From Stars to Diamonds: Counting and Listing Almost Complete Subgraphs in Large Networks

Irene Finocchi1,*, Renan Leon Garcia2 and Blerina Sinaimeri1

1Luiss University, Viale Romania 32, 00197, Rome, Italy
2Sapienza University of Rome, Via Salaria 113, 00139, Rome, Italy
*Corresponding author: finocchi@luiss.it

Listing dense subgraphs is a fundamental task with a variety of network analytics applications. A lot of research has been done focusing on k-cliques, i.e. complete subgraphs on k nodes. However, requiring complete connectivity between the nodes of a subgraph may be too restrictive in many real applications. Hence, in this paper, we consider a natural relaxation of cliques, called k-diamonds and defined as cliques of size k with one missing edge. We first provide a sequential algorithm that, in \(O(nm^{k-1/2})\) time, counts and lists all the k-diamonds in large graphs, for any constant \(k \geq 4\). A parallel extension of the sequential algorithm is then proposed and analyzed in a MapReduce-style model, achieving the same local and total space usage of the state-of-the-art algorithms for k-cliques. The running time is optimal on dense graphs and \(O(\sqrt{m})\) larger than k-clique counting if the graph is sparse. Our algorithms compute induced diamonds by analyzing the structure of directed stars formed by the graph nodes and their neighbors.

Keywords: clique relaxations; dense subgraphs; distributed algorithms; enumeration algorithms; graph mining; MapReduce

1. INTRODUCTION

One of the essential tasks in graph mining is finding dense subgraphs: density is associated to the amount of interconnections within a subgraph and influence many social phenomena. Counting dense subgraphs is therefore a fundamental task with a variety of applications such as social network analysis [1], link classification and recommendation [2], graph partition and compression [3], pattern discovery in biological networks [4]. A prominent example of a dense subgraph is a clique, where every pair of nodes are connected by an edge. A lot of research has been done in counting k-cliques (i.e. complete subgraphs on k nodes). However, in many real applications, requiring full connectivity between the nodes of a subgraph may be too restrictive. For instance, in community detection in social networks, a clique model may not capture realistic situations where some individuals do not know each other directly, but still belong to the same community. In a different context, the presence of errors in data may result in missing links and considering cliques could thus lead to a significant underestimation of the communities [5]. To deal with these issues, researchers in several applied fields have defined and studied less restrictive structures, typically referred to as clique relaxations [6]. Some of these relaxations include k-cores, maximal subgraphs that require each node to have degree at least k [7]; k-trusses, connected subgraphs in which each edge is incident to at least k − 2 triangles [8, 9]; k-plexes, which require each node to be connected to all but k other nodes [10, 11]; y-quasi-cliques, which require the density of the induced subgraph to be at least y [12]; and k-clubs, which require the diameter of the subgraph to be at most k [13, 14]. In this paper, we focus on a very natural and simple relaxation of cliques, called k-diamonds, defined as cliques of size k with one missing edge. Notice that k-diamonds generalize the well-known diamond graph that consists of a complete graph \(K_4\) minus one edge. We study the problem of listing k-diamonds algorithmically, both in a sequential and in a parallel setting.

1.1. Related work

A lot of research has been done in counting a large variety of clique-like subgraphs, both sequentially and in parallel (see, e.g. [8, 15–19]). Since k-diamonds are “almost” cliques, we will most closely relate the performance of our algorithms to bounds related to the counting of k-cliques, which are detailed in the following.

Counting triangles is the first non-trivial case of the counting of k-cliques (k = 3). Triangle counting has been largely studied, due to its large number of applications in network analysis. Several exact algorithms exist, and they can be generally categorized into two distinct groups. The first group only provides a count of triangles without listing (i.e. without enumerating) them. On the other hand, the second group, called enumeration-based algorithms, systematically lists all the triangles, making the counting process straightforward. A clear advantage of an enumeration-based approach is its capability to both count triangles and generate a comprehensive list, which can be valuable for tasks like community detection [20, 21] and spam filtering [22]. Our work aligns with the latter group of algorithms. We will hence consider enumeration-based counting both in a sequential and in a parallel context.

In the sequential setting, a simple yet practical algorithm was presented in [23]. Given an input graph \(G\) with \(n\) nodes and \(m\) edges, this algorithm requires \(O(nC)\) time to list (and hence also count) all triangles in \(G\), where \(C = O(\sqrt{m})\) is the arboricity.
of the graph. This running time is optimal in the worst case. In the same paper [23], the triangle counting algorithm was extended to compute \( k \)-cliques of a graph \( G \) in time \( O(k\alpha(G)^{k/2}m) \), where \( \alpha(G) \) is the arboricity of \( G \); for constant values of \( k \), this is \( O(m^{k/2}) \) in the worst case, since it can be proved that \( \alpha(G) = O(\sqrt{m}) \). Another algorithm initially proposed for counting triangles [1] can be also extended to \( k \)-cliques, reaching optimal \( O(m^{k/2}) \) time.

