Efficient k-Clique Listing: An Edge-Oriented Branching Strategy

Anonymous Author(s)

ABSTRACT

k-clique listing is a vital graph mining operator with diverse applications in various networks. The state-of-the-art algorithms all adopt a branch-and-bound (BB) framework with a vertex-oriented branching strategy (called `VBBkC`), which forms a sub-branch by expanding a partial k-clique with a *vertex*. These algorithms have the time complexity of $O(k \cdot m \cdot (\delta/2)^{k-2})$, where m is the number of edges in the graph and δ is the degeneracy of graph. In this paper, we propose a BB framework with a new *edge-oriented branching* (called `EBBkC`), which forms a sub-branch by expanding a partial k-clique with two vertices that connect each other (which correspond to an *edge*). We explore various edge orderings for `EBBkC` such that it achieves a time complexity of $O(k \cdot m \cdot (\tau/2)^{k-2})$, where τ is an integer related to the maximum truss number of the graph and we have $\tau < \delta$. Furthermore, we develop specialized algorithms for sub-branches on dense graphs so that we can early-terminate them and apply the specialized algorithms. We conduct extensive experiments on 16 real graphs, and the results show that our newly developed `EBBkC` based algorithms with the early termination technique consistently and largely outperform the state-of-the-art (`VBBkC` based) algorithms.

1 INTRODUCTION

Given a graph G, a k-clique is subgraph of G with k vertices such that each pair of vertices inside are connected [9]. k-clique listing, which is to list all k-cliques in a graph, is a fundamental graph mining operator that plays a crucial role in various data mining applications across different networks, including social networks, mobile networks, Web networks, and biological networks. Some significant applications include the detection of overlapping communities in social networks [24], identifying k-clique communities in mobile networks [14, 16, 25], detecting link spams in Web networks [28], and discovering groups of functionally related proteins (known as modules) in gene association networks [1]. Moreover, k-clique listing serves as a key component for several other tasks, such as finding large near cliques [32], uncovering the hierarchical structure of dense subgraphs [29], exploring k-clique densest subgraphs [33], identifying stories in social media [2], and detecting latent higher-order organization in real-world networks [5]. For more detailed information on how k-clique listing is applied in these contexts, please refer to references [19] and [36].

Quite a few algorithms have been proposed for listing k-cliques [9, 11, 19, 36]. The majority of these approaches adopt a *branch-and-bound* (BB) framework, which involves recursively dividing the problem of listing all k-cliques in graph G into smaller sub-problems of listing smaller cliques in G through *branching* operations [11, 19, 36]. This process continues until each sub-problem can be trivially solved. The underlying principle behind these methods is the observation that a k-clique can be constructed by merging two smaller cliques: a clique S and an l-clique, where $|S| + l = k$. A branch B is represented as a triplet (S, g, l), where S denotes a previously found clique with $|S| < k$, g represents a subgraph induced by vertices that connect each vertex in S, and l corresponds to $k - |S|$. Essentially, branch B encompasses all k-cliques, each comprising of S and an l-clique in g. To enumerate all k-cliques within branch B, a set of sub-branches is created through branching operations. Each sub-branch expands the set S by adding one vertex from g, updates the graph g accordingly (by removing vertices that disconnect the newly added vertex from S), and decrements l by 1. This recursive process continues until l for a branch reduces to 2, at which point the l-cliques (corresponding to edges) can be trivially listed within g. The original k-clique listing problem on graph G can be solved by initiating the branch (S, g, l) with $S = \emptyset$, $g = G$, and $l = k$. The branching step employed in these existing methods is referred to as *vertex-oriented branching* since each sub-branch is formed by adding a vertex to the set S. We term the $\underline{\text{v}}$ertex-oriented branching $\underline{\text{BB}}$ framework for $\underline{k}$-$\underline{\text{c}}$lique listing as `VBBkC`.

Existing research has focused on leveraging vertex information within the graph g to enhance performance by generating sub-branches with smaller graphs and pruning sub-branches more effectively. Specifically, when a new vertex v_i is added, the resulting sub-branch B_i can reduce the graph g to a smaller size by considering only the neighbors of v_i (e.g., removing vertices that disconnect v_i). Consequently, branch B_i can be pruned if v_i has fewer than $(l-1)$ neighbors in g. Furthermore, by carefully specifying the ordering of vertices in g during the branching process, it becomes possible to generate a group of sub-branches with compact graphs. The state-of-the-art `VBBkC` algorithms [19, 36] achieve a time complexity of $O(km(\delta/2)^{k-2})$, where m represents the number of edges in the graph, and δ denotes the degeneracy of the graph.

In this paper, we propose to construct a branch by simultaneously including *two* vertices that connect to each other (i.e., an *edge*) from g into S. Note that these two vertices must be connected, as only then can they form a larger partial k-clique together with S. This strategy allows for the consideration of additional information (i.e., an edge or two vertices instead of a single vertex) to facilitate the formation of sub-branches with smaller graphs and pruning. When a new edge is added, the resulting branch can reduce the graph to one induced by the common neighbors of the two vertices, which is smaller compared to the graph generated by vertex-oriented branching. To achieve this, we explore an edge ordering technique based on truss decomposition [34], which we refer to as the *truss-based edge ordering*. This ordering aids in creating sub-branches with smaller graphs than those by vertex-oriented branching. Additionally, more branches can be pruned based on the information derived from the added edge. For instance, a branch can be pruned if the vertices within the newly added edge have fewer than $(l-2)$ common neighbors in g. We term this branching strategy as *edge-oriented branching*. Consequently, the $\underline{\text{e}}$dge-oriented branching-based $\underline{\text{BB}}$ framework for $\underline{k}$-$\underline{\text{c}}$lique listing is denoted as `EBBkC`. Our `EBBkC` algorithm, combined with the proposed edge ordering, exhibits a time complexity of $O(km(\tau/2)^{k-2})$, where τ represents the truss number[1] of the graph. We formally prove that $\tau < \delta$, signifying that our EBBkC algorithm possesses a time complexity that is strictly lower than that of the state-of-the-art VBBkC algorithms [19, 36].

It is important to note that although a single branching step in our edge-oriented branching (i.e., including an edge) can be seen as two branching steps in the existing vertex-oriented branching (i.e., including two vertices of an edge via two steps), there exists a significant distinction between our EBBkC and VBBkC frameworks. In EBBkC, we have the flexibility to explore *arbitrary* edge orderings for each branching step, whereas VBBkC is inherently *constrained* by the chosen vertex ordering. For instance, once a vertex ordering is established for vertex-oriented branching, with vertex v_i appearing before v_j, the edges incident to v_i would precede those incident to v_j in the corresponding edge-oriented branching. Consequently, the existing VBBkC framework is encompassed by our EBBkC framework. In other words, for any instance of VBBkC with a given vertex ordering, there exists an edge ordering such that the corresponding EBBkC instance is equivalent to the VBBkC instance, *but not vice versa*. This elucidates why EBBkC, when based on certain edge orderings, achieves a superior time complexity compared to VBBkC.

To further enhance the efficiency of BB frameworks, we develop an *early termination* technique, which is based on two key observations. Firstly, if the graph g within a branch (S, g, l) is either a clique or a 2-plex[2], we can efficiently list l-cliques in g using a combinatorial approach. For instance, in the case of a clique, we can directly enumerate all possible sets of l vertices in g. Secondly, if the graph g is a t-plex (with $t \geq 3$), the branching process based on g can be converted to a procedure conducted on its inverse graph, denoted as g_{inv}[3]. Since g is dense, g_{inv} would be sparse, and the converted procedure operates more rapidly. Therefore, during the recursive branching process, we can employ early termination at a branch (S, g, l) if g transforms into a t-plex, utilizing efficient algorithms to list l-cliques within g. We note that the early termination technique is applicable to all BB frameworks, including our EBBkC framework, without impacting the worst-case time complexity of the BB frameworks.

Contributions. We summarize our contributions as follows.

- We propose a new branch-and-bound framework for k-clique listing problem, namely EBBkC, which is based on an edge-oriented branching strategy. We further explore different edge orderings for EBBkC such that it achieves a strictly smaller time complexity than that of the state-of-the-art VBBkC based algorithms, i.e., the former is $O(km(\tau/2)^{k-2})$ and the latter is $O(km(\delta/2)^{k-2})$, where τ is a number related to the maximum truss number of the graph and δ is the degeneracy of the graph and we have $\tau < \delta$. (Section 4)
- We further develop an early termination technique for boosting the efficiency of branch-and-bound frameworks including EBBkC, i.e., for branches of listing l-cliques in a dense graph (e.g., a t-plex), we develop more efficient algorithms based on combinatorial approaches (for a clique and a 2-plex) and conduct the branching process on its inverse graph (for a t-plex with $t \geq 3$), which would be faster. (Section 5)
- We conduct extensive experiments on 16 real graphs, and the results show that our EBBkC based algorithms with the early termination technique consistently and largely outperform the state-of-the-art (VBBkC based) algorithms. (Section 6)

The rest of the paper is organized as follows. Section 2 reviews the problem and presents some preliminaries. Section 3 summarizes the existing vertex-oriented branching-based BB framework. Section 7 reviews the related work and Section 8 concludes the paper.

2 PROBLEM AND PRELIMINARIES

We consider an *unweighted* and *undirected* simple graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. We denote by $n = |V|$ and $m = |E|$ the cardinalities of V and E, respectively. Given $u, v \in V$, both (u, v) and (v, u) denote the undirected edge between u and v. Given $u \in V$, we denote by $N(u, G)$ the set of neighbors of u in G, i.e., $N(u, G) = \{v \in V \mid (u, v) \in E\}$ and define $d(u, G) = |N(u, G)|$. Given $V_{sub} \subseteq V$, we use $N(V_{sub}, G)$ to denote the common neighbors of vertices in V_{sub}, i.e., $N(V_{sub}, G) = \{v \in V \mid \forall u \in V_{sub},\ (v, u) \in E\}$.

Given $V_{sub} \subseteq V$, we denote by $G[V_{sub}]$ the subgraph of G induced by V_{sub}, i.e., $G[V_{sub}]$ includes the set of vertices V_{sub} and the set of edges $\{(u, v) \in E \mid u, v \in V_{sub}\}$. Given $E_{sub} \subseteq E$, we denote by $G[E_{sub}]$ the subgraph of G induced by E_{sub}, i.e., $G[E_{sub}]$ includes the set of edges E_{sub} and the set of vertices $\{v \in V \mid (v, \cdot) \in E_{sub}\}$. Let g be a subgraph of G induced by either a vertex subset of V or an edge subset of E. We denote by $V(g)$ and $E(g)$ its set of vertices and its set of edges, respectively.

In this paper, we focus on a widely-used cohesive graph structure, namely k-clique [12], which is defined formally as below.

DEFINITION 2.1 (k-CLIQUE [12]). *Given a positive integer k, a subgraph g is said to be a k-clique if and only if it has k vertices and has an edge between every pair of vertices, i.e., $|V(g)| = k$ and $E(g) = \{(u, v) \mid u, v \in V(g), u \neq v\}$.*

We note that 1-clique ($k = 1$) and 2-clique ($k = 2$) correspond to single vertex and single edge, respectively, which are basic elements of a graph. Therefore, we focus on those k-cliques with k at least 3 (note that 3-clique is widely known as triangle and has found many applications [18, 22]). We now formulate the problem studied in this paper as follows.

PROBLEM 2.1 (k-CLIQUE LISTING [9]). *Given a graph $G = (V, E)$ and a positive integer $k \geq 3$, the k-clique listing problem aims to find all k-cliques in G.*

Hardness. The k-clique listing problem is a hard problem since the decision problem of determining whether a graph contains a k-clique is NP-hard [17] and this problem can be solved by listing all k-cliques and returning true if any k-clique is listed.

Remark. The problem of listing all 3-cliques (i.e., $k = 3$), known as *triangle listing problem*, has been widely studied [18, 22]. There are many efficient algorithms proposed for triangle listing, which run in *polynomial* time. We remark that these algorithms cannot be used to solve the general k-clique listing problem.

[1]The maximum truss number of the graph defined in [34], denoted by $k_{\max}$, has the following relationship with τ: $k_{\max} = \tau + 2$.
[2]A t-plex is a graph where each vertex inside disconnects at most t vertices, including itself.
[3]The inverse graph g_{inv} has the same set of vertices as g, with an edge between two vertices in g_{inv} if and only if they are disconnected in g.

