Maximal Directed Quasi-Clique Mining

Guimu Guo*, Da Yan*, Lyuheng Yuan*, Jalal Khalil*, Cheng Long†, Zhe Jiang‡, Yang Zhou+

*The University of Alabama at Birmingham {guimuguo, yanda, lyuan, jalalk}@uab.edu
†Nanyang Technological University c.long@ntu.edu.sg
‡University of Florida zhe.jiang@ufl.edu +Auburn University yangzhou@auburn.edu

***Abstract*—Quasi-cliques are a type of dense subgraphs that generalize the notion of cliques, important for applications such as community/module detection in various social and biological networks. However, the existing quasi-clique definition and algorithms are only applicable to undirected graphs. In this paper, we generalize the concept of quasi-cliques to directed graphs by proposing (γ_1, γ_2)-quasi-cliques which have density requirements in both inbound and outbound directions of each vertex in a quasi-clique subgraph. An efficient recursive algorithm is proposed to find maximal (γ_1, γ_2)-quasi-cliques which integrates many effective pruning rules that are validated by ablation studies. We also study the finding of top-k large quasi-cliques directly by bootstrapping the search from more compact quasi-cliques, to scale the mining to larger networks. The algorithms are parallelized with effective load balancing, and we demonstrate that they can scale up effectively with the number of CPU cores.**

I. Introduction

Given a degree threshold γ and an undirected graph G, a γ-quasi-clique is a subgraph of G, denoted by $g = (V_g, E_g)$, where each vertex connects to at least $\lceil \gamma \cdot (|V_g| - 1) \rceil$ other vertices in g. Quasi-clique is a natural generalization of clique which is useful in mining various networks, such as biological networks [13], [15], [17], [27], [37], [48], and social and communication networks [26], [32], [41], [47], [49], [50] to find significant clusters and communities. However, γ-quasi-cliques are defined for undirected graphs, while many real networks are directed such as gene regulatory networks and Twitter follower networks. It remains an open problem to define meaningful quasi-clique structures in directed graphs.

We identify the importance of having density requirements in both inbound and outbound directions of each vertex in a directed quasi-clique, and define the novel concept of (γ_1, γ_2)-quasi-clique, denoted by g, where each vertex connects to at least γ_1 fraction of other vertices in g, and is meanwhile pointed to by at least γ_2 fraction of other vertices in g. As an illustration, Fig. 1(a) shows a 0.6-quasi-clique (actually tighter, a 0.75-quasi-clique). If we make the edges directed as in Fig. 1(b) and add another vertex f that points to every other vertex, we can see that each vertex therein still points to at least 60% (i.e., 3) of the other 5 vertices. However, we cannot regard f as a member of the dense subgraph. Consider, for example, a group of Twitter users that frequently interact with each other; then the fact that a new user f follows and retweets many users in that group does not make f a member, unless a significant number of users in the group also pay attention to f. In contrast, the graph g in Fig. 1(c) forms a convincing dense group since every vertex points to and is also pointed to by at least 3 of the other 5 vertices, making

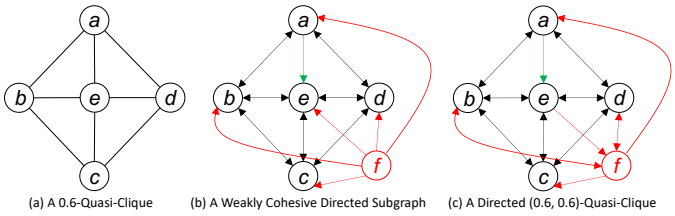

Fig. 1. Quasi-Cliques on Undirected & Directed Graphs

it a $(0.6, 0.6)$-quasi-clique (i.e., $\gamma_1 = \gamma_2 = 0.6$). Besides this example, directed quasi-cliques can also be used in many other applications in bioinformatics (e.g., finding co-regulated genes [55]) and cybersecurity (e.g., botnet detection [47]).

Mining maximal γ-quasi-cliques is notoriously expensive. In fact, [40] shows that even detecting if a given γ-quasi-clique is maximal is NP-hard, and the state-of-the-art algorithms [33], [39], [56] were only tested on very small graphs. Our (γ_1, γ_2)-quasi-clique definition generalizes γ-quasi-clique, and in fact, if we treat an undirected graph G as bidirected and mine (γ, γ)-quasi-cliques on it, this special case is equivalent to mining γ-quasi-cliques on G. Therefore, detecting if a given (γ_1, γ_2)-quasi-clique is maximal is also NP-hard.

In this paper, we design efficient algorithms to mine (γ_1, γ_2)-quasi-cliques, by designing effective pruning rules and search bootstrapping techniques. Despite the NP-hardness of our problem, these techniques allow maximal (γ_1, γ_2)-quasi-clique mining to practically scale to relatively large and dense real networks. Our main contributions are as follows:

- We develop a recursive mining algorithm following the set-enumeration search tree framework [33], [54], [58], by designing effective pruning rules to avoid searching unpromising subgraphs. Many rules require us to consider the intricate interactions between the in-neighbors and out-neighbors of the vertices under exploration.
- For big networks, mining (γ_1, γ_2)-quasi-cliques directly for small values of (γ_1, γ_2) is often expensive with many small subgraphs returned. We resolve this issue by first mining more compact (γ_1', γ_2')-quasi-cliques where $\gamma_1' \geq \gamma_1$ and $\gamma_2' \geq \gamma_2$, and then mining top-k large (γ_1, γ_2)-quasi-cliques from large (γ_1', γ_2')-quasi-cliques.
- The above algorithms are parallelized using a task-based framework, and timeout mechanism [25] is used to decompose straggler tasks to allow effective load balancing.
- Extensive experiments on real directed networks verify the scalability of our mining programs, and we provide a case study to visualize the mined (γ_1, γ_2)-quasi-cliques and to explain the necessity to consider edge directions.

The rest of this paper is organized as follows. Section II formally defines our notations, problem, and the set-enumeration search framework adopted by our algorithm. Section III reviews the related work on dense subgraph mining. Section IV then presents our pruning techniques, and Section V describes our mining algorithm and its parallelization, and Section VI describes our bootstrapping approach to directly mine top-k large quasi-cliques. Finally, Section VII reports our experiments and Section VIII concludes this paper.

II. Preliminaries

Graph Notations. We consider a directed graph $G = (V, E)$ where V (resp. E) is the set of vertices (resp. edges). The vertex set of a graph G can also be explicitly denoted as $V(G)$. We use $G(S)$ to denote the subgraph of G induced by a vertex set $S \subseteq V$, and use $|S|$ to denote the number of vertices in S. We also abuse the notation and use v to mean the singleton set $\{v\}$. We denote the set of outgoing (resp. incoming) neighbors of a vertex v in G by $N^+(v)$ (resp. $N^-(v)$), and denote the outdegree (resp. indegree) of v in G by $d^+(v) = |N^+(v)|$ (resp. $d^-(v) = |N^-(v)|$). We denote the bidirectional neighbors of v by $N^{\pm}(v) = N^+(v) \cap N^-(v)$.

Given a vertex subset $V' \subseteq V$, we define $N^+_{V'}(v) = N^+(v) \cap V'$, which is the set of v's out-neighbors in V'. Similarly, we define notations $N^-_{V'}(v) = N^-(v) \cap V'$, $N^{\pm}_{V'}(v) = N^+_{V'}(v) \cap N^-_{V'}(v)$, $d^+_{V'}(v) = |N^+_{V'}(v)|$ and $d^-_{V'}(v) = |N^-_{V'}(v)|$.

Problem Definition. We next formally define our problem.

Definition 1 (**(γ_1, γ_2)-Quasi-Clique**). A graph $g = (V_g, E_g)$ is a (γ_1, γ_2)-quasi-clique ($0 \leq \gamma_1, \gamma_2 \leq 1$) if g is connected, and for every vertex $v \in V_g$, we have $d^+_g(v) \geq \lceil \gamma_1 \cdot (|V_g| - 1) \rceil$ and $d^-_g(v) \geq \lceil \gamma_2 \cdot (|V_g| - 1) \rceil$.

If a graph is a (γ_1, γ_2)-quasi-clique, its subgraphs usually become uninteresting so we only mine **maximal** quasi-cliques. Here, given a vertex set $S \subseteq V$, $G(S)$ is a maximal (γ_1, γ_2)-quasi-clique if there does not exist a superset $S' \supset S$ such that $G(S')$ is also a (γ_1, γ_2)-quasi-clique.

For dense subgraph mining, researchers usually only strive to find big dense subgraphs, such as the largest one [21], [30], [34], [35], the top-k largest ones [40], and those larger than a predefined size threshold [21], [22], [33], since small dense subgraphs are common and thus statistically insignificant, and the number of dense subgraphs grows exponentially with the graph size. It is well recognized that mining clique relaxations (aka. pseudo-clique) such as quasi-clique [33] and k-plexes [14] are much more expensive than mining clique [14], [21], [22], [40] which is already NP-hard per se. We thus follow the convention and use a minimum size threshold τ_{size} to return only large quasi-cliques to be tractable.

Definition 2 (**Problem Statement**). Given a graph G, minimum degree thresholds $\gamma_1, \gamma_2 \in [0, 1]$ and a minimum size threshold τ_{size}, find all the vertex sets S such that $G(S)$ is a maximal (γ_1, γ_2)-quasi-cliques, and that $|S| \geq \tau_{size}$.

When $G(S)$ is a valid (γ_1, γ_2)-quasi-clique with $\geq \tau_{size}$ vertices, we simply say that S is a valid quasi-clique hereafter.

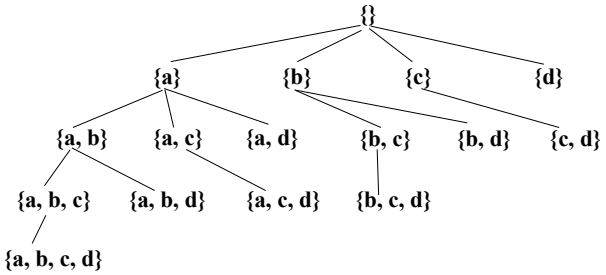

Fig. 2. Set-Enumeration Tree

Set-Enumeration Search Framework. The set-enumeration search tree framework is a popular approach for mining dense subgraphs, including quasi-cliques [33], [40], k-plexes [22], [58] and size-bounded max min-degree subgraph [54].

In the set-enumeration search, the giant search space of a graph $G = (V, E)$, i.e., V's power set, can be organized as a set-enumeration search tree [33]. Fig. 2 shows the set-enumeration tree T for a graph G with four vertices $\{a, b, c, d\}$ where $a < b < c < d$ (ordered by ID). Each tree node represents a vertex set S, and only vertices larger than the largest vertex in S are used to extend S. For example, in Fig. 2, node $\{a, c\}$ can be extended with d but not b as $b < c$; in fact, $\{a, b, c\}$ is obtained by extending $\{a, b\}$ with c.

Let us denote T_S as the subtree of the set-enumeration tree T rooted at a node with set S. Then, T_S represents the search space for all result subgraphs that contain all vertices in S. In other words, $Q \supseteq S$ for any result subgraph Q found by T_S.

We represent the task of mining T_S as a pair $\langle S, ext(S) \rangle$, where S is the set of vertices assumed to be already included, and $ext(S) \subseteq (V - S)$ keeps those vertices that can extend S further into a result subgraph. As we shall see, many vertices cannot form a (γ_1, γ_2)-quasi-clique together with S and can thus be safely pruned from $ext(S)$; therefore, $ext(S)$ is usually much smaller than $(V - S)$.

Mining of T_S can be recursively decomposed into mining of the subtrees rooted at the children of node S in T_S, denoted by $S' \supset S$. Since $ext(S') \subset ext(S)$, the subgraph induced by nodes of a child task $\langle S', ext(S') \rangle$ can be much smaller.

This set-enumeration approach requires postprocessing to remove non-maximal quasi-cliques from the set of valid quasi-cliques found [33]. For example, when processing the task that mines $T_{\{b\}}$, vertex a is not considered and thus the task has no way to determine that $\{b, c, d\}$ is not maximal, even if $\{b, c, d\}$ is invalidated by $\{a, b, c, d\}$ which happens to be a valid pseudo-clique, since $\{a, b, c, d\}$ is processed by the task mining $T_{\{a\}}$. But this postprocessing is efficient [25] especially when the number of valid pseudo-cliques is not big (as we only find large subgraphs).

III. Related Work

This section reviews the related work on set-enumeration based dense subgraph mining, quasi-clique mining on undirected graphs, and some recent efforts in generalizing dense subgraph definitions in undirected graphs to directed graphs.

Set-Enumeration Search. In the set-enumeration search tree framework, designing effective pruning rules is often the

key to the mining efficiency. For different dense subgraph structures, different tailor-made pruning methods need to be developed using the properties of the target structures, which are often the key contributions of such research to make these NP-hard problems tractable in practice, including quasi-cliques [33], [40], k-plexes [22], [58] and size-bounded max min-degree subgraph [54]. As we shall see in Section IV, we have designed a number of pruning rules tailor-made for the properties of (γ_1, γ_2)-quasi-cliques, which is non-trivial since they consider the intricate interactions between the in-neighbors and out-neighbors of the vertices under exploration.

Quasi-Clique Mining on Undirected Graphs. Quasi-cliques were originally defined as a relaxation of cliques on undirected graphs. Regarding γ-quasi-cliques, a few seminal works devised branch-and-bound subgraph searching algorithms such as Crochet [28], [39] and Cocain [56] which finally led to the Quick algorithm [33] that integrated all previous search space pruning techniques and added new effective ones. Our prior work Quick+ [25] further improved Quick's workflow to better utilize these pruning rules, and we parallelized and scaled it in our distributed graph mining framework called G-thinker [51], [52]. Yang et al. [53] adapted Quick to mine a set of diversified temporal γ-quasi-clique patterns from a temporal graph, where each subgraph is associated with the time interval that the pattern spans. Sanei-Mehri et al. [40] bootstraps the mining of top-k large γ-quasi-cliques by first using Quick to find the more compact γ'-quasi-cliques ($\gamma' > \gamma$) quickly, and then expanding large γ'-quasi-cliques into γ-quasi-cliques.

Some works consider a less costly problem variant which finds all quasi-cliques that contain a particular vertex or a set of query vertices [19], [20], [30] to aggressively narrow down the search space. There is another definition of quasi-clique based on edge density [11], [20], [38] rather than vertex degree, but it is essentially a different kind of dense subgraph definition. As [20] indicates, the edge-density based quasi-cliques are less dense than our degree-based quasi-cliques, so we focus on degree-based quasi-cliques. Brunato et al. [16] further consider both vertex degree and edge density. There are also other dense subgraph definitions such as k-plexes [14], [21], [22], [58] and max min-degree subgraph [54], all defined on undirected graphs. Quasi-cliques were also explored for bipartite graphs, i.e., quasi-bicliques [31], [42]–[44] which relax the concept of bicliques [12], [18], [23], [35].

Dense Subgraphs in Directed Graphs. Although many real networks are directed, research on directed dense subgraphs lags behind. Only recently, the idea of k-core has been generalized to directed graphs, called D-core [24]. However, the work focuses on finding a D-core that is strongly connected and contains a particular query vertex, and the degree thresholds are the actual degree values rather than ratios w.r.t. $|V_g|$ as in our case. We remark that using ratios is more desirable since they directly reflect subgraph density, and we find (γ_1, γ_2)-quasi-cliques g of all kinds of sizes (as long as $|V_g| \geq \tau_{size}$), without the need of providing a query vertex, and we find many dense subgraphs rather than a maximal D-core subgraph.

A more recent work, [36], finds the densest subgraph from a large directed graph, defined as a pair of vertex sets (S, T) that maximizes the density of edges from vertices in S to vertices in T, which is a different problem from ours since it only cares about the outbound (resp. inbound) edge density of vertices in S (resp. T), rather than bidirectional edge density of every vertex in a dense subgraph.

IV. Pruning Rules

Recall the set-enumeration tree in Fig. 2, where each node represents a mining task, denoted by $t_S = \langle S, ext(S)\rangle$. Task t_S mines the set-enumeration subtree T_S: it assumes that vertices in S are already included in a result quasi-clique to find, and continues to expand $G(S)$ with vertices of $ext(S) \subseteq (V - S)$ into a valid quasi-clique. Task t_S can be recursively decomposed into the mining of subtrees $\{T_{S'}\}$ where $S' \supset S$ are child nodes of node S, denoted as tasks $\{t_{S'}\}$. Our recursive serial algorithm basically examines the set-enumeration tree in depth-first order, while our parallel implementation utilizes the concurrency among tasks $\{t_{S'}\}$.

To reduce search space, we consider two categories of pruning rules that prune either candidate vertices in $ext(S)$ from expansion, or simply the entire subtree T_S:

- **Type I: Pruning $\mathbf{ext(S)}$:** if a vertex $u \in ext(S)$ satisfies certain conditions, u is pruned from $ext(S)$ since there must not exist a vertex set S' such that $(S \cup u) \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a (γ_1, γ_2)-quasi-clique.
- **Type II: Pruning S:** if a vertex $v \in S$ satisfies certain conditions, there must not exist a vertex set S' such that $S \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a (γ_1, γ_2)-quasi-clique, and thus there is no need to extend S further.

Type-II pruning invalidates the entire T_S. We also allow a variant of Type-II pruning that invalidates $G(S')$, $S \subset S' \subseteq (S \cup ext(S))$ from being a valid (γ_1, γ_2)-quasi-clique, but node S is not pruned since $G(S)$ may be a valid quasi-clique.

Overview of Pruning Rules and Contribution Highlights. We design tailor-made pruning rules for our (γ_1, γ_2)-quasi-clique definition that can be classified into 7 groups. Each rule either belongs to Type I, or Type II, or sometimes both.

We remark that these pruning rule types are commonly used in various set-enumeration algorithms, but the concrete forms are problem dependent and are the key contributions that make these NP-hard problems tractable on large graphs in practice. For example, our cover-vertex pruning to be described in Section has their counterparts in k-plexes (Lemma 2 in [58]) and size-bounded max min-degree subgraph (Lemma 4.7 in [54]), but their concrete forms are entirely different.

Due to page limit, we briefly summarize our pruning rules below. Our online appendix [8] gives details of pruning rule formulations and their complete proofs.

