Maximum Quasi-Clique Search via an Iterative Framework

Hongbo Xia[†], Kaiqiang Yu[‡], Shengxin Liu[†], Cheng Long[‡], Xun Zhou[†]

[§]*Harbin Institute of Technology, Shenzhen, China*

[‡]*Nanyang Technological University, Singapore*

{24s151167@stu., sxliu@, zhouxun2023@}hit.edu.cn, {kaiqiang002@e., c.long@}ntu.edu.sg

***Abstract*—Cohesive subgraph mining is a fundamental problem in graph theory with numerous real-world applications, such as social network analysis and protein-protein interaction modeling. Among the various cohesive subgraph definitions, the γ-quasi-clique is widely studied for its flexibility in requiring each vertex to connect to at least a γ proportion of other vertices in the subgraph. However, solving the maximum γ-quasi-clique problem is NP-hard and further complicated by the lack of the hereditary property, making efficient pruning strategies challenging. Existing algorithms, such as `DDA` and `FastQC`, struggle with scalability or exhibit significant performance decline for small values of γ. In this paper, we propose a novel algorithm, `IterQC`, which reformulates the maximum γ-quasi-clique problem as a series of k-plex problems that possess the hereditary property. `IterQC` introduces a non-trivial iterative framework and incorporates two key optimization techniques: the *fake lower bound (fake LB)* technique, which leverages information across iterations to improve the efficiency of branch-and-bound searches, and the *preprocessing* technique that reduces problem size and unnecessary iterations. Our extensive experiments demonstrate that `IterQC` achieves up to four orders of magnitude speedup and solves significantly more graph instances compared to state-of-the-art algorithms `DDA` and `FastQC`.**

***Index Terms*—Cohesive subgraph mining, γ-quasi-clique, iterative framework**

I. Introduction

Cohesive subgraph mining is a fundamental problem in graph theory with numerous real-world applications [12], [23], [31], [38]. For example, it plays a pivotal role in social network analysis [6], which capture relationships between users, and in modeling protein-protein interaction networks [54], which captures relationships between proteins. Unlike cliques, which require complete connectivity, cohesive subgraphs provide more flexibility by allowing the absence of certain edges. Various approaches have been proposed to relax the constraints of cohesive subgraph definitions, such as k-plex [53], k-core [52], and k-defective clique [62]. One widely studied definition is the γ-quasi-clique [45]. Given a fraction γ between 0 and 1, a γ-quasi-clique requires that every vertex in the subgraph is directly connected to at least a γ proportion of the other vertices in the subgraph. Cohesive subgraph mining in the context of γ-quasi-clique has recently attracted increasing interest [67], [39], [33], [47], [29], [51], [36], [63].

In this paper, we study the *maximum γ-quasi-clique* problem, which aims to identify the γ-quasi-clique with the largest number of vertices in the graph. Since this problem can be viewed as a natural extension of the classic maximum clique problem, it is unsurprising that it is NP-hard [45], [47]. An even more challenging lies in the fact that the maximum γ-quasi-clique lacks the hereditary property, meaning that any subgraph of a γ-quasi-clique is not guaranteed to also be a γ-quasi-clique. This limitation significantly complicates the design of efficient pruning strategies, especially when compared to problems like k-plex or k-defective subgraphs, where the hereditary property enables more optimizations. In this paper, we focus on designing practically efficient exact algorithms to tackle this challenging problem.

Existing state-of-the-art algorithms. The state-of-the-art algorithm for solving the maximum γ-quasi-clique problem is the `DDA` algorithm [47], which combines graph properties with techniques from operations research. A key feature of `DDA` is its iterative *enumerate (or guess)* of the degree of the minimum-degree vertex in the maximum γ-quasi-clique. For each candidate degree, `DDA` invokes an integer programming (IP) solver multiple times to solve equivalent sub-problems and identify the final solution. Unlike most other approaches, which either rely solely on IP formulations or exclusively utilize branch-and-bound techniques based on graph properties, `DDA` effectively integrates both. However, `DDA` has the following limitations: (1) it naively enumerates possible minimum degree values, potentially resulting in a large number of trials (e.g., in the worst case, `DDA` needs to enumerate $O(n)$ values, where n is the number of vertices in the graph); (2) the IP solver operates as a black box, making it difficult to optimize its internal processes for this specific problem.

Another closely related algorithm, `FastQC` [63], represents the state-of-the-art for enumerating all maximal γ-quasi-cliques in the graph. `FastQC` employs a sophisticated divide-and-conquer strategy combined with efficient pruning techniques that leverage intrinsic graph properties to systematically and effectively enumerate all maximal γ-quasi-cliques. It is clear that the largest maximal γ-quasi-clique identified corresponds to the maximum γ-quasi-clique. However, since `FastQC` is not specifically designed to solve the maximum γ-quasi-clique problem, even with additional pruning strategies tailored for the maximum solution, its efficiency remains suboptimal. Moreover, our experimental results in Section V show that the efficiency of the `FastQC` algorithm significantly decreases when the value of γ is relatively small.

In summary, while both DDA and FastQC have practical strengths, they fail to fully address the efficiency challenges of the maximum γ-quasi-clique problem. Their limitations, particularly in scalability and handling smaller values of γ, highlight the need for more specialized algorithmic solutions.

Our new methods. In this work, we propose a novel algorithm, IterQC, which reformulates the maximum γ-quasi-clique problem as a series of k-plex problems. Notably, k-plex problems possess the hereditary property, enabling the design of more effective pruning strategies. To achieve this, we introduce a non-trivial *iterative framework* that correctly solves the maximum γ-quasi-clique problem through a carefully designed iterative procedure, leveraging repeated calls to the maximum k-plex solver [58], [14], [27]. Building on this novel iterative framework, we further propose advanced optimization strategies, including the *fake lower bound (fake LB)* technique and the *preprocessing* technique. The fake LB technique effectively coordinates information between the inner iterations and the outer iterative framework, thereby boosting the overall efficiency of the iterative procedure. Meanwhile, the preprocessing technique performs preliminary operations before the iterative framework begins, reducing both the problem size by removing redundant vertices from the graph and reducing the number of iterations by skipping unnecessary iterations. Together, our novel algorithm IterQC equipped with these optimizations achieves speeds up to four orders of magnitude faster and solves more instances compared to the state-of-the-art algorithms DDA and FastQC.

Contributions. Our main contributions are as follows.

- We first propose a basic iterative framework, which correctly transforms non-hereditary problems into multiple problem instances with the hereditary property, effectively addressing the associated challenges. (Section III)
- Based on the basic iterative method, we design an improved algorithm IterQC, which incorporates two key components. First, we introduce the *fake lower bound (fake LB)* technique, which utilizes information from previous iterations to generate a fake lower bound. This technique optimizes the current branch-and-bound search, boosting the practical performance on challenging instances. Second, we develop the *preprocessing* technique that computes lower and upper bounds of the optimum size. By using these bounds, we can remove unpromising vertices from the graph, potentially reducing the number of iterations, and further improving the efficiency of the iterative algorithm. (Section IV)
- We conduct extensive experiments to assess the performance of IterQC. Specifically, we compare IterQC with state-of-the-art algorithms, DDA and FastQC, across different values of γ. Additionally, ablation studies are performed to validate the effectiveness of the proposed fake LB and preprocessing techniques. The results show that (1) in the 10th DIMACS and real-world graph collections, the number of instances solved by IterQC within 3 seconds exceeds the number solved by the other two baselines within 3 hours; (2) on 30 representative graphs, IterQC is up to four orders of magnitude faster than the state-of-the-art algorithms. (Section V)

Additionally, we give the problem definition in Section II, review the related work in Section VI, and conclude this paper in Section VII.

II. Preliminaries

Let $G = (V, E)$ be an undirected simple graph with $|V| = n$ vertices and $|E| = m$ edges. We denote $G[S]$ of G as the subgraph induced by the set of vertices S of V. We use $d_G(v)$ to denote the degree of v in G. Let g be an induced subgraph of G. We denote the vertex set and the edge set of g as $V(g)$ and $E(g)$, respectively. In this paper, we focus on the cohesive subgraph of γ-quasi-clique, which is defined below.

Definition 1. *Given a graph $G = (V, E)$ and a real number $0 \leq \gamma \leq 1$, an induced subgraph g is said to be a* γ-quasi-clique (γ-QC) *if $\forall v \in V(g)$, $d_g(v) \geq \gamma \cdot (|V(g)| - 1)$.*

We are ready to present our problem in this paper.

Problem 1 (Maximum γ-quasi-clique)**.** *Given a graph $G = (V, E)$ and a real number $\gamma \in [0.5, 1]$, the Maximum γ-quasi-clique Problem (MaxQC) aims to find the largest γ-quasi-clique in G.*

We first note that MaxQC has been proven to be NP-hard [45], [47]. Moreover, we let g^* be the largest γ-QC in G and $s^* = |V(g^*)|$ be the size of this maximum solution. We also note that, following previous studies [29], [36], [49], we consider MaxQC with $\gamma \geq 0.5$ since (1) conceptually, a QC with $\gamma < 0.5$ may not be cohesive in practice (note that each vertex in such a QC may connect to fewer than half of its neighbors); (2) technically, a QC with $\gamma \geq 0.5$ exhibits a small diameter, as demonstrated in the following lemma.

Property 1 (Two-Diameter [49])**.** *For $\gamma \geq 0.5$, the diameter of a γ-quasi-clique is at most 2, i.e., any two vertices in a γ-quasi-clique has a distance of at most 2.*

III. Our Basic Iterative Framework

The design of efficient exact algorithms for MaxQC exhibits various challenges. **First**, MaxQC has been proven to be NP-hard [45], [47]. Existing studies for solving similar maximum cohesive subgraph problems resort to the branch-and-bound algorithmic framework [13], [14], [58]. **Second**, the γ-quasi-clique is non-hereditary [63], i.e., an induced subgraph of a γ-QC is not necessarily a γ-QC. As illustrated in Figure 1, the graph on the left is a 0.75-quasi-clique. However, upon removing vertex 4, the graph no longer meets the requirements of a 0.75-quasi-clique. This non-hereditary property of the γ-QC prevents the design of effective bounding techniques in the branch-and-bound algorithmic framework.

To overcome the aforementioned challenges, our algorithm adopts an *iterative framework* that tackles MaxQC by iteratively invoking a solver for another cohesive subgraph

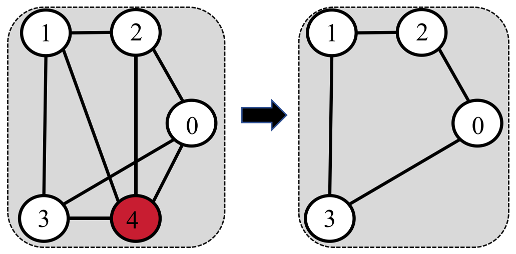


Fig. 1: Non-hereditary property of a 0.75-quasi-clique.

Algorithm 1: A Basic Iterative Framework

```
Input: A graph $G = (V, E)$ and a real value
       $0.5 \leq \gamma \leq 1$
Output: The size of the maximum $\gamma$-quasi-clique in $G$
1 $s_0 \leftarrow |V|$; $k_1 \leftarrow$ get-k($s_0$); $i \leftarrow 1$;
2 while true do
3     $s_i \leftarrow$ solve-plex($k_i$);
4     if $k_i =$ get-k($s_i$) then
5         $s^* \leftarrow s_i$; return $s^*$;
6     $k_{i+1} \leftarrow$ get-k($s_i$); $i \leftarrow i + 1$;
```

problem. The design rationale originates from the intention to *convert the non-hereditary subgraph problem into various implementations of the hereditary subgraph problem, so as to speed up the entire procedure*.

To begin, we introduce another useful cohesive subgraph structure as follows.

Definition 2 ([53]). *Given a graph $G = (V, E)$ and an integer $k \geq 1$, an induced subgraph g is said to be a k-plex if $\forall v \in V(g)$, $d_g(v) \geq |V(g)| - k$.*

Then, in the literature, a related problem for k-plex is defined [13], [25], [34], [35], [58], [60], [68], as below.