Algorithm 1: The vertex-oriented branching-based BB framework: VBBkC

```
  Input: A graph $G = (V, E)$ and an integer $k \geq 3$
  Output: All $k$-cliques within $G$
1 VBBkC_Rec($\emptyset, G, k$);
2 Procedure VBBkC_Rec($S, g, l$)
      /* Pruning */
3     if $|V(g)| < l$ then return;
      /* Termination when $l = 2$ */
4     if $l = 2$ then
5         for each edge $(u, v)$ in $g$ do Output a $k$-clique $S \cup \{u, v\}$;
6         return;
      /* Branching when $l \geq 3$ */
7     for each vertex $v_i \in V(g)$ based on a given vertex ordering do
8         Create branch $B_i = (S_i, g_i, l_i)$ based on Eq. (1);
9         VBBkC_Rec($S_i, g_i, l_i$);
```

3 THE BRANCH-AND-BOUND FRAMEWORK OF EXISTING ALGORITHMS: VBBKC

Many algorithms have been proposed for listing k-cliques in the literature [9, 11, 19, 36]. Most of them adopt a *branch-and-bound* (BB) framework, which recursively partitions the problem instance (of listing all k-cliques in G) into several sub-problem instances (of listing smaller cliques in G) via *branching* until each of them can be solved trivially [11, 19, 36]. The rationale behind these methods is that a k-clique can be constructed by merging two smaller cliques, namely a clique S and an l-clique with $|S| + l = k$. Specifically, a branch B can be represented as a triplet (S, g, l), where

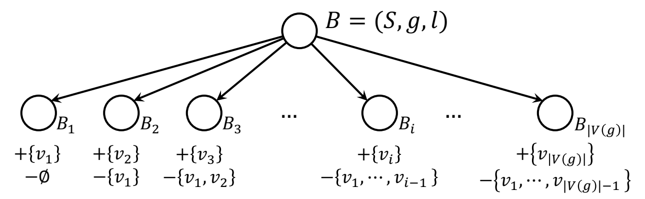
- **set** S induces a clique found so far with $|S| < k$,
- **subgraph** g is one induced by vertices that connect each vertex in S, and
- **integer** l is equal to $k - |S|$.

Essentially, branch B covers all k-cliques, each consisting of S and an l-clique in g. To list all k-cliques under branch B, it creates a group of sub-branches via a branching step such that for each sub-branch, the set S is expanded with one vertex from g, the graph g is updated accordingly (by removing those vertices that disconnect the vertex included in S), and l is decremented by 1. The recursive process continues until when the l for a branch reduces to 2, for which the l-cliques (which correspond to edges) can be listed trivially in g. The original k-clique listing problem on graph G can be solved by starting with the branch (S, g, l) with $S = \emptyset$, $g = G$ and $l = k$. We call the branching step involved in these existing methods *vertex-oriented branching* since each sub-branch is formed by including a *vertex* to the set S.

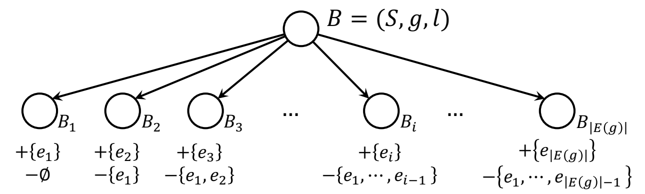
Consider the branching step at a branch $B = (S, g, l)$. Let $\langle v_1, v_2, \cdots, v_{|V(g)|}\rangle$ be an arbitrary ordering of vertices in g. The branching step would produce $|V(g)|$ new sub-branches from branch B. The i-th sub-branch, denoted by $B_i = (S_i, g_i, l_i)$, includes v_i to S and excludes $\{v_1, v_2, \cdots, v_{i-1}\}$ (and also those that disconnect v_i) from g. Formally, for $1 \leq i \leq |V(g)|$, we have

$$S_i = S \cup \{v_i\}, \quad g_i = \widehat{g}_i[N(v_i, \widehat{g}_i)], \quad l_i = l - 1, \tag{1}$$

where $\widehat{g}_i$ is a subgraph of g induced by the set of vertices $\{v_i, v_{i+1}, \cdots, v_{|V(g)|}\}$, i.e., $\widehat{g}_i = g[v_i, \cdots, v_{|V(g)|}]$. The branching can be explained by a recursive binary partition process, as shown in Figure 1(a). Specifically, it first divides the current branch B into two sub-branches based on v_1: one branch moves v_1 from g to S (this is the branch B_1 which will list those k-cliques in B that include v_1), and the other removes v_1 from g (so that it will list others in B that exclude v_1). For branch B_1, it also removes from g those vertices that disconnect to v_1 since they cannot form any k-clique with v_1. In summary, we have $g_1 = \widehat{g}_1[N(v_1, \widehat{g}_1)]$. Then, it recursively divides the latter into two new sub-branches: one branch moves v_2 from $\widehat{g}_2$ to S (this is the branch B_2 which will list those k-cliques in B that exclude v_1 and include v_2), and the other removes v_2 from $\widehat{g}_2$ (so that it will list others in B that exclude $\{v_1, v_2\}$). It continues the process, until the last branch $B_{|V(g)|}$ is formed. In summary, branch B_i will list those k-cliques in B that include v_i and exclude $\{v_1, \cdots, v_{i-1}\}$. Consequently, all k-cliques in B will be listed exactly once after branching.


$B = (S, g, l)$
B_1
B_2
B_3
B_i
$B_{|V(g)|}$
$+\{v_1\}$ $-\emptyset$
$+\{v_2\}$ $-\{v_1\}$
$+\{v_3\}$ $-\{v_1, v_2\}$
$+\{v_i\}$ $-\{v_1, \cdots, v_{i-1}\}$
$+\{v_{|V(g)|}\}$ $-\{v_1, \cdots, v_{|V(g)|-1}\}$

(a) Branching in VBBkC framework. The notation "+" means to include a vertex by adding it S and "−" means to exclude a vertex by removing it from the graph g.


$B = (S, g, l)$
B_1
B_2
B_3
B_i
$B_{|E(g)|}$
$+\{e_1\}$ $-\emptyset$
$+\{e_2\}$ $-\{e_1\}$
$+\{e_3\}$ $-\{e_1, e_2\}$
$+\{e_i\}$ $-\{e_1, \cdots, e_{i-1}\}$
$+\{e_{|E(g)|}\}$ $-\{e_1, \cdots, e_{|E(g)|-1}\}$

(b) Branching in EBBkC framework. The notation "+" means to include two vertices incident to an edge by adding them to S and "−" means to exclude an edge by removing it from the graph g.

Figure 1: Illustration of VBBkC and EBBkC.

We call the vertex-oriented branching-based BB framework for k-clique listing VBBkC. We present its pseudo-code in Algorithm 1. In particular, when $l = 2$, a branch (S, g, l) can be terminated by listing each of edges in g together with S (lines 4-6). We remark that Algorithm 1 presents the algorithmic idea only while the detailed implementations (e.g., the data structures used for representing a branch) often vary in existing algorithms [11, 19, 36]. Different variants of VBBkC have different time complexities. The state-of-the-art algorithms, including DDegCol [19], DDegree [19], BitCol [36] and SDegree [36], all share the time complexity of $O(km(\delta/2)^{k-2})$, where δ is the degeneracy of the graph. Some more details of variants of VBBkC will be provided in the related work (Section 7).

4 A NEW BRANCH-AND-BOUND FRAMEWORK: EBBKC

4.1 Motivation and Overview of EBBkC

Recall that for a branch $B = (S, g, l)$, the vertex-oriented branching forms a sub-branch by moving one vertex v_i from g to S. To improve the performance, existing studies consider the information of each

Algorithm 2: The edge-oriented branching-based BB framework: EBBkC

```
Input: A graph $G = (V, E)$ and an integer $k \geq 3$
Output: All $k$-cliques within $G$
1  EBBkC_Rec($\emptyset, G, k$);
2  Procedure EBBkC_Rec($S, g, l$)
       /* Pruning */
3      if $|V(g)| < l$ then return;
       /* Termination when $l = 1$ or $l = 2$ */
4      if $l = 1$ then
5          for each vertex $v$ in $g$ do Output a $k$-clique $S \cup \{v\}$;
6          return;
7      else if $l = 2$ then
8          for each edge $(u, v)$ in $g$ do Output a $k$-clique $S \cup \{u, v\}$;
9          return;
       /* Branching when $l \geq 3$ */
10     for each edge $e_i \in E(g)$ based on a given edge ordering do
11         Create branch $B_i = (S_i, g_i, l_i)$ based on Eq. (2);
12         EBBkC_Rec($S_i, g_i, l_i$);
```

vertex in g towards pruning more sub-branches and/or forming sub-branches with smaller graph instances. Specifically, with the newly added vertex v_i, the produced sub-branch B_i can shrink the graph instance g as a smaller one induced by the neighbors of v_i (e.g., removing those vertices that disconnect v_i). As a result, one can prune the branch B_i if v_i has less than $(l-1)$ neighbors in g. In addition, one can produce a group of sub-branches with small graph instances by specifying the ordering of vertices in g for branching.

In this paper, we propose to form a branch by *moving two vertices that connect with each other (correspondingly, an edge) from g to S at the same time*. Note that the two vertices are required to be connected since otherwise they will not form a larger partial k-clique with S. The intuition is that it would allow us to consider more information (i.e., an edge or two vertices instead of a single vertex) towards pruning more sub-branches and/or forming sub-branches with smaller graph instances. Specifically, with the newly added edge, the produced branch can shrink the graph instance as the one induced by the common neighbors of two vertices of the edge, which is smaller than that produced by the vertex-oriented branching (details can be found in Section 4.2). In addition, we can prune more branches based on the information of the added edge, e.g., a produced branch can be pruned if the vertices in the newly added edge have less than $(l-2)$ common neighbors in g (details can be found in Section 4.3). We call the above branching strategy *edge-oriented branching*, which we introduce as follows.

Consider the branching step at a branch $B = (S, g, l)$. Let $\langle e_1, e_2, \cdots, e_{|E(g)|} \rangle$ be an *arbitrary* ordering of edges in g. Then, the branching step would produce $|E(g)|$ new sub-branches from B. The i-th branch, denoted by $B_i = (S_i, g_i, l_i)$, includes to S the two vertices incident to the edge e_i, i.e., $V(e_i)$, and excludes from g those edges in $\{e_1, e_2, \cdots, e_{i-1}\}$ (and those vertices that disconnect the vertices incident to e_i). Formally, for $1 \leq i \leq |E(g)|$, we have

$$S_i = S \cup V(e_i), \quad g_i = \overline{g}_i[N(V(e_i), \overline{g}_i)], \quad l_i = l - 2, \tag{2}$$

where $\overline{g}_i$ is a subgraph of g induced by the set of edges $\{e_i, e_{i+1}, \cdots, e_{|E(g)|}\}$, i.e., $\overline{g}_i = g[e_i, \cdots, e_{|E(g)|}]$. We note that (1) $N(V(e_i), \overline{g}_i)$ is to filter out those vertices that are disconnected with the two vertices incident to e_i since they cannot form any k-cliques with e_i and (2) $\overline{g}_i[N(V(e_i), \overline{g}_i)]$ is the graph instance for the sub-branch induced by the vertex set $N(V(e_i), \overline{g}_i)$ in $\overline{g}_i$.

The edge-oriented branching also corresponds to a recursive binary partition process, as illustrated in Figure 1(b). Specifically, it first divides branch B into two sub-branches based on e_1: one moves e_1 from g to S (this is the branch B_1 which will list those k-cliques in B that include edge e_1), and the other removes e_1 from g (so that it will list others that exclude e_1). For branch B_1, it also removes from g those vertices that disconnect to one vertex in $V(e_1)$ (i.e., $g_1 = \overline{g}_1[N(V(e_1), \overline{g}_1)]$) since they cannot form any k-clique with e_1. Then, it recursively divides the latter into two new sub-branches: one moves e_2 from $\overline{g}_2$ to S (this the branch B_2 which will list those k-cliques in B that exclude e_1 and include e_2), and the other removes e_2 from $\overline{g}_2$ (so that it will list others that exclude $\{e_1, e_2\}$). It continues the process, until the last branch $B_{|E(g)|}$ is formed.

We call the *e*dge-oriented *b*ranching-based *BB* framework for *k*-*c*lique listing EBBkC. We present in Algorithm 2 the pseudo-code of EBBkC, which differs with VBBkC mainly in the branching step (lines 10-11). We note that while a branching step in our edge-oriented branching (i.e., including an edge) can be treated as two branching steps in the existing vertex-oriented branching (i.e., including two vertices of an edge via two steps), our EBBkC has a major difference from VBBkC as follows. For the former, we can explore *arbitrary* edge orderings for each branching step while for the latter, the underlying edge orderings are *constrained* by the adopted vertex ordering. For example, once a vertex ordering is decided for vertex-oriented branching with vertex v_i appearing before v_j, then the edges that are incident to v_i would appear before those that are incident to v_j for the corresponding edge-oriented branching. For this reason, the existing VBBkC framework is covered by our EBBkC framework, i.e., for any instance of VBBkC with a vertex ordering, there exists an edge ordering such that the corresponding EBBkC instance is equivalent to the VBBkC instance, *but not vice versa*. This explains why the time complexity of EBBkC based on some edge ordering is better than that of VBBkC (details can be found in Section 4.2).