Compared with pruning rules for undirected quasi-cliques (e.g., those of Quick [33]), designing rules for our directed setting is much more challenging since our rules need to additionally consider the intricate interactions between the in-neighbors and out-neighbors of the vertices under exploration. Pruning rules for undirected quasi-cliques are just special cases

Algorithm 1 *iterative_twohop_pruning*(v)

1: $O \leftarrow N^+(v) - N^{\pm}(v), \quad I \leftarrow N^-(v) - N^{\pm}(v)$
2: **repeat**
3: $\quad S_O \leftarrow O \cup N^{\pm}(v), \quad S_I \leftarrow I \cup N^{\pm}(v)$
4: $\quad O \leftarrow O \cap \bigcup_{w \in S_I} N^-(w), \quad I \leftarrow I \cap \bigcup_{w \in S_O} N^+(w)$
5: **until** neither O and I shrank in Line 4
6: $B \leftarrow \bigcup_{w \in S_O} N^+(w) \bigcap \bigcup_{w \in S_O} N^-(w) \bigcap \bigcup_{w \in S_I} N^+(w) \bigcap \bigcup_{w \in S_I} N^-(w)$
7: **return** $N^{\pm}(v) \cup O \cup I \cup B$

of ours when in-neighbors = out-neighbors for every vertex and $\gamma_1 = \gamma_2$; moreover, as we shall see, considering edge directions may bring more pruning opportunities non-existent in the undirected setting.

A. Graph-Diameter Based Pruning

In dense subgraph mining, it is common to conduct diameter-based pruning. For example, when mining k-plexes, it is common to require the subgraph diameter to be ≤ 2 [22], [58] which naturally holds for reasonably large subgraphs. For undirected γ-quasi-cliques, it is common to assume $\gamma \geq 0.5$ in which case it naturally holds that the subgraph diameter ≤ 2 [28], [33], [39], [56]. This assumption is reasonable since we prefer denser subgraphs, and the mining cost would be tractable since candidate vertices in $ext(S)$ are aggressively pruned. We remark that there exists a diameter upper bound for an arbitrary γ, denoted by $f(\gamma)$, as proved in [28], but this general bound is often much looser so the mining becomes very costly. The restriction of $\gamma \geq 0.5$ is a sweet spot in terms of both theoretical soundness and computational tractability.

In our new (γ_1, γ_2)-quasi-clique definition, we can also derive a subgraph diameter upper bound for arbitrary values of (γ_1, γ_2) in general. Let G^u be the undirected version of a directed graph G that ignores edge directions, then we have:

Theorem 1 (Diameter Upper Bound). *Let Q be a (γ_1, γ_2)-quasi-clique of graph G, and $\gamma_{max} = \max\{\gamma_1, \gamma_2\}$, then*

$$diam(Q^u) < \frac{3|Q|}{\gamma_{max}(|Q|-1)+1} < \frac{3}{\gamma_{max}}.$$

Proof. See Appendix A. □

For example, when $\gamma_1 = \gamma_2 = 0.5$, we have $diam(Q^u) < \frac{3}{0.5} = 6$, so we only consider vertices within 5 hops from every vertex in S on G^u. We can actually derive a tighter bound if we further divide γ into different cases, similar to Theorem 1 of [28]. Note that since the upper bound is $\frac{3}{\gamma_{max}}$, the smaller γ_1 and γ_2 are, the looser the upper bound is.

However, for the special case where $\gamma_1, \gamma_2 \geq 0.5$, we can derive a much tighter vertex candidate set that can form quasi-cliques with a vertex v, denoted by $\mathbb{B}(v)$. As we shall see, $\mathbb{B}(v)$ is a subset of all vertices within 2 hops from v on G^u, rather than 5 hops as computed by Theorem 1, so it is much tighter. Moreover, unlike the undirected setting where Quick [33] only prunes vertices beyond 2 hops from v when $\gamma \geq 0.5$, here we compute $\mathbb{B}(v)$ as follows to locate and remove unpromising vertices iteratively even when they are within 2 hops.

Algorithm 1 computes $\mathbb{B}(v)$. Specifically, when we consider (γ_1, γ_2)-quasi-clique where $\gamma_1, \gamma_2 > 0.5$, computation of $\mathbb{B}(v)$ is discussed using 4 cases (arrows below represent directed edges): (1) if $u \leftrightarrow v$, we always have $u \in \mathbb{B}(v)$; (2) if $u \rightarrow v$ only, then $u \in \mathbb{B}(v)$ only when $\exists w \in V - \{u, v\}, u \leftarrow w \leftarrow v$; (3) if $u \leftarrow v$ only, then $u \in \mathbb{B}(v)$ only when $\exists w \in V - \{u, v\}, u \rightarrow w \rightarrow v$; (4) if u and v have no neighboring relationship, then $u \in \mathbb{B}(v)$ only when $\exists w_1, w_2, w_3, w_4 \in V - \{u, v\}$, such that $u \leftarrow w_1 \leftarrow v$, $u \leftarrow w_2 \rightarrow v$, $u \rightarrow w_3 \leftarrow v$ and $u \rightarrow w_4 \rightarrow v$. The algorithm to compute $\mathbb{B}(v)$ can thus be formulated as in Algorithm 1, where $N^{\pm}(v) = N^+(v) \cap N^-(v)$, and O (resp. I, B) denotes vertices of Case 2 (resp. Case 3, Case 4) above. Iterative pruning is conducted in Lines 2–5 since $w \in ext(S)$ in Case 2 (resp. Case 3) could be pruned by Case 3 (resp. Case 2), making u invalid in Case 2 (resp. Case 3) anymore. See our appendix [8] for a complete proof of the 4 cases and a more detailed description of Algorithm 1. In summary, we have:

Theorem 2. *Let Q be a (γ_1, γ_2)-quasi-clique where $\gamma_1, \gamma_2 \geq 0.5$, and let v be a vertex in Q, then for any other vertex $u \in Q$, we have $u \in \mathbb{B}(v)$ where $\mathbb{B}(v)$ is computed by Algorithm 1.*

Proof. See Appendix B [8]. □

Note that $\mathbb{B}(v) \subseteq ext(S)$ is just the pruned candidate set by one vertex $v \in S$. A valid candidate $u \in ext(S)$ should survive the pruning of every vertex $v \in S$. Formally, we have

Theorem 3 (Diameter Pruning). *Given a task $\langle S, ext(S)\rangle$ where $\gamma_1, \gamma_2 \geq 0.5$, we have $ext(S) \subseteq \bigcap_{v \in S} \mathbb{B}(v)$.*

This is a Type-I pruning since if $u \notin \bigcap_{v \in S} \mathbb{B}(v)$, u can be immediately pruned from $ext(S)$. To efficiently handle the common setting $\gamma_1, \gamma_2 \geq 0.5$, we pre-compute $\mathbb{B}(v)$ for every vertex v so that they can be readily used for pruning as needed during set-enumeration search. While for the general (γ_1, γ_2) setting, we override $\mathbb{B}(v)$ with v's $\lceil 3/\gamma_{max} - 1 \rceil$-hop neighbor set in G^u according to Theorem 1. The neighbor sets can be computed on demand when it is needed and then cached in memory for reuse, so such computation can be skipped for pruned vertices (e.g., by (k_1, k_2)-core pruning below).

B. Size-Threshold Based Pruning

Theorem 4 (Size Threshold Pruning). *If a vertex u has $d^+(u) < \lceil \gamma_1 \cdot (\tau_{size} - 1) \rceil$ or $d^-(u) < \lceil \gamma_2 \cdot (\tau_{size} - 1) \rceil$, then u cannot appear in any (γ_1, γ_2)-quasi-clique Q with $|Q| \geq \tau_{size}$.*

Proof. See Appendix C [8]. □

For any vertex u in G, let us define $k_1 = \lceil \gamma_1 \cdot (\tau_{size} - 1) \rceil$ and $k_2 = \lceil \gamma_2 \cdot (\tau_{size} - 1) \rceil$, this rule shrinks G into its (k_1, k_2)-core, which is defined as the maximal subgraph of G such that every vertex has its outdegree $\geq k_1$ and indegree $\geq k_2$. The (k_1, k_2)-core of a graph $G = (V, E)$ can be computed in $O(|E|)$ time using a peeling algorithm, which repeatedly removes vertices with outdegree $< k_1$ and indegree $< k_2$ until there is no such vertex. We always shrink a graph G into its (k_1, k_2)-core before running our mining algorithm.

C. Degree-Based and Bound-Based Pruning

Four kinds of degrees are frequently used by our pruning rules. Taking outdegrees for example: (1) SS-degrees: $d_S^+(v)$ for all $v \in S$; (2) SE-degrees: $d_S^+(u)$ for all $u \in ext(S)$;

(3) ES-degrees: $d^+_{ext(S)}(v)$ for all $v \in S$; and (4) EE-degrees: $d^+_{ext(S)}(u)$ for all $u \in ext(S)$.

Three groups of pruning rules utilize these degrees: (i) degree-based pruning that solely uses the degrees of a vertex itself, (ii) upper-bound based pruning and (iii) lower-bound based pruning that look at the degrees of multiple (or even all) vertices in S and $ext(S)$. Each of the three groups contains one Type-I rule and one Type-II rule.

Here, the upper bound U_S (resp. lower bound L_S) is defined on the number of vertices in $ext(S)$ that can be added to S in order to form a valid (γ_1, γ_2)-quasi-clique. These bounds are defined based on the above-mentioned degrees of vertices in task $t_S = \langle S, ext(S)\rangle$ under exploration, the concrete forms can be found in Appendiices D–F [8] due to page limit.

D. Critical-Vertex Based Pruning

We next define the concepts of *outdegree critical vertex* and *indegree critical vertex* using lower bound L_S defined above.

Definition 3 (Outdegree Critical Vertex). *If there exists a vertex $v \in S$ such that $d^+_S(v) + d^+_{ext(S)}(v) = \lceil \gamma_1 \cdot (|S| + L_S - 1)\rceil$, then v is called an outdegree critical vertex of S.*

Definition 4 (Indegree Critical Vertex). *If there exists a vertex $v \in S$ such that $d^-_S(v) + d^-_{ext(S)}(v) = \lceil \gamma_2 \cdot (|S| + L_S - 1)\rceil$, then v is called an indegree critical vertex of S.*

Intuitively, v is outdegree (resp. indegree) critical if adding **all** its out-neighbors (resp. in-neighbors) inside $ext(S)$ into S merely allows the outdegree (resp. indegree) of v to meet the out-bound γ_1 (resp. in-bound γ_2) degree requirement to generate a quasi-clique. Then, we have the following theorems:

Theorem 5 (Outdegree Critical Vertex Pruning). *If $v \in S$ is an outdegree critical vertex, then for any vertex set S' such that $S \subset S' \subseteq (S \cup ext(S))$, if $G(S')$ is a (γ_1, γ_2)-quasi-clique, then S' must contain every out-neighbor of v in $ext(S)$, i.e., $N^+_{ext(S)}(v) \subseteq S'$.*

Theorem 6 (Indegree Critical Vertex Pruning). *If $v \in S$ is an indegree critical vertex, then for any vertex set S' such that $S \subset S' \subseteq (S \cup ext(S))$, if $G(S')$ is a (γ_1, γ_2)-quasi-clique, then S' must contain every in-neighbor of v in $ext(S)$, i.e., $N^-_{ext(S)}(v) \subseteq S'$.*

Proof. See Appendix G [8]. □

Therefore, when extending S, if we find that $v \in S$ is an outdegree (resp. indegree) critical vertex, we can directly add all vertices in $N^+_{ext(S)}(v)$ (resp. $N^-_{ext(S)}(v)$) to S for mining.

E. Cover-Vertex Based Pruning

Given a vertex $u \in ext(S)$, we next define a vertex set $C_S(u) \subseteq ext(S)$ such that for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C_S(u)$, $Q \cup u$ is also a (γ_1, γ_2)-quasi-clique. In other words, Q is not maximal and can thus be pruned. We say that $C_S(u)$ is the set of vertices in $ext(S)$ that are covered by u, and that u is the cover vertex.

To utilize $C_S(u)$ for pruning, we put vertices of $C_S(u)$ after all the other vertices in $ext(S)$ when checking the next level in the set-enumeration tree (see Fig. 2), and only check until vertices of $ext(S) - C_S(u)$ are examined (i.e., the extension of S using $V' \subseteq C_S(u)$ is pruned). To maximize the pruning effectiveness, we can find $u \in ext(S)$ with a large $|C_S(u)|$.

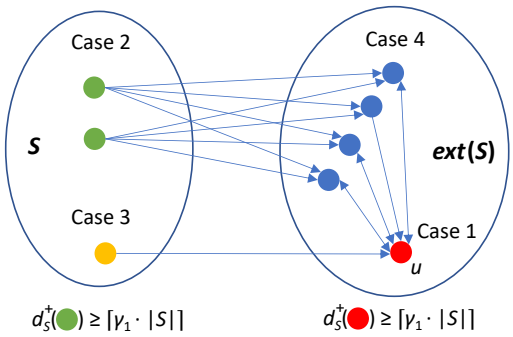

Fig. 3. Outbound Cover Set

We compute $C_S(u)$ as the intersection of an outbound cover set $C^+_S(u)$ and an inbound cover set $C^-_S(u)$. The outbound cover set $C^+_S(u)$ is computed as follows:

$$C^+_S(u) = N^+_{ext(S)}(u) \cap N^-_{ext(S)}(u) \cap \bigcap_{v \in S \wedge v \notin N^-(u)} N^+(v). \tag{1}$$

We call $C^+_S(u)$ the outbound cover set of u because, for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C^+_S(u)$, we have the guarantee that for any vertex $w \in Q \cup u$, $d^+_{Q \cup u}(w) \geq \lceil \gamma_1 \cdot (|Q \cup u| - 1)\rceil = \lceil \gamma_1 \cdot |Q| \rceil$.

We illustrate the computation in Eq (1) using Fig. 3, where the blue vertices are those in $C^+_S(u)$. Note that any blue vertex is bidirectionally connected with u (i.e., the red vertex in Fig. 3) due to the term $N^+_{ext(S)}(u) \cap N^-_{ext(S)}(u)$ in Eq (1). Additionally, any blue vertex is pointed to from all green vertices, where we highlight a vertex $v \in S$ in green if it is not an in-neighbor of u (other vertices in S are in yellow). This is because of the term $\bigcap_{v \in S \wedge v \notin N^-(u)} N^+(v)$ in Eq (1).

We can similarly define the inbound cover set of u:

$$C^-_S(u) = N^-_{ext(S)}(u) \cap N^+_{ext(S)}(u) \cap \bigcap_{v \in S \wedge v \notin N^+(u)} N^-(v). \tag{2}$$

Here, $C^-_S(u)$ is called the inbound cover set of u because, for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C^-_S(u)$, we have the guarantee that for any vertex $w \in Q \cup u$, $d^-_{Q \cup u}(w) \geq \lceil \gamma_2 \cdot |Q| \rceil$.

Finally, for u to cover a vertex $w \in ext(S)$, w should satisfy both $d^+_{Q \cup u}(w) \geq \lceil \gamma_1 \cdot |Q| \rceil$ and $d^-_{Q \cup u}(w) \geq \lceil \gamma_2 \cdot |Q| \rceil$. So,

$$C_S(u) = C^+_S(u) \cap C^-_S(u) \tag{3}$$

We compute $C_S(u)$ for pruning only if $d^+_S(u) \geq \lceil \gamma_1 \cdot |S| \rceil$ and $d^-_S(u) \geq \lceil \gamma_2 \cdot |S| \rceil$, and for any $v \in S$ that are not adjacent to u, it holds that $d^+_S(v) \geq \lceil \gamma_1 \cdot |S| \rceil$ and $d^-_S(v) \geq \lceil \gamma_2 \cdot |S| \rceil$. Otherwise, we deem this pruning inapplicable. The outbound degree requirements have been highlighted at the bottom of Fig. 3. Note that the constraint on v here is not too demanding. For example, consider the constraint $d^+_S(v) \geq \lceil \gamma_1 \cdot |S| \rceil$: note that since $v \in S$, we already have $d^+_S(v) \geq \lceil \gamma_1 \cdot (|S| - 1)\rceil$ which is very close, so the chance that the constraints are satisfied and so the pruning is applicable is not low.

Formally, we have the following theorem:

Theorem 7 (Cover-Vertex Pruning). *Given a vertex $u \in ext(S)$, assume that $d_S^+(u) \geq \lceil \gamma_1 \cdot |S| \rceil$ and $d_S^-(u) \geq \lceil \gamma_2 \cdot |S| \rceil$, and for any $v \in S$ that are not adjacent to u, it holds that $d_S^+(v) \geq \lceil \gamma_1 \cdot |S| \rceil$ and $d_S^-(v) \geq \lceil \gamma_2 \cdot |S| \rceil$. Then, for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C_S(u)$, $Q \cup u$ is also a (γ_1, γ_2)-quasi-clique.*

Proof. See Appendix H [8]. □

To maximize the pruning power, we can find our cover vertex $u \in ext(S)$ to maximize $|C_S(u)|$. However, this will require us to compute $C_S(u)$ for every $u \in ext(S)$, which is too expensive when $ext(S)$ is large. We, therefore, adopt a simple heuristic that chooses the cover vertex u as the vertex in $ext(S)$ with the largest value of $\min\{d_S^+(u), d_S^-(u)\}$, with tie broken by $\min\{d_{ext(S)}^+(u), d_{ext(S)}^-(u)\}$.

As we shall see in Section VII, even with this less expensive heuristic, enabling cover-vertex pruning can slow down the mining in many datasets since the pruning effectiveness cannot offset the pruning cost. This is a surprising finding since we can show that our cover-vertex pruning rule when applied on undirected graphs (i.e., every edge is treated as bidirected) is equivalent to the cover-vertex pruning of Quick [33] and Quick+ [25], but both Quick and Quick+ use cover-vertex pruning. For this reason, we also studied Quick+ by disabling its cover-vertex pruning, and also observed a couple of times of performance improvement on most datasets, and only observed a couple of times of performance degradation on some datasets. Still, cover-vertex pruning could be useful to protect against a long-running worst-case scenario which occurs on some datasets. We will elaborate on our findings and recommended pruning rule configurations in Section VII.

As a degenerate special case, initially when $S = \emptyset$, Eq (1) (resp. Eq (2)) becomes $C_S^+(u) = C_S^-(u) = N_{ext(S)}^+(u) \cap N_{ext(S)}^-(u)$ and all neighbors of u belong to $ext(S)$, so $C_S(u) = C_S^+(u) \cap C_S^-(u) = N_{ext(S)}^+(u) \cap N_{ext(S)}^-(u) = N^+(u) \cap N^-(u)$, i.e., we only need to find u as the vertex adjacent to the most number of bidirectional edges in G to maximize $|C_S(u)|$ for cover-vertex pruning.

We find u as the vertex adjacent to the most number of bidirectional edges after the (k_1, k_2)-core pruning (c.f. Section IV-B) since otherwise, we may find a high-degree vertex with limited pruning power, e.g., adjacent to many low-degree neighbors that will be pruned by (k_1, k_2)-core pruning.

V. The Recursive Mining Algorithm

We now describe our recursive mining algorithm that traverses the set-enumeration tree in depth-first order, and applies the pruning rules presented in Section IV wherever applicable.