Problem 2 (Maximum k-plex). *Given a graph $G = (V, E)$ and a positive integer k, the Maximum k-plex Problem aims to find the largest k-plex in G.*

Based on the maximum k-plex problem, we next describe our basic iterative framework for the `MaxQC` problem. To simplify the description, we define the following two functions.

- `get-k`$(x) := \lfloor(1-\gamma)\cdot(x-1)\rfloor+1$, which takes a number x of vertices as input and returns a suitable value of k;
- `solve-plex`$(y) :=$ the size of the largest y-plex in G, which takes a value y as input.

We present our basic iterative framework for solving `MaxQC` in Algorithm 1. Specifically, Algorithm 1 initializes s_0 as the number of vertices in the graph G, computes the corresponding value of k_1 using the function `get-k`, and set the index i as 1 (Line 1). The algorithm then iteratively computes s_i by repeatedly solving the maximum k_i-plex problem via the function `solve-plex`, where k_i is updated by the function `get-k` at each iteration (Lines 2-6). The iteration terminates when $k_i =$ `get-k`(s_i) in Line 4, at which point s^*, the size of the maximum γ-quasi-clique, is returned in Line 5.

We remark that while Algorithm 1 only returns the size of the maximum γ-quasi-clique, it can be easily adapted to return the subgraph that corresponds to this result.

Correctness. We next prove the correctness of our basic iterative framework in Algorithm 1. For the sake of simplicity of the proofs, we assume the termination condition (Line 4) of Algorithm 1 is met at the $(p+1)$-st iteration. Then, in our iterative framework, we can obtain two sequences $\{s_0, s_1, \ldots, s_p, s_{p+1}\}$ and $\{k_1, k_2, \ldots, k_p, k_{p+1}\}$, where $k_{p+1} =$ `get-k`(s_{p+1}). In our proof, we will show that the sequence of $\{s_0, s_1, \ldots, s_p\}$ generated by Algorithm 1 is strictly decreasing and s_{p+1} corresponds to the largest γ-quasi-clique in the graph, i.e, $s_{p+1} = s^*$.

We first consider the special case that the input graph G is already a γ-quasi-clique.

Lemma 1. *When the input graph $G = (V, E)$ is a γ-QC, Algorithm 1 correctly returns $|V|$ as the optimum solution.*

Proof. When G is a γ-quasi-clique, we have $d_G(v) \geq \gamma \cdot (|V|-1)$, $\forall v \in V$. Since $d_G(v)$ is an integer for each $v \in V$, it follows that $d_G(v) \geq \lceil\gamma\cdot(|V|-1)\rceil \geq |V| - (\lfloor(1-\gamma)\cdot(|V|-1)\rfloor+1) = |V| - k_1$. Thus, G is a k_1-plex and `solve-k`$(k_1) = |V|$, which implies that $s_1 = s_0$. At this point, we have $k_1 =$ `get-k`(s_1), and the algorithm results in $s_1 = |V|$, which completes the proof. □

In the following, we discuss the correctness of the algorithm where G itself is not a γ-quasi-clique. We first prove the properties of the sequence for s in the following lemmas.

Lemma 2. *For the sequence $\{s_0, s_1, \ldots, s_p\}$, two consecutive elements are not identical, i.e., $\forall 0 \leq i \leq p-1$, $s_i \neq s_{i+1}$.*

Proof. Assume, to the contrary, that $s_i = s_{i+1}$ for $0 \leq i \leq p-1$. Then, it follows that $k_{i+1} =$ `get-k`$(s_i) =$ `get-k`(s_{i+1}), which satisfies the termination condition in Line 4. This implies that the algorithm terminates at the $(i+1)$-st iteration, thus $i = p$, leading to a contradiction to $i \leq p-1$. □

Lemma 3. *The sequence $\{s_0, s_1, \ldots, s_p\}$ is strictly decreasing.*

Proof. We prove this lemma by the mathematical induction.
① $p = 1$. As we consider the case where G itself is not a γ-quasi-clique (by Lemma 1), we have $s_p = s_1 =$ `solve-plex`$(k_1) < s_0$. The base case holds true.
② $p \geq 2$. Assume that the induction holds for $p-1$, i.e., we have $s_{p-1} < s_{p-2}$. We observe that `get-k` is a non-decreasing function, which implies that `get-k`$(s_{p-1}) \leq$ `get-k`(s_{p-2}). Moreover, by the definition

of the `solve-plex` function, for any $y_1 > y_2$, the inequality $\texttt{solve-plex}(y_1) \geq \texttt{solve-plex}(y_2)$ holds. Then we have

$$\begin{aligned} s_p &= \texttt{solve-plex}(k_p) = \texttt{solve-plex}(\texttt{get-k}(s_{p-1})) \\ &\leq \texttt{solve-plex}(\texttt{get-k}(s_{p-2})) \\ &= \texttt{solve-plex}(k_{p-1}) = s_{p-1}. \end{aligned}$$

Note that based on Lemma 2 the sequence $\{s_0, s_1, \dots, s_p\}$ ensures that no two adjacent elements share the same value, which implies $s_p < s_{p-1}$, completing the inductive step.

Based on ①, ②, and the principle of mathematical induction, we complete the proof of Lemma 3. □

Based on Lemma 3, we can obtain the following corollary.

Corollary 1. *The sequence $\{s_0, s_1, \dots, s_p\}$ has a finite number of elements.*

Proof. By Lemma 3, the sequence is strictly decreasing. Further, it is easy to see that $s_p \geq 1$ since an induced subgraph with a single vertex is a trivial solution to the maximum k-plex problem with any $k \geq 1$. Moreover, as $\{s_0, s_1, \dots, s_p\}$ is an integer sequence, the difference between any two consecutive elements is at least 1, which implies that $p \leq s_0 = |V| = n$. □

We then show the relationship between the sequence $\{s_0, s_1, \dots, s_p, s_{p+1}\}$ and the size of the largest γ-quasi-clique.

Lemma 4. *$\forall s_i \in \{s_0, s_1, \dots, s_{p+1}\}$, $s_i \geq s^*$ holds.*

Proof. Recall that s^* is the size of the optimum solution. Let u be the vertex with the minimum degree in the largest γ-quasi-clique g^*. Let $\gamma^* = \frac{d_{g^*}(u)}{s^*-1}$. Then we have $s^* = \texttt{solve-plex}(1 + \lfloor (1-\gamma^*) \cdot (s^*-1) \rfloor)$. Additionally, from the definition of a γ-quasi-clique, it also follows that $\gamma^* \geq \gamma$. We then use the mathematical induction to prove this lemma.
① $i = 0$. $s_i = s_0 = |V| \geq s^*$. The base case holds true.
② $i \geq 1$. Assume that the induction holds for $i-1$, i.e., $s_{i-1} \geq s^*$, it follows that

$$\begin{aligned} s_i &= \texttt{solve-plex}(k_i) = \texttt{solve-plex}(\texttt{get-k}(s_{i-1})) \\ &= \texttt{solve-plex}(1 + \lfloor (1-\gamma) \cdot (s_{i-1}-1) \rfloor) \\ &\geq \texttt{solve-plex}(1 + \lfloor (1-\gamma^*) \cdot (s_{i-1}-1) \rfloor) \\ &\geq \texttt{solve-plex}(1 + \lfloor (1-\gamma^*) \cdot (s^*-1) \rfloor) \\ &= s^*. \end{aligned}$$

Based on ①, ②, and the principle of mathematical induction, we complete the proof of Lemma 4. □

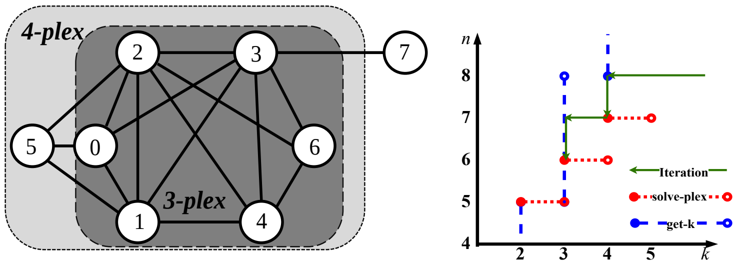
Lemma 5. *Algorithm 1 correctly identifies the largest γ-quasi-clique, i.e., $s_{p+1} = s^*$.*

Proof. We prove this lemma by contradiction. By Lemma 4, we assume, to the contrary, that $s_{p+1} > s^*$. According to the termination condition in Line 4 of Algorithm 1, we have $k_{p+1} = \texttt{get-k}(s_{p+1}) = 1 + \lfloor (1-\gamma) \cdot (s_{p+1}-1) \rfloor$. Thus, it is clear that the result obtained by $\texttt{solve-plex}(k_{p+1})$ is a γ-quasi-clique of size s_{p+1}, which contradicts s^* being the size of the maximum γ-quasi-clique. The proof is complete. □

Example. We present an example of Algorithm 1 and illustrate the corresponding iterative process in Figures 2(a) and 2(b), respectively. In this example graph with $\gamma = 0.55$, the number of vertices is 8, which means $s_0 = 8$. By applying $k_1 = \texttt{get-k}(s_0) = \lfloor (1-0.55) \cdot (8-1) \rfloor + 1 = 4$, the solution of `solve-plex` yields the largest 4-plex highlighted by the light-colored box in Figure 2(a), resulting in $s_1 = 7$. Subsequently, $k_2 = \texttt{get-k}(s_1) = \lfloor (1-0.55) \cdot (7-1) \rfloor + 1 = 3$, and solving for `solve-plex` produces the largest 3-plex, indicated by the dark-colored box in Figure 2(a), with $s_2 = 6$ vertices. At this point, the stopping condition $k_2 = \texttt{get-k}(s_2)$ is satisfied, and the iteration terminates. The maximum 3-plex shown in Figure 2(a) corresponds to the desired maximum 0.55-quasi-clique. In Figure 2(b), the horizontal axis represents values of k, while the vertical axis represents values of n. Moreover, (1) the sparse light blue dashed line depicts the relationship between n and the corresponding k obtained in `get-k`; (2) the dense red dashed line shows the relationship between k and the resulting n obtained in `solve-plex`; (3) the green solid line represents the overall iterative process in Algorithm 1, demonstrating how n evolves from an initial graph of size 8 to the final result of maximum γ-quasi-clique with size 6. Specifically, we start with $s_0 = 8$. After the first iteration, by solving $\texttt{get-k}(s_0)$, the state transitions to $(4, 8)$ in Figure 2(b) on the corresponding axes (i.e., $k_1 = 4$, $s_0 = 8$). Subsequently, solving $s_1 = \texttt{solve-plex}(k_1)$ results in a state transition to $(4, 7)$ in Figure 2(b), completing one full iteration. Repeating this process eventually leads to termination at $(3, 6)$, and the corresponding vertical coordinate 6 represents the optimum solution s^*.

Discussions. As in Lemma 5, our iterative framework in Algorithm 1 can correctly solve `MaxQC`. The time complexity analysis is simple, since there are at most n iterations (Lines 2-6) and each iteration invokes a solver for the maximum k-plex problem. Note that the current best time complexities of the maximum k-plex algorithms are $O^*((k+1)^{\delta+k-s^*})$ and $O^*(\gamma_k^\delta)$ [58], [14], [27], where O^* suppresses the polynomial factors, $\gamma_k < 2$ is the largest real root of $x^{k+2} - x^{k+1} - 2x^k + 2 = 0$, and δ is the degeneracy of the graph.

However, Algorithm 1 still suffers from efficiency issues due to the following reasons. **First**, Algorithm 1 utilizes a solver for the function `solve-plex` as a black box. This solver conservatively relies on a lower bound within `solve-plex` to reduce the graph, resulting in inefficiencies when handling certain challenging instances. **Second**, the graph processed iteratively in `solve-plex` (Line 3) contains several unpromising vertices, which adversely affects overall performance, even after graph reduction within the function `solve-plex`. Furthermore, the initial iteration, where the first value of k is set as $k_1 = \texttt{get-k}(|V|)$ with $|V|$ serving as the trivial upper bound of s^* in Line 1, could potentially result in numerous unnecessary iterations of Lines 2-6.