In the sequel, we explore different orderings of edges in EBBkC. Specifically, with the proposed truss-based edge ordering, EBBkC would have the worst-case time complexity of $O(km(\tau/2)^{k-2})$ with $\tau < \delta$, i.e., the time complexity is strictly better than that of the state-of-the-art VBBkC algorithms (which is $O(km(\delta/2)^{k-2})$) (Section 4.2). With the proposed color-based edge ordering, EBBkC can apply some pruning rules to improve the efficiency in practice (Section 4.3). Finally, with the proposed hybrid edge ordering, EBBkC would inherit both the above theoretical result and practical performance (Section 4.4).

4.2 EBBkC-T: EBBkC with the Truss-based Edge Ordering

Consider the edge-oriented branching at $B = (S, g, l)$ based on an ordering of edges $\langle e_1, e_2, \cdots, e_{|E(g)|} \rangle$. For a sub-branch B_i ($1 \leq i \leq |E(g)|$) produced from B, we observe that the size of graph

Algorithm 3: EBBkC with truss-based edge ordering: EBBkC-T

```
Input: A graph $G = (V, E)$ and an integer $k \geq 3$
Output: All $k$-cliques within $G$
/* Initialization and branching at $(\emptyset, G, k)$ */
1  $\pi_\tau(G) \leftarrow$ the truss-based ordering of edges in $G$;
2  for each edge $e_i \in E(G)$ following $\pi_\tau(G)$ do
3      Obtain $S_i$ and $g_i$ according to Eq. (2);
4      Initialize $V[e_i] \leftarrow V(g_i)$ and $E[e_i] \leftarrow E(g_i)$;
5      EBBkC-T_Rec $(S_i, g_i, k-2)$;
6  Procedure EBBkC-T_Rec $(S, g, l)$
7      Conduct Pruning and Termination (lines 3-9 of Algorithm 2);
       /* Branching when $l \geq 3$ */
8      for each edge $e$ in $E(g)$ do
9          $S' \leftarrow S \cup V(e)$ and $g' \leftarrow (V(g) \cap V[e], E(g) \cap E[e])$;
10         EBBkC-T_Rec $(S', g', l-2)$;
```

instance g_i (i.e., the number of vertices in g_i) is equal to the number of common neighbors of vertices in $V(e_i)$ in $\overline{g}_i$, which depends on the ordering of edges. Formally, we have

$$|V(g_i)| = |N(V(e_i), \overline{g}_i)|, \text{ where } \overline{g}_i = g[e_i, \cdots, e_{|E(g)|}]. \quad (3)$$

Recall that the smaller a graph instance is, the faster the corresponding branch can be solved. Therefore, to reduce the time costs, we aim to minimize the sizes of graph instances by determining the ordering of edges via the following greedy procedure.

Truss-based edge ordering. We determine the ordering of edges by an iterative process. Specifically, it iteratively removes from g the edge with the smallest number of common numbers (correspondingly, the smallest size of a graph instance), and then adds it to the end of the ordering. Consequently, for the i-th edge e_i in the produced ordering ($1 \leq i \leq |E(g)|$), we have

$$e_i = \min_{e \in E(g) \setminus \{e_1, \cdots, e_{i-1}\}} |N(V(e_i), g[E(g) \setminus \{e_1, \cdots, e_{i-1}\}])|. \quad (4)$$

We call the above ordering *truss-based edge ordering* of g and denote it by $\pi_\tau(g)$ since the corresponding iterative process is the same to the *truss decomposition* [8, 10, 34], which can be done in $O(|E(g)|^{1.5})$ time.

The EBBkC-T algorithm. When the truss-based edge ordering is adopted in EBBkC, we call the resulting algorithm EBBkC-T. The pseudo-code of EBBkC-T is presented in Algorithm 3. In particular, it only computes the truss-based edge ordering of G (i.e., $\pi_\tau(G)$) for the branching at the initial branch $(\emptyset, G, k)$ (lines 2-7). Then, for any other branching step at a following branch (S, g, l), edges in $\langle e_1, e_2, \cdots, e_{|E(g)|}\rangle$ adopt the same ordering to those used in $\pi_\tau(G)$, which could differ with the truss-based edge ordering of g. Formally, e_i comes before e_j in $\langle e_1, e_2, \cdots, e_{|E(g)|}\rangle$ (i.e., $i < j$) if and only if it does so in $\pi_\tau(G)$. To implement this efficiently, it maintains two additional auxiliary sets, i.e., $V[\cdot]$ and $E[\cdot]$ (lines 2-4). The idea is that for an edge e, all edges in $E[e]$ are ordered behind e in π_τ and the vertices incident to these edges are both connected with those incident to e. Therefore, the branching steps (with the introduced edge ordering) can be efficiently conducted via set intersections (line 9) for those branches following $(\emptyset, G, k)$. The correctness of this implementation can be easily verified.

Time complexity. Given a branch $B = (S, g, l)$, let $\tau(g)$ be the largest size of a produced graph instance, i.e.,

$$\tau(g) = \max_{e_i \in E(g)} |V(g_i)|. \quad (5)$$

We have the following observation.

Lemma 4.1. *When applying the truss-based edge ordering $\pi_\tau(g)$ at (S, g, l), we have $\tau(g) < \delta(g)$, where $\delta(g)$ is the degeneracy of g.*

Proof. We prove by contradiction. Suppose that $\tau \geq \delta$. Since δ is defined as the largest value of k such that the k-core of g is non-empty, there must not have a $(\delta+1)$-core in g, i.e., $V(C_{\delta+1}) = \emptyset$. Here, C_k refers to a k-core. Consider the branching step at such an edge $e_i \in E(g)$ that produces the largest size of graph instance, i.e., $|V(g_i)| = \tau$. According to Eq. (4), e_i has the minimum number of common neighbors of its end points in the graph $\overline{g}_i = g[e_i, \cdots, e_{|E(g)|}]$. This means for each edge $e \in E(\overline{g}_i)$, the number of common neighbors of the end points of e is no less than τ, i.e., $|N(V(e), \overline{g}_i)| \geq \tau$, Obviously, $\overline{g}_i$ is non-empty, i.e., $V(\overline{g}_i) \neq \emptyset$. Then for each vertex $v \in V(\overline{g}_i)$, the number of its neighbors is at least $\tau+1$, i.e., $|N(v, \overline{g}_i)| \geq \tau+1$. Therefore, $\overline{g}_i$ is a subgraph of $(\tau+1)$-core, i.e., $V(\overline{g}_i) \subseteq V(C_{\tau+1})$. According to the hereditary property of k-core[4] and the hypothesis $\tau \geq \delta$, we have $V(C_{\tau+1}) \subseteq V(C_{\delta+1})$, which leads to a contradiction that $V(\overline{g}_i) \subseteq V(C_{\tau+1}) \subseteq V(C_{\delta+1}) = \emptyset$. □

Based on the above result, we derive that the time complexity of EBBkC-T is strictly smaller than that of the state-of-the-art algorithms, i.e., $O(km(\delta/2)^{k-2})$, which we show in the following theorem.

Theorem 4.2. *Given a graph $G = (V, E)$ and an integer $k \geq 3$, the time complexity of EBBkC-T is $O(km(\tau/2)^{k-2})$ where $\tau = \tau(G)$ is strictly smaller than the degeneracy δ of G.*

Proof. We give a sketch of the proof and put the details in the technical report [3]. The running time of EBBkC-T is dominated by the recursive listing procedure (lines 6-10 of Algorithm 3). Given a branch $B = (S, g, l)$, we denote by $T(g, l)$ the upper bound of time cost of listing l-cliques under such a branch. When $k \geq 3$, with different values of l, we have the following recurrences.

$$T(g, l) \leq \begin{cases} O(k \cdot |V(g)|) & l = 1 \\ O(k \cdot |E(g)|) & l = 2 \\ \sum_{e \in E(g)} \left(T(g', l-2) + T'(g')\right) & 3 \leq l \leq k-2 \end{cases} \quad (6)$$

where $T'(g')$ is the time for constructing g' given $B = (S, g, l)$ (line 9 of Algorithm 3). We show that with different values of l, $T'(g')$ satisfies the following equation.

$$\sum_{e \in E(g)} T'(g') = \begin{cases} O(\tau \cdot |E(g)|) & l = 3 \\ O(\tau^2 \cdot |E(g)|) & l > 3 \end{cases} \quad (7)$$

The reason is as follows. When given a branch $B = (S, g, l)$ with $l = 3$, for each edge, we just need to compute $V(g')$ for the sub-branch (since the termination when $l = 1$ only cares about the vertices in g'), which can be done in $O(\tau)$. When given a branch $B = (S, g, l)$

[4]The hereditary property claims that given a graph G and two integers k and k' with $k \leq k'$, then the k'-core of G is a subgraph of the k-core of G [4].

Algorithm 4: `EBBkC` with color-based edge ordering: `EBBkC-C`

```
  Input: A graph $G = (V, E)$ and an integer $k \geq 3$
  Output: All $k$-cliques within $G$
1 Conduct vertex coloring on $G$ and get $id(v)$ for each vertex in $V$;
2 $\overrightarrow{G} \leftarrow (V, \overrightarrow{E})$ where $\overrightarrow{E} = \{u \rightarrow v \mid (u, v) \in E \wedge id(u) < id(v)\}$;
3 EBBkC-C_Rec ($\emptyset, \overrightarrow{G}, k$);
4 Procedure EBBkC-C_Rec ($S, \overrightarrow{g}, l$)
5     Conduct Pruning and Termination (line 3-9 of Algorithm 2);
      /* Branching when $l \geq 3$ */
6     for each edge $u \rightarrow v$ in $E(\overrightarrow{g})$ do
7         $S' \leftarrow S \cup \{u, v\}$ and $\overrightarrow{g}' \leftarrow \overrightarrow{g}[N^+(\{u, v\}, \overrightarrow{g})]$;
8         if either of the rules of pruning applies then continue;
9         EBBkC-C_Rec ($S', \overrightarrow{g}', l - 2$);
```

with $l > 3$, for each edge, we need to construct both $V(g')$ and $E(g')$, which can be done in $O(\tau^2)$ since there are at most $\tau(\tau-1)/2$ edges in g. Besides, we show that given a branch $B = (S, g, l)$ and the sub-branches $B' = (S', g', l')$ produced at B, we have

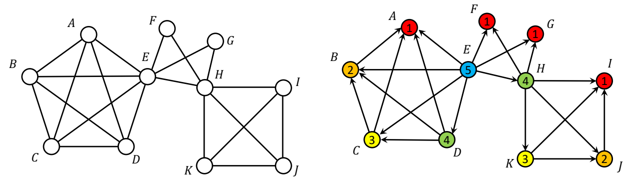
$$\sum_{e \in E(g)} |E(g')| < \begin{cases} \frac{\tau^2}{4} \cdot |E(g)| & l < k \\ \frac{\tau^2}{2} \cdot |E(g)| & l = k \end{cases} \tag{8}$$

This inequality is proven by a lemma in the technical report [3]. With the above three inequalities, we can prove the theorem by induction on l. □

Remark. (1) We note that another option of designing `EBBkC-T` is to compute the truss-based edge ordering for each individual branch and use it for branching at the branch. However, it would introduce additional time cost without achieving better theoretical time complexity. Thus, we choose not to adopt this option. (2) It is worthy noting that for a truss-based edge ordering, there does not always exist a vertex ordering such that the instance of `VBBkC` with the vertex ordering is equivalent to the instance of `EBBkC-T`. We include a counter-example in the technical report [3] for illustration.

4.3 `EBBkC-C`: `EBBkC` with the Color-based Edge Ordering

While the truss-based edge ordering helps to form sub-branches with small sizes at a branch, it does not offer much power to prune the formed sub-branches - all we can leverage for pruning are some size constraints (line 3 of Algorithm 2). On the other hand, some existing studies of `VBBkC` have successfully adopted color-based *vertex* ordering for effective pruning [15, 35]. Specifically, consider a branch $B = (S, g, l)$. They first color the vertices in g by iteratively assigning to an uncolored vertex v the smallest color value taken from $\{1, 2, \cdots\}$ that has not been assigned to v's neighbours. Let c be the number of color values used by the coloring procedure. They then obtain a vertex orderng by sorting the vertices in a non-increasing order based on the color values $\langle v_1, v_2, \cdots, v_{|V(g)|}\rangle$ (with ties broken by node ID), i.e., for v_i and v_j with $i < j$, we have $col(v_i) \geq col(v_j)$, where $col(\cdot)$ is the color value of a vertex. As a result, they prune the sub-branch $B_i = (S_i, g_i, l_i)$, which includes v_i to S, if $col(v_i) < l$. The rationale is that since all vertices in g_i have



(a) An example graph G.
(b) The DAG built upon G (the numbers inside circles are the color values).