Data Structures and Preprocessing. Before running our recursive main algorithm, we first pre-compute $\mathbb{B}(v)$ for each vertex $v \in V$ (c.f. end of Section IV-A). In this way, each vertex v has direct access to $\mathbb{B}(v)$, $N^+(v)$ and $N^-(v)$.

Recall that each task $t = \langle S, ext(S) \rangle$ corresponds to a node S in the set-enumeration tree. Our main algorithm uses an array A to organize vertices where vertices in S are positioned before those in $ext(S)$. In other words elements in A are always maintained to have 2 segments: (1) vertex objects in S, (2) vertex objects in $ext(S)$. For each vertex object v in A, we also maintain five degrees (1) $d_S^+(v)$, (2) $d_S^-(v)$, (3) $d_{ext(S)}^+(v)$, (4) $d_{ext(S)}^-(v)$ and (5) $d_{ext(S)}^{\mathbb{B}}(v) \triangleq |\mathbb{B}(v) \cap ext(S)|$, which are incrementally updated so that they can be accessed in $O(1)$ time for checking the conditions of our various pruning rules. More details can be found in Appendix I [8].

Look-Ahead Pruning. This technique examines if $S \cup ext(S)$ gives a valid quasi-clique, and if so, we output it and avoid the unnecessary depth-first traversal of the subtree T_S. Notably, since we maintain $d_{ext(S)}^{\mathbb{B}}(v)$ for each vertex $v \in ext(S) \subseteq A$, it allows an additional pruning opportunity that combines diameter-based pruning with look-ahead pruning. Please refer to Appendix J [8] for the algorithm of look-ahead pruning.

Look-ahead pruning comes with a cost since it needs to be checked at each recursive step. In fact, we find in our experiments in Section VII that look-ahead pruning does not improve the mining performance much on most datasets, but it serves as a faster pruning alternative to cover-vertex pruning to protect against a potential long-running worst-case scenario which occurs on some datasets.

Iterative Bound-Based Pruning. Whenever we remove a candidate vertex from $ext(S)$ and/or add a candidate vertex to S, the degrees of the vertices in array A w.r.t. S and $ext(S)$ would be incrementally updated, creating new opportunities for degree-based pruning (c.f. Appendix D [8]). Moreover, the degree updates would also cause the bounds L_S and U_S to be updated (c.f. Fig. 10 and 11 in our appendix [8]), creating new opportunities for upper bound pruning (c.f. Appendix E [8]) and lower bound based pruning (c.f. Appendix F [8]).

Note that some of the above pruning rules could be Type I rules, causing $ext(S)$ to shrink, which in turn reduces $d_{ext(S)}^+(.)$ and $d_{ext(S)}^-(.)$ and thus triggers another round of bound-based pruning.

Algorithm 5 in Appendix K [8] details the pseudocode of *iterative_bound_pruning*(S, $ext(S)$) for iterative bound-based pruning, which returns if the entire set-enumeration subtree T_S is pruned or not (i.e., *PRUNED* or *NOT_PRUNED*), and shrinks $ext(S)$ by Type-I pruning. Note that if *iterative_bound_pruning*(S, $ext(S)$) returns *NOT_PRUNED* but $ext(S)$ is shrunk into $\emptyset$, we still need to examine if $G(S)$ is a valid quasi-clique but not any other descendant in T_S.

Advanced Pruning by Critical Vertices. Recall critical-vertex based pruning from Section IV-D. These pruning rules are the most effective when they are used together with diameter-based pruning. Algorithm 2 gives the details.

Specifically, recall from Theorem 5 (resp. Theorem 6) that if v is an outdegree (resp. indegree) critical vertex, then its neighbors $N_{ext(S)}^+(v)$ (resp. $N_{ext(S)}^-(v)$) should be moved to S in order to form a valid quasi-clique. Therefore, in Algorithm 2, we collect those vertices into V_{nb} in Lines 1–3, which will be added to S in Line 9.

Before this vertex batch movement, we first check if the new S can create a valid quasi-clique. Two Type-II pruning rules apply here. **(R1)** if there exists $v \in S$ such that a newly added

Algorithm 2 *critical_vertex_pruning*(S, $ext(S)$)

1: gather $N^+_{ext(S)}(u)$ of all outdegree critical vertices u into V_O
2: gather $N^-_{ext(S)}(u)$ of all indegree critical vertices u into V_I
3: $V_{nb} = V_O \cup V_I$
4: **for each** $w \in S \cup ext(S)$ **do** $counter[w] \leftarrow |\mathbb{B}(w) \cap V_{nb}|$
5: **for each** $v \in S$ **do**
6: **if** $counter[v] \neq |V_{nb}|$ **do** **return** *PRUNED*
7: **for each** $w \in V_{nb}$ **do**
8: **if** $counter[w] \neq |V_{nb}| - 1$ **do** **return** *PRUNED*
9: add all vertices of V_{nb} into S
10: prune all those $u \in ext(S)$ for which $counter[u] \neq |V_{nb}|$
11: **return** *iterative_bound_pruning*(S, $ext(S)$)

vertex $u \notin \mathbb{B}(v)$, then S is Type-II pruned. This is because $u \in V_{nb}$ has to be added to S to form a valid quasi-clique, but then u will occur together with v in such a quasi-clique, which is impossible given $u \notin \mathbb{B}(v)$. **(R2)** if there exists $u, u' \in V_{nb}$ such that $u \notin \mathbb{B}(u')$, then S is Type-II pruned. This is because both u and u' have to be added to S to form a valid quasi-clique, which is impossible given $u \notin \mathbb{B}(u')$.

To facilitate the condition checking for (R1) and (R2), Line 4 computes for each vertex $w \in S \cup ext(S)$ a counter $counter[w]$ to keep the number of vertices in V_{nb} that are within set $\mathbb{B}(w)$. Note that we want $V_{nb} \subseteq \mathbb{B}(w)$, or S is Type-II pruned. Lines 5–6 (resp. Lines 7–8) implements the Type-II pruning check for (R1) (resp. (R2)). Note that in Line 8 "-1" is because we want every other vertex of V_{nb} to be contained by $\mathbb{B}(w)$, but $w \in V_{nb}$ itself is not added to its counter $counter[w]$ (since $w \notin \mathbb{B}(w)$).

If S passes the Type-II pruning, Line 10 then conducts Type-II pruning over remaining vertices in $ext(S)$ to ensure that they are contained by $\mathbb{B}(v)$ of every vertex $v \in V_{nb}$ newly added to S. Since this may trigger degree and bound updates, Line 11 then runs the previously described procedure *iterative_bound_pruning*(S, $ext(S)$) (i.e., Algorithm 5 in Appendix K) [8] for degree- and bound-based pruning.

The Recursive Main Algorithm. We now put all previously discussed techniques together, which gives our main algorithm shown in Algorithm 3. This algorithm is recursive, and we start the entire mining process by calling *recursive_mine*($\emptyset$, V). The function *recursive_mine*(.) returns if the current task $t_S = \langle S, ext(S)\rangle$ finds any valid quasi-clique in the subtree T_S not counting node S itself. This information is maintained by variable *result_found* which is initialized as *false* in Line 1. Then, we iterate through each vertex $v \in ext(S)$ (kept in array A) via the for-loop starting from Line 2, to move v from $ext(S)$ to S for recursive mining later in Line 10.

Before moving v to S, Line 3 first conducts look-ahead pruning by checking if moving all vertices from $ext(S)$ to S gives a valid quasi-clique (i.e., by running Algorithm 4 in Appendix J), and if so, Line 4 outputs $G(S \cup ext(S))$ and then returns *true* indicating that a result is found in T_S, skipping the iterating through vertices in $ext(S)$. If look-ahead pruning fails, we create a new array $A' = [S', ext(S')]$ in Lines 5 and 6 for the child task $t_{S'}$ that mines subtree $T_{S'}$. Note that candidate v has been considered, so Line 5 excludes it from $ext(S)$ for future iterations. Also, vertices in $ext(S')$ should

Algorithm 3 *recursive_mine*(S, $ext(S)$)

1: $results\_found \leftarrow$ *false*
2: **for each** vertex $v \in ext(S)$ not covered by the cover vertex **do**
3: **if** *look_ahead*(S, $ext(S)$) **then**
4: output $Q = S \cup ext(S)$; **return** $true$
5: $S' \leftarrow S \cup v$, $ext(S) \leftarrow ext(S) - v$
6: $ext(S') \leftarrow ext(S) \cap \mathbb{B}(v)$; update degrees for v's neighbors
7: **if** *iterative_bound_pruning*($S', ext(S')$) = *PRUNED* **continue**
8: **if** *critical_vertex_pruning*(S', $ext(S')$) = *PRUNED* **continue**
9: find a cover vertex $u \in ext(S')$; mask its covered vertices
10: $tag \leftarrow$ *recursive_mine*(S', $ext(S')$)
11: **if** $tag = true$ **then** $results\_found \leftarrow$ *true*
12: **else if** $G(S')$ is a valid quasi-clique **then**
13: $results\_found \leftarrow$ *true*; output $G(S')$
14: **return** $results\_found$

now also be contained in $\mathbb{B}(v)$ of the newly added v, so Line 6 excludes unsatisfied vertices in $ext(S)$ to compute $ext(S')$.

Now that $t_{S'} = \langle S', ext(S')\rangle$ is constructed, we need to determine if it can be Type-II pruned before processing $t_{S'}$ recursively in Line 10. Specifically, since v has been moved, Line 6 updates the degrees of v's neighbors w.r.t. S' and $ext(S')$ in the new array A'. Line 7 then uses *iterative_bound_pruning*(S, $ext(S)$) (i.e., Algorithm 5 in Appendix K [8]) to conduct iterative degree- and bound-based pruning, and if S' is Type-II pruned, Line 7 directly moves on to check the next v in $ext(S)$. Otherwise, Line 8 uses Algorithm 2 to conduct the advanced critical-vertex pruning.

If $t_{S'}$ survives both pruning, Line 9 then finds a cover vertex $u \in ext(S')$ using the method described in Section IV-E, and masks out its covered vertex in $ext(S')$ before running $t_{S'}$ over A' recursively in Line 10, so that Line 2 will skip those vertices when running $t_{S'}$.

Line 10 then runs $t_{S'}$ by recursively calling Algorithm 3 over A', and the return value tag indicates if $t_{S'}$ finds any quasi-clique not counting S' itself. If $tag = true$, it implies that t_S also finds a quasi-clique so we update *results_found* as *true* in Line 11; there is no need to check $G(S')$ since it cannot be maximal. In contrast, if $tag = false$, we need to examine if $G(S')$ itself is a valid quasi-clique in Line 12, and if so, output $G(S')$ and set *results_found* as *true* in Line 13. Finally, Line 14 returns *results_found* indicating if any iteration of the for-loop in Lines 2–13 ever finds any valid quasi-clique, which is needed when its parent task runs Line 10 to set tag.

Load-Balanced Parallel Implementation. Algorithm 3 directly traverses the entire set-enumeration tree in depth-first order. One way to parallelize the algorithm is to create the level-1 tasks T_v for all $v \in V$ so that different tasks can run concurrently by different CPU cores. However, a task may have a giant subtree, causing the straggler problem. We use the timeout mechanism recently proposed in [25] to further decompose a task $t_S = \langle S, ext(S)\rangle$: t_S traverses T_S in depth-first order by the serial recursive algorithm till when it runs for a time longer than a timeout threshold, after which the recursion backtracks and wraps every unprocessed task $\langle S'', ext(S'')\rangle$ as an independent one that can be assigned to another CPU core for processing, rather than process $t_{S''}$ recursively by the current task's CPU core. While this approach

can generate small-grained tasks to avoid stragglers, it is also important to schedule expensive tasks early for decomposition to keep the number of tasks abundant, and to keep the task workloads among different CPU cores balanced. Our G-thinker system achieves this goal by maintaining a globally shared queue of expensive tasks for prioritized execution, and please refer to Section 5 of [25] for the details. In this work, we use a shared-memory parallel counterpart called T-thinker as proposed in [29], on top of which we parallelize Algorithm 3 using the timeout mechanism. This single-machine parallel engine facilitates our implementation of the parallel kernel-expansion solution to be presented in the next section, which relies on a shared concurrent trie to avoid redundant search.

Recall that we initially need to pre-compute $\mathbb{B}(v)$ for all vertices $v \in V$, and we implement this operation by a parallel for-loop (in OpenMP) since each call of Algorithm 1 takes a non-negligible amount of time. However, we find in our experiments in Section VII that this parallel for-loop hits a speedup barrier as we increase the number of threads, because each call of Algorithm 1 visits a different portion of the original graph G, causing a lot of CPU cache misses that make the computation memory-bandwidth bound. The problem does not occur in the depth-first set-enumeration search since the cache miss rate is low, and we see an ideal speedup.

VI. Mining Top-k Large Quasi-Cliques

Motivated by the kernel expansion approach proposed by [40] to find top-k large maximal γ-quasi-cliques in an undirected graph, we also develop a similar approach for finding top-k large (γ_1, γ_2)-quasi-cliques. Specifically, we first **(1)** mine (γ_1', γ_2')-quasi-cliques ($\gamma_1' > \gamma_1$ and $\gamma_2' > \gamma_2$) using the previously described algorithm, and then **(2)** select the k' largest (γ_1, γ_2)-quasi-cliques as "kernels". For each kernel S, we then **(3)** expand S into (γ_1, γ_2)-quasi-cliques that are maximal locally in $G(S \cup ext(S))$, i.e., it mines maximal quasi-cliques in subtree T_S. **(4)** For all the quasi-cliques found by expanding the k' kernels, the top-k largest results are then returned. In a nutshell, a kernel-expansion job can be represented by $(\gamma_1', \gamma_2', k', \gamma_1, \gamma_2, k)$ besides the minimum size threshold τ_{size}, and it finds k large γ-quasi-cliques with at least τ_{size} vertices (if available).

Referring to Fig. 2 again, we can first mine $k' = 2$ kernels like $S_1 = \{a, c\}$ and $S_2 = \{b, c\}$, and then continue to mine larger quasi-cliques in T_{S_1} and T_{S_2}, which leads to result subgraphs such as $\{a, c, d\}$ and $\{b, c, d\}$. However, expansions from different kernels may search the same set-enumeration tree nodes (and hence their entire subtrees) repeatedly. To illustrate, let us consider the following two scenarios. **Scenario 1:** there is only one kernel $S_1 = \{a, c\}$ found. In this case, if we use the node ordering $a < b < c < d$ as in Fig. 2, we will only find quasi-clique $\{a, c, d\}$ but will miss $\{a, b, c\}$ even if the latter is a large valid result. To find $\{a, b, c\}$ from S_1, we need to assume that a and c are before all other nodes in ordering, so $c < b$ and b should be included in $ext(S_1)$ unlike in Fig. 2 where we assume $b < c$. While this assumption avoids missing results, it can lead to redundant search as we explain next. **Scenario 2:** there are two kernels $S_1 = \{a, c\}$ and $S_2 = \{b, c\}$.

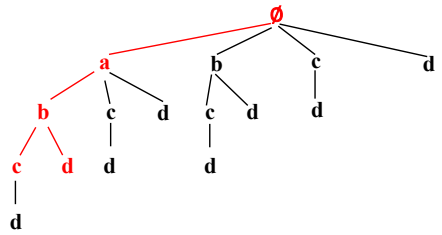

Fig. 4. The Trie that Incorporates All Sets in Fig. 2

According to our discussion above, when expanding S_1 we are assuming $a < b$ (i.e., $a, c \in S_1$ and $b \in ext(S_1)$) so S_1 can be expanded with b, but when expanding S_2 we are assuming $b < a$ (i.e., $b, c \in S_2$ and $a \in ext(S_2)$) so S_2 can be expanded with a. As a result, expansions from both S_1 and S_2 will reach node $\{a, b, c\}$, meaning that both of them search for the entire subtree $T_{\{a,b,c\}}$! Such redundancy is common among other vertex pairs beyond (a, b), and [40] did not address this issue.

Our solution is to maintain a data structure $\mathcal{T}$ to keep all those nodes that have been explored, so that, for example, if the expansion from S_1 has visited node $\{a, b, c\}$, it will be detected by the expansion from S_2 so this expansion will skip node $\{a, b, c\}$ and hence the entire $T_{\{a,b,c\}}$, which is already being explored by the expansion from S_1. Note that since we are only expanding from k' kernels, we expect the number of nodes being searched to be acceptable so that $\mathcal{T}$ can keep all of them in memory. We implement $\mathcal{T}$ as a trie (i.e., prefix tree) to be memory-compact. Fig. 4 shows a trie that encodes all the sets in Fig. 2, and we can see that fewer elements are saved compared with in Fig. 2. In fact, the red parts in Fig. 4 are those resulted after inserting both $\{a, b, c\}$ and $\{a, b, d\}$ into trie $\mathcal{T}$, and we can see that a and b are kept only once for these 2 sets. We remark that a trie basically executes a fixed vertex ordering. For example, in Fig. 4 we assume $a < b < c < d$. While different kernel expansions may use different vertex orders, when they check a node S against $\mathcal{T}$ for redundancy avoidance, S is basically reordered using the vertex ordering of trie $\mathcal{T}$.

In our parallel implementation, each kernel serves as an initial task $t_S = \langle S, ext(S) \rangle$ with all nodes of the kernel being put in S. Since different tasks that are concurrently processed may be expanded from different kernels, we adopt the highly-concurrent trie implementation proposed in [29] where each trie node is protected by a read-write lock.

VII. Experiments

This section reports our experiments. Our code has been released at https://github.com/guimuguo/Tthinker_DQC where a directly executable demo with Google Colab is provided.

Datasets. We use 8 real directed graphs as summarized in Table 1 spanning a wide range of graph sizes and categories: web graphs *PolBlogs* [9], *Google* [6], *Baidu* [2] and *ClueWeb* [4]; social networks *Bitcoin* [3] and *Epinions* [5]; contact network *MathOverflow* [7]; and road network *USA Road* [10].

Experimental Setup. Our experiments were run on a server with IBM POWER8 CPU (32 cores, 3491 MHz) and 1TB RAM. For the T-thinker system, we used the default parameter

TABLE II
MINING TIME ON VARIOUS DATASETS (UNIT: SECOND)

Data	all rules	w/o cover	w/o lookahead	w/o cover&lookahead
PolBlogs	8.68	1.46	7.96	1.67
Bitcoin	36.71	9.88	37.00	8.48
MathOverflow	551.49	17.05	550.49	17.37
Epinions	15.66	6.82	14.74	6.15
Google	0.77	0.79	0.78	216.89
Baidu	9.21	8.82	10.58	20.15
USA Road	9.81	11.06	10.94	10.20
ClueWeb	172.85	174.79	175.42	198.66

setting in [29] where the task timeout threshold is 1 second, and a task $t_S = \langle S, ext(S)\rangle$ is considered a big task for prioritized scheduling if $|ext(S)| \geq 200$. T-thinker keeps memory usage bounded by spilling superfluous tasks from in-memory task queues to the disk for later processing, if too many tasks were generated out of a timeout-based task decomposition. Unless otherwise stated, our parallel programs were run with 32 concurrent mining threads. All reported results were averaged over 10 repeated runs. In our reports, the unit of time (resp. space) is second (resp. MB).

