Guimei Liu and Limsoon Wong. 2008. Effective pruning techniques for mining quasi-cliques. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 33–49.

[29] Xiaowen Liu, Jinyan Li, and Lusheng Wang. 2008. Quasi-bicliques: Complexity and binding pairs. In *International Computing and Combinatorics Conference*. Springer, 255–264.

[30] Grigory Pastukhov, Alexander Veremyev, Vladimir Boginski, and Oleg A Prokopyev. 2018. On maximum degree-based-quasi-clique problem: Complexity and exact approaches. *Networks* 71, 2 (2018), 136–152.

[31] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. 2013. On the maximum quasi-clique problem. *Discrete Applied Mathematics* 161, 1-2 (2013), 244–257.

[32] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*. 228–238.

[33] Ronald L Rivest. 1976. Partial-match retrieval algorithms. *SIAM J. Comput.* 5, 1 (1976), 19–50.

[34] Seyed-Vahid Sanei-Mehri, Apurba Das, Hooman Hashemi, and Srikanta Tirthapura. 2021. Mining Largest Maximal Quasi-Cliques. *ACM Trans. Knowl. Discov. Data (TKDD)* 15, 5 (2021), 1–21.

[35] Seyed-Vahid Sanei-Mehri, Apurba Das, and Srikanta Tirthapura. 2018. Enumerating top-k quasi-cliques. In *2018 IEEE international conference on big data (big data)*. IEEE, 1107–1112.

[36] Iztok Savnik. 2013. Index data structure for fast subset and superset queries. In *Availability, Reliability, and Security in Information Systems and HCI: IFIP WG 8.4, 8.9, TC 5 International Cross-Domain Conference, CD-ARES 2013, Regensburg, Germany, September 2-6, 2013. Proceedings 8*. Springer, 134–148.

[37] Iztok Savnik, Mikita Akulich, Matjaž Krnc, and Riste Škrekovski. 2021. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. *Plos one* 16, 2 (2021), e0245122.

[38] Brian K Tanner, Gary Warner, Henry Stern, and Scott Olechowski. 2010. Koobface: The evolution of the social botnet. In *2010 eCrime Researchers Summit*. IEEE, 1–10.

[39] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* 363, 1 (2006), 28–42.

[40] Zhengren Wang, Yi Zhou, Mingyu Xiao, and Bakhadyr Khoussainov. 2022. Listing Maximal k-Plexes in Large Real-World Graphs. In *Proceedings of the ACM Web Conference 2022*. 1517–1527.

[41] Daniel Weiss and Gary Warner. 2015. Tracking criminals on Facebook: A case study from a digital forensics REU program. (2015).

[42] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, John CS Lui, and Weida Tan. 2019. T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 411–412.

[43] Da Yan, Guimu Guo, Jalal Khalil, M Tamer Özsu, Wei-Shinn Ku, and John Lui. 2022. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *The VLDB Journal* 31, 2 (2022), 287–320.

[44] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John CS Lui. 2016. Diversified temporal subgraph pattern mining. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*. 1965–1974.

[45] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinformatics* 22, 7 (2006), 823–829.

[46] Kaiqiang Yu and Cheng Long. 2021. Graph Mining Meets Fake News Detection. In *Data Science for Fake News*. Springer, 169–189.

[47] Kaiqiang Yu and Cheng Long. 2023. Fast Maximal Quasi-clique Enumeration: A Pruning and Branching Co-Design Approach (Technical report). https://personal.ntu.edu.sg/c.long/paper/24-SIGMOD-FastQC-report.pdf.

[48] Kaiqiang Yu and Cheng Long. 2023. Maximum k-Biplex Search on Bipartite Graphs: A Symmetric-BK Branching Approach. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[49] Kaiqiang Yu, Cheng Long, P Deepak, and Tanmoy Chakraborty. 2021. On efficient large maximal biplex discovery. *IEEE Trans. Knowl. Data Eng.* (2021).

[50] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 860–873.

[51] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2006. Coherent closed quasi-clique discovery from large dense graph databases. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*. 797–802.

[52] Yi Zhou, Jingwei Xu, Zhenyu Guo, Mingyu Xiao, and Yan Jin. 2020. Enumerating maximal k-plexes with worst-case time guarantee. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 2442–2449.

Received January 2023; revised April 2023; accepted May 2023