Figure 2: Color-based edge ordering and pruning rules.

their color values strictly smaller than $col(v_i)$ (according to Eq. (1) and the definition of the color-based vertex ordering), they do not have $l-1$ different color values, indicating g_i does not contain any $(l-1)$-cliques, and therefore B_i can be pruned.

Recall that in Section 4.1, for an instance of `VBBkC` with a vertex ordering, there would exist an edge ordering such that the corresponding `EBBkC` instance based on the edge ordering is equivalent to the `VBBkC` instance. Motivated by this, we propose to adopt the edge ordering that corresponds to the color-based vertex ordering, which we call *color-based edge ordering*, for our `EBBkC`. One immediate benefit is that it would naturally inherit the pruning power of the color-based vertex ordering, which has been demonstrated for `VBBkC` [15, 35]. Furthermore, it would introduce new opportunities for pruning, compared with existing `VBBkC` with the color-based vertex ordering, since it can leverage the two vertices incident to an edge collectively (instead of a single vertex twice as in `VBBkC`) for designing new pruning rules, which we explain next.

Color-based edge ordering and pruning rules. Consider the branch $B = (S, g, l)$. We first color the graph g using the graph coloring technique [15, 35] and obtain the color-based vertex ordering $\langle v_1, v_2, \cdots, v_{|V(g)|}\rangle$ such that $col(v_i) \geq col(v_j)$ for $i < j$. Then for each vertex $u \in V(g)$, let $id(u)$ be the position of a vertex u in the ordering. For each edge $e = (u, v) \in E(g)$ with $id(u) < id(v)$, we define $str(e)$ as a string, which is the concatenation of $id(u)$ and $id(v)$ (i.e., $str(e)$ ='$id(u)$+$id(v)$'). Finally, we define the color-based edge ordering as the alphabetical ordering based on $str(e)$ for edges $e \in E(g)$. That is, one edge $e = (u, v)$ with $id(u) < id(v)$ comes before another edge $e' = (u', v')$ with $id(u') < id(v')$ if (1) $id(u) < id(u')$ or (2) $id(u) = id(u')$ and $id(v) < id(v')$.

Consider a branch $B = (S, g, l)$ and and a sub-branch B_i that includes edge $e_i = (u, v)$ with $col(u) > col(v)$ to S. We can apply the following two pruning rules.

- **Rule (1).** If $col(u) < l$ or $col(v) < l - 1$, we prune sub-branch B_i;
- **Rule (2).** If the vertices in produced sub-branch have less than $l-2$ distinct color values, we prune sub-branch B_i.

We note that Rule (1) is equivalent to the pruning that `VBBkC` with the color-based vertex ordering conducts at two branching steps of including u and v [19]. Rule (2) is a new one, which applies only in our `EBBkC` framework with the color-based edge ordering. In addition, Rule (2) is more powerful than Rule (1) in the sense that if Rule (2) applies, then Rule (1) applies, but not vice versa. The reason is that the color values of u and v are sometimes much larger than the number of distinct color values in the sub-branch since both color values consider the information of their own neighbors instead of their common neighbors. For illustration, consider the example in Figure 2 and assume that we aim to list 4-cliques (i.e., $k = 4$). We focus on the edge EH. It is easy to check that Rule (1) does not apply, but Rule (2) applies since the vertices in the produced sub-branch, i.e., F and G, have only one color value. Given that it takes $O(1)$ time to check if Rule (1) applies and $O(|V(g_i)|)$ time to check if Rule (2) applies, our strategy is to check Rule (1) first, and if it does not apply, we further check Rule (2).

The `EBBkC-C` algorithm. When the color-based edge ordering is adopted in `EBBkC`, we call the resulting algorithm `EBBkC-C`. The pseudo-code of `EBBkC-C` is presented in Algorithm 4. With the color-based edge ordering, a directed acyclic graph (DAG), denoted by $\overrightarrow{G}$, is built for efficiently conducting the branching steps [19, 36]. Specifically, $\overrightarrow{G}$ is built upon G by orienting each edge (u, v) in E with $id(u) < id(v)$ from u to v (line 2). For illustration, consider Figure 2. Given $V_{sub} \subseteq V$, let $N^+(V_{sub}, \overrightarrow{g})$ be the common out-neighbours of the vertices in V_{sub} in $\overrightarrow{g}$. Then, given a branch $B = (S, \overrightarrow{g}, l)$[5], the edge-oriented branching at a branch with the color-based ordering can be easily conducted by calculating the common out-neighbors of the vertices incident to an edge in $\overrightarrow{g}$ (line 7). Then, we will prune the produced branch if either of the above two rules applies (line 8).

Time complexity. The time cost of `EBBkC-C` is $O(km(\Delta/2)^{k-2})$, which is worse than that of `EBBkC-T` (i.e., $O(km(\tau/2)^{k-2})$) though `EBBkC-C` runs faster in practice.

THEOREM 4.3. *Given a graph $G = (V, E)$ and an integer $k \geq 3$, the time complexity of `EBBkC-C` is $O(km(\Delta/2)^{k-2})$ where Δ is the maximum degree of G.*

PROOF. The proof is similar to that of Theorem 4.2. The difference is that the largest size of the produced graph instance in `EBBkC-C` can only be bounded by Δ. □

4.4 `EBBkC-H`: `EBBkC` with Hybrid Edge Ordering

Among `EBBkC-T` and `EBBkC-C`, the former has a better theoretical time complexity and the latter enables effective pruning in practice. We aim to achieve the merits of both algorithms by adopting both the truss-based edge ordering (used by `EBBkC-T`) and the color-based edge ordering (used by `EBBkC-C`) in the `EBBkC` framework. Specifically, we first apply the truss-based edge ordering for the branching step at the initial branch $(\emptyset, G, k)$. Then, for the following branches, we adopt the color-based ordering for their branching steps. We call this algorithm based on the hybrid edge ordering `EBBkC-H`. The pseudo-code of `EBBkC-H` is presented in Algorithm 5 and the implementations of branching steps are similar to those in `EBBkC-T` and `EBBkC-C`. The size of a produced problem instance for `EBBkC-H` is bounded by τ (due to the branching at the initial branch based on the truss-based edge ordering), and thus `EBBkC-H` achieves the same time complexity as `EBBkC-T`. In addition, `EBBkC-H` enables effective pruning for all branches except for the initial branch (since color-based edge coloring is adopted at these branches), and thus it runs fast in practice as `EBBkC-C` does.

Time complexity The time complexity of `EBBkC-H` is $O(km(\tau/2)^{k-2})$, which is the same as that of `EBBkC-T`.

THEOREM 4.4. *Given a graph $G = (V, E)$ and an integer $k \geq 3$, the time complexity of `EBBkC-H` is $O(km(\tau/2)^{k-2})$, where $\tau = \tau(G)$ is strictly smaller than the degeneracy δ of G.*

[5]Note that $\overrightarrow{g}$ is a subgraph of $\overrightarrow{G}$, whose edge orientations are the same as those of $\overrightarrow{G}$.

Algorithm 5: `EBBkC` with hybrid edge ordering: `EBBkC-H`

```
Input: A graph $G = (V, E)$ and an integer $k \geq 3$
Output: All $k$-cliques within $G$
/* Initialization and branching at $(\emptyset, G, k)$ */
1 $\pi_\tau(G) \leftarrow$ the truss-based ordering of edges in $G$;
2 for each edge $e_i \in E(G)$ following $\pi_\tau(G)$ do
3     Obtain $S_i$ and $g_i$ according to Eq. (2);
4     Do vertex coloring on $g_i$ and get $id(v)$ for each vertex in $V(g_i)$;
5     $\overrightarrow{g_i} \leftarrow (V(g_i), \{u \rightarrow v \mid (u, v) \in E(g_i) \wedge id(u) < id(v)\})$;
6     EBBkC-C_Rec $(S_i, \overrightarrow{g_i}, k-2)$;
```

PROOF. The proof is similar to that of Theorem 4.2. Since the largest size of the produced graph instance in `EBBkC-H` can also be bounded by τ, it has the same worst-case time complexity as that of `EBBkC-T`. □

5 EARLY TERMINATION TECHNIQUE

Suppose that we are at a branch $B = (S, g, l)$, where graph g is dense (e.g., g is a clique or nearly a clique), and the goal is to list l-cliques in g (and merge them with S). Based on the `EBBkC` framework, we would conduct branching at branch B and form sub-branches. Nevertheless, since g is dense, there would be many sub-branches to be formed (recall that we form $|E(g)|$ sub-branches), which would be costly. Fortunately, for such a branch, we can list the l-cliques efficiently without continuing the recursive branching process of `EBBkC`, i.e., we can *early terminate* the branching process. Specifically, we have the following two observations.

- If g is a clique or a 2-plex (recall that a t-plex is a graph where each vertex inside disconnects at most t vertices including itself), we can list l-cliques in g efficiently in a combinatorial manner. For the former case, we can directly enumerate all possible sets of l vertices in g. For the latter case, we can do similarly, but in a bit more complex manner (details will be discussed in Section 5.1).
- If g a t-plex (with $t \geq 3$), the branching procedure based on g can be converted to that on its inverse graph g_{inv} (recall that g_{inv} has the same set of vertices as g, with an edge between two vertices in g_{inv} if and only if they are disconnected in g), which is sparse, and the converted procedure would run faster (details will be discussed in Section 5.2).

5.1 Listing k-Cliques from 2-Plex in Nearly Optimal Time

Consider a branch $B = (S, g, l)$ with g as a 2-plex. The procedure for listing k-cliques inside, called `kC2Plex`, utilizes the *combinatorial technique*. The rationale behind is that listing k-cliques from a large clique can be solved in the optimal time by directly enumerating all possible combinations of k vertices.

Specifically, we first partition $V(g)$ into three disjoint sets, namely F, L and R, each of which induces a clique. This can be done in two steps. First, it partitions $V(g)$ into two disjoint parts: one containing those vertices that disconnect itself only (this is F) and the other containing the remaining vertices that disconnect two vertices including itself (this is $L \cup R$). Note that $L \cup R$ always involves an even number of vertices and can be regarded as a collection of pairs of vertices $\{u, v\}$ such that u disconnects v. Second, it further partitions

Algorithm 6: List k-cliques in a 2-plex: `kC2Plex`

```
  Input: A branch $(S, g, l)$ with $g$ corresponding to a 2-plex
  Output: All $k$-cliques within $(S, g, l)$
1 Partition $V(g)$ into three disjoint sets $F$, $L$ and $R$;
2 if $|F| + |L| < l$ then return;
3 for $c_1 \in [\max\{0, l - |L|\}, \min\{l, |F|\}]$ and each $c_1$-combination $F_{sub}$ over $F$ do
4     for $c_2 \in [0, \min\{l - c_1, |L|\}]$ and each $c_2$-combination $L_{sub}$ over $L$ do
5         for $c_3 \leftarrow l - c_1 - c_2$ and each $c_3$-combination $R_{sub}$ over $R \setminus \overline{N}(L_{sub}, g)$ do
6             Output a $k$-clique $S \cup F_{sub} \cup L_{sub} \cup R_{sub}$;
```

$L \cup R$ into two parts by breaking each pair in $L \cup R$, that is, L and R contain the first and the second vertex in each pair, respectively. As a result, every vertex in one set connects all others within the same set while disconnecting one vertex from the other set. Note that the partition of $L \cup R$ is not unique and can be an arbitrary one. Below, we elaborate on how the partition $V(g) = F \cup L \cup R$ helps to speedup the k-clique listing.

Recall that the set of k-cliques in B can be listed by finding all l-cliques in g and merging each of them with S. Consider a l-clique in g. Based on the partition $V(g) = F \cup L \cup R$, it consists of three disjoint subsets of F, L and R, namely F_{sub}, L_{sub} and R_{sub}, each of which induces a small clique. Therefore, all l-cliques with the form of $F_{sub} \cup L_{sub} \cup R_{sub}$ can be found by iteratively enumerating all possible $|F_{sub}|$-combinations over F, $|L_{sub}|$-combinations over L and $|R_{sub}|$-combinations over R such that $|F_{sub}| + |L_{sub}| + |R_{sub}| = l$.

We present the pseudo-code of `kC2Plex` in Algorithm 6. In particular, the integers c_1, c_2 and c_3 are used to ensure the satisfaction of $|F_{sub}| + |L_{sub}| + |R_{sub}| = l$. Specifically, it first finds a c_1-clique F_{sub} from F and a c_2-clique L_{sub} from L. Recall that every vertex in L disconnects one vertex in R and vice versa. Hence, it removes from R those vertices that disconnect to one vertex in L_{sub}, which we denote by $\overline{N}(L_{sub}, g)$, and this can be done efficiently in $\Theta(|L_{sub}|)$ (as verified by Theorem 5.1), and then finds a c_3-clique from $R \setminus \overline{N}(L_{sub}, g)$. Besides, when $|F| + |L| < l$, it terminates the procedure since no l-clique will be found in g (line 2).