Default Quasi-Clique Parameters. Recall that we have quasi-clique parameters $(\tau_{size}, \gamma_1, \gamma_2)$. We tune their values on each dataset so that some selective results are returned (in contrast to 0) but not overwhelmingly many. In Appendix I, [8], we illustrate how the number of results changes as the values of τ_{size}, γ_1, and γ_2 vary on *Bitcoin* and *Epinions*. Unless otherwise stated, we use the tuned default parameters in the experiments reported hereafter. These default parameters are summarized in Table I along with the number of maximal (γ_1, γ_2)-quasi-cliques of size $\geq \tau_{size}$.

Performance on Various Datasets. Table II reports the performance of our parallel mining algorithms on all the datasets in Table I. As we shall see later in our ablation study, cover-vertex pruning and look-ahead pruning could slow down a program in some cases, but all the other rules always speed up the mining significantly and are thus indispensable. Therefore, in Table II, we consider four variants: (1) all rules are used, (2) all but cover-vertex pruning, (3) all but look-ahead pruning, and (4) all but cover-vertex and look-ahead pruning.

Comparing the first two columns, we can see that enabling cover-vertex pruning slows down the mining by quite a few times on the first four datasets, and only slightly improves the performance on the last two. This is because computing the cover set of a selected cover vertex has a non-negligible cost, and this overhead is often not properly offset by the pruned search space. This is a new finding since prior works all adopt a similar pruning rule [25], [33], [54], [58] (though in different forms). We tested Quick and Quick+ for mining undirected quasi-cliques and observed a similar speedup on many tested datasets when we disabled their cover-vertex pruning.

Comparing "all rules" with "w/o lookahead", we see that their running times are very close, which means that the effectiveness of look-ahead pruning alone is very limited.

Since both cover-vertex and look-ahead pruning rules are not effective, a natural question to ask is: what if we disable both? As can be seen from the last column, the running time increases by over two orders of magnitude on *Google*! doubles on *Baidu*, and increases a lot on *ClueWeb*, while being best or near-best on the other datasets. This shows that we should enable at least one of the two rules, as otherwise, some high-degree candidates that were able to be selected by the rules to aggressively prune the search space would not be selected early, causing a huge slowdown.

TABLE I
DATASETS AND DEFAULT ALGORITHM PARAMETERS

Data	$\|V\|$	$\|E\|$	$\|E\|$ / $\|V\|$	max $d^+(v)$	max $d^-(v)$	τ_{size}	γ_1	γ_2	# Maximal	Category
PolBlogs	1,224	33,430	27.31	351	351	25	0.90	0.90	3,050	web graph
Bitcoin	5,881	35,592	6.05	763	535	10	0.70	0.60	287,139	social network
MathOverflow	24,818	506,550	20.41	5,931	5,378	20	0.87	0.85	5	contact network
Epinions	75,879	508,837	6.71	1,801	3,035	20	0.80	0.90	469	social network
Google	875,713	5,105,039	5.83	456	6,326	15	0.75	0.80	1,827	web graph
Baidu	2,141,300	17,794,839	8.31	2,596	97,950	20	0.70	0.80	802	web graph
USA Road	23,947,347	57,708,624	2.41	9	9	7	0.50	0.50	16	road network
ClueWeb	147,925,593	454,072,685	3.07	1,055	308,477	35	0.90	0.90	2,373	web graph

Since the effectiveness of these rules are very data-dependent, we recommend to disable cover-vertex pruning but enable look-ahead pruning by default, which avoids a non-negligible slowdown.

Speedup by Parallelization. Hereafter, we focus on two algorithm variants, "all rules" and the recommended "w/o cover." Tables III and IV show (1) performance of our serial mining program, (2) that of the parallel program, and (3) speedup achieved for "all rules" and "w/o cover," respectively.

Recall that our program has two steps: (i) a preprocessing stage to compute $\mathbb{B}(v)$ for all vertices, and (ii) the set-enumeration search for quasi-clique mining. In Tables III and IV, we report metrics including: (1) the total mining time, (2) the time for Step 1 (denoted by "2-hop"), and (3) the peak memory and disk space usage. Note that the speedup with 32 threads is good on all datasets except for *Baidu* and *ClueWeb*, for which the speedup ratio merely passes 7. After looking into the reason, we find that for these two datasets, Step 1 consumes the majority of the running time and the set-enumeration search afterwards is actually fast and has a good speedup. Recall that we implement Step 1 by OpenMP parallel for-loop, and since each call of Algorithm 1 visits a different portion of the original graph G, Step 1 causes a lot of CPU cache misses that make the computation memory-bandwidth bound. The problem does not occur in the depth-first set-enumeration search which is more cache-friendly. Since Step 1 hits a speedup ceiling with merely 8 threads due to the limited memory bandwidth, the overall speedup achieved by *Baidu* and *ClueWeb* is limited.

In contrast, when Step 1 only accounts for a tiny portion of the overall time, the speedup ratio can even go beyond 32 as on *Bitcoin* and *MathOverflow* in Table III. This is because, when we spawn a task t_S for each root task $S = \{v\}$, we create a smaller task-subgraph eliminating those vertices not in $\mathbb{B}(v)$, which avoids the scanning of adjacency list items for those vertices in subsequent task computation, making computation faster than our serial program that directly checks the adjacency lists of the original graph.

Also, we can see that the memory and disk space consumption is reasonable, though *ClueWeb* uses a lot of memory due to its sheer graph size. Our parallel program often uses a few times more memory than the serial program, which is within expectation since more threads imply a larger concurrency fanout in the set-enumeration search tree; but the additional space used is much smaller than 32× since our task queues spill tasks to the disk when they become full to keep memory space bounded. Since spilled tasks are refilled into task queues for computation first before considering generating new tasks from vertices for refill, the disk space used by buffered tasks is kept small; only *MathOverflow* (all rules) used quite some disk space due to the frequent task decompositions done by the 32 threads that caused a lot of tasks to be spilled on disk.

Recall from Section II that we need a postprocessing step to remove non-maximal results from the set of valid pseudo-cliques found, but this time is found to be negligible compared with the mining time, and it has been included in the total time.

Ablation Study on Pruning Techniques. To verify the effectiveness of our techniques, we consider those algorithm variants which use all but one technique that gets disabled. Table V shows the results on *Bitcoin* and *MathOverflow*, and those for the other 6 datasets are in Appendix M [8]. We can see that upper- and lower-bound based pruning is the most effective, without which the running time can increase significantly. This is followed by critical-vertex pruning. As we have discussed previously, the impact of cover-vertex pruning and look-ahead pruning are not always positive, but they serve to protect against a long-running worst-case scenario which occurs on *Google*.

TABLE V
ABLATION STUDY: ALL BUT ONE

Algorithm	Bitcoin		MathOverflow	
	Runtime	Memory	Runtime	Memory
full version	36.71	180	551.49	738
w/o lookahead	37.00	256	550.49	742
w/o critical	70.45	283	663.06	760
w/o cover	9.88	89	17.05	209
w/o bound	200.91	358	3,105.62	1,251

We also consider our algorithm variants by starting from a baseline with basic diameter-based, size-threshold, and degree-based pruning, and incrementally adding bound-based, critical-vertex, look-ahead, and cover-vertex pruning, one at a time. This gives algorithm variants denoted by "baseline," "+bound," "+critical," "+lookahead," and "+cover." Note that our baseline still adopts those basic pruning techniques which are quick to check but highly effective in pruning, so that its mining program can finish in reasonable time.

TABLE III
SPEEDUP WITH ALL RULES

Data	Serial (all rules)			32 Threads (all rules)				Speedup
	Runtime	2-hop	Memory	Runtime	2-hop	Memory	Disk	
PolBlogs	231.04	0.093	18	8.68	0.010	79	26	26.63
Bitcoin	1,604.49	0.141	100	36.71	0.010	180	1,352	43.71
MathOverflow	20,548.69	1.269	153	551.49	0.110	738	30,520	37.26
Epinions	434.48	1.523	68	15.66	0.110	479	109	27.75
Google	15.74	6.386	231	0.77	0.420	564	0	20.36
Baidu	70.84	50.920	587	9.21	7.280	1,524	0	7.69
USA Road	209.32	117.405	4,064	9.81	5.790	15,170	0	21.34
ClueWeb	1,219.55	186.348	62,869	172.85	105.950	255,371	0	7.06

TABLE IV
SPEEDUP W/O COVER-VERTEX PRUNING

Data	Serial (w/o cover)			32 Threads (w/o cover)				Speedup
	Runtime	2-hop	Memory	Runtime	2-hop	Memory	Disk	
PolBlogs	26.21	0.092	12	1.46	0.006	31	2	17.97
Bitcoin	294.69	0.143	63	9.88	0.010	89	48	29.81
MathOverflow	276.64	1.273	45	17.05	0.104	209	139	16.23
Epinions	117.12	1.528	54	6.82	0.109	290	9	17.18
Google	15.35	6.323	241	0.79	0.445	565	1	19.48
Baidu	68.92	43.593	536	8.82	6.667	1,576	15	7.82
USA Road	206.30	117.597	3,750	11.06	6.869	14,990	27	18.66
ClueWeb	1,361.91	223.447	62,853	174.79	105.431	255,437	7	7.79

Table VI shows the results on *Bitcoin* and *Google*, and those for the other 6 datasets are in Appendix M. We can see that bound-based pruning significantly speeds up the baseline on *Bitcoin* though the performance remains similar as baseline on *Google*. Critical-vertex pruning further speeds up "+bound." As we have discussed previously, adding lookahead and cover-vertex pruning on *Bitcoin* slows down "+critical" on *Bitcoin*, but significantly speeds up "+critical" on *Google* and avoids the long-running worst-case scenario.

TABLE VI
ABLATION STUDY: INCREMENTAL ADDITION

Algorithm	Bitcoin		Google	
	Runtime	Memory	Runtime	Memory
baseline	123.55	195	207.84	2,086
+bound	17.01	121	222.34	2,445
+critical	9.38	89	214.24	2,475
+lookahead	9.88	89	0.79	565
+cover	36.71	180	0.77	564

Scalability with # of Threads. We illustrate the scalability of our parallel program by running 1, 2, 4, 8, 16 and 32 threads. Fig. 5 shows the results of "w/o cover" on *Bitcoin*, where the red curve shows the program running time, and the blue curve shows the peak memory usage. We can see that the running time almost halves each time the number of threads doubles. Also, the memory usage increases with the number of threads but slowly and gradually flattens at the maximum allowed task queue capacity [25]. Note that here even our parallel program with one thread is faster than the serial one in Table IV, because we spawn tasks with smaller subgraphs and hence avoid scanning the long adjacency lists

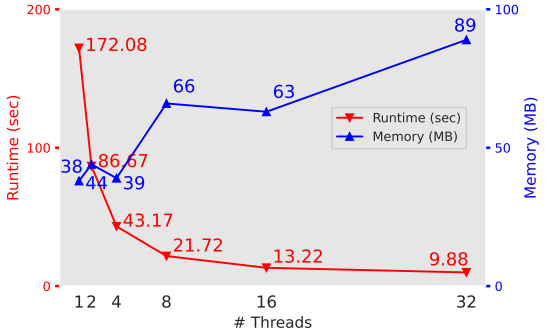

Fig. 5. Scalability of "w/o cover" on *Bitcoin*

of the original graph. In our online appendices [8], we show the complete scalability plots for all our datasets running with "full version" and "w/o cover", where the observation is similar except that on *Baidu* and *ClueWeb*, the time curve hits a floor higher than 0 as the memory-bound computing of $\mathbb{B}(v)$ dominates the runtime.

TABLE VII
EFFECTIVENESS OF KERNEL EXPANSION

Data	γ_1	γ_2	# Maximal	Stage 2 Runtime	Stages 1&2 Runtime	Without Kernel Runtime	Speedup	Top-50 Recall	Top-100 Recall	Top-200 Recall
PolBlogs	0.87	0.87	415	0.201	1.660	19.89	11.98	0.820	0.410	0.650
Bitcoin	0.6	0.5	4,144	0.334	8.917	5,620.88	630.36	0.960	0.890	0.945
Epinions	0.75	0.85	430	1.115	7.934	194.60	24.53	0.960	0.930	0.965

Finding Top-k Large Quasi-Cliques by Kernel Expansion. Recall from Section VI that in order to directly mine top-k large (γ_1, γ_2)-quasi-cliques faster than from scratch, we first find the top-k' largest (γ_1', γ_2')-quasi-cliques as "kernels" (where $\gamma_1 < \gamma_1'$ and $\gamma_2 < \gamma_2'$) as Stage 1, which are then expanded to generate (γ_1, γ_2)-quasi-cliques and return top-k maximal ones from the results as Stage 2. Thus, such a job takes parameters $(\gamma_1', \gamma_2', k', \gamma_1, \gamma_2, k)$. Following the default setting by [40] for mining large undirected γ-quasi-cliques, we set $k' = 3k$ and $k = 100$, and use the same τ_{size} for mining both kernels and result subgraphs. We use the default values of $(\tau_{size}, \gamma_1, \gamma_2)$ in Table I for bootstrapping, i.e., they are used as parameters $(\tau_{size}, \gamma_1', \gamma_2')$ here, to mine quasi-cliques with even smaller values of (γ_1, γ_2).

We illustrate the effectiveness of this kernel bootstrapping approach on three datasets in Table VII, which shows the values of (γ_1, γ_2) that we aim to mine quasi-cliques, the number of maximal quasi-cliques found, the time/speedup metrics, and the result quality metrics. Note that (γ_1, γ_2) here are smaller than the bootstrapping parameters as given by Table I, so directly mining quasi-cliques from scratch is much more time consuming, as shown by the Column "Without Kernel Runtime." When using kernel expansion, Stage 1 runtime is given by Table II where we adopt the recommended configuration "w/o cover," and Stage 2 runtime is shown in Table VII. They sum up to be the total runtime as shown by the Column "Stages 1&2 Runtime," which is then compared with the from-scratch runtime to obtain the speedup achieved by our kernel expansion method. We observe impressive speedup of up to 630× on *Bitcoin*, which shows that kernel expansion can effectively scale mining to large quasi-cliques even when the values of γ_1 and γ_2 are small (hence mining from scratch could be very time consuming).

To evaluate the result quality of the kernel expansion technique, we define the concept of "top-k recall" as the fraction of the exact top-k largest γ-quasi-cliques that are within the top-k largest quasi-cliques found by Stage 2. Intuitively, top-k recall measures how many large quasi-cliques are missed when using kernel expansion rather than exact mining. Note that the result precision is always 100% since Stage 2 always finds valid quasi-cliques that meet the $(\tau_{size}, \gamma_1, \gamma_2)$ requirement. Table VII reports the top-k recall for $k = 50, 100, 150$. We can see that the recall is high in general, meaning that most large cliques are not missed by this bootstrapping technique (note, however, that quasi-cliques are not hereditary so it is inevitable to miss results). Also note that the recall value is not monotonic to k, since as k changes, so is the set of top-k exact largest quasi-cliques that determine the numerator of the recall ratio computation (although the denominator increases with k).

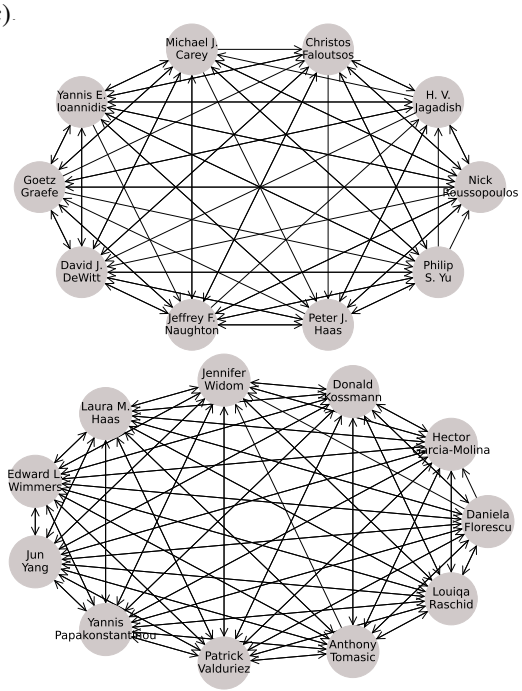

Fig. 6. Two Real (0.75, 0.75)-Quasi-Cliques in G

Case Study. We conducted a case study using the citation network from [1] which consists of papers chosen from ArnetMiner [45], [46] that fall in 10 topics such as Data Mining and Database Systems, as well as their citation relationship. Since the vertices of this network are papers, but the citation relationship is one-way in time, it is improper to directly mine quasi-cliques on this network. We, therefore, preprocess this network to obtain an author citation network G, where an edge (u, v) is added if there exists a paper (co-)authored by u that

cited another paper (co-)authored by v. In total, G contains 2,160 authors and 19,192 citation edges.

We mined (0.75, 0.75)-quasi-cliques on G with $\tau_{size} = 10$, and obtained 22 quasi-cliques in total in 0.7 second. Fig. 6 top (resp. bottom) shows a result quasi-clique with 10 (resp. 11) authors where each author cites at least 75% other members and is cited by at least 75% other members. We can see that these two quasi-cliques correspond to two communities of renowned database researchers that frequently cite each others' works; in fact, Jennifer Widom was Jun Yang's PhD advisor, so they know each other.

An alternative method without using (γ_1, γ_2)-quasi-clique mining is to mine γ-quasi-cliques from the undirected graph G^u. We mined 0.75-quasi-cliques on G^u with $\tau_{size} = 30$, and obtained 37 undirected quasi-cliques in total in 18.1 seconds. Fig. 7 shows a result 0.75-quasi-clique with 30 authors. Note that the 10 authors plotted on the inner circle are exactly those in Fig. 6 top. However, some authors on the outer circle may have a low inbound or outbound degree. For example, 'Theodore Johnson' has 18 outgoing edges (highlighted in red) and merely 7 incoming edges (in blue). This is because γ-quasi-cliques do not care about edge directions. We remark that the community in Fig. 6 top cannot be found here since it is not maximal given the quasi-clique in Fig. 7. Moreover, we also cannot further reduce τ_{size} from 30 to as low as 10, since there are already 961,389 0.75-quasi-cliques when we reduce τ_{size} to 25. This shows the necessity of our (γ_1, γ_2)-quasi-clique formulation for finding highly cohesive directed subgraphs in real directed networks.