(a) An example graph. (b) Illustration of the example.

Fig. 2: An example of Algorithm 1 with $\gamma = 0.55$.

IV. Our Improved Algorithm

To address the shortcomings in our basic iterative framework (Algorithm 1), our advanced iterative method `IterQC` incorporates the following two key components. **First**, we make use of a *fake lower bound (fake LB)* inside the function `Plex-Search` (similar to `solve-plex` in Algorithm 1 but incorporating our newly proposed fake LB technique) to boost the practical performance in handling challenging instances, while maintaining the correctness of the iterative algorithm for solving the `MaxQC` problem. The novel technique of fake LB is discussed in Section IV-A. **Second**, we can *preprocess* the graph by first obtaining both lower and upper bounds lb and ub of the size s^* of the largest γ-quasi-clique g^*. Using lb and ub, we can (1) reduce the graph by removing unpromising vertices and edges from G that cannot appear in g^*, and (2) set a smaller initial value of k for the function `Plex-Search`, so as to potentially reduce the number of iterations. The preprocessing technique is introduced in Section IV-B, along with our overall improved iterative algorithm.

A. A Novel Fake Lower Bound Technique

The basic iterative framework in Algorithm 1 can correctly solve the `MaxQC` problem by iteratively invoking the maximum k-plex solver `solve-plex` as a black box using different values of k. To improve the practical efficiency, we have the following key observation.

Key observation. We remark that existing studies [60], [13], [34], [58], [14], [27] on the maximum k-plex solver primarily have two steps: (1) heuristically obtaining a lower bound for the maximum k-plex, and (2) exhaustively conducting the branch-and-bound search for the final solution. It is well known that Step (1) has a polynomial complexity while Step (2) requires an exponential complexity. Thus, both theoretically and practically, in most cases, the time cost of the branch-and-bound search exceeds that of obtaining the lower bound.

With the above property, our key idea is to *balance* the time cost of both steps. In particular, we introduce a *fake lower bound*, denoted by $\textit{fake-lb}$, which is the average of the sum of the heuristic lower bound calculated in Step (1) and a known upper bound of the maximum k-plex solution. In other words, $\textit{fake-lb}$ is always at least the heuristic lower bound in Step (1) and at most the known upper bound. Then, by incorporating the fake lower bound $\textit{fake-lb}$ into the branch-and-bound search in Step (2), we may improve pruning effectiveness, resulting in a reduction in overall search time as verified in our experiments. We note that we use `Plex-Heu` and `Plex-BRB`, which correspond to Step (1) for computing a heuristic solution and Step (2) for conducting the branch-and-bound search in the maximum k-plex algorithms, respectively.

The technique of fake LB. We call this technique as *fake lower bound (fake LB)*, which is incorporated into a branch-and-bound search algorithm called `Plex-Search` in Algorithm 2. `Plex-Search` takes $\textit{ub-plex}$ as an upper bound of the size of the maximum k-plex found in `Plex-Search`, which implies that there does not exist any maximum k-plex of size greater than $\textit{ub-plex}$. The output of `Plex-Search` is a fake lower bound $\textit{fake-lb}$ and a fake size $|S|$ which corresponds to (1) the size of the maximum k-plex of size at least $\textit{fake-lb}$ if exists, or (2) 0, if there does not exist a k-plex with size at least $\textit{fake-lb}$. This is because, when we input a fake lower bound $\textit{fake-lb}$ that is larger than the size of the maximum k-plex in the graph, `Plex-BRB` will utilize this $\textit{fake-lb}$ for pruning the graph, which results in an empty graph as the returned vertex set S.

Our `Plex-Search` method. We describe `Plex-Search` in Algorithm 2. Specifically, in Line 1, we invoke `Plex-Heu` for obtaining a lower bound $\textit{lb-plex}$. If $\textit{lb-plex}$ is equal to $\textit{ub-plex}$, we can directly return $\textit{lb-plex}$ as both the fake lower bound and the fake size in Line 2, since `Plex-Heu` already finds the maximum k-plex in this case. Then, Line 3 computes our fake lower bound $\textit{fake-lb}$ as the average of the sum of $\textit{lb-plex}$ and $\textit{ub-plex}$. We conduct `Plex-BRB` using $\textit{fake-lb}$ for pruning and obtain a corresponding k-plex S (either the vertex set of maximum k-plex or the empty set). Finally, we return $\textit{fake-lb}$ and $|S|$ in Line 5.

Our improved iterative search algorithm. Having our newly proposed `Plex-Search` with the technique of fake LB, we next present our improved iterative search method `Improved-Iter-Search` in Algorithm 3, which is similar to our basic iterative framework in Algorithm 1. Specifically, Algorithm 3 initializes s_0 to be ub, calculates the corresponding value of k_1 using the function `get-k`, and

Algorithm 2: `Plex-Search`$(G, k, \textit{ub-plex})$

```
Output: a fake lower bound $fake\text{-}lb$ and a fake size
1 $lb\text{-}plex \leftarrow$ Plex-Heu($G$); // $G$ is locally reduced.
2 if $lb\text{-}plex = ub\text{-}plex$ then return $lb\text{-}plex, lb\text{-}plex$;
3 $fake\text{-}lb \leftarrow \lfloor (lb\text{-}plex + ub\text{-}plex)/2 \rfloor$;
4 $S \leftarrow$ Plex-BRB($G, fake\text{-}lb$); // The maximum $k$-plex of size at least $fake\text{-}lb$ if exists; empty, otherwise.
5 return $fake\text{-}lb, |S|$;
```

Algorithm 3: `Improved-Iter-Search`(G, γ, ub)

```
Output: The size of the maximum $\gamma$-quasi-clique in $G$
1 $s_0 \leftarrow ub$; $k_1 \leftarrow$ get-k($s_0$); $i \leftarrow 1$;
2 while $true$ do
3     $fake\text{-}lb, fake\text{-}size \leftarrow$ Plex-Search($G$, $k_i$, $s_{i-1}$); $s_i \leftarrow \max\{fake\text{-}lb, fake\text{-}size\}$;
4     if $k_i =$ get-k($s_i$) and $fake\text{-}size \geq fake\text{-}lb$ then
5         $s^* \leftarrow s_i$; return $s^*$;
6     $k_{i+1} \leftarrow$ get-k($s_i$); $i \leftarrow i + 1$;
```

set the index i as 1 (Line 1). The algorithm then iteratively computes s_i by repeatedly invoking `Plex-Search` (instead of `solve-plex` in Algorithm 1), where k_i is updated by the function `get-k` at each iteration (Lines 2-6). Note that we obtain both $fake\text{-}lb$ and $fake\text{-}size$ from our `Plex-Search` and set s_i to be the greater of these two values in Line 3. The iteration terminates when $k_i = \texttt{get-k}(s_i)$ and $fake\text{-}size \geq fake\text{-}lb$ in Line 4, meaning that (1) the output returned from `Plex-Search` is the maximum k-plex even with the use of a fake lower bound; (2) the current iteration yields the same value of k for the next iteration from `Plex-Search`. At this point, we return s^* as the size of the maximum γ-quasi-clique in Line 5.

Remark. We note that our improved iterative search method in Algorithm 3 can use adaptions of any existing maximum k-plex algorithm [60], [13], [34], [58], [14], [27] for `Plex-Search`, provided that the existing solver can be decomposed into two steps: (1) heuristically obtaining a lower bound for the maximum k-plex, `Plex-Heu`, and (2) exhaustively conducting the exact branch-and-bound search for the final solution, `Plex-BRB`. In our work, we utilize one of the state-of-the-art maximum k-plex algorithms `KPEX` [27].

Correctness. We prove the correctness of our improved iterative search algorithm `Improved-Iterative-Search` in Algorithm 3, which includes the fake LB technique in Algorithm 2. We first show a property on the sequence generated by our basic iterative framework in Algorithm 1. To simplify the proofs, in the following discussion, let s_p denote the last element of the sequence $\{s_0, s_1, \ldots, s_p\}$ generated by the iterative frameworks (either Algorithm 1 or Algorithm 3). It is important to highlight that, in contrast to s_p in the correctness proof of the basic iterative framework, which represents the penultimate computed value, here s_p specifically corresponds to the value at the iteration where the algorithm 2 terminates, i.e., $k_p = \texttt{get-k}(s_p)$.

Lemma 6. *For any positive integer $\overline{s} \geq s^*$, the last element s_p of the sequence $\{\overline{s}, s_1, s_2, \ldots, s_p\}$ generated by Algorithm 1 is equal to s^*.*

Proof. The proof is similar to the proofs of Lemmas 4 and 5, which are based on a trivial upper bound, i.e., $|V| \geq s^*$. By using $\overline{s}$ instead of $|V|$, we can prove Lemma 6. □

Subsequently, in Lemma 7, we make a connection between the sequences generated by Algorithm 1 and Algorithm 3.

Lemma 7. *Consider the i-th iteration of Lines 2-6 in Algorithm 3, where `Plex-Search`$(G, k, \textit{ub-plex})$ is called in Line 3. Let $\overline{s} \leftarrow \texttt{solve-plex}(k)$, i.e., $\overline{s}$ is the size of maximum k-plex. If $k = \texttt{get-k}(\textit{ub-plex})$ and $\textit{ub-plex} = s_{i-1} \geq s^*$, then $s_i \geq \overline{s} \geq s^*$.*

Proof. In Algorithm 2, we invoke the branch-and-bound algorithm `Plex-BRB` with a fake lower bound of $fake\text{-}lb$ in Line 4, where $fake\text{-}lb = \lfloor (lb\text{-}plex + ub\text{-}plex)/2 \rfloor$ is not the true lower bound. In other words, with $fake\text{-}lb$, the size of the returned vertex set S may not be larger than our fake lower bound $fake\text{-}lb$. We have two possible cases as follows.

1) $\overline{s} > fake\text{-}lb$. In this case, since the pruning in `Plex-BRB` using $fake\text{-}lb$ does not affect the generation of the correct solution of maximum k-plex, the returned vertex set S from `Plex-BRB` corresponds to the maximum k-plex. Thus, we have $|S| = \overline{s}$. Moreover, as $s_i = \max\{fake\text{-}lb, |S|\}$, we know that $s_i \geq \overline{s}$.
2) $\overline{s} \leq fake\text{-}lb$. In this case, we note that in Line 4 of Algorithm 3, we have $s_i = \max\{fake\text{-}lb, |S|\}$, which directly implies that $s_i \geq \overline{s}$.

In either case mentioned above, we have $s_i \geq \overline{s}$. According to Lemmas 4 and 6 and $\textit{ub-plex} = s_{i-1} \geq s^*$, we know that $\overline{s} \geq s^*$. The proof is thus complete. □

Utilizing the connection, we verify the correctness of Algorithm 3 that uses the technique of fake LB in the following.

Lemma 8. *Algorithm 3 correctly identifies the largest γ-quasi-clique.*

Proof. According to Lemma 7, we can guarantee that $s_i \geq s^*$ for each iteration i in Lines 2-6 of Algorithm 3. Thus, we can view Algorithm 3 as a procedure that continuously seeks upper bounds for the maximum γ-quasi-clique across all iterations. Let s_p denote the last element of the sequence $\{s_0, s_1, \ldots, s_p\}$ generated by Algorithm 3. Next, we prove that s_p is equal to s^*.

Consider the i-th iteration in Lines 2-6 of Algorithm 3, where `Plex-Search`$(G, k, \textit{ub-plex})$ is called in Line 3. Note that $\textit{ub-plex} = s_{i-1}$. We focus on this `Plex-Search` and have the following two cases.

1) $\textit{lb-plex} = \textit{ub-plex}$. Since $\textit{ub-plex} = s_{i-1}$, it follows that $\textit{ub-plex} = s_{i-1} \geq s^*$. We also know that there

exists a k-plex of size *lb-plex* computed in Line 1 of Algorithm 2. Thus, we have a k-plex of size s_{i-1}, where k = `get-k`(s_{i-1}). Similar to Lemma 5, it is easy to conclude that the returned result in Line 5 of Algorithm 3 is exactly s^*. Thus, $s^* = s_{i-1} = s_i$.