Time complexity. We analyze the time complexity of `kC2Plex` as follows.

THEOREM 5.1. *Given a branch $B = (S, g, l)$ with g being a 2-plex, `kC2Plex` lists all k-cliques within B in $O(|E(g)| + k \cdot c(g, l))$ time where $c(g, l)$ is the number of l-cliques in g.*

PROOF. Algorithm 6 takes $O(|E(g)|)$ for partitioning $V(g)$ by obtaining the degree of each vertex inside (line 1). For each round of lines 3-6, the algorithm can guarantee exactly one k-clique to be outputted at line 6 based on the settings of c_1, c_2 and c_3. Besides, the operation $R \setminus \overline{N}(L_{sub}, g)$ only takes $\Theta(|L_{sub}|)$ time. Specifically, we (1) maintain two arrays $L = \{u_1, u_2, \cdots\}$ and $R = \{v_1, v_2, \cdots\}$ such that u_i disconnects to v_i for $1 \le i \le |L|$, and (2) reorder R by switching $|L_{sub}|$ vertices with the same indices as those in L_{sub} to the tail of R which runs in $\Theta(|L_{sub}|)$ time, and take the first $|R| - |L_{sub}|$ vertices in R as $R \setminus \overline{N}(L_{sub}, g)$. □

Algorithm 7: List k-cliques in a t-plex ($t \ge 3$): `kCtPlex`

```
   Input: A branch $(S, g, l)$ with $g$ corresponding to a $t$-plex
   Output: All $k$-cliques within $(S, g, l)$
1  Construct the inverse graph $g_{inv}$ of $g$;
2  $I \leftarrow$ set of vertices in $V(g_{inv})$ that disconnect all others;
3  kCtPlex_Rec ($S$, $V(g_{inv}) \setminus I$, $l$)
4  Procedure kCtPlex_Rec ($S'$, $C$, $l'$)
       /* Termination when $l' = 0$ */
5      if $l' = 0$ then
6          Output a $k$-clique $S'$;
7          return;
       /* Choose all rest $l'$ vertices from $I$ */
8      if $|I| \ge l'$ then
9          for each $l'$-combination $I_{sub}$ over $I$ do
10             Output a $k$-clique $S' \cup I_{sub}$;
       /* Choose at least one vertex from $C$ */
11     for each $v_i \in C$ do
12         Create a branch $(S_i, C_i, l_i)$ based on Eq. (9);
13         if $|C_i| + |I| \ge l_i$ then kCtPlex_Rec ($S_i$, $C_i$, $l_i$) ;
```

Remark. We remark that `kC2Plex` achieves the *input-output sensitive* time complexity since the time cost depends on both the size of input $|E(g)|$ and the number of k-cliques within the branch. Besides, we note that $O(k \cdot c(g, k))$ is the optimal time for listing k-cliques within the branch and thus `kC2Plex` only takes extra $O(|E(g)|)$ time, i.e., `kC2Plex` is *nearly optimal.*

5.2 Listing k-Cliques from t-Plex with $t \ge 3$

Consider a branch $B = (S, g, l)$ with g as a t-plex with $t \ge 3$. The procedure for listing k-clique inside, called `kCtPlex`, differs in the way of branching (i.e., forming new branches). Specifically, it branches based on the inverse graph g_{inv} instead of g. The rationale is that since g is a t-plex and tends to be dense, its inverse graph g_{inv} would be sparse. As a result, branching on g_{inv} would run empirically faster. Below, we give the details.

Specifically, it maintains the inverse graph g_{inv} of g and represents a branch (S, g, l) by the new form of (S, C, l) where C is the set of vertices in g, i.e., $C = V(g)$. We note that the new form omits the information of edges in g, which is instead stored in g_{inv}. Consider the branching step of `kCtPlex` at a branch (S, C, l). Let $\langle v_1, v_2, \cdots, v_{|C|} \rangle$ be an arbitrary ordering of vertices in C. Then, the branching step would produce $|C|$ new sub-branches. The i-th branch, denoted by $B_i = (S_i, C_i, l_i)$, includes v_i to S and excludes $\{v_1, v_2, .., v_i\}$ from C. Formally, for $1 \le i \le |C|$, we have

$$S_i = S \cup \{v_i\},\ C_i = C \setminus \{v_1, v_2, \cdots, v_i\} \setminus N(v_i, g_{inv}),\ l_i = l - 1. \quad (9)$$

Note that we need to remove from C those vertices that disconnect v_i in g (since they cannot form any k-clique with v_i) and they connect to v_i in g_{inv}. Clearly, all k-cliques in (S, C, l) will be listed exactly once after branching. We note that the branching strategy we used in `kCtPlex` differs from that for `EBBkC` (and that for `VBBkC`). Specifically, the former (resp. the latter) is based on a sparse inverse graph g_{inv} (resp. a dense t-plex g) and maintains a vertex set C_i (resp. a graph instance g_i) for the produced branches. We remark that the

Table 1: Dataset Statistics.

Graph (Name)	$\lvert V\rvert$	$\lvert E\rvert$	Δ	δ	τ	ω
nasasrb (NA)	54,870	1,311,227	275	35	22	24
fbwosn (FB)	63,731	817,090	2K	52	35	30
wikitrust (WK)	138,587	715,883	12K	64	31	25
shipsec5 (SH)	179,104	2,200,076	75	29	22	24
youtube (YO)	1,157,828	2,987,624	29K	49	18	17
pokec (PO)	1,632,803	22,301,964	15K	47	27	29
wikicn (CN)	1,930,270	8,956,902	30K	127	31	33
baidu (BA)	2,140,198	17,014,946	98K	82	29	31
websk (WE)	121,422	334,419	590	81	80	82
citeseer (CI)	227,320	814,134	1K	86	85	87
stanford (ST)	281,904	1,992,636	39K	86	61	61
dblp (DB)	317,080	1,049,866	343	113	112	114
dielfilter (DE)	420,408	16,232,900	302	56	43	45
digg (DG)	770,799	5,907,132	18K	236	72	50
skitter (SK)	1,696,415	11,095,298	35K	111	67	67
orkut (OR)	2,997,166	106,349,209	28K	253	74	47

former (correspondingly, the early stop strategy with t at least 3) runs faster than the latter in practice, as verified in our experiments.

We summarize the procedure `kCtPlex` in Algorithm 7. In particular, it also utilizes the combinatorial technique for boosting the performance (lines 8-10). Specifically, it figures out the set of vertices, denoted by I, in g_{inv}, which disconnect all others (line 2). Consider a k-clique in B. It may involve c vertices in I where $c \in [0, \min\{|I|, k\}]$. Hence, we remove from $V(g_{inv})$ those vertices in I for branching at line 3 while adding them back to a k-clique at lines 8-10. It is not difficult to verify the correctness. Due to the page limit, we include the time complexity analysis of `kCtPlex` in the technical report [3].

Remark. We remark that (1) with the early termination strategy, the BB algorithms retain the same time complexity provided before but run practically faster as verified in the experiments and (2) the early termination strategy is supposed to set a small threshold of t so as to apply the alternative procedures only on dense graph instances (i.e., t-plexes). We test different choices of t in the experiments; the results suggest that `EBBkC` with t being set from 3 to 5 runs comparably faster than other choices while the best one among them varies for different settings of k.

6 EXPERIMENTS

6.1 Experimental Setup

Datasets. We use 16 real datasets in our experiments, which can be obtained from an open-source network repository [27]. For each graph, we ignore the directions, weights and self-loops (if any) at the very beginning. Following the existing study [19], we divide the real datasets into two groups based on the size of a maximum clique ω: small-ω graphs and large-ω graphs. For small-ω graphs, we list all k-cliques for all k, while for large-ω graphs, we only list k-cliques for small k values and large k values which are near ω. We collect the graph statistics and report the maximum degree Δ, the degeneracy number δ, the truss related number τ and the maximum clique size ω, which are shown in Table 1. We select four datasets, namely WK, PO, ST and OR, which are bold in the table, as default ones since they cover different size of graphs.

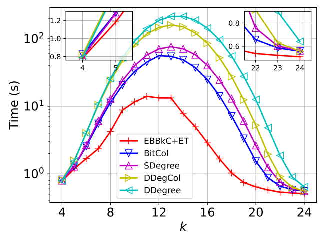
Baselines and Metrics. We choose `EBBkC-H` as the default edge-oriented branching-based BB framework for comparison, and denote it by `EBBkC` for brevity in the experimental results. We compare our algorithm `EBBkC+ET` with four existing algorithms, namely `DDegCol` [19], `DDegree` [19], `SDegree` [36] and `BitCol` [36] in terms of running time. Specifically, `EBBkC+ET` employs the edge-oriented branching-based BB framework with hybrid edge ordering, and applies the early-termination technique. We remark that we choose the best possible parameter t (in t-plex) for early-termination. All baselines are the state-of-the-art algorithms for k-cliques listing with vertex-oriented branching-based BB framework. Following the existing study [19], we vary k from 4 since k-clique listing problem reduces to triangle listing problem when $k = 3$ and there are efficient algorithms [18, 22] for triangle listing which run in polynomial time. For a fair comparison, we only collect the time of listing k-clique and omit the time of reading the graph from the disk.

Settings. The source codes of all algorithms are written in C++ and the experiments are conducted on a Linux machine with a 2.10GHz Intel CPU and 128GB memory. We note that we have not utilized SIMD instructions for data-level parallelism in our implementation, despite the potential to further enhance the acceleration of our algorithm. We set the time limit as 24 hours (i.e., 86,400 seconds) and the running time of any algorithm that exceeds the time limit is recorded with "INF". Implementation codes and datasets can be found via this link https://anonymous.4open.science/r/EBBkC.

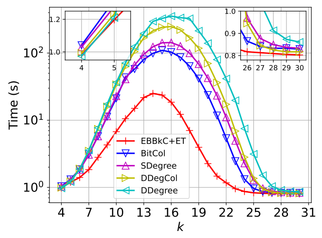
6.2 Experimental Results

(1) Comparison among algorithms (on small-ω graphs). Figure 3 shows the results of listing k-cliques on small-ω graphs. We observe that `EBBkC+ET` (indicated with red lines) runs faster than all baselines on all datasets. This is consistent with our theoretical analysis that the time complexity of `EBBkC+ET` is better than those of the baseline algorithms. Besides, on PO, CN and BA, we observe that the running time of `EBBkC+ET` first decreases when k is small. There are two possible reasons: (1) on some dataset, the number of k-cliques decreases as k increases when k is small. On BA, for example, the number of 4-cliques ($k = 4$) is nearly 28M while the number of 7-cliques ($k = 7$) is 21M; (2) as k increases, a larger number of branches can be pruned by the size constraint with the truss-based edge ordering, which provides the opportunity to run faster. To see this, we collect the number of promising branches after the branching step with the truss-based edge ordering. On PO, there are 11M branches left when enumerating 4-cliques ($k = 4$) while there are only less than 1M branches left when enumerating 10-cliques ($k = 10$).

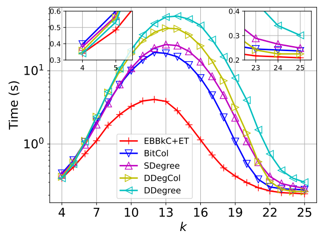
(2) Comparison among algorithms (on large-ω graphs). Figure 4 shows the results of listing k-cliques on large-ω graphs. We find that `EBBkC+ET` still runs the fastest. It is worthy noting that `EBBkC+ET` can greatly improve the efficiency by 1-2 orders of magnitude over the baselines when k is near the size of a maximum clique ω, e.g., `EBBkC+ET` runs 9.2x and 97.7x faster than `BitCol` on DB (when $k = 109$) and on DE (when $k = 40$), respectively. The reasons are two-fold: (1) when k is near ω, a large number of branches can be pruned by the size constraint with the truss-based edge ordering, and as a result, the remaining branches are relatively dense; (2) `EBBkC+ET` can quickly enumerate cliques within a dense structure, e.g., t-plex, without making branches for the search space, which dramatically reduces the running time.