Recall that our algorithm can also handle the case when γ_1 and/or $\gamma_2 < 0.5$, by replacing $\mathbb{B}(v)$ with v's $\lceil 3/\gamma_{max} - 1 \rceil$-hop neighbor set in G^u (where $\gamma_{max} = \max\{\gamma_1, \gamma_2\}$) following Theorem 1. We mined (0.6, 0.4)-quasi-cliques on G with $\tau_{size} = 22$ in 1.7 seconds, and obtained only 1 valid quasi-clique. Fig. 8 shows the result quasi-clique, where 'Laura M. Hass' has 13 outgoing edges (in red) and 9 incoming edges (in blue). Laura actually has the lowest indegree in this community, but she is still connected by 42% of the other researchers so the community is a valid (0.6, 0.4)-quasi-clique. Note that Laura was not in all previously visualized communities when our threshold values are 0.75, but she is still in a relatively dense community if we lower our density criteria a bit. Since the community in Fig. 8 is the only (0.6, 0.4)-quasi-clique with ≥ 22 vertices, it is clearly statistically significant and Laura should be considered research active. If we reduce τ_{size} to 20, mining (0.6, 0.4)-quasi-cliques would take 5.0 seconds and return 394 results.

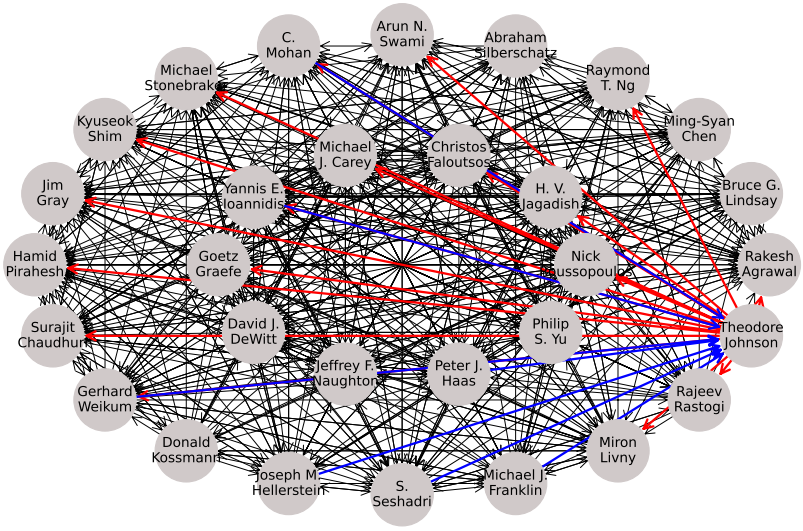

Fig. 7. A Real 0.75-Quasi-Clique in G^u

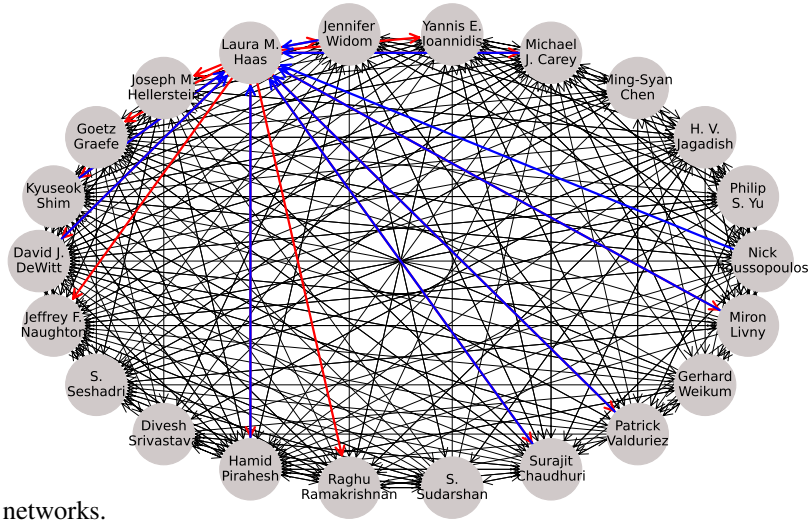

Fig. 8. A Real (0.6, 0.4)-Quasi-Clique in G

VIII. Conclusion

We formulated the concept of (γ_1, γ_2)-quasi-cliques on directed graphs, and proposed an efficient recursive algorithm with many effective pruning rules to find maximal (γ_1, γ_2)-quasi-cliques. We also studied a kernel-expansion approach to find top-k large quasi-cliques directly by bootstrapping. The algorithms are parallelized following a task-based framework with effective load balancing. Extensive experiments demonstrated the scalability of our algorithms, and we analyzed the effectiveness of various pruning rules to obtain a recommended default configuration that is generally efficient on various datasets. A case study was conducted to demonstrate the necessity of our new (γ_1, γ_2)-quasi-clique formulation.

Acknowledgment

This work was supported by NSF OAC-1755464, OAC-2106461, OAC-2152085 and DGE-1723250. Guimu Guo acknowledges financial support from the Alabama Graduate Research Scholars Program (GRSP) funded through the Alabama Commission for Higher Education and administered by the Alabama EPSCoR. The research of Cheng Long is supported by the Nanyang Technological University Start-Up Grant from the College of Engineering under Grant M4082302.

REFERENCES

[1] ArnetMiner Citation Network. https://lfs.aminer.cn/lab-datasets/soinf/.
[2] Baidu. http://konect.cc/networks/zhishi-baidu-internallink/.
[3] Bitcoin. http://konect.cc/networks/soc-sign-bitcoinotc/.
[4] ClueWeb. http://networkrepository.com/web-ClueWeb09-50m.php.
[5] Epinions. http://konect.cc/networks/soc-Epinions1/.
[6] Google Web. https://snap.stanford.edu/data/web-Google.html.
[7] MathOverflow. http://konect.cc/networks/sx-mathoverflow/.
[8] Online Appendices. https://github.com/guimuguo/Tthinker_DQC/blob/main/dqc_appendix.pdf.
[9] Political Blogs. http://konect.cc/networks/dimacs10-polblogs/.
[10] USA Road. http://konect.cc/networks/dimacs9-USA/.
[11] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *LATIN*, volume 2286 of *Lecture Notes in Computer Science*, pages 598–612. Springer, 2002.
[12] A. Abidi, R. Zhou, L. Chen, and C. Liu. Pivot-based maximal biclique enumeration. In *IJCAI*, pages 3558–3564. ijcai.org, 2020.
[13] G. D. Bader and C. W. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics*, 4(1):2, 2003.
[14] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *SIGMOD*, pages 431–444. ACM, 2015.
[15] M. Bhattacharyya and S. Bandyopadhyay. Mining the largest quasi-clique in human protein interactome. In *2009 International Conference on Adaptive and Intelligent Systems*, pages 194–199. IEEE, 2009.
[16] M. Brunato, H. H. Hoos, and R. Battiti. On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*, pages 41–55. Springer, 2007.
[17] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.
[18] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li. Efficient exact algorithms for maximum balanced biclique search in bipartite graphs. In *SIGMOD Conference*, pages 248–260. ACM, 2021.
[19] Y. H. Chou, E. T. Wang, and A. L. P. Chen. Finding maximal quasi-cliques containing a target vertex in a graph. In *DATA*, pages 5–15. SciTePress, 2015.
[20] 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*, 8(1):20, 2018.
[21] A. Conte, D. Firmani, C. Mordente, M. Patrignani, and R. Torlone. Fast enumeration of large k-plexes. In *SIGKDD*, pages 115–124. ACM, 2017.
[22] A. Conte, T. D. Matteis, D. D. Sensi, R. Grossi, A. Marino, and L. Versari. D2K: scalable community detection in massive networks via small-diameter k-plexes. In Y. Guo and F. Farooq, editors, *SIGKDD*, pages 1272–1281. ACM, 2018.
[23] A. Das and S. Tirthapura. Incremental maintenance of maximal bicliques in a dynamic bipartite graph. *IEEE Trans. Multi Scale Comput. Syst.*, 4(3):231–242, 2018.
[24] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *IEEE Trans. Knowl. Data Eng.*, 31(11):2093–2107, 2019.
[25] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.
[26] J. Hopcroft, O. Khan, B. Kulis, and B. Selman. Tracking evolving communities in large linked networks. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5249–5253, 2004.
[27] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl_1):i213–i221, 2005.
[28] D. Jiang and J. Pei. Mining frequent cross-graph quasi-cliques. *ACM Trans. Knowl. Discov. Data*, 2(4):16:1–16:42, 2009.
[29] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *The VLDB Journal (accepted and to appear)*, 2021.
[30] P. Lee and L. V. S. Lakshmanan. Query-driven maximum quasi-clique search. In *SDM*, pages 522–530. SIAM, 2016.
[31] J. Li, K. Sim, G. Liu, and L. Wong. Maximal quasi-bicliques with balanced noise tolerance: Concepts and co-clustering applications. In *SDM*, pages 72–83. SIAM, 2008.
[32] J. Li, X. Wang, and Y. Cui. Uncovering the overlapping community structure of complex networks by maximal cliques. *Physica A: Statistical Mechanics and its Applications*, 415:398–406, 2014.
[33] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In W. Daelemans, B. Goethals, and K. Morik, editors, *ECML/PKDD*, volume 5212 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2008.
[34] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. *Proc. VLDB Endow.*, 10(11):1538–1549, 2017.
[35] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou. Maximum biclique search at billion scale. *Proc. VLDB Endow.*, 13(9):1359–1372, 2020.
[36] C. Ma, Y. Fang, R. Cheng, L. V. S. Lakshmanan, W. Zhang, and X. Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *SIGMOD Conference*, pages 1051–1066. ACM, 2020.
[37] H. Matsuda, T. Ishihara, and A. Hashimoto. Classifying molecular sequences using a linkage graph with their pairwise similarities. *Theor. Comput. Sci.*, 210(2):305–325, 1999.
[38] J. Pattillo, A. Veremyev, S. Butenko, and V. Boginski. On the maximum quasi-clique problem. *Discret. Appl. Math.*, 161(1-2):244–257, 2013.
[39] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *SIGKDD*, pages 228–238. ACM, 2005.
[40] S. Sanei-Mehri, A. Das, and S. Tirthapura. Enumerating top-k quasi-cliques. In *IEEE BigData*, pages 1107–1112. IEEE, 2018.
[41] S. Sheng, B. Wardman, G. Warner, L. Cranor, J. Hong, and C. Zhang. An empirical analysis of phishing blacklists. In *6th Conference on Email and Anti-Spam (CEAS)*. Carnegie Mellon University, 2009.
[42] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu. Mining maximal quasi-bicliques to co-cluster stocks and financial ratios for value investment. In *ICDM*, pages 1059–1063. IEEE Computer Society, 2006.
[43] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu. Mining maximal quasi-bicliques: Novel algorithm and applications in the stock market and protein networks. *Stat. Anal. Data Min.*, 2(4):255–273, 2009.
[44] K. Sim, G. Liu, V. Gopalkrishnan, and J. Li. A case study on financial ratios via cross-graph quasi-bicliques. *Inf. Sci.*, 181(1):201–216, 2011.
[45] J. Tang, J. Sun, C. Wang, and Z. Yang. Social influence analysis in large-scale networks. In *KDD*, pages 807–816. ACM, 2009.
[46] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, pages 990–998. ACM, 2008.
[47] B. K. Tanner, G. Warner, H. Stern, and S. Olechowski. Koobface: The evolution of the social botnet. In *eCrime*, pages 1–10. IEEE, 2010.
[48] D. Ucar, S. Asur, U. Catalyurek, and S. Parthasarathy. Improving functional modularity in protein-protein interactions graphs using hub-induced subgraphs. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 371–382. Springer, 2006.
[49] C. Wei, A. Sprague, G. Warner, and A. Skjellum. Mining spam email to identify common origins for forensic application. In R. L. Wainwright and H. Haddad, editors, *ACM Symposium on Applied Computing*, pages 1433–1437. ACM, 2008.
[50] D. Weiss and G. Warner. Tracking criminals on facebook: A case study from a digital forensics reu program. In *Proceedings of Annual ADFSL Conference on Digital Forensics, Security and Law*, 2015.
[51] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE*, pages 1369–1380. IEEE, 2020.
[52] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W.-S. Ku, and J. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *The VLDB Journal*, pages 1–34, 2021.
[53] Y. Yang, D. Yan, H. Wu, J. Cheng, S. Zhou, and J. C. S. Lui. Diversified temporal subgraph pattern mining. In *SIGKDD*, pages 1965–1974. ACM, 2016.
[54] K. Yao and L. Chang. Efficient size-bounded community search over large networks. *Proc. VLDB Endow.*, 14(8):1441–1453, 2021.
[55] K. Y. Yeung, M. Medvedovic, and R. E. Bumgarner. From co-expression to co-regulation: how many microarray experiments do we need? *Genome biology*, 5(7):1–11, 2004.
[56] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *SIGKDD*, pages 797–802. ACM, 2006.
[57] 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.*, 32(2):13, 2007.

[58] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin. Enumerating maximal k-plexes with worst-case time guarantee. In *AAAI*, pages 2442–2449. AAAI Press, 2020.

APPENDIX

A. Proof of Theorem 1

Proof. Consider a (γ_1, γ_2)-quasi-clique Q of graph G. For an arbitrary vertex v in Q, let us use V_j to denote the set of vertices whose shortest distance in G^u is j hops away from v, and assume that we can decompose Q into $V_0, V_1, \ldots, V_\ell$. Then, we have

$$|V_0| = 1, \quad (4)$$
$$|V_1| \geq \gamma_{max} \cdot (|Q| - 1), \quad (5)$$
$$|V_{i-1}| + |V_i| + |V_{i+1}| \geq \gamma_{max} \cdot (|Q| - 1) + 1, \quad (6)$$
$$|V_{\ell-1}| + |V_\ell| \geq \gamma_{max} \cdot (|Q| - 1) + 1, \quad (7)$$

where Eq (4) is because $V_0 = \{v\}$; Eq (5) is because V_1 contain neighbors of v including γ_1 in-neighbors and γ_2 out-neighbors; Eq (6) is because for a vertex u in V_i, its neighbors must be within $V_{i-1} \cup V_i \cup V_{i+1}$ (recall that V_j's are defined over G^u), and u plus its neighbors contain at least $(\gamma_{max} \cdot (|Q| - 1) + 1)$ vertices; Eq (7) is because for a vertex u in V_ℓ, its neighbors must be within $V_{\ell-1} \cup V_\ell$. Then we can add the following formulas:

$$\begin{aligned}
|V_0| + |V_1| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_0| + |V_1| + |V_2| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_1| + |V_2| + |V_3| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_2| + |V_3| + |V_4| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
&\cdots , \\
|V_{\ell-5}| + |V_{\ell-4}| + |V_{\ell-3}| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_{\ell-4}| + |V_{\ell-3}| + |V_{\ell-2}| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_{\ell-3}| + |V_{\ell-2}| + |V_{\ell-1}| &\geq \gamma_{max} \cdot (|Q| - 1) + 1, \\
|V_{\ell-1}| + |V_\ell| &\geq \gamma_{max} \cdot (|Q| - 1) + 1.
\end{aligned}$$

After summation, we have $3 \cdot |Q| >$ the left hand side $\geq \ell \cdot (\gamma_{max} \cdot (|Q| - 1) + 1)$, so:

$$\ell < \frac{3|Q|}{\gamma_{max}(|Q| - 1) + 1},$$

which completes the proof since the vertex farthest from v in Q can be at most ℓ hops away. □

B. Proof of Theorem 2

Proof. Consider any two vertices u, v in a (γ_1, γ_2)-quasi-clique Q where $\gamma_1, \gamma_2 \geq 0.5$, we can easily show that u and v are at most 2 hops apart in G^u (c.f., Fig. 9). Specifically, we prove below that any two vertices u, v in Q cannot be more than 2 hops apart (i.e., cannot fall out of the 6 cases in Fig. 9).

Without loss of generality, we only consider the path from v to u where the first edge is outbound from v, i.e., Cases 1(a)–(c). Cases 2(a)–(c) are symmetric and can be similarly proved.

If v directly points to u, we are done since Case 1(a) occurs. Now assume that edge (v, u) does not exist in G, and we show that:

- **Case (I): edge (u, v) does not exist in G**, then both Case 1(b) and Case 1(c) should be satisfied. **(i) We first prove Case 1(b).** Note that $u \notin N^+(v)$ and $v \notin N^+(u)$. Since $\gamma_1 \geq 0.5$, u and v each points to at least $\lceil 0.5 \cdot (|Q| - 1) \rceil$ other vertices in Q, so they must share an out-neighbor; otherwise, there exist $2 \cdot \lceil 0.5 \cdot (|Q| - 1) \rceil \geq |Q| - 1$ vertices other than u and v, leading to a contradiction since there will be at least $(|Q| + 1)$ vertices in Q when adding u and v. **(ii) We next prove Case 1(c).** Note that $v \notin N^-(u)$ and $u \notin N^+(v)$. Since $\gamma_1 \geq 0.5$, v points to at least $\lceil 0.5 \cdot (|Q| - 1) \rceil$ other vertices in Q (here, u is excluded since $u \notin N^+(v)$); also since $\gamma_2 \geq 0.5$, u is pointed to by at least $\lceil 0.5 \cdot (|Q| - 1) \rceil$ other vertices in Q (here, v is excluded since $v \notin N^-(u)$). So, $N^+(v)$ and $N^-(u)$ must intersect as illustrated by Fig. 9 Case 1(c); otherwise, there will be $(|Q| + 1)$ vertices in Q when adding u and v.
- **Case (II): edge (u, v) exists in G,** then Case 1(c) should be satisfied. The proof is the same as (ii) above. Note that we cannot guarantee Case 1(b) anymore, since $v \in N^+(u)$, i.e., v can be one of the at least $\lceil 0.5 \cdot (|Q| - 1) \rceil$ neighbors of u, invalidating the prove for (i) above.

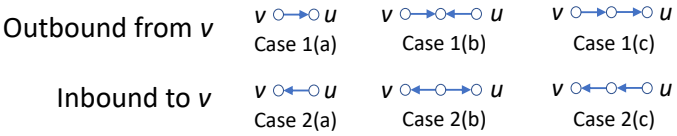

Fig. 9. Cases for Two-Hop Diameter Upper Bound

Symmetrically, consider the path from v to u where the first edge is inbound to v, i.e., Cases 2(a)–(c). If u directly points to v, we are done since Case 2(a) occurs. If edge (u, v) does not exist in G:

- **Case (III): edge (v, u) does not exist in G**, then both Case 2(b) and Case 2(c) should be satisfied. The proof is symmetric to Case (I) above and thus omitted.
- **Case (IV): edge (v, u) exists in G**, then Case 2(c) should be satisfied. The proof is symmetric to Case (II) above.