2) *lb-plex* < *ub-plex*. It is easy to see that $fake\text{-}lb$ < *ub-plex*. If $s_i = fake\text{-}lb$ in Line 3 of Algorithm 3, it guarantees that $s_i < s_{i-1}$. Otherwise, i.e., $s_i = fake\text{-}size$, we use $\overline{s}_i$ to denote the size of the largest k-plex in the graph G, according to the proof of Lemma 7, $s_i = \overline{s_i}$, which still satisfies $s_i = \overline{s_i} < s_{i-1}$. Thus, in any case, we have $s_i < s_{i-1}$.

Since $\{s_0, s_1, \dots, s_p\}$ is an integer sequence with each $s_i \geq s^*$, the case in 2) is always finite. In other words, the case in 1) will definitely occur. The proof is complete. □

Example of the fake LB technique. To illustrate the effect of the fake LB technique, we present an example in Figure 3, where Figure 3(a) shows an example with 7 vertices and Figure 3(b) presents the corresponding iterative process of Figure 3(a). For better visualization of the iterative process, the line segments near $k = 2$ in Figure 3(b) are slightly offset along the positive k-axis. This adjustment is purely for visualization purposes and does not imply that the k-values used in the iterative process are non-integer. Assume that the heuristic solution obtained in each iteration corresponds to the subgraph in the dark-colored box, i.e., $G[\{0, 5, 6\}]$, in Figure 3(a), where *lb-plex* = 3 according to Line 1 of Algorithm 2.

As shown in Figure 3(b), using the fake LB technique, solving the problem with $\gamma = 0.75$ in Figure 3(a) requires two iterations. The first iteration, represented by the double-arrowed purple solid line in the figure, corresponds to the second case discussed in the proof of Lemma 7. At this stage, $fake\text{-}lb = \lfloor (3 + 7)/2 \rfloor = 5 > \overline{s}_1 = 4$. Consequently, the branch-and-bound search in Line 4 of Algorithm 2 fails to find a solution, resulting in $S = \emptyset$. Therefore, $s_1 = \max\{5, 0\} = 5$, to start the second iteration. In the second iteration, $fake\text{-}lb = \lfloor (5 + 3)/2 \rfloor = 4 \leq \overline{s}_2 = 4$, which corresponds to the first case discussed in the proof of Lemma 7. Since the pruning during the branch-and-bound search uses $fake\text{-}lb = 3$, it does not affect the computation of the solution S. Thus, $s_2 = \max\{3, 4\} = 4$, satisfying the condition in Line 4 of Algorithm 3. The final solution, i.e., $G[\{1, 2, 3, 4\}]$, is shown in Figure 3(a) within the light-colored box.

B. Our Improved Iterative Algorithm with Preprocessing

As mentioned, our improved iterative algorithm `IterQC` contains two key stages. One stage is the *iterative search stage* that iteratively conducts the maximum k-plex search with the technique of fake LB, which is discussed in Section IV-A. The next stage is the *preprocessing stage*, where we (1) compute lower and upper bounds lb and ub of the optimum solution s^*, (2) reduce the input graph using lb, and (3) set a tighter initial value for our iterative search using ub. Below, we elaborate on the details of the preprocessing technique in `IterQC`.

Algorithm 4: `Get-Bounds`(G, γ)

```
Output: The lower and upper bounds $lb$ and $ub$ of $s^*$
1 $lb \leftarrow 0$; $ub \leftarrow 0$; $max\_core \leftarrow 0$;
2 while $V(G) \neq \emptyset$ do
3     $u \leftarrow \arg\min_{v \in V(G)} d_G(v)$;
4     $max\_core \leftarrow \max(max\_core, d_G(u))$;
5     $ub \leftarrow \max(ub, \min(1 + \lceil max\_core/\gamma \rceil, |V(G)|))$;
6     if $G$ is a $\gamma$-quasi-clique and $lb = 0$ then
7         $lb \leftarrow |V(G)|$;
8     Remove $u$ from $G$;
9 return $lb$, $ub$;
```

Computation of lower and upper bounds. We first introduce some relevant concepts that are useful for our bounds.

Definition 3. *(k-core [52]) Given a graph G and an integer k, the k-core of G is the maximal subgraph g of G such that every vertex $u \in V(g)$ has degree $d_g(u) \geq k$ in g.*

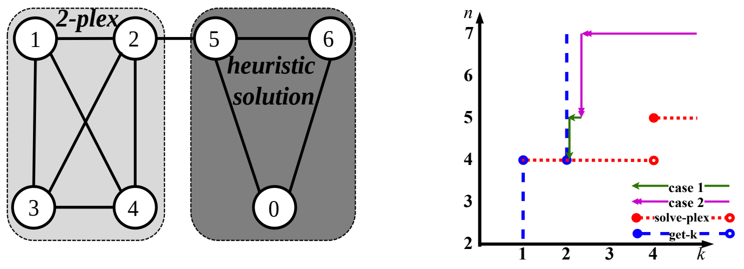
Based on the k-core, a related concept is the *core number* of a vertex u, denoted as $core(u)$, which refers to the largest k such that u is part of a k-core in the graph. In other words, u cannot belong to any subgraph where all vertices have a degree of at least $core(u)+1$. Let max_core be the maximum core number in the graph, i.e., $max\_core = \max_{u \in V} core(u)$. Note that given a graph $G = (V, E)$, the core numbers for all vertices can be computed in $O(|V| + |E|)$ time via the famous peeling algorithm [46], [5] which iteratively removes the vertices with the smallest degree from the current graph. We remark that in the peeling algorithm used to compute the core numbers, we can obtain the lower bound lb of s^* as a by-product. In particular, we check whether the current graph qualifies as a γ-quasi clique and keeps the size of this qualified γ-quasi clique. Moreover, as the current graph keeps shrinking during the peeling algorithm, our lower bound lb clearly corresponds to the size of the first current graph that meets the γ-quasi clique requirements. For the upper bound ub, we first have the following lemma by the core number.

Lemma 9. *Given a graph G and a real number $0 \leq \gamma \leq 1$, we have $s^* \leq 1 + \lceil max\_core/\gamma \rceil$, where s^* is the size of the maximum γ-quasi clique in G.*

Proof. The minimum degree of a γ-QC is at most max_core. By the definition of γ-QC, we complete the proof. □

Our method `Get-Bounds` to compute both lower and upper bounds is summarized in Algorithm 4. In particular, as mentioned, Algorithm 4 follows the peeling algorithm by iteratively removing the vertex with the smallest degree from the current graph and dynamically updating ub and lb.

The preprocessing stage. After obtaining both lower and upper bounds through `Get-Bounds`, we next show how to utilize these bounds in our preprocessing stage. Recall that, with lb and ub, we can (1) reduce the graph by removing unpromising vertices from G that cannot appear in g^* using



(a) An example graph. (b) Illustration of the example.

Fig. 3: An example of Algorithm 3 with $\gamma = 0.75$.

Algorithm 5: Our Improved Algorithm: `IterQC`

```
Input: A graph $G = (V, E)$ and a real value $0.5 \leq \gamma \leq 1$
Output: The size of the maximum $\gamma$-quasi-clique in $G$
// Stage 1: Preprocessing
1 $lb, ub \leftarrow$ Get-Bounds($G, \gamma$) ;
2 while $\min_{v \in V} d_G(v) \leq \lfloor (lb - 1) \cdot \gamma \rfloor$ do
3     $u \leftarrow \arg\min_{v \in V} d_G(v)$; Remove $u$ from $G$;
// Stage 2: Iterative Search
4 Improved-Iter-Search($G, \gamma, ub$);
```

lb, and (2) set a smaller initial value of k for the function `Plex-Search` using ub. The details are as follows.

For (1), we can iteratively remove vertices from G with degrees at most $\lfloor (lb-1) \cdot \gamma \rfloor$ and their incident edges, since it is clear that no solution of size greater than lb can still include these vertices. For (2), we can easily derive from Lemma 6 that instead of initializing s_0 to be $|V|$, we can set s_0 as an integer that is larger than s^*, and our iterative framework can still return the correct solution. This allows us to use the obtained upper bound ub to directly update s_0. We then utilize this ub to generate the initial value of k for `Plex-Search` through `get-k` while maintaining the correctness of the iterative framework.

Our improved iterative algorithm. Our improved iterative algorithm `IterQC` is summarized in Algorithm 5. In particular, Lines 1-3 correspond to Stage 1 of the preprocessing while Line 4 corresponds to Stage 2 of the iterative search. In Algorithm 5, we first call `Get-Bounds` to obtain both lower and upper bounds in Line 1 and iteratively remove unpromising vertices from the current graph using lb in Lines 2-3. Finally, we invoke `Improved-Iter-Search` using ub to get the final solution of the largest γ-quasi-clique.

Discussions. As mentioned before, in our implementation of `Improved-Iter-Search` (or specifically, `Plex-Solve` in Algorithm 2), we make use of the two components, i.e., the heuristic solution component and the branch-and-bound search component, of the maximum k-plex solver `kPEX` in [27]. The time complexity of our `IterQC` is dominated by `Plex-BRB` in Algorithm 2, which is invoked at most n times. The time complexities of `kPEX` are $O^*((k+1)^{\delta+k-s^*})$ and $O^*(\gamma_k^{\delta})$, where γ_k is the largest real root of $x^{k+2} - x^{k+1} - 2x^k + 2 = 0$, using the branching methods in [58], [14], which are the current best time complexities for the maximum k-plex problem, where O^* suppresses the polynomial factors, $\gamma_k < 2$ depending on the value of k, and δ is the degeneracy of the graph. As the sequence $\{s_1, s_2, \ldots, s_p\}$ generated by our improved iterative framework is monotonically decreasing and `get-k` is a non-decreasing function, each value of k_i is thus upper bounded by k_1. Consequently, the time complexity of each iteration depends on solving the maximum k-plex problem with k_1. Furthermore, with the preprocessing technique, k_1 can be expressed as $k_1 = \delta \frac{1-\gamma}{\gamma} \leq \delta$ (as $\gamma \in [0.5, 1]$). Consequently, the time complexity of `IterQC` is given by

$$O^*\left(\min\left\{(\delta+1)^{2\delta-s^*}, \gamma_\delta^\delta\right\}\right),$$

which improves upon the trivial time complexity of $O^*(2^n)$.

V. Experiments

In this section, we conduct extensive experiments to evaluate the practical performance of our `IterQC` algorithm, compared with the following competitors.

- `DDA`[1]: the state-of-the-art algorithm [47].
- `FastQC`[2]: the baseline adapted from the state-of-the-art algorithm for enumerating all maximal γ-QCs that are of size at least a given lower bound [63]. We adapt `FastQC` for our `MaxQC` problem by keeping track of the largest maximal γ-QC seen so far and dynamically adjusting the lower bound to facilitate better pruning, since the largest γ-QC is the largest one among all maximal γ-QCs.

Setup. All algorithms are implemented in C++, compiled with g++ -O3, and run on a machine with an Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz and 256GB main memory running Ubuntu 20.04.6. We set the time limit as 3 hours (i.e., 10,800s) and use **OOT** (Out Of Time limit) to represent the time exceeds the limit. Our source codes can be found in [1].

[1]We re-implemented the algorithm based on the description in [47]. Our implementation has better performances compared with the results shown in their paper.

[2]https://github.com/KaiqiangYu/SIGMOD24-MQCE

TABLE I: Statistics of 30 representative graphs.