In order to speed up the listing task, parallel approaches have been proposed in many models of computation (see, e.g. [24, 25] and the references therein). In particular, Suri and Vassilvitskii [26] describe MapReduce algorithms to list triangles in \( O(m^{3/2}) \) total space and \( O(m^{3/2}) \) work, where work is the total computational effort required by all the computational units. Cliques of size \( k \) are considered in [27], presenting an algorithm which is work-optimal in the worst case and lists \( k \)-cliques in \( O(m^{k/2}) \) total space and \( O(m^{k/2}) \) time. Other algorithms have been introduced that exploit different techniques in order to increase the performance of these algorithms in practice (see, e.g. [28]).

It is worth noticing that instead of considering cliques of fixed size \( k \), several works in the literature have also addressed the maximal clique problem (see, e.g. [17, 29–33]).

1.2. Our contributions

In this paper, we present both sequential and parallel MapReduce-style algorithms for listing \( k \)-diamonds, for any \( k \geq 4 \).

As a first contribution, we introduce a structural classification of \( k \)-diamonds, based on node degrees and edge orientations, which is crucial in the design of our algorithmic solutions. We then design and analyze a sequential algorithm for listing \( k \)-diamonds: we start from the simpler case of \( 4 \)-diamonds and later present an extension to \( k > 4 \). Our sequential algorithm computes the number of \( k \)-diamonds in \( O(m^{(k-1)/2}) \) time.

A parallel approach is then proposed, based on MapReduce, which is among the most restrictive distributed programming models at a large scale. Our algorithm lists the number of \( k \)-diamonds, without repetitions, using worst-case optimal \( O(m^{3/2}) \) total space. The work is \( O(m^{(k-2)/2}) \) and matches the running time of its sequential counterpart. Mappers and reducers use \( O(m) \) local space and their local running time is \( O(m^{k-2}/2) \). We remark that our parallel algorithm achieves the same local and total space usage of the state-of-the-art algorithms for listing \( k \)-cliques. Work and local running times are optimal on dense graphs, and \( O(\sqrt{m}) \) larger than the bounds for \( k \)-clique enumeration if the graph is sparse.

2. PRELIMINARIES

In this section, we introduce basic notation and graph-theoretical properties that will be used throughout the paper. We also briefly describe the programming model behind MapReduce, a widely used big data computing platform, and the performance metrics that we will consider in the analysis of MapReduce algorithms.

2.1. Graph-theoretical definitions

2.1.1. Graph terminology

Let \( G^a = (V^a, E^a) \) be a simple undirected graph without self loops. For each node \( u \in V \), let \( d(u) \) and \( \Gamma(u) \) denote its degree and its neighborhood in \( G^a \), respectively (\( u \) is not included). Given an integer \( k \geq 1 \), a \( k \)-clique is a complete subgraph on \( k \) nodes. A \( k \)-diamond is a clique of size \( k \) with one missing edge, and can be represented as two \( (k-1) \)-cliques with one \( (k-2) \)-clique in common. For instance, for \( k = 4 \), each \( 4 \)-diamond has four nodes and five edges, represented by two triangles (i.e. two \( (k-1) \)-cliques) with one common edge (i.e. a \( (k-2) \)-clique). Examples are given in Figure 1, where edges in one \( (k-1) \)-clique are bold, edges in the other \( (k-1) \)-clique are dashed, and the \( (k-2) \)-clique in their intersection is both bold and dashed.

2.1.2. Total order

We define a total order \( \prec \) over the nodes of \( G^a \) as follows:

\[ u \prec v \text{ iff } d(u) < d(v) \text{ or } d(u) = d(v) \text{ and } u < v \]

assuming nodes to have comparable and unique labels. Order \( \prec \) implicitly defines a directed graph \( G = (V, E) \) as follows: \( V = V^a \) and \( E = \{(u, v) \in E^a \text{ such that } u < v \in G^a\} \). Throughout the paper, we will denote with \( n \) and \( m \) the number of nodes and the number of edges in \( G \), respectively, i.e. \( n = |V| = |V^a| \) and \( m = |E| = |E^a| \). For each node \( u \in V \), \( \Gamma^+(u) \) denotes the high-neighborhood of \( u \), i.e. the set of neighbors \( v \) such that \( u \prec v \). Symmetrically, \( \Gamma^-(u) = \Gamma(u) \) and \( \Gamma^+(u) \) is the set of neighbors \( v \) of \( u \) such that \( u \prec v \).

A graph \( H \) is a subgraph of \( G \) if \( V(H) \subseteq V \) and \( E(H) \subseteq E \). \( H \) is an induced subgraph if, in addition to the above conditions, for each pair of nodes \( u, v \in V(H) \), it also holds \( (u, v) \in E \) if and only if \( (u, v) \in E \). We denote the subgraph induced by the neighborhood \( \Gamma(u) \) of a node \( u \) as \( G(u) \), and the subgraph induced by the high-neighborhood \( \Gamma^+(u) \) as \( G^+(u) \).

In this paper, we focus on subgraphs induced by subsets of \( k \) nodes that have the structure of a \( k \)-diamond.

In order to count diamonds in an undirected graph \( G^a \), we first compute the degree of its nodes and, based on the total order \( \prec \), the corresponding directed graph \( G \). In the remainder of the paper we will thus assume that \( G^a \) has been preprocessed as described above and will work directly on \( G \).