Putting the above discussions together, we obtain the following 4 cases, for each of which we explain how to exclude an impossible candidate u from $ext(S)$ given a vertex $v \in S$.

- **Case A: $(v, u) \in E$ and $(u, v) \in E$.** In this case, we always have $u \in ext(S)$.
- **Case B: $(v, u) \notin E$ and $(u, v) \in E$.** Based on Case (II) above, we have $u \in ext(S)$ only if a path $u \leftarrow w \leftarrow v$ exists in G for some $w \in V$ $(w \neq u, v)$.
- **Case C: $(v, u) \in E$ and $(u, v) \notin E$.** Based on Case (IV) above, we have $u \in ext(S)$ only if a path $u \rightarrow w \rightarrow v$ exists in G for some $w \in V$ $(w \neq u, v)$.
- **Case D: $(v, u) \notin E$ and $(u, v) \notin E$.** Based on Case (I) above, we have Condition (C1): $u \in ext(S)$ only if both Case 1(b) and Case 1(c) in Fig. 9 are satisfied. Similarly, based on Case (III) above, we have Condition (C2): $u \in ext(S)$ only if both Case 2(b) and Case 2(c) are satisfied. Combining both conditions, $u \in ext(S)$ only if there exist $w_1, w_2, w_3, w_4 \in V - \{u, v\}$ such that $u \leftarrow w_1 \leftarrow v$ and $u \leftarrow w_2 \rightarrow v$ and $u \rightarrow w_3 \leftarrow v$ and $u \rightarrow w_4 \rightarrow v$.

Once we have applied the above rules to prune $ext(S)$ to exclude invalid candidates u, let us abuse the notation to use G again to denote the resulting graph induced by $S \cup ext(S)$ after pruning. Note that we can apply this diameter-based pruning on the pruned G again, since some vertex w in Case B (resp. Case C) could have been pruned by Case C (resp. Case B) in the previous iteration, causing some required paths to disappear, further invalidating more vertices u from $ext(S)$. This pruning can be iteratively run over G.

Based on the above idea, Algorithm 1 computes the set of vertices in $ext(S)$ that are not 2-hop pruned by a vertex $v \in S$. Specifically, Line 1 computes O (resp. I) as the set of v's out-neighbors (resp. in-neighbors) u that belong to Case B (resp. Case C).

Then, Line 3 recovers S_O (resp. S_I) as the set of v's all non-pruned out-neighbors (resp. in-neighbors) w in Case B (resp. Case C) with path $v \to w \to u$ (resp. $v \leftarrow w \leftarrow u$). Note that $N^{\pm}(v) \subseteq ext(S)$ based on Case A so its vertices cannot be further pruned, so the iterative pruning is contributed by the shrink of sets O and I.

Next, Line 4 prunes away those vertices $u \in O$ (resp. $u \in I$) that cannot find a path $u \leftarrow w$ (resp. $u \to w$) for some non-pruned $w \in N^-(v)$ (resp. $w \in N^+(v)$), which is based on Case B (resp. Case C). Note that if O or I shrinks in Line 4, Line 5 will trigger another iteration of pruning. When the loop of Lines 2–5 exits, we have O (resp. I) being the remaining vertices $u \in ext(S)$ in Case B (resp. Case C) after iterative pruning.

Finally, Line 6 computes the set B of vertices where u satisfies Case D w.r.t. v, and Line 7 unions the 4 disjoint candidate sets that correspond to Cases A, B, C and D, respectively, to obtain the final 2-hop pruned $ext(S)$ for a vertex $v \in S$. We denote this set as $\mathbb{B}(v)$, which is returned by Line 7. □

C. Proof of Theorem 4

Proof. A valid (γ_1, γ_2)-quasi-clique $Q \subseteq V$ should contain at least τ_{size} vertices (i.e. $|Q| \geq \tau_{size}$), and therefore, for any $v \in Q$, its outdegree $d^+(v) \geq \lceil \gamma_1 \cdot (|Q|-1) \rceil \geq \lceil \gamma_1 \cdot (\tau_{size}-1) \rceil$ and indegree $d^-(v) \geq \lceil \gamma_2 \cdot (|Q|-1) \rceil \geq \lceil \gamma_2 \cdot (\tau_{size}-1) \rceil$. □

D. Degree-Based Pruning

Recall that $d^+_{V'}(v) = |N^+_{V'}(v)|$ and $d^-_{V'}(v) = |N^-_{V'}(v)|$. Thus, $d^+_S(v)$ (resp. $d^-_S(v)$) denotes the number of v's out-neighbors (resp. in-neighbors) in S, and $d^+_{ext(S)}(v)$ (resp. $d^-_{ext(S)}(v)$) denotes the number of v's out-neighbors (resp. in-neighbors) in $ext(S)$.

Theorem 8 (Type I Degree Pruning). *Given a vertex $u \in ext(S)$, if Condition (i): $d^+_S(u) + d^+_{ext(S)}(u) < \lceil \gamma_1 \cdot (|S| + d^+_{ext(S)}(u)) \rceil$ or Condition (ii): $d^-_S(u) + d^-_{ext(S)}(u) < \lceil \gamma_2 \cdot (|S| + d^-_{ext(S)}(u)) \rceil$ holds, then u can be pruned from $ext(S)$.*

This theorem is a result of the following lemma proven by [57].

Lemma 1. *If $a + n < \lceil \gamma \cdot (b + n) \rceil$ where $a, b, n \geq 0$, then $\forall i \in [0, n]$, we have $a + i < \lceil \gamma \cdot (b + i) \rceil$.*

Proof of Theorem 8. Theorem 8 follows since for any valid (γ_1, γ_2)-quasi-clique $Q = S \cup V'$ where $u \in V'$ and $V' \subseteq ext(S)$, we have

$$\begin{aligned} d^+_Q(u) &= d^+_S(u) + d^+_{V'}(u) && (8)\\ &< \lceil \gamma_1 \cdot (|S| + d^+_{V'}(u)) \rceil && (9)\\ &\leq \lceil \gamma_1 \cdot (|Q| - 1) \rceil, && (10) \end{aligned}$$

where Eq (8) is because $Q = S \cup V'$; Eq (9) is derived using Lemma 1, based on Condition (i) and the fact that $V' \subseteq ext(S)$; Eq (10) is because $\big(S \cup N^+_{V'}(u)\big) \subseteq \big(S \cup V' - \{u\}\big) = Q - \{u\}$. This result contradicts with the fact that Q is a (γ_1, γ_2)-quasi-clique. Condition (ii) is symmetric and a contradiction can be similarly derived. Therefore, if u satisfies either Condition (i) or (ii), we can safely prune u from $ext(S)$. □

Theorem 9 (Type II Degree Pruning). *Given vertex $v \in S$, if (1) $d^+_S(v) < \lceil \gamma_1 \cdot |S| \rceil$ and $d^+_{ext(S)}(v) = 0$, or (2) if $d^+_S(v) + d^+_{ext(S)}(v) < \lceil \gamma_1 \cdot (|S| - 1 + d^+_{ext(S)}(v)) \rceil$, then for any S' such that $S \subset S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.*

Given vertex $v \in S$, if (1) $d^-_S(v) < \lceil \gamma_2 \cdot |S| \rceil$ and $d^-_{ext(S)}(v) = 0$, or (2) if $d^-_S(v) + d^-_{ext(S)}(v) < \lceil \gamma_2 \cdot (|S| - 1 + d^-_{ext(S)}(v)) \rceil$, then for any S' such that $S \subset S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.

Proof. We hereby prove the pruning rule w.r.t. outdegrees, and the other rule w.r.t. indegrees is symmetric and can be similarly proved. First consider Condition (2), we have

$$\begin{aligned} d^+_Q(v) &= d^+_S(v) + d^+_{V'}(v) && (11)\\ &< \lceil \gamma_1 \cdot (|S| - 1 + d^+_{V'}(v)) \rceil && (12)\\ &\leq \lceil \gamma_1 \cdot (|Q| - 1) \rceil, && (13) \end{aligned}$$

where Eq (11) is because $Q = S \cup V'$; Eq (12) is derived using Lemma 1, based on Condition (2) and the fact that $V' \subseteq ext(S)$; Eq (13) is because $\big(S \cup N^+_{V'}(v)\big) \subseteq \big(S \cup V'\big) = Q$. This result contradicts with the fact that Q is a (γ_1, γ_2)-quasi-clique. Note that as long as we find one such $v \in S$, there is no need to extend S further. If $d^+_{ext(S)}(v) = 0$ in Condition (2), then we obtain $d^+_S(v) < \lceil \gamma_1 \cdot (|S| - 1) \rceil$ which is contained in Condition (1). Note that Condition (2) applies to the case $S = S'$ since i can be 0 in Lemma 1 (in contrast to Condition (1) to be explained below).

Now let us consider Condition (1). Condition (1) allows more effective pruning and is correct since for any valid quasi-clique $Q \supset S$ extended from S, we have $V' \neq \emptyset$ and

$$\begin{aligned} d^+_Q(v) &\leq d^+_S(v) + d^+_{ext(S)}(v) && (14)\\ &= d^+_S(v) && (15)\\ &< \lceil \gamma_1 \cdot (|Q| - 1) \rceil, && (16) \end{aligned}$$

where Eq (14) is because $Q = S \cup V'$ and $V' \subseteq ext(S)$; Eq (15) is because $d^+_{ext(S)}(v) = 0$ in Condition (1); Eq (16)

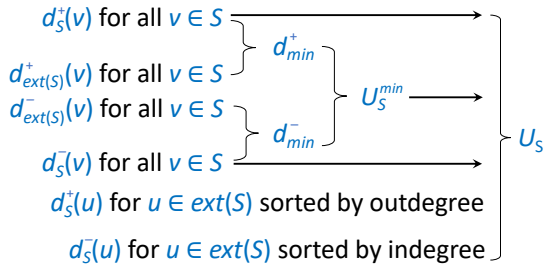

Fig. 10. Upper Bound Derivation

is because $d_S^+(v) < \lceil \gamma_1 \cdot |S| \rceil$ in Condition (1) and the fact that $|S| \leq |Q| - 1$ (recall that $V' \neq \emptyset$ and $Q = S \cup V'$). This result contradicts with the fact that Q is a (γ_1, γ_2)-quasi-clique. Note that the pruning of Condition (1) does not include the case where $S' = S$. □

E. Upper Bound Based Pruning

We next define an upper bound, denoted by U_S, on the number of vertices in $ext(S)$ that can be added to S concurrently to form a (γ_1, γ_2)-quasi-clique. The definition of U_S is based on $d_S^{\pm}(v)$ and $d_{ext(S)}^{\pm}(v)$ of all vertices $v \in S$ and on $d_S^{\pm}(u)$ of vertices $u \in ext(S)$ as summarized by Fig. 10, which we describe next.

We first define d_{min}^+ (resp. d_{min}^-) as the minimum outdegree (resp. minimum indegree) of any vertex in S, where the degrees are counted w.r.t. the other vertices in $S \cup ext(S)$ (c.f. Fig. 10):

$$d_{min}^+ = \min_{v \in S} \{d_S^+(v) + d_{ext(S)}^+(v)\}$$

$$d_{min}^- = \min_{v \in S} \{d_S^-(v) + d_{ext(S)}^-(v)\}$$

Now consider any quasi-clique S' such that $S \subseteq S' \subseteq (S \cup ext(S))$. For any $v \in S$, we have $d_S^+(v) + d_{ext(S)}^+(v) \geq d_{S'}^+(v) \geq \lceil \gamma_1 \cdot (|S'|-1) \rceil$ and therefore, $d_{min}^+ \geq \lceil \gamma_1 \cdot (|S'|-1) \rceil$. As a result, $\lfloor d_{min}^+ / \gamma_1 \rfloor \geq \lfloor \lceil \gamma_1 \cdot (|S'|-1) \rceil / \gamma_1 \rfloor \geq \lfloor \gamma_1 \cdot (|S'|-1)/\gamma_1 \rfloor = |S'| - 1$, which gives the following upper bound on $|S'|$:

$$|S'| \leq \lfloor d_{min}^+ / \gamma_1 \rfloor + 1. \tag{17}$$

We can similarly derive the other upper bound on $|S'|$ w.r.t. d_{min}^-:

$$|S'| \leq \lfloor d_{min}^- / \gamma_2 \rfloor + 1. \tag{18}$$

Combining Eq (17) and Eq (18), we obtain:

$$|S'| \leq \min\{\lfloor d_{min}^+ / \gamma_1 \rfloor, \lfloor d_{min}^- / \gamma_2 \rfloor\} + 1. \tag{19}$$

Let us define U_S^{min} as an upper bound on the number of vertices from $ext(S)$ that can further extend S to form a valid quasi-clique. Using Eq (19) and the fact that vertices in S are already included in a quasi-clique to find (i.e., $S \subseteq S'$), we obtain (c.f. Fig. 10):

$$U_S^{min} = \min\{\lfloor d_{min}^+ / \gamma_1 \rfloor, \lfloor d_{min}^- / \gamma_2 \rfloor\} + 1 - |S|. \tag{20}$$

We next tighten this upper bound using vertices in $ext(S) = \{u_1^+, u_2^+, \cdots u_n^+\}$[1], assuming that the vertices are listed in non-increasing order of outdegree $d_S^+(.)$. Similarly, we can also tighten this upper bound using vertices in $ext(S) = \{u_1^-, u_2^-, \cdots u_n^-\}$, assuming that the vertices are listed in non-increasing order of indegree $d_S^-(.)$. Then we have:

Lemma 2. *Given an integer k such that $1 \leq k \leq n$, if $\sum_{v \in S} d_S^+(v) + \sum_{i=1}^{k} d_S^-(u_i^-) < |S| \cdot \lceil \gamma_1(|S| + k - 1) \rceil$, then for any vertex set $Z \subseteq ext(S)$ with $|Z| = k$, $S \cup Z$ is not a (γ_1, γ_2)-quasi-clique.*

Proof. If S' is a (γ_1, γ_2)-quasi-clique, then for any $v \in S'$:

$$d_{S'}^+(v) \geq \lceil \gamma_1 \cdot (|S'| - 1) \rceil,$$

and therefore, for any $S \subseteq S'$, we have

$$\sum_{v \in S} d_{S'}^+(v) \geq |S| \cdot \lceil \gamma_1(|S'| - 1) \rceil. \tag{21}$$

Thus, to prove Lemma 2, we only need to show that

$$\sum_{v \in S} d_{S \cup Z}^+(v) < |S| \cdot \lceil \gamma_1(|S| + |Z| - 1) \rceil, \tag{22}$$

That is, Eq (21) is not satisfied for $S' = S \cup Z$, so a contradiction happens that invalidates S' from being a (γ_1, γ_2)-quasi-clique.

We now show that Eq (22) is correct below:

$$\begin{aligned}
\sum_{v \in S} d_{S \cup Z}^+(v) &= \sum_{v \in S} d_S^+(v) + \sum_{v \in S} d_Z^+(v) && (23)\\
&= \sum_{v \in S} d_S^+(v) + \sum_{u \in Z} d_S^-(u) && (24)\\
&\leq \sum_{v \in S} d_S^+(v) + \sum_{i=1}^{|Z|} d_S^-(u_i^-) && (25)\\
&< |S| \cdot \lceil \gamma_1(|S| + |Z| - 1) \rceil, && (26)
\end{aligned}$$

where Eq (23) is because $Z \subseteq ext(S)$ so $Z \cap S = \emptyset$; Eq (24) is because $\sum_{v \in S} d_Z^+(v) = \sum_{u \in Z} d_S^-(u)$ = the number of edges pointing from vertices in S to vertices in Z; Eq (25) is because $u_1^-, \cdots, u_{|Z|}^-$ are the k ($= |Z|$) vertices with the highest $d_S^-(.)$ in $ext(S)$; Eq (26) is because of Lemma 2 ($k = |Z|$). □

Symmetrically, we can also prove the following lemma:

Lemma 3. *Given an integer k such that $1 \leq k \leq n$, if $\sum_{v \in S} d_S^-(v) + \sum_{i=1}^{k} d_S^+(u_i^+) < |S| \cdot \lceil \gamma_2(|S| + k - 1) \rceil$, then for any vertex set $Z \subseteq ext(S)$ with $|Z| = k$, $S \cup Z$ is not a (γ_1, γ_2)-quasi-clique.*

Based on Lemma 2 and Lemma 3, we define a tightened upper bound U_S as follows (c.f. Fig. 10):

$$\begin{aligned}
U_S = \max \Bigg\{ t \;\Bigg|\; & \left(1 \leq t \leq U_S^{min}\right) \bigwedge \left(\sum_{v \in S} d_S^+(v) + \sum_{i=1}^{t} d_S^-(u_i^-) \right.\\
& \left. \geq |S| \cdot \lceil \gamma_1(|S| + t - 1) \rceil \right) \bigwedge \left(\sum_{v \in S} d_S^-(v) + \sum_{i=1}^{t} d_S^+(u_i^+) \right.\\
& \left. \geq |S| \cdot \lceil \gamma_2(|S| + t - 1) \rceil \right) \Bigg\}. \qquad (27)
\end{aligned}$$

[1] The superscript "+" is to indicate that vertices in $ext(S)$ are ordered by outdegree.

If such a t cannot be found, then S cannot be extended to generate a valid quasi-clique, which is a Type-II pruning. Otherwise, we further consider the 4 pruning rules to be described below which are based on U_S. Below, we only prove the theorems for outdegree-based upper bound pruning; the indegree-based rules are symmetric and can be similarly proved. We first describe Type-I pruning rules:

Theorem 10 (Type-I Outdegree Upper Bound Pruning)**.** *Given a vertex $u \in ext(S)$, if $d_S^+(u)+U_S-1 < \lceil \gamma_1 \cdot (|S|+U_S-1)\rceil$, then u can be pruned from $ext(S)$.*

Proof. Consider any valid quasi-clique $Q = S \cup V'$ where $u \in V'$ and $V' \subseteq ext(S)$. If the condition in Theorem 10 holds, i.e., $d_S^+(u)+U_S-1 < \lceil \gamma_1 \cdot (|S|+U_S-1)\rceil$, then based on Lemma 1 and the fact that $|V'| \le U_S$, we have:

$$d_S^+(u) + |V'| - 1 < \lceil \gamma_1 \cdot (|S| + |V'| - 1)\rceil = \lceil \gamma_1 \cdot (|Q| - 1)\rceil, \quad (28)$$

and therefore, $d_Q^+(u) = d_S^+(u)+d_{V'}^+(u) \le d_S^+(u)+|V'|-1 < \lceil \gamma_1 \cdot (|Q|-1)\rceil$ (where the last step is due to Eq (28)), which contradicts with the fact that Q is a quasi-clique. □

Symmetrically, we can also prove the following theorem:

Theorem 11 (Type-I Indegree Upper Bound Pruning)**.** *Given a vertex $u \in ext(S)$, if $d_S^-(u)+U_S-1 < \lceil \gamma_2 \cdot (|S|+U_S-1)\rceil$, then u can be pruned from $ext(S)$.*

We next describe Type-II pruning rules:

Theorem 12 (Type-II Outdegree Upper Bound Pruning)**.** *Given a vertex $v \in S$, if $d_S^+(v)+U_S < \lceil \gamma_1 \cdot (|S|+U_S-1)\rceil$, then for any S' such that $S \subseteq S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.*

Proof. Consider any valid quasi-clique $Q = S \cup V'$ where $v \in S$ and $V' \subseteq ext(S)$. If the condition in Theorem 12 holds, i.e., $d_S^+(v)+U_S < \lceil \gamma_1 \cdot (|S|+U_S-1)\rceil$, then based on Lemma 1 and the fact that $|V'| \le U_S$, we have:

$$d_S^+(v)+|V'| < \lceil \gamma_1 \cdot (|S|+|V'|-1)\rceil = \lceil \gamma_1 \cdot (|Q|-1)\rceil, \quad (29)$$

and therefore, $d_Q^+(v) = d_S^+(v) + d_{V'}^+(v) \le d_S^+(v) + |V'| < \lceil \gamma_1 \cdot (|Q|-1)\rceil$ (where the last step is due to Eq (29)), which contradicts with the fact that Q is a quasi-clique.