ID	Graph	n	m	density	δ
G1	tech-WHOIS	7476	56943	$2.04 \cdot 10^{-3}$	88
G2	scc_retweet	17151	24015	$1.63 \cdot 10^{-4}$	19
G3	socfb-Berkeley	22900	852419	$3.25 \cdot 10^{-3}$	64
G4	ia-email-EU	32430	54397	$1.03 \cdot 10^{-4}$	22
G5	socfb-Penn94	41536	1362220	$1.58 \cdot 10^{-3}$	62
G6	sc-nasasrb	54870	1311227	$8.71 \cdot 10^{-4}$	35
G7	socfb-OR	63392	816886	$4.07 \cdot 10^{-4}$	52
G8	soc-Epinions1	75879	405740	$1.41 \cdot 10^{-4}$	67
G9	rec-amazon	91813	125704	$2.98 \cdot 10^{-5}$	4
G10	ia-wiki-Talk	92117	360767	$8.50 \cdot 10^{-5}$	58
G11	soc-LiveMocha	104103	2193083	$4.05 \cdot 10^{-4}$	92
G12	soc-gowalla	196591	950327	$4.92 \cdot 10^{-5}$	51
G13	delaunay_n18	262144	786395	$2.29 \cdot 10^{-5}$	4
G14	auto	448695	3314610	$3.29 \cdot 10^{-5}$	9
G15	soc-digg	770799	5907132	$1.99 \cdot 10^{-5}$	236
G16	ca-hollywood-2009	1069126	56306653	$9.85 \cdot 10^{-5}$	2208
G17	inf-belgium_osm	1441295	1549969	$1.49 \cdot 10^{-6}$	3
G18	soc-orkut	2997166	106349209	$2.37 \cdot 10^{-5}$	230
G19	rgg_n_2_22_s0	4194301	30359197	$3.45 \cdot 10^{-6}$	19
G20	inf-great-britain_osm	7733821	8156516	$2.73 \cdot 10^{-7}$	3
G21	inf-asia_osm	11950756	12711602	$1.78 \cdot 10^{-7}$	3
G22	hugetrace-00010	12057441	18082178	$2.49 \cdot 10^{-7}$	2
G23	inf-road_central	14081816	16933412	$1.71 \cdot 10^{-7}$	3
G24	hugetrace-00020	16002413	23998812	$1.87 \cdot 10^{-7}$	2
G25	rgg_n_2_24_s0	16777215	132557199	$9.42 \cdot 10^{-7}$	20
G26	delaunay_n24	16777216	50331600	$3.58 \cdot 10^{-7}$	4
G27	hugebubbles-00020	21198119	31790178	$1.41 \cdot 10^{-7}$	2
G28	inf-road-usa	23947347	28854312	$1.01 \cdot 10^{-7}$	3
G29	inf-europe_osm	50912018	54054659	$4.17 \cdot 10^{-8}$	3
G30	socfb-uci-uni	58790782	92208195	$5.34 \cdot 10^{-8}$	16

Datasets. We evaluate the algorithms on two graph collections (223 graphs in total) that are widely used in previous studies.

- The **real-world graphs** collection[3] contains 139 real-world graphs from Network Repository with up to 5.87×10^7 vertices.
- The **10th DIMACS graphs** collection[4] contains 84 graphs from the 10th DIMACS implementation challenge with up to 5.09×10^7 vertices.

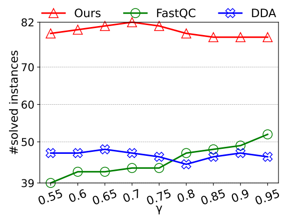
To provide a detailed comparison, we also select 30 representative graph instances as shown in Table I (where the density is computed as $\frac{2m}{n(n-1)}$). The selection criteria are as follows: (1) we first exclude those graph instances that are either overly simple or extremely difficult to be solved, i.e., those that can be solved within 3 seconds by all algorithms or cannot be solved within 3 hours by any algorithm; (2) We then select 10 small graphs (i.e., G1 to G10) with $n < 10^5$, 10 medium graphs (i.e., G11 to G20) with $10^5 \leq n < 10^7$, and 10 large graphs (i.e., G21 to G30) with $n \geq 10^7$.

A. Comparison with Baselines

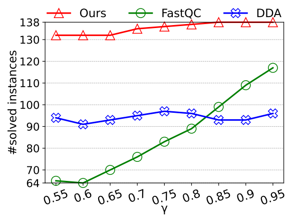
Number of solved instances. We first compare our algorithm `IterQC` with the baselines `DDA` and `FastQC` by considering the number of instances solved within 3-hour and 3-second limit on two collections in Figure 4. In addition, we present the number of instances solved over time for the two collections at γ values of 0.65, 0.75, 0.85, and 0.95, in Figures 5, and 6. We have the following observations. **First**, our algorithm

[3]http://lcs.ios.ac.cn/ caisw/Resource/realworld%20graphs.tar.gz

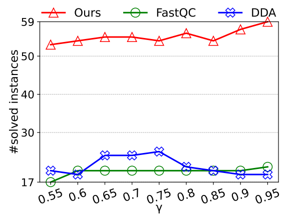
[4]https://sites.cc.gatech.edu/dimacs10/downloads.shtml



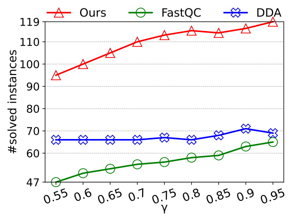
(a) 10th DIMACS (3-hour limit)



(b) real-world (3-hour limit)

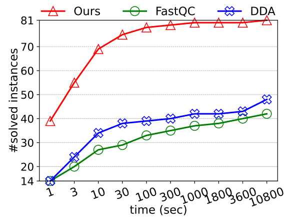


(c) 10th DIMACS (3-second limit)

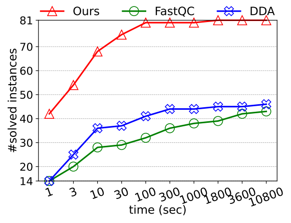


(d) real-word (3-second limit)

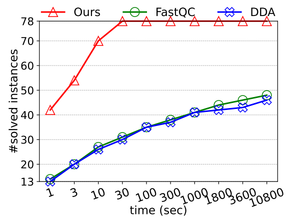
Fig. 4: Number of solved instances with different values of γ.



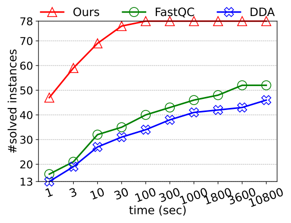
(a) $\gamma = 0.65$



(b) $\gamma = 0.75$



(c) $\gamma = 0.85$



(d) $\gamma = 0.95$

Fig. 5: Number of solved instances on 10th DIMACS graphs.

`IterQC` solves a greater number of instances across different values of γ, compared with the baselines `FastQC` and `DDA`. For example, in the real-world dataset with $\gamma = 0.7$ (in Figure 4(b)), `IterQC` solves 135 out of 139 instances, while `DDA` and `FastQC` only solve 95 and 76 instances, respectively. **Second**, in general, as the value of γ decreases, `IterQC` tends to use relatively larger values of k during each iteration, leading to higher computational costs and longer overall running time. As shown in Figures 4(b), 4(c) and 4(d), the number of instances solved by `IterQC` generally increases with γ. However, this characteristic is less apparent in Figure 4(a). Specifically, as γ decreases from 0.8 to 0.7, `IterQC` solves more instances. This phenomenon may be due to the following factors. `IterQC` computes a heuristic upper bound ub in the preprocessing stage, where the gap of this ub and the

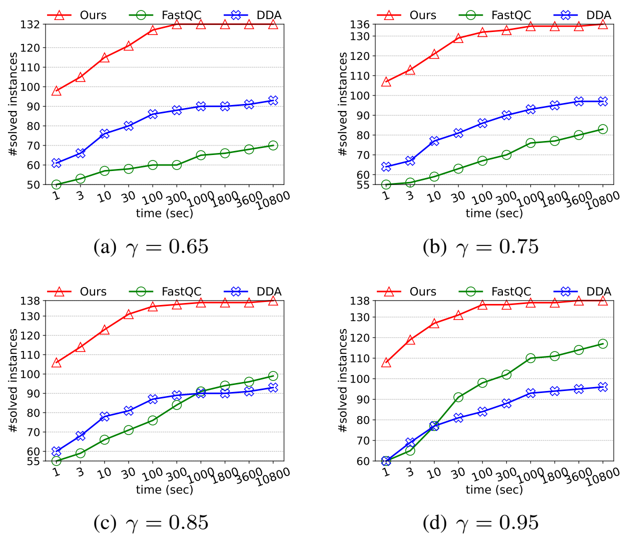


Fig. 6: Number of solved instances on real-world graphs.

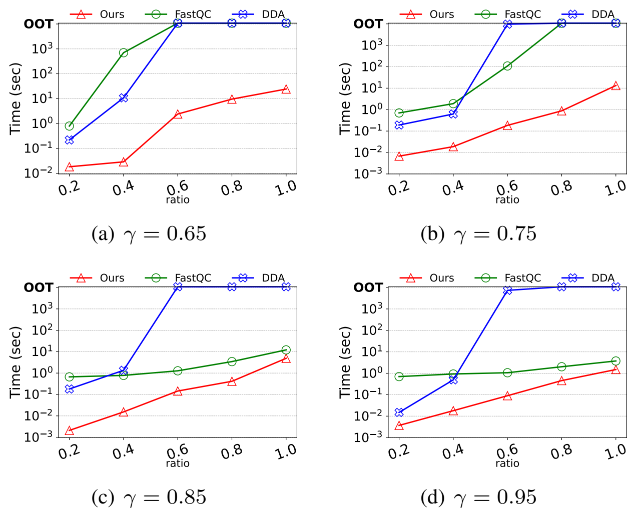


Fig. 7: Scalability test on **bio-pdb1HYS**.

optimum solution s^* is unpredictable for different values of γ. A smaller gap clearly results in a potentially shorter overall running time. Additionally, in our graph reduction process in the preprocessing stage, a smaller γ ensures a non-decreasing lower bound lb, enhancing the effectiveness of graph reduction and reducing the subsequent search costs. Moreover, the increased computational complexity associated with smaller values of γ primarily arises from the branch-and-bound search process. Intuitively, as γ decreases, the relaxation of the clique condition becomes more significant, which makes it harder to prune branches that could previously be terminated early. The fake LB technique we utilized essentially serves to accelerate the branch-and-bound approach. Consequently, it also helps reduce the increased difficulty introduced by smaller values of γ. **Third**, we also observe in Figures 5 and 6 that the number of instances that IterQC can solve within 3 second exceeds the number solved by the other two baselines within three hours. For example, on 10th DIMACS graphs with $\gamma = 0.75$, IterQC solves 54 instances within 3 seconds, while DDA and FastQC solve 46 and 43 instances within 3 hours, respectively. Moreover, as shown in Figure 6(c), IterQC can complete 105 instances in 1 second, while DDA and FastQC finish 93 and 99 instances within 3 hours, respectively. This significant practical efficiency improvements achieved by IterQC may be due to the following aspects. **(1)** IterQC solves MaxQC by iteratively solving multiple k-plex-related problem, which exhibits structural properties to support efficient computations. **(2)** Before performing the iterative framework, IterQC utilizes the preprocessing stage to effectively reduce the graph size and provide a tight upper bound of the size of the optimum solution. **(3)** For the iterative search stage, we propose the technique of fake LB to accelerate the algorithmic ability to solve challenging graphs efficiently.

Performance on representative instances. The runtime performance comparison between IterQC and the two baseline algorithms with $\gamma = 0.65$, 0.75, and 0.85 across 30 representative instances is shown in Tables II to IV. As illustrated in these tables, IterQC consistently demonstrates superior efficiency, outperforming both baselines FastQC and DDA across nearly all instances. In particular, IterQC can solve all the graph instances, while both baseline algorithms FastQC and DDA exhibit a high occurrence of timeouts, failing to yield solutions within the 3-hour limit; for example, 25 out of 30 instances for FastQC and 18 out of 30 instances for DDA result in **OOT** in Table II and at $\gamma = 0.65$, they fail to solve 23 and 16 instances, respectively, as shown in Table III. For each γ value (0.65, 0.75, or 0.85), the IterQC algorithm successfully solves 14 out of the 30 representative instances where both baseline algorithms exceed the 3-hour time limit. Moreover, on G22 in Table IV, IterQC uses only 0.19 second, while both baselines cannot complete in 3 hours, achieving at least 56,000$\times$ speed-ups. This result further suggests that efficiency superiority of IterQC over both FastQC and DDA.