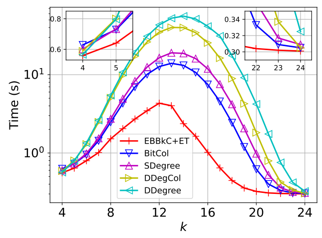
(a) NA



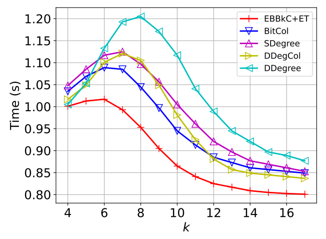
(b) FB



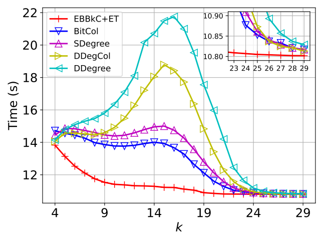
(c) WK



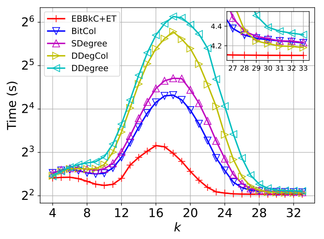
(d) SH



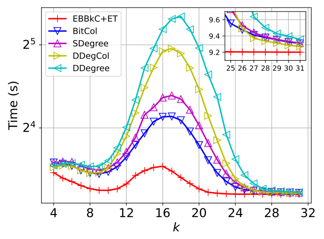
(e) YO



(f) PO

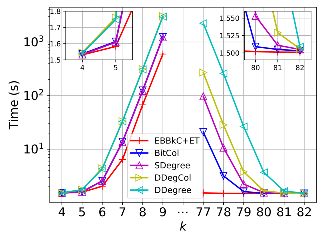


(g) CN

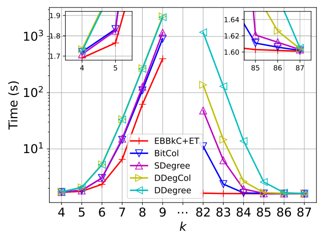


(h) BA

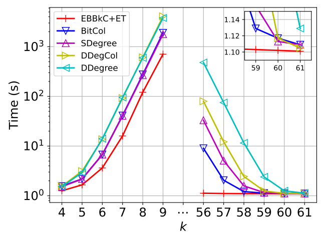
Figure 3: Comparison with baselines on the small-ω graphs, varying k from 4 to ω.



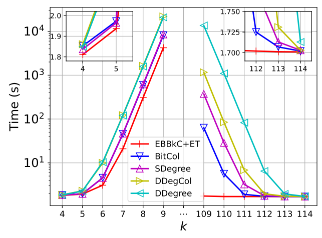
(a) WE



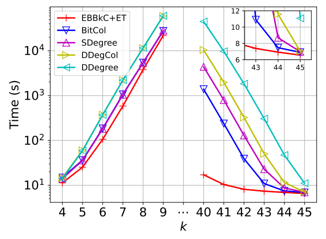
(b) CI



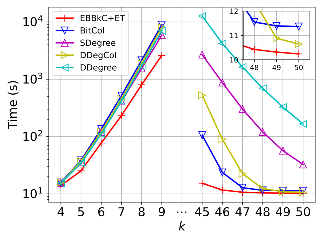
(c) ST



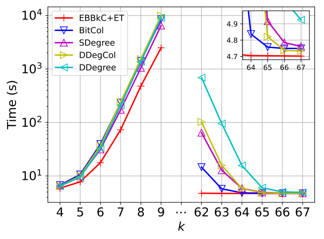
(d) DB



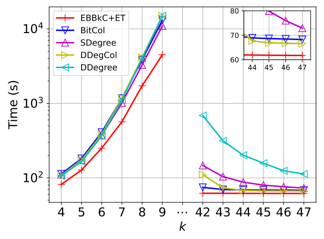
(e) DE



(f) DG



(g) SK



(h) OR

Figure 4: Comparison with baselines on the large-ω graphs, varying k from 4 to 9 and from $\omega - 5$ to ω.

(3) Effect of the enumeration frameworks (comparison between `EBBkC` and `VBBkC`). To ensure a fair comparison, we maintain consistent settings for both frameworks, specifically employing all color-based pruning rules and the early-termination technique. These two techniques are orthogonal to the frameworks and remain unchanged in both cases. For the branching steps under `VBBkC` framework, we follow [19] by first using degeneracy ordering to branch the universal search space and then using color ordering to branch the produced search spaces. The results are shown in Figure 5, indicated by red lines and blue lines. We observe that `EBBkC+ET` consistently outperforms `VBBkC+ET` on both small-ω graphs and large-ω graphs, which is consistent with our theoretical analysis since the parameter τ is strictly smaller than δ on these graphs as shown in Table 1.

(4) Effects of the edge ordering (comparison among `EBBkC-T`, `EBBkC-C` and `EBBkC-H`). For the sake of fairness, we employ all color-based pruning rules for `EBBkC-C` and `EBBkC-H` frameworks and employ the early-termination technique for all frameworks. The results are shown in Figure 6. Consider `EBBkC-H` and `EBBkC-T`. Although both frameworks have the same time complexity, `EBBkC-H` runs much faster since it can prune more unpromising search paths by color-based pruning rules than `EBBkC-T`. Consider `EBBkC-H` and `EBBkC-C`. `EBBkC-H` outperforms `EBBkC-C` since the largest sub-problem instances produced by `EBBKC-H` is smaller than that of `EBBkC-C`, which also conforms our theoretical analysis.

(5) Effect of the color-based pruning rules. Recall that in Section 4.3, we introduce two color-based pruning rules, where the first rule is adapted from the existing studies [19] and the second rule

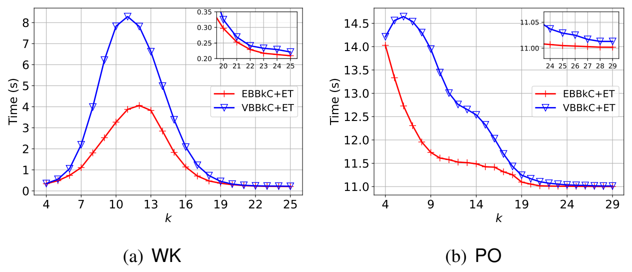


Figure 5: Effects of the enumeration frameworks (comparison between `EBBkC` and `VBBkC`).

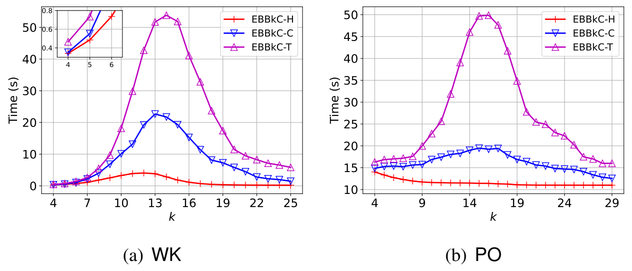


Figure 6: Effects of the edge ordering (comparison among `EBBkC-T`, `EBBkC-C` and `EBBkC-H`).

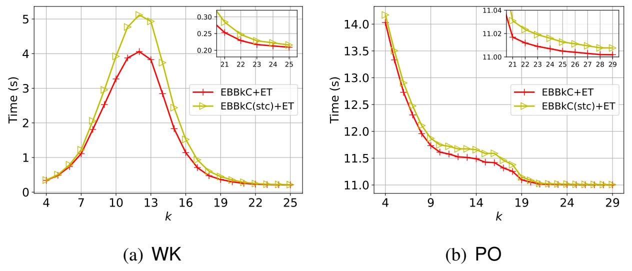


Figure 7: Effects of the color-based pruning rules (comparison between the algorithms with and without the Rule (2)).

is newly proposed in this paper. Therefore, we study the effect of the second pruning rule by making comparison between the running time of the algorithms with and without this rule, respectively. We denote the algorithm without this rule by `EBBkC(stc)+ET`. The results are shown in Figure 7, indicated by red lines and yellow lines. We observe that the second pruning rule brings more advantages as k increases. A possible reason is that when k is small, the graph instance g usually has more than l colors ($l \le k$), which cannot be pruned by the second rule while as k increases, sparse graph instances can easily violate the rule and they can be safely pruned.

(6) Effect of early-termination technique. We first study the effect of early-termination technique by making comparison between `EBBkC+ET` and `EBBkC`. Both algorithms use edge-oriented BB framework and all color-based pruning rules. The results are shown in Figure 8, indicated by red lines and magenta lines. We observe that as k increases, early-termination technique can significantly reduce the running time. Besides, we also study the effects of choosing different parameter t (in t-plex) for early-termination. We vary t in the range $\{1, 2, 3, 4, 5\}$ and report the corresponding running time of `EBBkC+ET` under different values of k. The results are shown in Figure 9. We have the following observations. First, for all values of k,

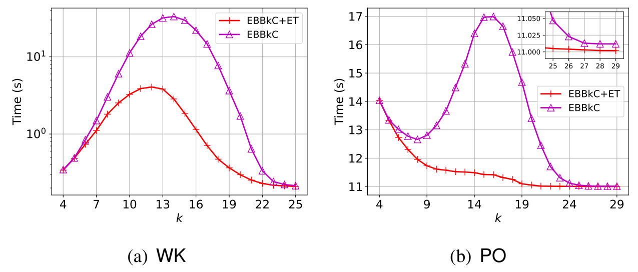


Figure 8: Effects of early-termination technique (comparison between the algorithms with and without the technique).

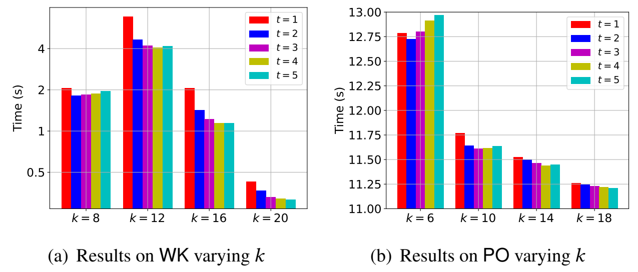


Figure 9: Effects of early-termination technique (varying t).

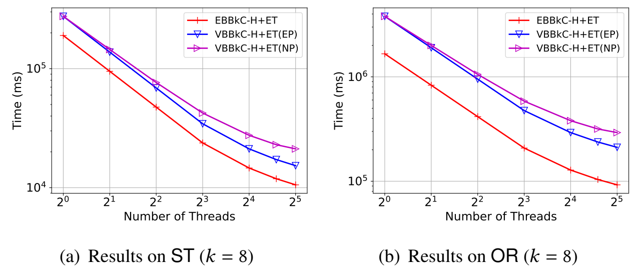


Figure 10: Comparison among different parallel schemes, varying the number of threads.

`EBBkC+ET` with $t = 2$ always runs faster than that with $t = 1$, since we can list all k-cliques inside a t-plex in optimal time when $t = 1$ and $t = 2$ but we can early-terminate the recursion EBBkC_Rec in an earlier phase when $t = 2$ than that when $t = 1$ (Section 5.1). Second, when the value of k is small, `EBBkC+ET` with smaller t runs faster while when the value of k is large, `EBBkC+ET` with larger t runs faster. On WK, for example, when $k = 8$, `EBBkC+ET` with $t = 2$ runs the fastest; when $k = 12$ and $k = 16$, `EBBkC+ET` with $t = 4$ runs the fastest; and when $k = 20$, `EBBkC+ET` with $t = 5$ runs the fastest. This phenomenon conforms our theoretical analysis in Section 5 that when the value of k increases, listing k-cliques with early-termination on sparser plexes, i.e., t-plex with larger t, can also be efficient.

(7) Parallelization. We compare different algorithms in a parallel computing setting. Specifically, for `EBBkC` framework, since each produced sub-branch produced from $B = (\emptyset, G, k)$ can be solved independently (see line 6 of Algorithm 5), we process all of such sub-branches in parallel. For existing studies under `VBBkC` framework [11, 19, 36], there are two parallel schemes. One is called `NodeParallel` (`NP` for short). This strategy processes each produced sub-branch at $B = (\emptyset, G, k)$ in parallel since they are independent (see line 9 of Algorithm 1). The other is called EdgeParallel (EP for short). This strategy first aggregates the first two consecutive branching steps at $B = (\emptyset, G, k)$ as a unit, and as a result, it would produce $|E(G)|$ sub-branches, then it processes each sub-branch in parallel since they are independent [11, 19, 36]. The results of the comparison are shown in Figure 10. Consider the comparison between VBBkC+ET(NP) and VBBkC+ET(EP), indicated by blue lines and magenta lines. The algorithm with edge-level parallelization strategy achieves a higher degree of parallelism since it would produce a number of problem instances with smaller and similar scales, which balances the computational loads across the threads. Then, we consider the comparison between EBBkC+ET and VBBkC+ET(EP), indicated by red lines and blue lines. Both algorithms can be regarded as adopting edge-level parallelization strategy but differ in the ordering of the edges. We observe that EBBkC+ET runs faster than VBBkC+ET(EP). The reason is that EBBkC uses truss-based edge ordering, which would produce even smaller problem instances than those of VBBkC+ET(EP).