Since i can be 0 in Lemma 1, the pruning of Theorem 12 includes the case where $S' = S$, which is different from Theorem 9. □

Symmetrically, we can also prove the following theorem:

Theorem 13 (Type-II Indegree Upper Bound Pruning)**.** *Given a vertex $v \in S$, if $d_S^-(v)+U_S < \lceil \gamma_2 \cdot (|S|+U_S-1)\rceil$, then for any S' such that $S \subseteq S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.*

F. Lower Bound Based Pruning

Given a vertex set S, if some vertex $v \in S$ has $d_S^+(v) < \lceil \gamma_1 \cdot (|S|-1)\rceil$ (or $d_S^-(v) < \lceil \gamma_2 \cdot (|S|-1)\rceil$), then at least a certain number of vertices need to be added to S to increase the outdegree (or indegree) of v in order to form a (γ_1, γ_2)-quasi-clique. We denote this lower bound as L_{min}, which is

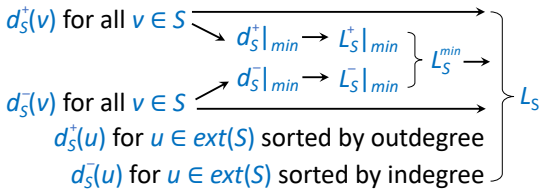

Fig. 11. Lower Bound Derivation

defined based on $d_S^{\pm}(v)$ of all vertices $v \in S$ and based on $d_S^{\pm}(u)$ of vertices $u \in ext(S)$ as summarized by Fig. 11, which we describe next.

We first define $d_S^+|_{min}$ as the minimum outdegree of any vertex in S and $d_S^-|_{min}$ as the minimum indegree of any vertex in S:

$$d_S^+|_{min} = \min_{v \in S} d_S^+(v), \qquad d_S^-|_{min} = \min_{v \in S} d_S^-(v)$$

Then, we can immediately derive the following two lower bounds:

$$L_S^+|_{min} = \min\{t \mid d_S^+|_{min} + t \ge \lceil \gamma_1 \cdot (|S| + t - 1)\rceil\} \quad (30)$$

$$L_S^-|_{min} = \min\{t \mid d_S^-|_{min} + t \ge \lceil \gamma_2 \cdot (|S| + t - 1)\rceil\} \quad (31)$$

Note that if even when all t newly added vertices are counted towards the degree of $v \in S$, the degree requirements w.r.t. γ_1 and γ_2 are still not satisfied, then we cannot make $S \cup Z$ (where $Z \subseteq ext(S)$ and $|Z| = t$) a valid quasi-clique, hence t is not valid. The lower bounds are taken as the smallest valid t.

To find such $L_S^+|_{min}$ (resp. $L_S^-|_{min}$), we check $t = 0, 1, \cdots, |ext(S)|$, and if none of them satisfies the inequality in Eq (30) (resp. Eq (31)), then S and its extensions cannot produce a valid quasi-clique, which is a Type-II pruning.

Otherwise, we obtain a lower bound:

$$L_S^{min} = \max\{L_S^+|_{min},\ L_S^-|_{min}\}. \quad (32)$$

We can further tighten this lower bound into L_S below using Lemma 2, assuming that vertices in $ext(S) = \{u_1^+, u_2^+, \cdots, u_n^+\}$ are listed in non-increasing order of $d_S^+(.)$, and $ext(S) = \{u_1^-, u_2^-, \cdots, u_n^-\}$ are listed in non-increasing order of $d_S^-(.)$:

$$L_S = \min\Bigg\{t \;\Bigg|\; \Big(L_S^{min} \le t \le n\Big) \bigwedge \Bigg(\sum_{v \in S} d_S^+(v) + \sum_{i=1}^{t} d_S^-(u_i^-) \ge |S| \cdot \lceil \gamma_1(|S| + t - 1)\rceil\Bigg) \bigwedge \Bigg(\sum_{v \in S} d_S^-(v) + \sum_{i=1}^{t} d_S^+(u_i^+) \ge |S| \cdot \lceil \gamma_2(|S| + t - 1)\rceil\Bigg)\Bigg\}. \quad (33)$$

If such a t cannot be found, then S cannot be extended to generate a valid quasi-clique, which is Type-II pruning. Otherwise, we further consider 4 pruning rules based on L_S which we list below. There, we only prove the theorems w.r.t. outdegree, since those w.r.t. indegree are symmetric. We first describe Type-I pruning rules:

Theorem 14 (Type-I Outdegree Lower Bound Pruning)**.** *Given a vertex $u \in ext(S)$, if $d_S^+(u) + d_{ext(S)}^+(u) < \lceil \gamma_1 \cdot (|S| + L_S - 1) \rceil$, then u can be pruned from $ext(S)$.*

Proof. Consider any valid quasi-clique $Q = S \cup V'$ where $u \in V'$ and $V' \subseteq ext(S)$. If the condition in Theorem 14 holds, we have $d_Q^+(u) = d_S^+(u) + d_{V'}^+(u) \leq d_S^+(u) + d_{ext(S)}^+(u) < \lceil \gamma_1 \cdot (|S| + L_S - 1) \rceil$ (due to the condition in Theorem 14) $\leq \lceil \gamma_1 \cdot (|Q| - 1) \rceil$ (since $L_S \leq |V'|$), which contradicts the fact that Q is a quasi-clique. □

Symmetrically, we can also prove the following theorem:

Theorem 15 (Type-I Indegree Lower Bound Pruning)**.** *Given a vertex $u \in ext(S)$, if $d_S^-(u) + d_{ext(S)}^-(u) < \lceil \gamma_2 \cdot (|S| + L_S - 1) \rceil$, then u can be pruned from $ext(S)$.*

We next describe Type-II pruning rules:

Theorem 16 (Type-II Outdegree Lower Bound Pruning)**.** *Given a vertex $v \in S$, if $d_S^+(v) + d_{ext(S)}^+(v) < \lceil \gamma_1 \cdot (|S| + L_S - 1) \rceil$, then for any S' such that $S \subseteq S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.*

Proof. Consider any valid quasi-clique $Q = S \cup V'$ where $v \in S$ and $V' \subseteq ext(S)$. If the condition in Theorem 16 holds, we have $d_Q^+(v) = d_S^+(v) + d_{V'}^+(v) \leq d_S^+(v) + d_{ext(S)}^+(v) < \lceil \gamma_1 \cdot (|S| + L_S - 1) \rceil$ (due to the condition in Theorem 16) $\leq \lceil \gamma_1 \cdot (|Q| - 1) \rceil$ (since $L_S \leq |V'|$), which contradicts the fact that Q is a quasi-clique. □

Symmetrically, we can also prove the following theorem:

Theorem 17 (Type-II Indegree Lower Bound Pruning)**.** *Given a vertex $v \in S$, if $d_S^-(v) + d_{ext(S)}^-(v) < \lceil \gamma_2 \cdot (|S| + L_S - 1) \rceil$, then for any S' such that $S \subseteq S' \subseteq (S \cup ext(S))$, $G(S')$ cannot be a (γ_1, γ_2)-quasi-clique.*

G. Proof of Theorems 5 and 6

Proof. This theorem is correct because if $u \in N_{ext(S)}^+(v)$ is not in S', then $d_{S'}^+(v) < d_S^+(v) + d_{ext(S)}^+(v) = \lceil \gamma_1 \cdot (|S| + L_S - 1) \rceil$ (due to Definition 3) $\leq \lceil \gamma_1 \cdot (|S'| - 1) \rceil$, which contradicts with the fact that S' is a (γ_1, γ_2)-quasi-clique. □

Symmetrically, we can also prove Theorems 6.

H. Proof of Theorem 7

We first prove that for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C_S^+(u)$, we have $d_{Q \cup u}^+(w) \geq \lceil \gamma_1 \cdot (|Q \cup u| - 1) \rceil = \lceil \gamma_1 \cdot |Q| \rceil$ for any vertex $w \in Q \cup u$. The other guarantee w.r.t. $C_S^-(u)$ is symmetric and can be similarly proved.

Proof. Recall from Fig. 3 that we only compute $C_S^+(u)$ for pruning if we have

$$d_S^+(u) \geq \lceil \gamma_1 \cdot |S| \rceil \quad (34)$$
$$d_S^+(v) \geq \lceil \gamma_1 \cdot |S| \rceil, \quad \forall\, v \in S \wedge v \notin N^-(u) \quad (35)$$

We divide the vertices $w \in Q \cup u$ in 3 disjoint sets (1) S, (2) $C_S^+(u) \subseteq ext(S)$, and (3) $\{u\}$ into 4 categories as follows, and prove that $d_{Q \cup u}^+(w) \geq \lceil \gamma_1 \cdot |Q| \rceil$ for any vertex w.

- **Case 1:** $w = u$ **(red in Fig. 3).** Then, we have

$$\begin{aligned} d_{Q \cup u}^+(u) &= d_S^+(u) + |Q| - |S| && (36)\\ &\geq \lceil \gamma_1 \cdot |S| \rceil + |Q| - |S| && (37)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil + |Q| - |Q| && (38)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil, \end{aligned}$$

  where Eq (36) is because u points to all the blue vertices in $C_S^+(u)$ (c.f. Fig. 3); Eq (37) is because of Eq (34); and Eq (38) is because $S \subseteq Q$ and $\lceil \gamma_1 - 1 \rceil \leq 0$.
- **Case 2:** $w \in S$ **and** $w \notin N^-(u)$ **(green in Fig. 3).**

$$\begin{aligned} d_{Q \cup u}^+(w) &= d_S^+(w) + |Q| - |S| && (39)\\ &\geq \lceil \gamma_1 \cdot |S| \rceil + |Q| - |S| && (40)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil + |Q| - |Q| && (41)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil, \end{aligned}$$

  where Eq (39) is because all the green vertices point to all the blue vertices in $C_S^+(u)$ (c.f. Fig. 3); Eq (40) is because of Eq (35); and Eq (41) is because $S \subseteq Q$ and $\lceil \gamma_1 - 1 \rceil \leq 0$.
- **Case 3:** $w \in S$ **and** $w \in N^-(u)$ **(yellow in Fig. 3).**

$$\begin{aligned} d_{Q \cup u}^+(w) &= d_Q^+(w) + 1 && (42)\\ &\geq \lceil \gamma_1 \cdot (|Q| - 1) \rceil + 1 && (43)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil, && (44) \end{aligned}$$

  where Eq (42) is because any yellow vertex in S should point to u (c.f. Fig. 3); Eq (43) is because Q is a (γ_1, γ_2)-quasi-clique; and Eq (44) is because $\lceil 1 - \gamma_1 \rceil \geq 0$.
- **Case 4:** $w \in C_S^+(u)$ **(blue in Fig. 3).**

$$\begin{aligned} d_{Q \cup u}^+(w) &= d_Q^+(w) + 1 && (45)\\ &\geq \lceil \gamma_1 \cdot (|Q| - 1) \rceil + 1 && (46)\\ &\geq \lceil \gamma_1 \cdot |Q| \rceil, && (47) \end{aligned}$$

  where Eq (45) is because any blue vertex in $C_S^+(u)$ should point to u (c.f. Fig. 3); Eq (46) is because Q is a (γ_1, γ_2)-quasi-clique; and Eq (47) is because $\lceil 1 - \gamma_1 \rceil \geq 0$.

As a special case, if all vertices in S points to u, then we do not have any vertex in Case 2, and our proof still holds. Here, we just need to compute $C_S^-(u) = N_{ext(S)}^+(u) \cap N_{ext(S)}^-(u)$ (c.f. Eq (1)). □

The other guarantee w.r.t. $C_S^-(u)$ (c.f. Eq (2)) is symmetric and can be similarly proved by reversing the directions of all edges. That is, for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C_S^-(u)$, $d_{Q \cup u}^-(w) \geq \lceil \gamma_2 \cdot |Q| \rceil$ for any $w \in Q \cup u$. Combining both guarantees, for any (γ_1, γ_2)-quasi-clique Q generated by extending S with vertices in $C_S(u) = C_S^+(u) \cap C_S^-(u)$, $Q \cup u$ is also a (γ_1, γ_2)-quasi-clique so Q is not maximal.

As for the degenerate special case when initially $S = \emptyset$, Eq (1) (resp. Eq (2)) becomes $C_S^+(u) = C_S^-(u) = N_{ext(S)}^+(u) \cap N_{ext(S)}^-(u)$ and all neighbors of u belong to $ext(S)$, so $C_S(u) = C_S^+(u) \cap C_S^-(u) = N_{ext(S)}^+(u) \cap$

Algorithm 4 *look_ahead*(S, $ext(S)$)

1: **if** $|Q| < \tau_{size}$ **return** *FAIL*
2: **for each** $w \in S \cup ext(S)$ **do** {here, we are iterating array A}
3: **if** $d^+_S(w) + d^+_{ext(S)}(w) < d^+_{min}(|S| + |ext(S)|)$ **return** *FAIL*
4: **if** $d^-_S(w) + d^-_{ext(S)}(w) < d^-_{min}(|S| + |ext(S)|)$ **return** *FAIL*
5: **if** $w \in ext(S)$ **and** $d^{\mathbb{B}}_{ext(S)}(w) < |ext(S)| - 1$ **return** *FAIL*
6: **return** *SUCCEED*

$N^-_{ext(S)}(u) = N^+(u) \cap N^-(u)$, i.e., we only need to find u as the vertex adjacent to the most number of bidirectional edges in G to maximize $|C_S(u)|$ for cover-vertex pruning. This is correct, since in our previous proof, there are no vertex in Cases 2 and 3 so no vertex breaks the requirement $d^+_{Q\cup u}(w) \geq \lceil \gamma_1 \cdot |Q| \rceil$ for any vertex $w \in Q \cup u$.

I. Degree Fields Maintained by Vertices in Array A

Each vertex object v in A (i.e., $S \cup ext(S)$) maintains five degrees (1) $d^+_S(v)$, (2) $d^-_S(v)$, (3) $d^+_{ext(S)}(v)$, (4) $d^-_{ext(S)}(v)$ and (5) the number of 2-hop neighbors $\mathbb{B}(v)$ that are in $ext(S)$, denoted by $d^{\mathbb{B}}_{ext(S)}(v) = |\mathbb{B}(v) \cap ext(S)|$. These five degree values are kept up-to-date whenever S and/or $ext(S)$ changes during the recursive mining, so that they can be accessed in $O(1)$ time when needed. Recall that these degree values are frequently needed when evaluating the conditions of our pruning rules, such as computing the bounds U_S and L_S as summarized in Fig. 10 and 11, so accessing them in $O(1)$ time is performance-critical. In fact, incrementally maintaining these degree values has a low cost: if a vertex v is moved or pruned, we only need to access those vertices in $N^+(v)$, $N^-(v)$, and $\mathbb{B}(v)$ to increment/decrement their degree values w.r.t. S and/or $ext(S)$.

J. Quasi-Clique Validation & Look-Ahead Pruning

Quasi-Clique Validation. Recall that τ_{size} is the size threshold for a valid quasi-clique. Let us define two functions:

$$d^+_{min}(size) = \lceil \gamma_1 \cdot (\max\{size, \tau_{size}\} - 1) \rceil, \quad (48)$$
$$d^-_{min}(size) = \lceil \gamma_2 \cdot (\max\{size, \tau_{size}\} - 1) \rceil. \quad (49)$$

Then, the following theorem directly follows from the definition of (γ_1, γ_2)-quasi-clique:

Theorem 18. *Let Q be a vertex set, then Q is a valid quasi-clique if and only if (i) $|Q| \geq \tau_{size}$ and (ii) for any vertex $v \in Q$, we have $d^+_Q(v) \geq d^+_{min}(|Q|)$ and $d^-_Q(v) \geq d^-_{min}(|Q|)$.*

The Look-Ahead Technique. This technique examines if $S \cup ext(S)$ gives a valid quasi-clique, and if so, we output it and avoid the unnecessary depth-first traversal of the subtree T_S. The rationale is that when $G(S \cup ext(S))$ is a valid quasi-clique and hence dense, traversing T_S can be expensive since pruning is less likely to be applicable during the traversal. In fact, when mining structures with a hereditary property such as k-plexes, look-ahead pruning is even essential since if $G(S \cup ext(S))$ is a k-plex, every node S in T_S is also a k-plex (and will thus be explored) but not maximal [58].

Algorithm 4 checks if $G(S \cup ext(S))$ is a (γ_1, γ_2)-quasi-clique, and returns *SUCCEED* if so to skip the traversal of subtree T_S. Specifically, we first make sure $|Q| \geq \tau_{size}$ in Line 1. Then, we check each vertex w in array $A = [S, ext(S)]$ one by one (Line 2), to examine the conditions of Theorem 18. If the conditions hold for all vertices in $S \cup ext(S)$, then $G(S \cup ext(S))$ is a valid quasi-clique so we return *SUCCEED* in Line 6; while if they do not hold for some vertex w, we return *FAIL* immediately as in Lines 3 and 4.

Line 5 provides an additional pruning if w is a vertex in $ext(S)$: if $d^{\mathbb{B}}_{ext(S)}(w) < |ext(S)| - 1$ (recall that we keep $d^{\mathbb{B}}_{ext(S)}(w) = |\mathbb{B}(w) \cap ext(S)|$ with w), then there must exist another vertex $u \in ext(S)$ such that $u \notin \mathbb{B}(w)$, so u and w cannot appear together in any valid quasi-clique, including $G(S \cup ext(S))$. Note that we do not need to consider $w \in S$ since when we add w into S, we always make sure that $w \in \mathbb{B}(v)$ for any $v \in S$, and we always prune away those vertices in $ext(S)$ that are not in $\mathbb{B}(w)$.