Scalability test. We use the **bio-pdb1HYS** graph (with 36,417 vertices and 2,154,174 edges) from the 10th DIMACS collection to evaluate the scalability of IterQC. In our experiment, we randomly extract 20% to 100% of the vertices and test the performance of IterQC and two baselines with γ values of 0.65, 0.75, 0.85, and 0.95 in Figure 7. The results demonstrate two findings: First, in all cases, IterQC consistently uses the smallest running time. Second, the advantage of IterQC is more obvious at smaller values of γ, where it outperforms the baselines by 2 to 3 orders of magnitude. With $\gamma = 0.75$ and an extraction ratio of 80%, IterQC solves the case in 0.88 seconds, while the other two algorithms timeout, demonstrating a speedup of at least 12000$\times$. These results validate the efficiency and scalability of IterQC.

B. Ablation Studies

We conduct ablation studies to evaluate the effectiveness of the techniques of preprocessing and fake LB proposed in

TABLE II: Runtime performance (in seconds) of `IterQC`, `FastQC`, and `DDA` on 30 instances with $\gamma = 0.65$.

ID	IterQC	FastQC	DDA	ID	IterQC	FastQC	DDA
G1	**5.01**	98.83	OOT	G16	**0.02**	3.89	1.92
G2	**5.87**	OOT	OOT	G17	**4.08**	OOT	26.85
G3	**0.49**	33.99	OOT	G18	**19.65**	OOT	91.48
G4	**26.13**	OOT	OOT	G19	**0.01**	OOT	1.04
G5	**9.70**	OOT	OOT	G20	**6.19**	OOT	OOT
G6	**4.82**	OOT	6845.30	G21	OOT	OOT	OOT
G7	**6.47**	OOT	OOT	G22	**3.06**	OOT	OOT
G8	**0.04**	2211.78	101.39	G23	**0.55**	OOT	OOT
G9	**36.31**	OOT	OOT	G24	**0.33**	OOT	OOT
G10	**2.42**	OOT	21.64	G25	**6.07**	OOT	OOT
G11	**0.30**	1041.55	3.22	G26	**23.28**	OOT	OOT
G12	**13.80**	OOT	OOT	G27	**3.38**	OOT	OOT
G13	**1.80**	OOT	13.39	G28	**135.13**	OOT	OOT
G14	**5.33**	OOT	26.90	G29	OOT	OOT	OOT
G15	**6.49**	OOT	40.70	G30	**0.01**	OOT	0.25

TABLE III: Runtime performance (in seconds) of `IterQC`, `FastQC`, and `DDA` on 30 instances with $\gamma = 0.75$.

ID	IterQC	FastQC	DDA	**ID**	IterQC	FastQC	DDA
G1	**0.03**	OOT	0.16	G16	**5.70**	OOT	OOT
G2	**0.004**	652.72	0.10	G17	**0.24**	1073.59	0.34
G3	**0.37**	OOT	OOT	G18	**341.59**	OOT	OOT
G4	**0.02**	4.93	69.71	G19	4.06	OOT	**2.73**
G5	**0.91**	OOT	OOT	G20	**1.38**	OOT	1.49
G6	**4.46**	80.61	OOT	G21	**1.87**	OOT	2.14
G7	**0.24**	OOT	3059.20	G22	**8.70**	OOT	OOT
G8	**0.32**	OOT	OOT	G23	3.98	OOT	**3.45**
G9	**0.02**	3.88	0.20	G24	**13.07**	OOT	OOT
G10	**2.46**	OOT	OOT	G25	18.00	OOT	**9.30**
G11	**17.71**	OOT	OOT	G26	**14.86**	OOT	OOT
G12	**0.41**	OOT	OOT	G27	**18.64**	OOT	OOT
G13	**0.26**	34.60	297.66	G28	**4.75**	OOT	42.54
G14	**3.28**	106.46	OOT	G29	**9.377**	OOT	9.378
G15	**772.93**	OOT	OOT	G30	**25.50**	OOT	OOT

TABLE IV: Runtime performance (in seconds) of `IterQC`, `FastQC`, and `DDA` on 30 instances with $\gamma = 0.85$.

ID	IterQC	FastQC	DDA	ID	IterQC	FastQC	DDA
G1	**3.48**	102.83	OOT	G16	**0.02**	3.90	0.48
G2	**5.90**	OOT	OOT	G17	**4.14**	OOT	16.63
G3	**0.38**	34.95	2490.80	G18	**17.61**	OOT	46.32
G4	**29.11**	OOT	OOT	G19	**0.13**	OOT	0.88
G5	**18.28**	OOT	OOT	G20	**5.02**	5.06	OOT
G6	**8.84**	OOT	OOT	G21	**57.88**	OOT	OOT
G7	**13.03**	OOT	OOT	G22	**0.19**	OOT	OOT
G8	**0.01**	1.18	21.53	G23	**0.30**	OOT	OOT
G9	**0.99**	OOT	OOT	G24	**0.21**	203.62	OOT
G10	**2.65**	OOT	21.88	G25	**0.50**	85.27	OOT
G11	**0.24**	1115.24	517.41	G26	**21.31**	OOT	OOT
G12	**12.32**	OOT	93.58	G27	**0.45**	OOT	OOT
G13	**1.90**	OOT	OOT	G28	**7.95**	OOT	OOT
G14	**5.13**	OOT	25.93	G29	**91.45**	OOT	OOT
G15	**6.49**	OOT	39.78	G30	**0.08**	OOT	3.43

Section IV. We compare `IterQC` with the following variants:

- `IterQC-PP`: it removes the preprocessing technique in `IterQC`, which includes (1) initial estimation of lower and upper bounds, and (2) graph reduction. Specifically, `IterQC-PP` replaces Lines 1-3 in Algorithm 5 with $ub \leftarrow |V|$.
- `IterQC-FKLB`: it removes the fake lower bound $fake\text{-}lb$ and utilizes the true heuristic lower bound by replacing Line 3 in Algorithm 2 with $fake\text{-}lb \leftarrow lb\text{-}plex$.

Table V presents the runtime performance of `IterQC`, `IterQC-PP`, and `IterQC-FKLB` on 30 representative instances with $\gamma = 0.75$. The results for other values of γ can be found in Appendix of the technical report [1].

TABLE V: Runtime performance (in seconds) of `IterQC`, `IterQC-PP`, and `IterQC-FKLB` on 30 instances with $\gamma = 0.75$.

ID	IterQC	-PP	-FKLB	**ID**	IterQC	-PP	-FKLB
G1	**0.034**	0.10	0.035	G16	**5.70**	35.54	6.31
G2	**0.0043**	0.039	0.0044	G17	**0.24**	2.96	0.32
G3	**0.37**	2.21	0.41	G18	**341.59**	4946.34	OOT
G4	0.019	0.08	**0.015**	G19	**4.06**	16.96	4.14
G5	0.91	6.64	**0.80**	G20	**1.38**	17.11	1.73
G6	**4.46**	6.65	4.86	G21	**1.87**	29.47	2.50
G7	0.24	1.36	**0.17**	G22	**8.70**	49.07	9.29
G8	**0.32**	1.24	0.61	G23	**3.98**	37.37	5.15
G9	0.021	0.17	**0.018**	G24	**13.07**	53.49	14.76
G10	**2.46**	2.51	29.79	G25	**18.00**	81.90	18.60
G11	**17.71**	97.73	134.01	G26	**14.86**	113.84	16.36
G12	**0.41**	1.05	0.66	G27	**18.64**	79.77	21.07
G13	0.26	0.99	**0.22**	G28	**4.75**	87.08	4.94
G14	**3.28**	11.15	3.42	G29	**9.38**	191.62	9.99
G15	**772.93**	1385.18	OOT	G30	**25.50**	195.29	25.97

Effectiveness of the preprocessing technique. We can see from Table V that `IterQC` consistently outperforms `IterQC-PP`, achieving a speedup factor of at least 5 in 17 instances and at least 10 in 6 instances, with a remarkable speedup factor of 20.32 on G29. We also summarize additional information for our preprocessing technique in Table VI, which details the percentages of vertices and edges pruned in the preprocessing stage, as well as the lower and upper bounds (lb and ub) for the optimum solution s^*. From Table VI, we observe that in 3 instances (G17, G20, and G29), the preprocessing technique prunes all vertices and edges, effectively obtaining the solution directly, while in 15 instances, it removes at least 90% of the vertices. Moreover, across the 30 instances, we observe that the preprocessing technique enables the iteration process to start from a smaller initial value (closer to the optimum solution s^*), as indicated by the upper bound ub (in contrast to the trivial upper bound $|V|$ in Table I).

Effectiveness of the fake LB technique. We can see in Table V that applying the fake LB technique leads to an improved performance in 25 of these 30 instances. Moreover, compared to `IterQC-FKLB`, `IterQC` successfully solves 2 additional **OOT** instances, i.e., G15 and G18. For G18, `IterQC` solves in 341.59 seconds while `IterQC-FKLB` times out, implying a speedup factor of at least 31.62. This improvement is due to the acceleration of the branch-and-bound search by the fake LB technique in Algorithm 2, especially leveraging the graph structure: dense local regions increase branch-and-bound search costs, resulting in improved speedups. Conversely, instances like G4, G5, G7, G9, and G13 show lower effectiveness when the running time is dominated by the computations of heuristic lower bound in Line 4 of Algorithm 2. Despite this, even in these instances, the impact on running time is minor, with all such instances completing in under 1 second.

VI. Related Work

Maximum γ-quasi-clique search problem. The maximum γ-quasi-clique search problem has been proven to be NP-hard [45], [47] and has been proven to be W[1]-hard parameterized by several graph parameters [4], [3]. The state-of-the-art exact algorithm for solving this problem is `DDA`,

TABLE VI: Preprocessing information with $\gamma = 0.75$, where Red-V/Red-E represent the percentages of reduced vertices/edges.

ID	Red-V (%)	Red-E (%)	*lb*	*ub*	s^*
G1	98.29	87.61	116	119	117
G2	80.85	62.72	230	233	230
G3	65.86	46.16	74	87	74
G4	98.38	84.10	17	31	21
G5	60.22	39.17	57	84	66
G6	1.56	0.84	24	48	30
G7	93.32	77.40	54	71	59
G8	96.85	67.91	57	91	58
G9	98.36	97.56	6	7	6
G10	96.15	63.47	29	79	33
G11	42.27	8.76	13	124	45
G12	98.08	86.33	37	69	45
G13	0.00	0.00	3	7	6
G14	0.00	0.00	6	13	10
G15	97.49	53.99	86	316	127
G16	99.79	95.67	2209	2945	2209
G17	100.00	100.00	5	5	5
G18	76.82	52.51	66	308	130
G19	99.87	99.84	20	27	25
G20	100.00	100.00	5	5	5
G21	25.34	23.82	3	5	5
G22	0.00	0.00	2	4	2
G23	0.00	0.00	2	5	5
G24	0.00	0.00	2	4	2
G25	99.99	99.99	25	28	27
G26	1.15	1.15	5	7	6
G27	0.00	0.00	2	4	2
G28	29.35	24.36	3	5	5
G29	100.00	100.00	5	5	5
G30	99.72	98.72	10	23	10

proposed in [47] and extensively discussed in Section I. In contrast, Bhattacharyya and Bandyopadhyay [8] provided a greedy heuristic that iteratively either removes a vertex with minimum degree or adds a vertex with maximum degree. Further studies [17], [37] addressed related problems of finding the largest maximal quasi-cliques that include a given target vertex or vertex set in a graph. Moreover, Marinelli et al. [44] proposed a method based on Integer Programming to compute upper bounds of the maximum γ-quasi-clique.