7 RELATED WORK

Listing k-cliques for arbitrary k values. Existing exact k-clique listing algorithms for arbitrary k values can be classified into two categories: backtracking based algorithms [9] and branch-and-bound based algorithms [19, 35, 36]. Specifically, the algorithm Arbo [9] is the first practical algorithm for listing all k-cliques, whose time complexity (we focus on the worst-case time complexies in this paper) is $O(km\alpha^{k-2})$, where m and α are the number of edges and the arboricity of the graph, respectively. However, it is difficult to paralleize Arbo since it involves a depth-first backtracking procedure. To solve this issue, several (vertex-oriented branching-based) branch-and-bound (BB) based algorithms are proposed, including Degree [13, 22], Degen [11], DegenCol [19], DegCol [19], DDegCol [19], DDegree [19], SDegree [36] and BitCol [36]. As introduced in Section 3, these algorithms follow the same framework but differ in how the vertex orderings are adopted and some implementation details. Specifically, Degree uses a global degree ordering, with which the size of the largest problem instance produced can be bounded by η (i.e., the h-index of the graph). Degree has a time complexity of $O(km(\eta/2)^{k-2})$. Degen uses a global degeneracy ordering, with which the size of the largest problem instance produced can be bounded by δ (i.e., the degeneracy of the graph). Degen has a time complexity of $O(km(\delta/2)^{k-2})$. Since it has been proven that $\delta \leq 2\alpha - 1$ [37], Degen is the first algorithm that outperforms Arbo theoretically and practically. However, Degen suffers from the issue that it cannot list the clique whose size is near ω (i.e., the size of a maximum clique). To solve this issue, the authors in [19] propose several algorithms, which are based on color-based vertex orderings. DegCol and DegenCol first color the graph with some graph coloring algorithms (e.g., inverse degree based [35] and inverse degeneracy based [15]) and generate the an ordering of vertices based on the color values of the vertices. While the color values of the vertices can significantly prune the unpromising search paths and make the algorithm efficient to list near-ω cliques, DegCol and DegenCol both have the time complexity of $O(km(\Delta/2)^{k-2})$, where Δ is the maximum degree of a vertex in the graph since the color-based ordering cannot guarantee a tighter size bound for the produced problem instance. To overcome this limitation, DDegCol adopts a hybrid ordering, where it first uses degeneracy ordering to branch the universal search space such that the size of each produced problem instance is bounded by δ, then it uses color-based orderings to branch each produced sub-branch. Following a similar procedure, DDegree also combines degeneracy ordering and degree ordering to branch the universal search space and the subspaces, respectively. In this way, DDegCol and DDegree have the time complexity of $O(km(\delta/2)^{k-2})$. BitCol and SDegree implement DDegCol and DDegree in a more efficient way with SIMD instructions, respectively, and retain the same time complexity as that of DDegCol and DDegree. In contrast, our EBBkC algorithm is an edge-oriented branching-based BB algorithm based on edge orderings and has its time complexity (i.e., $O(km(\tau/2)^{k-2})$) better than that of the state-of-the-art algorithms including DDegCol, DDegree, BitCol and SDegree (i.e., $O(km(\delta/2)^{k-2})$), where τ is strictly smaller than δ.

Some other algorithms, e.g., MACE [21], which are originally designed for the maximal clique enumeration problem, can also be adapted to listing k-cliques problem [21, 30, 31]. These adapted algorithms are mainly based on the well-known Bron-Kerbosch (BK) algorithm [6] with some size constraints to ensure that each clique to be outputted has exactly k vertices. However, these adapted algorithm are even less efficient than Arbo theoretically and practically, e.g., MACE has a time complexity of $O(kmn\alpha^{k-2})$ [19, 30], and cannot handle large real graphs.

Listing k-cliques for special k values. There are two special cases for k-clique listing problem: when $k = 3$ and when $k = \omega$. When $k = 3$, the problem reduces to triangle listing problem [9, 18, 22]. The state-of-the-art algorithm for triangle listing problem follows a vertex ordering based framework [22], whose time complexity is $O(m\alpha)$, where α is the arboricity of the graph. When $k = \omega$, the problem reduces to maximum clique search problem [7, 20, 23, 26]. The state-of-the-art algorithm for maximum clique search problem first transforms the maximum clique problem to a set of clique finding sub-problems, then it conducts a branch-and-bound framework to iteratively check whether a clique of a certain size can be found in the sub-problem [7], whose time complexity is $O(n2^n)$. We note that these existing algorithms cannot solve the k-clique listing problem for arbitrary k's.

8 CONCLUSION

In this paper, we study the k-clique listing problem, a fundamental graph mining operator with diverse applications in various networks. We propose a new branch-and-bound framework, named EBBkC, which incorporates an edge-oriented branching strategy. This strategy expands a partial k-clique using two connected vertices (an edge), offering new opportunities for optimization. Furthermore, to handle dense graph sub-branches more efficiently, we develop specialized algorithms that enable early termination, contributing to improved performance. We conduct extensive experiments on 16 real graphs, and the results consistently demonstrate EBBkC's superior performance compared to state-of-the-art VBBkC-based algorithms. In the future, we will explore the possibility of adopting our algorithms to list k-cliques' counterparts in other types of graphs such as bipartite graphs.

REFERENCES

[1] Balázs Adamcsek, Gergely Palla, Illés J Farkas, Imre Derényi, and Tamás Vicsek. 2006. CFinder: locating cliques and overlapping modules in biological networks. *Bioinformatics* 22, 8 (2006), 1021–1023.

[2] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *The VLDB journal* 23, 2 (2014), 175–199.

[3] Anonymous Authors. 2023. Efficient k-Clique Listing: An Edge-Oriented Branching Strategy (Technical Report). https://anonymous.4open.science/r/EBBkC.

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

[5] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

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

[7] Lijun Chang. 2019. Efficient maximum clique computation over large sparse graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 529–538.

[8] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating truss decomposition on heterogeneous processors. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1751–1764.

[9] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[10] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008).

[11] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[12] Paul Erdös and George Szekeres. 1935. A combinatorial problem in geometry. *Compositio mathematica* 2 (1935), 463–470.

[13] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. 2015. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)* 20 (2015), 1–20.

[14] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. 2012. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 8 (2012), 1651–1660.

[15] William Hasenplaugh, Tim Kaler, Tao B Schardl, and Charles E Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. 166–177.

[16] Pan Hui and Jon Crowcroft. 2008. Human mobility models and opportunistic communications system design. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1872 (2008), 2005–2016.

[17] Richard M Karp. 2010. *Reducibility among combinatorial problems*. Springer.

[18] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.

[19] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* (2020).

[20] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the maximum clique in massive graphs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1538–1549.

[21] Kazuhisa Makino and Takeaki Uno. 2004. New algorithms for enumerating all maximal cliques. In *Scandinavian workshop on algorithm theory*. Springer, 260–272.

[22] Mark Ortmann and Ulrik Brandes. 2014. Triangle listing algorithms: Back from the diversion. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 1–8.

[23] Patric RJ Östergård. 2002. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics* 120, 1-3 (2002), 197–207.

[24] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814–818.

[25] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814–818.

[26] Bharath Pattabiraman, Md Mostofa Ali Patwary, Assefaw H Gebremedhin, Weikeng Liao, and Alok Choudhary. 2015. Fast algorithms for the maximum clique problem on massive graphs with applications to overlapping community detection. *Internet Mathematics* 11, 4-5 (2015), 421–448.

[27] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[28] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. 2007. A large-scale study of link spam detection by graph algorithms. In *Proceedings of the 3rd international workshop on Adversarial information retrieval on the web*. 45–48.

[29] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. 2015. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*. 927–937.

[30] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. 2009. A scalable, parallel algorithm for maximal clique enumeration. *Journal of parallel and distributed computing* 69, 4 (2009), 417–428.

[31] UNO Takeaki. 2012. Implementation issues of clique enumeration algorithm. *Special issue: Theoretical computer science and discrete mathematics, Progress in Informatics* 9 (2012), 25–30.

[32] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*. 1122–1132.

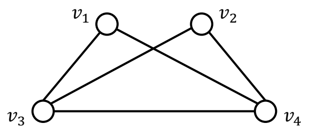
[33] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 104–112.

[34] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment* 5, 9 (2012), 812–823.

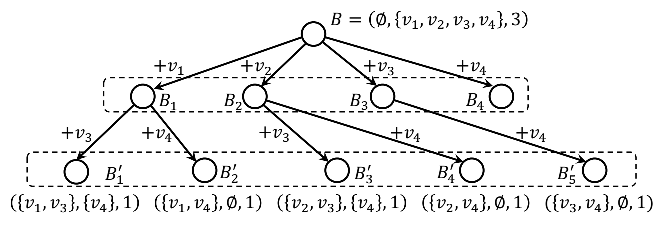
[35] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Effective and efficient dynamic graph coloring. *Proceedings of the VLDB Endowment* 11, 3 (2017), 338–351.

[36] Zhirong Yuan, You Peng, Peng Cheng, Li Han, Xuemin Lin, Lei Chen, and Wenjie Zhang. 2022. Efficient k-clique listing with set intersection speedup. *ICDE. IEEE* (2022).

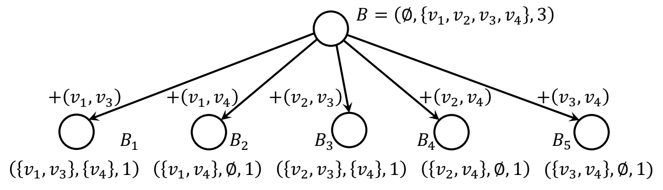
[37] Xiao Zhou and Takao Nishizeki. 1994. Edge-coloring and f-coloring for various classes of graphs. In *Algorithms and Computation: 5th International Symposium, ISAAC'94 Beijing, PR China, August 25–27, 1994 Proceedings 5*. Springer, 199–207.



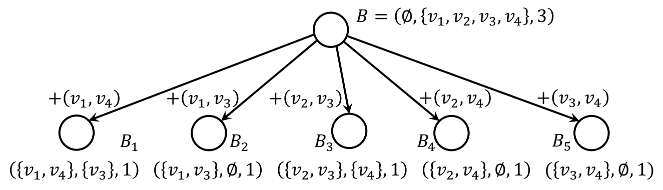
(a) An example graph G.


$B = (\emptyset, \{v_1, v_2, v_3, v_4\}, 3)$
$+v_1$
$+v_2$
$+v_3$
$+v_4$
B_1
B_2
B_3
B_4
$+v_3$
$+v_4$
$+v_3$
$+v_4$
$+v_4$
B'_1
B'_2
B'_3
B'_4
B'_5
$(\{v_1, v_3\}, \{v_4\}, 1)$
$(\{v_1, v_4\}, \emptyset, 1)$
$(\{v_2, v_3\}, \{v_4\}, 1)$
$(\{v_2, v_4\}, \emptyset, 1)$
$(\{v_3, v_4\}, \emptyset, 1)$

(b) Branching with VBBkC on G.


$B = (\emptyset, \{v_1, v_2, v_3, v_4\}, 3)$
$+(v_1, v_3)$
$+(v_1, v_4)$
$+(v_2, v_3)$
$+(v_2, v_4)$
$+(v_3, v_4)$
B_1
B_2
B_3
B_4
B_5
$(\{v_1, v_3\}, \{v_4\}, 1)$
$(\{v_1, v_4\}, \emptyset, 1)$
$(\{v_2, v_3\}, \{v_4\}, 1)$
$(\{v_2, v_4\}, \emptyset, 1)$
$(\{v_3, v_4\}, \emptyset, 1)$

(c) Branching with EBBkC that can generate the same branches as those with VBBkC.


$B = (\emptyset, \{v_1, v_2, v_3, v_4\}, 3)$
$+(v_1, v_4)$
$+(v_1, v_3)$
$+(v_2, v_3)$
$+(v_2, v_4)$
$+(v_3, v_4)$
B_1
B_2
B_3
B_4
B_5
$(\{v_1, v_4\}, \{v_3\}, 1)$
$(\{v_1, v_3\}, \emptyset, 1)$
$(\{v_2, v_3\}, \{v_4\}, 1)$
$(\{v_2, v_4\}, \emptyset, 1)$
$(\{v_3, v_4\}, \emptyset, 1)$

(d) Branching with EBBkC that can generate the branches that VBBkC cannot generate.

Figure 11: Illustration of the advantages of EBBkC over BBkC.

A PROOF OF THEOREM 4.2

The running time of EBBkC−T is dominated by the recursive listing procedure (lines 6-10). Given a branch $B = (S, g, l)$, we denote by $T(g, l)$ the upper bound of time cost of listing l-cliques under such branch. When $k \geq 3$, with different values of l, we have the following recurrences.

$$T(g,l) \leq \begin{cases} O(k \cdot |V(g)|) & l = 1 \\ O(k \cdot |E(g)|) & l = 2 \\ \sum_{e_i \in E(g)} \Big(T(g_i, l-2) + T'(g_i)\Big) & 3 \leq l \leq k-2 \end{cases} \tag{10}$$

where $T'(g_i)$ is the time for constructing g_i given $B = (S, g, l)$ (line 9 of Algorithm 3). We first show the a lemma which builds the relationship between g_i and g, then we present the complexity of $T'(g_i)$.