Algorithm 5 *iterative_bound_pruning*(S, $ext(S)$)

1: compute bounds U_S and L_S
2: conduct degree-based, upper- and lower-bound based pruning using Type-II pruning rules for every $v \in S$
3: **if** S is Type-II pruned **do** **return** *PRUNED*
4: **if** $L_S \leq U_S$ **then**
5: conduct degree-based, upper- and lower-bound based pruning using Type-I pruning rules for every $u \in ext(S)$
6: **while** $ext(S) \neq \emptyset$ **and** $ext(S)$ shrank **do**
7: update $d^+_{ext(S)}(.)$ and $d^-_{ext(S)}(.)$
8: compute bounds U_S and L_S
9: conduct degree-based, upper- and lower-bound based pruning using Type-II pruning rules for every $v \in S$
10: **if** S is Type-II pruned **then** **return** *PRUNED*
11: **if** $L_S > U_S$ **then** $ext(S) \leftarrow \emptyset$; **return** *NOT_PRUNED*
12: conduct degree-based, upper- and lower-bound based pruning using Type-I pruning rules for every $u \in ext(S)$
13: **else** $ext(S) \leftarrow \emptyset$
14: **return** *NOT_PRUNED*

K. Iterative Bound-Based Pruning

Whenever we remove a candidate vertex from $ext(S)$ and/or add a candidate vertex to S, the degrees of the vertices in array A w.r.t. S and $ext(S)$ would be incrementally updated, creating new opportunities for degree-based pruning (c.f. Appendix D). Moreover, the degree updates would also cause the bounds L_S and U_S to be updated (c.f. Fig. 10 and 11), creating new opportunities for upper bound pruning (c.f. Appendix E) and lower bound based pruning (c.f. Appendix F).

Note that some of the above pruning rules could be Type I rules, causing $ext(S)$ to shrink, which in turn reduces $d^+_{ext(S)}(.)$ and $d^-_{ext(S)}(.)$ and thus triggers another round of bound-based pruning.

Algorithm 5 shows the process of iterative bound-based pruning. Specifically, Line 1 first computes U_S and L_S following the procedures summarized in Fig. 10 and 11. Type-II pruning may occur during the process of computing U_S and L_S (c.f., the paragraphs after Eq (27), after Eq (33) and before Eq (32)). If Type-II pruning occurs in Line 1, Algorithm 5 will return tag *PRUNED* directly so that the main algorithm will skip subtree T_S. Otherwise, Line 2 conducts degree-based Type-II pruning (i.e., Theorem 9), upper-bound based Type-II pruning (i.e., Theorems 12 and 13), and lower-bound

TABLE VIII
EFFECT OF QUASI-CLIQUE PARAMETERS ON *Bitcoin*

τ_{size}	γ_1	γ_2	Runtime (all rules)	Runtime (w/o cover)	# Maximal
10	0.67	0.6	72.32	16.29	166,014
	0.68		68.53	15.49	166,014
	0.69		56.13	14.40	174,785
	0.7		36.71	9.88	287,139
	0.71		21.40	3.28	24,962
	0.72		16.20	2.78	34,470
	0.73		12.09	1.88	9,446
10	0.7	0.57	54.32	13.39	261,451
		0.58	43.23	12.09	281,868
		0.59	38.20	8.99	287,139
		0.6	36.71	9.88	287,139
		0.61	23.40	4.18	72,215
		0.62	22.71	4.39	72,333
		0.63	23.42	4.29	72,333
7	0.7	0.6	41.65	12.54	320,836
8			41.82	12.82	320,763
9			39.44	11.21	289,114
10			36.71	9.88	287,139
11			36.00	9.00	287,138
12			23.83	1.99	24,344

TABLE IX
EFFECT OF QUASI-CLIQUE PARAMETERS ON *Epinions*

τ_{size}	γ_1	γ_2	Runtime (all rules)	Runtime (w/o cover)	# Maximal
20	0.77	0.9	15.54	6.24	469
	0.78		15.86	6.15	469
	0.79		15.95	5.94	469
	0.8		15.54	6.82	469
	0.81		10.84	5.05	345
	0.82		10.84	4.25	345
	0.83		10.63	4.74	345
20	0.8	0.87	46.49	11.27	2,669
		0.88	34.88	9.28	2,669
		0.89	18.48	7.37	2,669
		0.9	15.54	6.82	469
		0.91	6.24	4.24	0
		0.92	5.44	3.13	0
		0.93	3.84	2.74	0
17	0.8	0.9	18.60	7.14	687
18			17.11	7.62	477
19			17.28	6.38	473
20			15.54	6.82	469
21			14.95	5.84	469
22			10.48	4.07	24

based Type-II pruning (i.e., Theorems 16 and 17). Algorithm 5 returns *PRUNED* directly in Line 3 if S is Type-II pruned. Otherwise, if we find $L_S > U_S$ in Line 4 meaning that S cannot be expanded further into a valid quasi-clique, we set $ext(S) \leftarrow \emptyset$ in Line 13 and return *NOT_PRUNED* in Line 14 to indicate that T_S is not Type-II pruned. Note that if *iterative_bound_pruning*$(S, ext(S))$ returns *NOT_PRUNED* but $ext(S)$ has been set to $\emptyset$, we still need to examine if $G(S)$ is a valid quasi-clique but not any other descendant in T_S.

Otherwise, Line 5 then conducts degree-based Type-I pruning (i.e., Theorem 8), upper-bound based Type-I pruning (i.e., Theorems 10 and 11), and lower-bound based Type-I pruning (i.e., Theorems 14 and 15). If some vertices have been Type-I pruned from $ext(S)$, and $ext(S) \neq \emptyset$ (Line 6), then since the degrees $d^+_{ext(S)}(.)$ and $d^-_{ext(S)}(.)$ may decrease triggering the update of U_S and L_S and hence more pruning opportunities, we enter the iterative pruning procedure given by Lines 6–12. Specifically, Line 7 updates $d^+_{ext(S)}(.)$ and $d^-_{ext(S)}(.)$, and Line 8 updates U_S and L_S, to reflect the removal of Type-I pruned vertices from $ext(S)$. Then, Line 9 conducts Type-II pruning once more, followed by Line 12 for Type-I pruning once more, and the iterative pruning repeats by going back to Line 6 for another iteration of pruning.

Note that Line 10 and Line 11 help skip unnecessarily executing the expensive checking in Line 12 before checking the loop-exiting conditions in Line 6. Another detail not shown in Algorithm 5 is with Condition (1) in Theorem 9 which Type-II prunes T_S except for S itself, in which case instead of returning *PRUNED*, we set $ext(S) = \emptyset$ and return *NOT_PRUNED* so $G(S)$ will still be examined.

L. Effect of Quasi-Clique Parameters

Effect of Quasi-Clique Parameters. Recall that we tuned the quasi-clique parameters $(\tau_{size}, \gamma_1, \gamma_2)$ and used them in our experiments by default. Here, we show how the mining time and number of results vary as the parameters change, using *Bitcoin* and *Epinions*.

We show the effect of changing the quasi-clique parameters $(\tau_{size}, \gamma_1, \gamma_2)$ by varying one parameter while fixing the other two. Table VIII shows the results on *Bitcoin* for illustration. We can see that a small change of a parameter value can change the number of results a lot. For example, when changing γ_1 from $(10, 0.7)$ to $(10, 0.71)$, the result number decreases from 287,139 to 24,962 due to the stricter density requirements. The change of result number is, however, not monotonic. For example, when changing γ_1 from $(10, 0.69)$ to $(10, 0.70)$, the result number actually increases which might appear counter-intuitive. The reason is that some previously valid quasi-cliques get split into multiple smaller quasi-cliques rather than being eliminated. Table IX shows the results

TABLE X
ABLATION STUDY: ALL BUT ONE

Algorithm	PolBlogs		Epinions		Google	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
full version	8.68	79	15.66	479	0.77	564
w/o lookahead	7.96	72	14.74	520	0.78	611
w/o critical	9.08	88	16.25	532	0.78	583
w/o cover	1.46	31	6.82	290	0.79	565
w/o bound	48.90	184	268.40	924	0.76	572

Algorithm	Baidu		USA Road		ClueWeb	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
full version	9.21	1,524	9.81	15,170	172.85	25,5371
w/o lookahead	10.58	1,436	10.94	15,250	175.42	25,5690
w/o critical	10.20	1,509	10.79	15,223	171.75	25,5349
w/o cover	8.82	1,576	11.06	14,990	174.79	25,5437
w/o bound	10.67	1,439	10.52	15,041	176.47	25,5513

TABLE XI
ABLATION STUDY: INCREMENTAL ADDITION

Algorithm	MathOverflow		PolBlogs		Epinions	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
baseline	385.29	798	2.30	45	94.76	665
+bound	22.82	217	1.40	24	5.81	291
+critical	17.51	186	1.40	23	6.71	273
+lookahead	17.05	209	1.46	31	6.82	290
+cover	551.49	738	8.68	79	15.66	479

Algorithm	Baidu		USA Road		ClueWeb	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
baseline	20.57	1,549	10.34	15,575	191.63	257,502
+bound	19.60	1,473	9.62	15,450	199.79	255,568
+critical	19.97	1,604	8.71	15,433	196.47	255,567
+lookahead	8.82	1,576	9.82	15,225	174.79	255,437
+cover	9.21	1,524	9.81	15,170	172.85	255,371

on *Epinions* for illustration, and we can obtain a similar observation.

M. Ablation Study

We report the ablation study results of those algorithm variants which use all but one technique on the other 6 datasets in Table X. We can see that bound-based pruning is very effective, without which the running time can be much longer as on *PolBlogs* and *Epinions*. Also, our recommended configuration "w/o cover" is consistently the fastest or near-fastest, and exhibits much better performance on *PolBlogs* and *Epinions* than other configurations.

We also report our algorithm variants starting from a baseline with basic diameter-based, size-threshold, and degree-based pruning, and incrementally adding bound-based, critical-vertex, look-ahead, and cover-vertex pruning, one at a time. This gives algorithm variants denoted by "baseline," "+bound," "+critical," "+lookahead," and "+cover."

Table XI reports the results on the other 6 datasets. We can see that bound-based pruning significantly speeds up the baseline, especially on *MathOverflow* and *Epinions*. As we have discussed previously, adding look-ahead and cover-vertex pruning generally slows down the computation but can speed up web graphs such as *Baidu* and *ClueWeb* (as well as *Google* as shown in Table VI).

N. Scalability

We report the scalability study results of our parallel algorithm with all pruning rules enabled in Fig. 12, and we report the scalability study results of our parallel algorithm with all but cover-vertex pruning in Fig. 13.

We can observe on most datasets that the running time almost halves each time the number of threads doubles, except that the time curve hits a floor higher than 0 on *Baidu* and *ClueWeb* as the memory-bound computing of $\mathbb{B}(v)$ dominates the runtime.

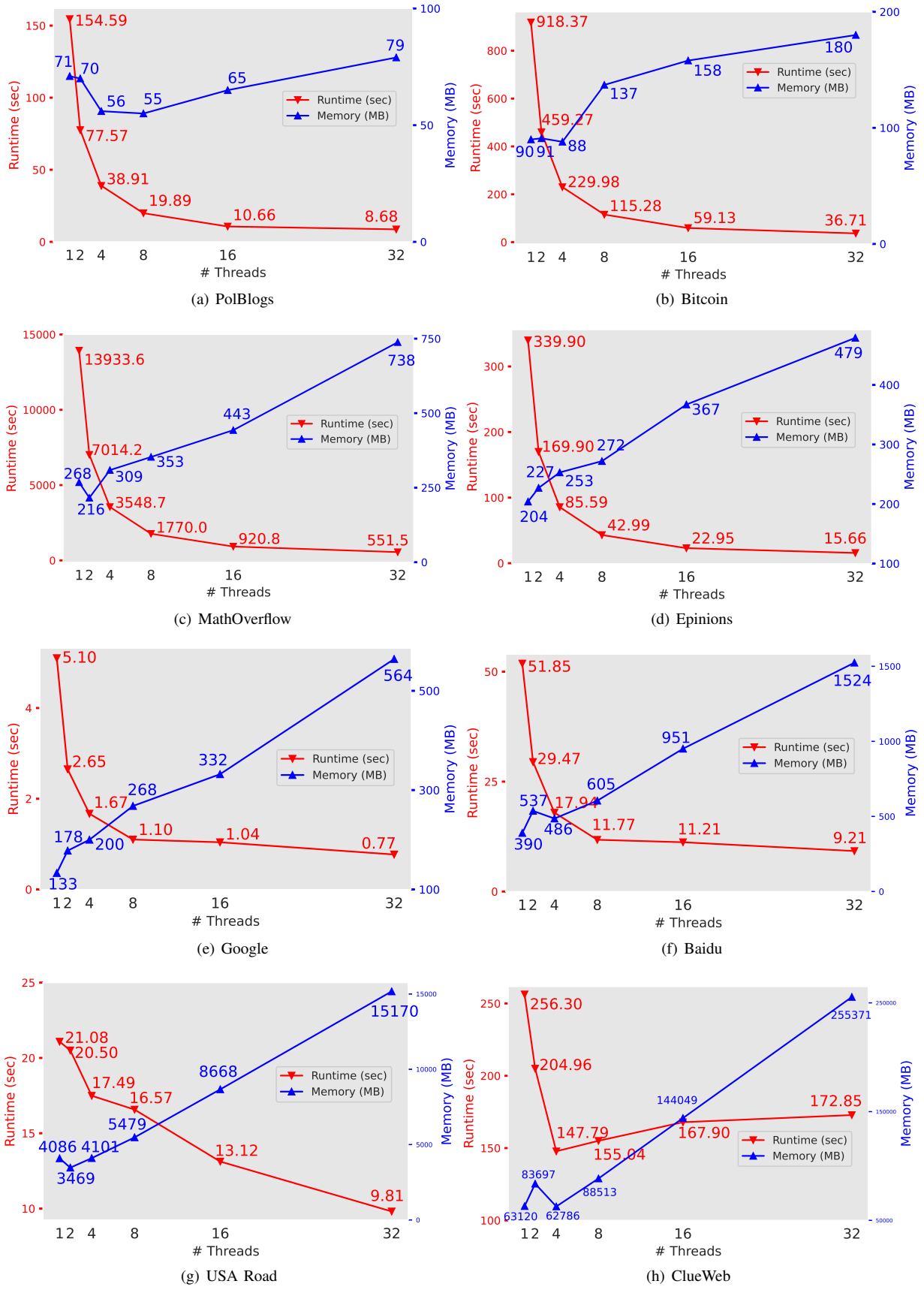

Fig. 12. Scalability of "full version"

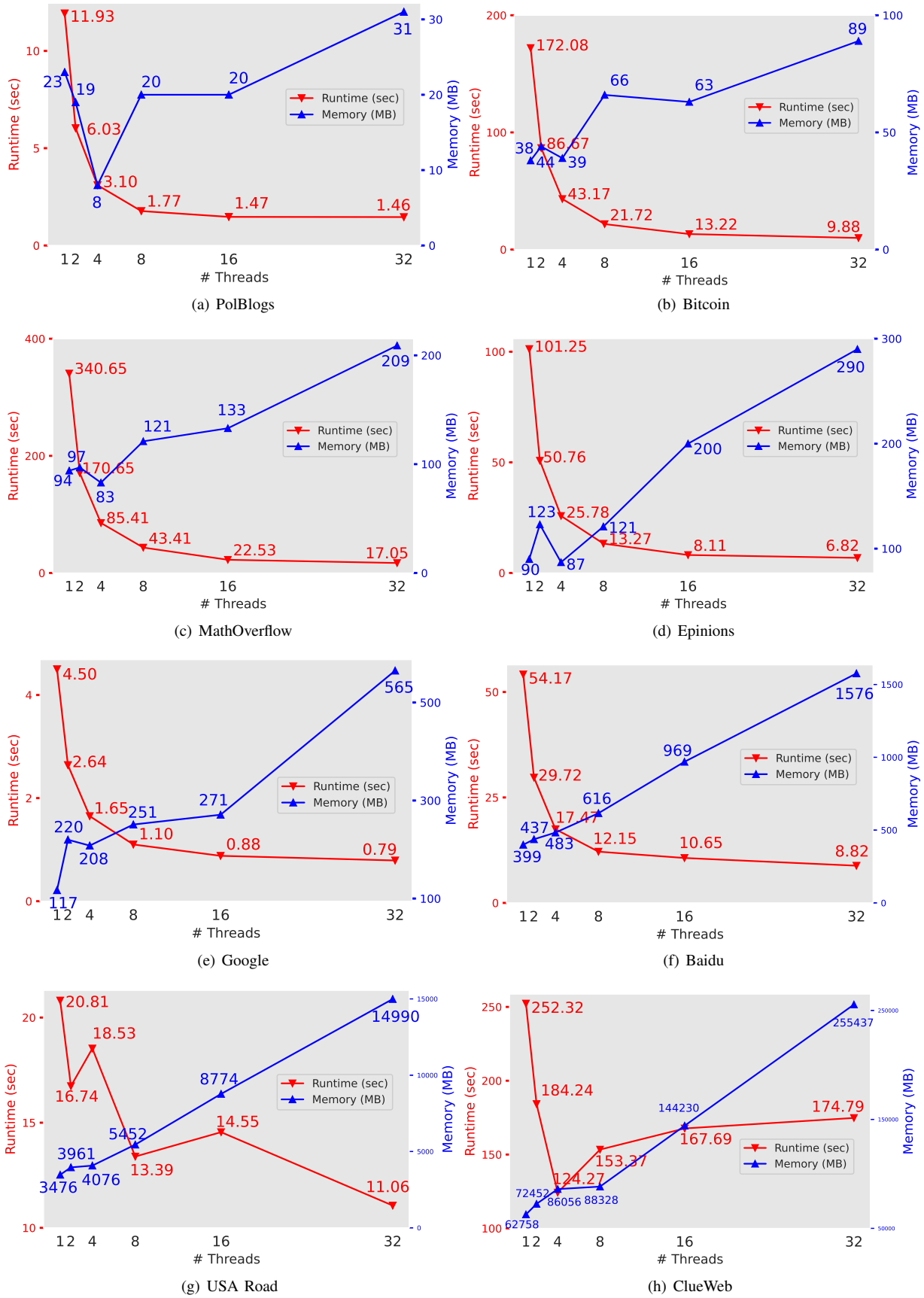

Fig. 13. Scalability of "w/o cover"