Maximal γ-quasi-clique enumeration problem. A closely related problem is the enumeration of all maximal γ-quasi-cliques in a given graph [39], [36], [63]. A γ-QC g is considered *maximal* if and only if there exists no supergraph g' of g such that g' is also a γ-QC. Several branch-and-bound algorithms have been proposed to tackle this problem by using multiple pruning techniques to reduce the search space during enumeration. In particular, Liu and Wong [39], Guo et al. [29], and Khalil et al. [36] developed such algorithms to improve efficiency. Recently, Yu and Long [63] introduced `FastQC`, the current state-of-the-art algorithm, which combines a pruning and branching co-design approach. This method achieves significant improvements by breaking the trivial time complexity of $O^*(2^n)$ and demonstrating improved efficiency in practice. We remark that the maximum γ-QC search problem can be solved using algorithms designed for maximal γ-QC enumeration, as the maximum γ-QC is always a maximal one in the graph. In our experimental studies, we adapt the state-of-the-art maximal γ-QC enumeration algorithm, `FastQC`, as the baseline method to solve our problem.

Some studies adopt slightly different problem constraints on γ-quasi-cliques. For example, Sanei-Mehri et al. [51] focused on enumerating only the top-k largest quasi-cliques in a graph, rather than enumerating all of them. Guo et al. [30] studied the problem of mining top-k large (γ_1, γ_2)-quasi-cliques in directed graphs. In addition, other extensions considered graph databases instead of a single graph. For example, Jiang and Pei [33] addressed the problem of jointly mining cross-graph quasi-cliques in a graph database. Specifically, a vertex set S is defined as a cross-graph quasi-clique if $G_i[S]$ is a γ_i-quasi-clique for all graphs G_i in the database and if no superset of S satisfies this. Similarly, Zeng et al. [67] considered a similar problem, but reported a vertex set S when the number of subgraphs $G_i[S]$ that form a γ-quasi-clique exceeds a specified minimum threshold (unlike the requirement in [33], which considers all graphs). Note that the γ-quasi-clique problem is a special case of the cross-graph quasi-cliques problem when the graph database contains only a single graph.

Other cohesive subgraph mining problems. Another approach to cohesive subgraph mining involves relaxing the clique definition from the perspective of edges. This approach gives rise to the concept of the edge-based γ-quasi-clique [2], [18], [48], which is also referred to as pseudo-cliques [57], dense subgraphs [41], or near-cliques [9], [55]. In this cohesive subgraph model, the total number of edges in a subgraph must be at least $\gamma \cdot \binom{n}{2}$. Very recently, Rahman et al. [50] introduced a novel pruning strategy based on Turán's theorem [32] to obtain an exact solution, building on the PCE algorithm proposed by Uno [57]. Similar to the γ-quasi-clique problem, several studies have also explored non-exact approaches to solve the edge-based γ-quasi-clique problem [2], [10], [15], [40]. For instance, Tsourakakis et al. [56] proposed an objective function that unifies the concepts of average-degree-based quasi-cliques and edge-based γ-quasi-clique.

There also exist many other types of cohesive subgraphs, including k-plex [21], [59], [69], [13], [25], [34], [35], [58], [60], [68], k-defective clique [11], [20], [26], and densest subgraph [43], [61]. Moreover, the topic of cohesive subgraphs has also been widely studied in other types of graphs, including bipartite graphs [16], [22], [42], [66], [64], [65], directed graphs [28], temporal graphs [7], and uncertain graphs [19]. For an overview on cohesive subgraphs, see the excellent books and survey, e.g., [12], [23], [24], [31], [38].

VII. Conclusion

In this paper, we studied the maximum γ-quasi clique problem. We first proposed an innovative iterative framework that transform the problem, which lacks with the hereditary property, into multiple problems possessing the favorable hereditary property. To further improve the practical efficiency of our algorithm, we also developed techniques such as the fake lower bound and preprocessing. These methods reduce the challenges posed by the increased difficulty of solving the problem for smaller values of γ. Finally, extensive experimental studies confirmed the superiority of our proposed algorithm `IterQC` over existing state-of-the-art algorithms `DDA` and `FastQC`. In the future, we plan to apply our iterative method to other cohesive graph models.

References

[1] https://github.com/SmartProbiotics/IterQC, Technical Report and Source Code.

[2] J. Abello, M. G. Resende, and S. Sudarsky, "Massive quasi-clique detection," in *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*, 2002, pp. 598–612.

[3] A. Baril, A. Castillon, and N. Oijid, "On the parameterized complexity of non-hereditary relaxations of clique," *Theoretical Computer Science*, vol. 1003, p. 114625, 2024.

[4] A. Baril, R. Dondi, and M. M. Hosseinzadeh, "Hardness and tractability of the γ-complete subgraph problem," *Information Processing Letters*, vol. 169, p. 106105, 2021.

[5] V. Batagelj and M. Zaveršnik, "An $O(m)$ algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[6] P. Bedi and C. Sharma, "Community detection in social networks," *Data Mining and Knowledge Discovery*, vol. 6, no. 3, pp. 115–135, 2016.

[7] M. Bentert, A.-S. Himmel, H. Molter, M. Morik, R. Niedermeier, and R. Saitenmacher, "Listing all maximal k-plexes in temporal graphs," *ACM J. Exp. Algorithmics*, vol. 24, no. 1, pp. 1.13:1–1.13:27, 2019.

[8] M. Bhattacharyya and S. Bandyopadhyay, "Mining the largest quasi-clique in human protein interactome," in *Proceedings of International Conference on Adaptive and Intelligent Systems*, 2009, pp. 194–199.

[9] Z. Brakerski and B. Patt-Shamir, "Distributed discovery of large near-cliques," *Distributed Computing*, vol. 24, pp. 79–89, 2011.

[10] M. Brunato, H. H. Hoos, and R. Battiti, "On effectively finding maximal quasi-cliques in graphs," in *Proceedings of the International Conference on Learning and Intelligent Optimization*, 2008, pp. 41–55.

[11] L. Chang, "Efficient maximum k-defective clique computation with improved time complexity," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 3, pp. 1–26, 2023.

[12] L. Chang and L. Qin, *Cohesive Subgraph Computation over Large Sparse Graphs*. Springer, 2018.

[13] L. Chang, M. Xu, and D. Strash, "Efficient maximum k-plex computation over large sparse graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 2, pp. 127–139, 2022.

[14] L. Chang and K. Yao, "Maximum k-plex computation: Theory and practice," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 2, no. 1, pp. 1–26, 2024.

[15] J. Chen, S. Cai, S. Pan, Y. Wang, Q. Lin, M. Zhao, and M. Yin, "NuQClq: An effective local search algorithm for maximum quasi-clique problem," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021, pp. 12 258–12 266.

[16] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient exact algorithms for maximum balanced biclique search in bipartite graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2021, pp. 248–260.

[17] Y. H. Chou, E. T. Wang, and A. L. P. Chen, "Finding maximal quasi-cliques containing a target vertex in a graph," in *Proceedings of the International Conference on DATA*, 2015, pp. 5–15.

[18] P. Conde-Cespedes, B. Ngonmang, and E. Viennet, "An efficient method for mining the maximal α-quasi-clique-community of a given node in complex networks," *Social Network Analysis and Mining*, vol. 8, no. 1, pp. 1–18, 2018.

[19] Q. Dai, R.-H. Li, M. Liao, H. Chen, and G. Wang, "Fast maximal clique enumeration on uncertain graphs: A pivot-based approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2022, pp. 2034–2047.

[20] Q. Dai, R.-H. Li, M. Liao, and G. Wang, "Maximal defective clique enumeration," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 1, pp. 1–26, 2023.

[21] Q. Dai, R.-H. Li, H. Qin, M. Liao, and G. Wang, "Scaling up maximal k-plex enumeration," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 345–354.

[22] Q. Dai, R.-H. Li, X. Ye, M. Liao, W. Zhang, and G. Wang, "Hereditary cohesive subgraphs enumeration on bipartite graphs: The power of pivot-based approaches," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 2, pp. 1–26, 2023.

[23] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, pp. 353–392, 2020.

[24] Y. Fang, K. Wang, X. Lin, and W. Zhang, "Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2021, pp. 2829–2838.

[25] J. Gao, J. Chen, M. Yin, R. Chen, and Y. Wang, "An exact algorithm for maximum k-plexes in massive graphs," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 1449–1455.

[26] J. Gao, Z. Xu, R. Li, and M. Yin, "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 (AAAI)*, 2022, pp. 10 174–10 183.

[27] S. Gao, K. Yu, S. Liu, and C. Long, "Maximum k-plex search: An alternated reduction-and-bound method," *Proceedings of the VLDB Endowment*, 2025.

[28] S. Gao, K. Yu, S. Liu, C. Long, and Z. Qiu, "On searching maximum directed (k, ℓ)-plex," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2024, pp. 2570–2583.

[29] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil, "Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 573–585, 2020.

[30] G. Guo, D. Yan, L. Yuan, J. Khalil, C. Long, Z. Jiang, and Y. Zhou, "Maximal directed quasi-clique mining," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2022, pp. 1900–1913.

[31] X. Huang, L. V. S. Lakshmanan, and J. Xu, *Community Search over Big Graphs*. Morgan & Claypool Publishers, 2019.

[32] S. Jain and C. Seshadhri, "A fast and provable method for estimating clique counts using Turán's theorem," in *Proceedings of the International Conference on World Wide Web (WWW)*, 2017, pp. 441–449.

[33] D. Jiang and J. Pei, "Mining frequent cross-graph quasi-cliques," *ACM Trans. Knowl. Discov. Data*, vol. 2, no. 4, pp. 16:1–16:42, 2009.

[34] H. Jiang, F. Xu, Z. Zheng, B. Wang, and W. Zhou, "A refined upper bound and inprocessing for the maximum k-plex problem," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2023, pp. 5613–5621.

[35] H. Jiang, D. Zhu, Z. Xie, S. Yao, and Z.-H. Fu, "A new upper bound based on vertex partitioning for the maximum k-plex problem," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2021, pp. 1689–1696.

[36] J. Khalil, D. Yan, G. Guo, and L. Yuan, "Parallel mining of large maximal quasi-cliques," *The VLDB Journal*, vol. 31, no. 4, pp. 649–674, 2022.

[37] P. Lee and L. V. Lakshmanan, "Query-driven maximum quasi-clique search," in *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2016, pp. 522–530.

[38] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*. Springer, 2010, pp. 303–336.

[39] G. Liu and L. Wong, "Effective pruning techniques for mining quasi-cliques," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, 2008, pp. 33–49.

[40] S. Liu, J. Zhou, an Wang, Z. Zhang, and M. Lei, "An optimization algorithm for maximum quasi-clique problem based on information feedback model," *PeerJ Comput. Sci.*, vol. 10, p. e2173, 2024.

[41] J. Long and C. Hartman, "Odes: An overlapping dense sub-graph algorithm," *Bioinformatics*, vol. 26, no. 21, pp. 2788–2789, 2010.

[42] W. Luo, K. Li, X. Zhou, Y. Gao, and K. Li, "Maximum biplex search over bipartite graphs," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2022, pp. 898–910.

[43] C. Ma, Y. Fang, R. Cheng, L. V. S. Lakshmanan, W. Zhang, and X. Lin, "On directed densest subgraph discovery," *ACM Transactions on Database Systems*, vol. 46, no. 4, pp. 1–45, 2021.

[44] F. Marinelli, A. Pizzuti, and F. Rossi, "LP-based dual bounds for the maximum quasi-clique problem," *Discrete Applied Mathematics*, vol. 296, pp. 118–140, 2021.

[45] H. Matsuda, T. Ishihara, and A. Hashimoto, "Classifying molecular sequences using a linkage graph with their pairwise similarities," *Theoretical Computer Science*, vol. 210, no. 2, pp. 305 – 325, 1999.

[46] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.

[47] G. Pastukhov, A. Veremyev, V. Boginski, and O. A. Prokopyev, "On maximum degree-based γ-quasi-clique problem: Complexity and exact approaches," *Networks*, vol. 71, no. 2, pp. 136–152, 2018.

[48] J. Pattillo, A. Veremyev, S. Butenko, and V. Boginski, "On the maximum quasi-clique problem," *Discrete Applied Mathematics*, vol. 161, no. 1-2, pp. 244–257, 2013.

[49] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2005, pp. 228–238.

[50] A. Rahman, K. Roy, R. Maliha, and T. F. Chowdhury, "A fast exact algorithm to enumerate maximal pseudo-cliques in large sparse graphs," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2024, pp. 2479—2490.