LEMMA A.1. *Given a branch $B = (S, g, l)$ and the sub-branches $B_i = (S_i, g_i, l_i)$ produced at B. We have (1) when $l < k$,*

$$\sum_{e_i \in E(g)} |E(g_i)| < \frac{\tau^2}{4} \cdot |E(g)| \tag{11}$$

and (2) when $l = k$ (we note that the branch B corresponds to the universal branch $B = (\emptyset, G, k)$),

$$\sum_{e_i \in E(G)} |E(g_i)| < \frac{\tau^2}{2} \cdot |E| \tag{12}$$

PROOF. **Case $l < k$.** $E(g_i)$ can be obtained by checking for each edge in $\{e_{i+1}, \cdots, e_{|E(g)|}\}$ whether it is in $E[e_i]$. Clearly, we have $|E(g_i)| \leq |E(g)| - i$. Then

$$\sum_{e_i \in E(g)} |E(g_i)| \leq \sum_{i=1}^{|E(g)|} (|E(g)| - i) = \frac{|E(g)|(|E(g) - 1|)}{2} \tag{13}$$

According to Lemma 4.1, each branch contains at most τ vertices, which indicates that $|E(g)|$ is at most $\tau(\tau-1)/2$. Therefore,

$$\frac{|E(g)|(|E(g) - 1|)}{2} \leq \frac{\tau(\tau-1)}{4} \cdot (|E(g) - 1|) < \frac{\tau^2}{4} \cdot |E(g)| \tag{14}$$

Case $l = k$. According Lemma 4.1, $V[e] \leq \tau$. Then $E[e]$ contains at most $\tau(\tau-1)/2$ edges. Thus,

$$\sum_{e_i \in E} |E(g_i)| \leq \sum_{e_i \in E} \frac{\tau(\tau-1)}{2} < \frac{\tau^2}{2} \cdot |E| \tag{15}$$

which completes the proof. □

Consider $T'(g_i)$. When $l = 3$, for each edge $e_i \in E(g)$, we just need to compute $V(g_i)$ by checking for each vertex in $V(g)$ whether it is in $V[e_i]$, which costs at most $O(\tau)$ time since $|V(g)| \leq \tau$. When $l > 3$, for each edge $e_i \in E(g)$, we need to compute both $V(g_i)$ and $E(g_i)$. Therefore,

$$\sum_{e_i \in E(g)} T'(g_i) = \begin{cases} O(\tau \cdot |E(g)|) & l = 3 \\ O(\tau^2 \cdot |E(g)|) & l > 3 \end{cases} \tag{16}$$

Given the above analyses, we prove the Theorem 4.2 as follows.

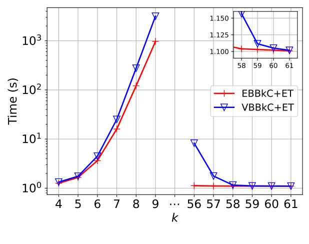
PROOF. We prove by induction on l to show that

$$T(g,l) \leq \lambda \cdot (k + 2l) \cdot |E(g)| \cdot \left(\frac{\tau}{2}\right)^{l-2} \tag{17}$$

where λ is positive constant. Since the integer l decreases by 2 when branching, then when k is odd (resp. even), l would always be odd (resp. even) in all branches. Thus, we need to consider both cases.

Base Case ($l = 2$ *and* $l = 3$). When $l = 2$, it is easy to verify that there exists λ such that $T(g, 2) \leq \lambda \cdot k \cdot |E(g)|$ satisfies Eq. (17). When $l = 3$, we put Eq. (16) into Eq. (10), and it is easy to verify that $T(g, 3) \leq \lambda \cdot k \cdot |E(g)| \cdot \tau$, which also satisfies Eq. (17).

Induction Step. We first consider the case when $3 < l < k$. Assume the induction hypothesis, i.e., Eq. (17), is true for $l = p$ ($p + 2 < k$).


(a) ST

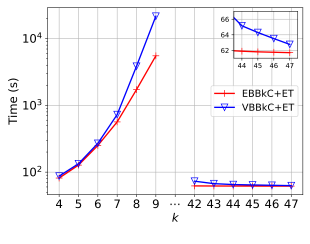
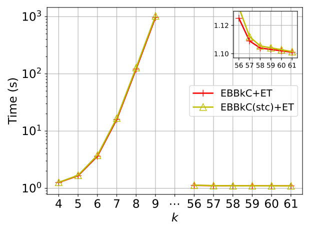

(b) OR

Figure 12: Effects of the enumeration frameworks (comparison between EBBkC and VBBkC).


(a) ST

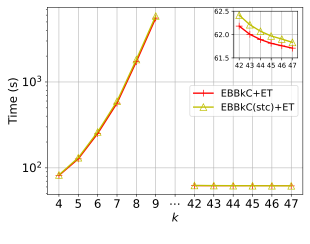
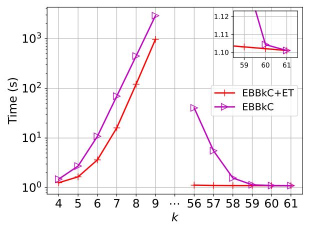

(b) OR

Figure 13: Effects of the color-based pruning rules (comparison between the algorithms with and without the Rule (2)).


(a) ST

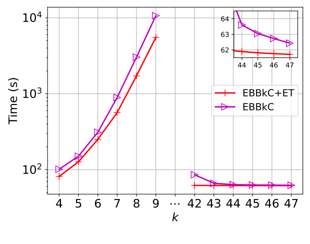

(b) OR

Figure 14: Effects of early-termination technique (comparison between the algorithms with and without the technique).

When $l = p + 2$,

$$\begin{aligned} T(g, p+2) &\leq \sum_{e_i \in E(g)} \Big(T(g_i, p) + T'(g_i)\Big) \\ &\leq \lambda \cdot \tau^2 \cdot |E(g)| + \sum_{e_i \in E(g)} \lambda \cdot (k+2p) \cdot |E(g_i)| \cdot \left(\frac{\tau}{2}\right)^{p-2} \\ &< \lambda \cdot \tau^2 \cdot |E(g)| + \lambda \cdot (k+2p) \cdot |E(g)| \cdot \left(\frac{\tau}{2}\right)^{p} \\ &\leq \lambda \cdot (k+2p+4) \cdot |E(g)| \cdot \left(\frac{\tau}{2}\right)^{p} \end{aligned} \tag{18}$$

The first inequality derives from recurrence. The second inequality derives from Eq. (16) and the induction hypothesis. The third inequality derives from Lemma A.1 (case $l < k$). The fourth inequality derives from the fact that $\tau^2 \leq 4 \cdot (\tau/2)^p$ when $p \geq 2$.

Then consider the cases when $l = k$.

$$\begin{aligned} T(G, k) &\leq \sum_{e_i \in E(G)} \Big(T(g_i, k-2) + T'(g_i)\Big) \\ &\leq \lambda \cdot \tau^2 \cdot E(G) + \sum_{e_i \in E(G)} \lambda \cdot (3k-4) \cdot |E(g_i)| \cdot \left(\frac{\tau}{2}\right)^{k-4} \\ &< \lambda \cdot \tau^2 \cdot E(G) + 2\lambda \cdot (3k-4) \cdot |E(G)| \cdot \left(\frac{\tau}{2}\right)^{k-2} \\ &< 6\lambda \cdot k \cdot |E(G)| \cdot \left(\frac{\tau}{2}\right)^{k-2} \end{aligned} \tag{19}$$

The first inequality derives from recurrence. The second inequality derives from Eq. (16) and the induction hypothesis. The third inequality derives from Lemma A.1 (case $l = k$). The fourth inequality derives from the fact that $\tau^2 \leq 4 \cdot (\tau/2)^{k-2}$ when $k \geq 4$.

Conclusion. We conclude that given a graph $G = (V, E)$ and an integer $k \geq 3$, the time complexity of EBBkC can be upper bounded by $O(k \cdot |E(G)| \cdot (\tau/2)^{k-2})$. □

B A COUNTER-EXAMPLE FOR THE STATEMENT IN SECTION 4.2

Example. To illustrate the statement in Section 4.2, we consider the example in Figure 11. The graph G involves 4 vertices and 5 edges. Assume that we aim to list 3-cliques in G, i.e., $k = 3$. Consider the degeneracy ordering of the vertices $\pi_\delta = \langle v_1, v_2, v_3, v_4\rangle$. Then VBBkC would generate 5 branches following the vertex ordering π_δ, and the illustration is shown in Figure 11(b). Correspondingly, we can construct an edge ordering $\pi_e = \langle (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\rangle$ by following which we can produce the same branches as that of VBBkC. Consider the truss-based edge ordering $\pi_\tau = \langle (v_1, v_4), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4)\rangle$. The produced branches are shown in Figure 11(d). We note that these produced branches cannot be generated with any vertex orderings. To see this, consider the branch B_1 and B_3 in Figure 11(d). If there exists a vertex ordering π_v that can generate B_1, then vertex v_3 must be ordered behind v_4 in π_v. Similarly, for branch B_3, vertex v_4 must be ordered behind v_3 in π_v. This derives a contradiction.

C TIME COMPLEXITY OF ALGORITHM 7

THEOREM C.1. *Given a branch $B = (S, g, l)$ with g being a t-plex ($t \geq 3$), kCtPlex lists all k-cliques within B in at most $O(|E(g)| + t \cdot \binom{|V(g)|}{l}) + k \cdot c(g, l))$ time, where $c(g, l)$ is the number of l-cliques in g.*

PROOF. Let $T(g, l)$ be the total time complexity. Lines 1-2 of Algorithm 7 can be done in $O(E(g))$ time for partitioning $V(g)$ and constructing the inverse graph g_{inv}. Let $T'(|C|, l')$ be the time complexity to produce all sub-branches excluding the output part (lines 5-10). We have $T'(0, \cdot) = 0$ and $T'(\cdot, 0) = 0$.

$$T \leq O(E(g) + k \cdot c(g, l)) + T'(|V(g) \setminus I|, l) \tag{20}$$

According to Eq. (9), for each $1 \le i \le |C|$, the size of the produced C_i is at most $|C| - i$. Thus, we have the following recurrence.

$$\begin{aligned} T'(|C|, l') &= \sum_{i=1}^{|C|} \Big(T'(|C_i|, l' - 1) + O(t) \Big) \\ &= T'(|C| - 1, l' - 1) + O(t) + \sum_{j=0}^{|C|-2} \Big(T'(j, l' - 1) + O(t) \Big) \\ &= T'(|C| - 1, l' - 1) + T'(|C| - 1, l') + O(t) \end{aligned} \tag{21}$$

We note that $O(t)$ is the time to filter out the vertices that are connected with v_i in g_{inv}, i.e., $N(v_i, g_{inv})$. Let $T^*(|C|, l') = T'(|C|, l') + O(t)$ and apply it to the above equation,

$$T^*(|C|, l') = T^*(|C| - 1, l' - 1) + T^*(|C| - 1, l') \tag{22}$$

with the initial conditions $T^*(\cdot, 0) = O(t)$ and $T^*(0, \cdot) = O(t)$. Observe that Eq. (22) is similar to an identity equation $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Then it is easy to verify that $T^*(|C|, l') = O\big(\binom{|C|}{l'} \cdot t\big)$. Therefore, $T'(|C|, l')$ can also be bounded by $O\big(\binom{|C|}{l'} \cdot t\big)$. Finally, since $|V(g) \setminus I| \le |V(g)|$, the time complexity of Algorithm 7 is at most $O(|E(g)| + t \cdot \binom{|V(g)|}{l} + k \cdot c(g, l))$. □

Remark. Recall that in Eq. (9), we need to filter out the vertices that are connected with v_i in g_{inv} to create the sub-branch. There are two possible ways to implement this procedure. One way is to use a shared array to store $V(g) \setminus I$, denoted by C. Every time we execute line 11 and 12, the pointer in C moves forward and marks those vertices in $N(v_i, g_{inv})$ as invalid, respectively. The benefit is that the procedure can be done in $O(t)$ in each round, as we show in Eq. (21), but will be hard to be parallelized. Another way is to use additional $O(C_i)$ to ensure that each sub-branch has its own copy of C_i, which makes the procedure easy to be parallelized. In our experiments, we implement the algorithm in the former way.

D ADDITIONAL EXPERIMENTAL RESULTS

The experimental results on studying the effects of the enumeration framework (comparison between EBBkC and VBBkC), the effects of the color-based pruning rules (comparison between the algorithms with and without the Rule (2)) and the effects of the early-termination technique (comparison between the algorithms with and without the technique) on datasets ST and OR are shown in Figure 12, Figure 13 and Figure 14, respectively.