[51] S.-V. Sanei-Mehri, A. Das, H. Hashemi, and S. Tirthapura, "Mining largest maximal quasi-cliques," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 5, pp. 81:1–81:21, 2021.

[52] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[53] S. B. Seidman and B. L. Foster, "A graph-theoretic generalization of the clique concept," *Journal of Mathematical Sociology*, vol. 6, no. 1, pp. 139–154, 1978.

[54] A. Suratanee, M. H. Schaefer, M. J. Betts, Z. Soons, H. Mannsperger, N. Harder, M. Oswald, M. Gipp, E. Ramminger, G. Marcus *et al.*, "Characterizing protein interactions employing a genome-wide siRNA cellular phenotyping screen," *PLoS Computational Biology*, vol. 10, no. 9, p. e1003814, 2014.

[55] S. Tadaka and K. Kinoshita, "Ncmine: Core-peripheral based functional module detection using near-clique mining," *Bioinformatics*, vol. 32, no. 22, pp. 3454–3460, 2016.

[56] C. E. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. A. Tsiarli, "Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2013, pp. 104–112.

[57] T. Uno, "An efficient algorithm for solving pseudo clique enumeration problem," *Algorithmica*, vol. 56, no. 1, pp. 3–16, 2010.

[58] Z. Wang, Y. Zhou, C. Luo, and M. Xiao, "A fast maximum k-plex algorithm parameterized by the degeneracy gap," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2023, pp. 5648–5656.

[59] Z. Wang, Y. Zhou, M. Xiao, and B. Khoussainov, "Listing maximal k-plexes in large real-world graphs," in *Proceedings of the ACM Web Conference (WWW)*, 2022, pp. 1517–1527.

[60] M. Xiao, W. Lin, Y. Dai, and Y. Zeng, "A fast algorithm to compute maximum k-plexes in social network analysis," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2017, pp. 919–925.

[61] Y. Xu, C. Ma, Y. Fang, and Z. Bao, "Efficient and effective algorithms for densest subgraph discovery and maintenance," *The VLDB Journal*, 2024.

[62] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein, "Predicting interactions in protein networks by completing defective cliques," *Bioinformatics*, vol. 22, no. 7, pp. 823–829, 2006.

[63] K. Yu and C. Long, "Fast maximal quasi-clique enumeration: A pruning and branching co-design approach," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, vol. 1, no. 3, 2023, pp. 1–26.

[64] ——, "Maximum k-biplex search on bipartite graphs: A symmetric-bk branching approach," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 1, pp. 1–26, 2023.

[65] K. Yu, C. Long, P. Deepak, and T. Chakraborty, "On efficient large maximal biplex discovery," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 824–829, 2023.

[66] K. Yu, C. Long, S. Liu, and D. Yan, "Efficient algorithms for maximal k-biplex enumeration," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2022, pp. 860–873.

[67] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Out-of-core coherent closed quasi-clique mining from large dense graph databases," *ACM Trans. Database Syst.*, vol. 32, no. 2, pp. 13:1–13:40, 2007.

[68] S. H. Zhou, M. Xiao, and Z.-H. Fu, "Improving maximum k-plex solver via second-order reduction and graph color bounding," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021, pp. 12 453–12 460.

[69] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin, "Enumerating maximal k-plexes with worst-case time guarantee," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020, pp. 2442–2449.

Appendix

A. Additional Experiments

Number of solved instances. Figures 8 and 9 illustrate the trends in the number of solved instances over time for the `IterQC`, `FastQC`, and `DDA` algorithms on the 10th DIMACS and real-world datasets, under γ values of 0.55, 0.6, 0.7, 0.8, and 0.9. From the results, it is easy to see that under the same time constraint, for any given value of γ or dataset, `IterQC` consistently solves more instances compared to the baseline algorithms. Remarkably, regardless of the dataset or γ, `IterQC` solves more instances within 3 seconds than the larger number achieved by either baseline algorithm within 3 hours. For instance, at $\gamma = 0.6$, `IterQC` solves 53 instances in only 3 seconds, while `FastQC` and `DDA` solve only 41 and 58 instances, respectively, even after 3 hours of computation. Across all tested values of γ, `IterQC` achieves improved practical performance. On the 10th DIMACS dataset, it solves at least 79 out of 84 instances, while on the real-world dataset, it solves at least 132 out of 139 instances.

Ablation Studies. Table VII to IX present the results of the ablation studies conducted on 30 representative instances, demonstrating the impact of preprocessing and the fake lower bound (fake LB) techniques on practical performance. The results clearly indicate significant improvements across all values of γ. For the preprocessing technique, the optimization effect is clear in nearly every case. For example, at $\gamma = 0.65$, instance G30 achieves a speedup of approximately 135000× compared to the setting without the preprocessing technique. Regarding the fake LB technique, substantial benefits are observed in most instances, particularly for smaller values of γ. At $\gamma = 0.65$, 23 out of 30 instances show accelerated running time. Specifically, for G28 at $\gamma = 0.65$ and G21 at $\gamma = 0.85$, the introduction of the fake LB technique transforms previously unsolvable instances within the 3-hour limit into solvable ones. For G21 at $\gamma = 0.85$, the running time is reduced to 57.88 seconds, corresponding to a speedup of 186×. These findings highlight the critical contributions of both preprocessing and fake LB techniques in improving efficiency of the proposed method.

TABLE VII: Runtime performance (in seconds) of `IterQC`, `IterQC-PP`, and `IterQC-FKLB` on 30 instances with $\gamma = 0.65$.

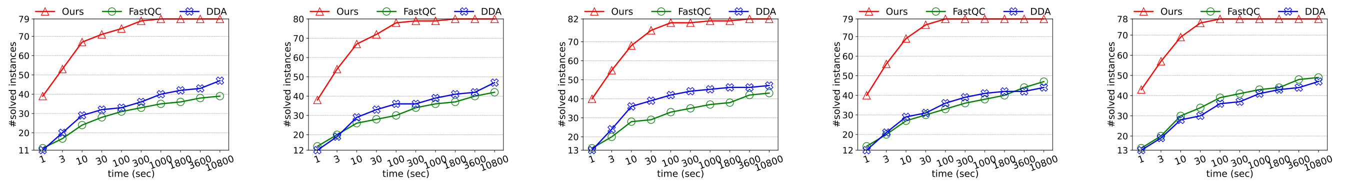
ID	`IterQC`	`-PP`	`-FKLB`	ID	`IterQC`	`-PP`	`-FKLB`
G1	**5.01**	9.74	5.87	G16	**0.02**	0.23	0.03
G2	**5.87**	102.00	6.52	G17	**4.08**	29.05	4.33
G3	0.49	1.74	**0.46**	G18	19.65	144.75	**19.20**
G4	**26.13**	159.50	29.94	G19	0.014	0.07	**0.007**
G5	**9.70**	116.42	11.26	G20	6.19	7.53	**6.47**
G6	**4.82**	77.83	6.55	G21	OOT	OOT	OOT
G7	**6.47**	79.45	7.78	G22	**3.06**	4.07	12.26
G8	**0.04**	0.18	0.05	G23	0.547	2.11	**0.553**
G9	**35.66**	36.31	180.13	G24	0.33	1.57	**0.23**
G10	**2.42**	73.74	2.95	G25	**6.07**	10.14	14.32
G11	0.30	4.95	**0.28**	G26	**23.28**	460.12	27.91
G12	**13.80**	368.10	13.91	G27	**3.38**	3.64	7.35
G13	**1.80**	32.42	2.71	G28	**135.13**	501.39	OOT
G14	**5.33**	90.88	5.93	G29	OOT	OOT	OOT
G15	**6.49**	134.90	6.57	G30	**0.08**	OOT	3.43

TABLE VIII: Runtime performance (in seconds) of `IterQC`, `IterQC-PP`, and `IterQC-FKLB` on 30 instances with $\gamma = 0.85$.

ID	`IterQC`	`-PP`	`-FKLB`	ID	`IterQC`	`-PP`	`-FKLB`
G1	**3.48**	7.30	4.13	G16	0.021	0.09	**0.020**
G2	**5.90**	17.33	6.34	G17	**4.14**	12.31	5.96
G3	**0.38**	0.94	0.44	G18	**17.61**	59.02	18.97
G4	**29.11**	82.92	31.36	G19	0.132	0.46	**0.128**
G5	**18.28**	69.84	19.63	G20	**5.02**	7.39	2.83
G6	**8.84**	33.29	12.81	G21	**57.88**	107.74	OOT
G7	**13.03**	45.33	13.29	G22	0.19	0.81	**0.17**
G8	**0.01**	0.08	0.02	G23	0.30	0.86	**0.28**
G9	**0.99**	1.64	3.51	G24	0.21	0.85	**0.14**
G10	**2.65**	25.22	3.31	G25	0.50	3.15	**0.38**
G11	**0.24**	1.67	0.26	G26	**21.31**	163.58	25.44
G12	**12.32**	86.57	13.46	G27	**0.45**	1.12	0.40
G13	**1.90**	11.16	2.05	G28	7.95	92.42	**7.42**
G14	**5.13**	35.33	6.90	G29	**91.45**	5334.55	74.72
G15	**6.49**	58.82	6.95	G30	**0.08**	0.14	0.10

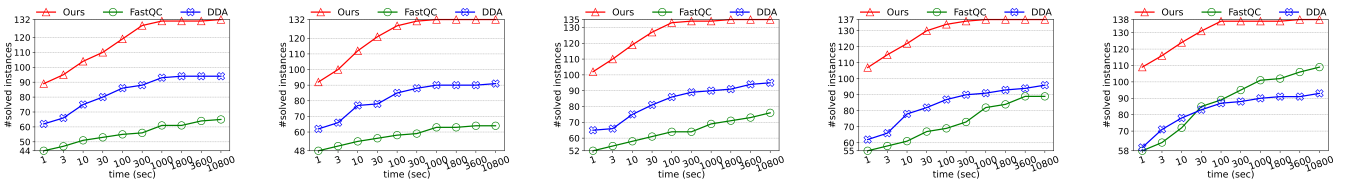
TABLE IX: Runtime performance (in seconds) of `IterQC`, `IterQC-PP`, and `IterQC-FKLB` on 30 instances with $\gamma = 0.95$.

ID	`IterQC`	`-PP`	`-FKLB`	ID	`IterQC`	`-PP`	`-FKLB`
G1	**1.13**	4.25	1.28	G16	**0.019**	0.08	0.023
G2	**5.38**	12.09	5.48	G17	**2.93**	7.81	3.23
G3	**0.41**	0.57	0.42	G18	**17.85**	46.38	20.23
G4	**28.77**	48.56	35.45	G19	0.21	0.33	**0.18**
G5	**18.96**	48.75	27.01	G20	**0.55**	2.97	0.66
G6	**8.70**	24.29	10.02	G21	85.18	127.56	**53.86**
G7	**13.01**	38.80	15.01	G22	0.50	0.83	**0.34**
G8	0.02	0.06	**0.01**	G23	**0.31**	1.36	0.40
G9	1.29	1.69	**1.05**	G24	**0.30**	0.60	0.31
G10	**2.70**	17.39	3.28	G25	**0.41**	2.37	0.50
G11	0.29	1.06	**0.28**	G26	**23.09**	121.08	24.65
G12	**12.30**	77.92	12.95	G27	0.334	0.90	**0.328**
G13	**2.00**	7.67	2.38	G28	8.70	16.19	**7.91**
G14	**5.72**	22.83	6.15	G29	85.68	4646.00	**72.59**
G15	**6.53**	39.07	6.88	G30	0.10	0.16	**0.08**



(a) $\gamma = 0.55$ (b) $\gamma = 0.6$ (c) $\gamma = 0.7$ (d) $\gamma = 0.8$ (e) $\gamma = 0.9$

Fig. 8: Number of solved instances on 10th DIMACS graphs.



(a) $\gamma = 0.55$ (b) $\gamma = 0.6$ (c) $\gamma = 0.7$ (d) $\gamma = 0.8$ (e) $\gamma = 0.9$

Fig. 9: Number of solved instances on real-world graphs.