2.1.3. Two useful bounds

The following properties will be crucial in the analysis of our algorithms.

Lemma 1. [34] Let \( G \) be the directed graph with \( m \) edges obtained from an undirected graph \( G^a \) according to the total order \( < \). For each node \( u \in G \), \( |\Gamma^+(u)| \leq 2\sqrt{m} \).

Proof. We report the short proof for completeness. Let \( h \) be the number of nodes with degree larger than \( \sqrt{m} \), since there are \( m \) edges, it must be \( h \leq 2\sqrt{m} \). If \( d(u) > \sqrt{m} \), then nodes in \( \Gamma^+(u) \)
must also have degree larger than \( \sqrt{m} \) and their number is upper bounded by \( h \leq 2\sqrt{m} \). If \( d(u) \leq \sqrt{m} \), the claim trivially holds since \( \Gamma^+(u) \subseteq \Gamma(u) \).

As an example, Figure 2 shows an undirected graph \( G^u \) in which only node 1 has a high degree, equal to \( n - 1 \). When preprocessed, it turns out that node 1 has outdegree 0 in \( G \) because \( v < 1 \) for each other node \( v \). Intuitively, the preprocessing makes it possible to get rid in \( G \) of all nodes with very high degree in \( G^u \).

**Lemma 2.** Let \( n \) and \( m \) be, respectively, the numbers of nodes and edges of an undirected graph \( G^u \) without isolated nodes. Then it holds the following: 
\[
\sqrt{m}/2 \leq n \leq 2m.
\]

**Proof.** We separately prove the upper and the lower bound on \( n \).

Since there are no isolated nodes in \( G^u \), the degree of each node is at least one and thus \( n = \sum_{x \in V} 1 \leq \sum_{x \in V} d(x) = 2m \). Notice that this inequality is tight, as \( G^u \) is not required to be connected. Now consider the directed graph \( G \) obtained from \( G^u \) according to the total order. Notice that \( m = \sum_{x \in V} |\Gamma^+(x)| \) and that, for each node \( x \) in \( G \), \( |\Gamma^+(x)| \leq 2\sqrt{m} \) by Lemma 1. Hence, \( m = \sum_{x \in V} |\Gamma^+(x)| \leq \sum_{x \in V} 2\sqrt{m} = 2m\sqrt{m} \). This implies that \( n \geq \sqrt{m}/2 \).

### 2.2. MapReduce

Throughout this paper, we give parallel implementations of our algorithms based on MapReduce [35], which is among the oldest and most restricted distributed programming models. MapReduce offers programmers the possibility to easily run their code on large clusters while neglecting low-level issues related to scheduling, synchronization, communication and error detection. Many recent workflow systems, such as Apache Spark [36], extend MapReduce in several ways, offering programming primitives that are both more efficient and more flexible. We remark that our MapReduce-based algorithms also afford a natural implementation in such systems.

The programming model behind MapReduce consists of a sequence of parallel rounds. Each round is conceptually divided into a map and a reduce phase. Input/output values of a round, as well as intermediate data, are stored as (key, value) pairs. In the map phase, pairs are arbitrarily distributed among mappers and a programmer-defined map function is applied to each of them. Mappers are stateless and process each input pair independently from the others. During a shuffle phase, which is transparent to the programmer, the intermediate output pairs emitted by the mappers are grouped by key and all pairs with the same key are sent to the same reducer, that will process each of them by executing a programmer-defined reduce function. Parallelism comes from the concurrent execution of mappers as well as reducers.

Several theoretical abstractions for distributed programming frameworks à la MapReduce have been proposed in the literature [37–40]. All these models assume that the input data is distributed arbitrarily across various machines and that the computation occurs in rounds. The number of rounds (i.e., synchronization barriers) during the computation is a prominent metric in the theoretical analysis and should be constant, whenever possible (shuffling is indeed an expensive operation). Since both local computation and data communication are performed at each round, this will be taken into account in our analyses, considering the local memory available to each mapper/reducer, the global memory across all mappers/reducers (also called communication cost), the total work and the local running time. Notice that a local memory sub-linear with respect to the total input size allows us to exclude trivial algorithms that simply map the whole input to a single reducer, which then solves the problem via a sequential algorithm. We remark that analyses along these lines could be adapted to the Massively Parallel Computation model [40], which is popular in the database community.

### 3. LISTING DIAMONDS SEQUENTIALLY

In this section, we present our sequential algorithm for counting the number of \( k \)-diamonds in a graph with \( n \) nodes and \( m \) edges in \( O(nm(k^{-1/2}) \log k) \). Though we focus on counting, the algorithm can be easily adapted to the listing problem. Assuming that the input undirected graph has been preprocessed as described in Section 2, we first introduce a structural classification of \( k \)-diamonds based on node degrees and edge orientations in the directed graph \( G \) (Section 3.1). We then describe (Section 3.2) and analyze (Section 3.3) an algorithm to compute the number of \( k \)-diamonds. The extension to \( k > 4 \) is addressed in Section 3.4.

#### 3.1. Diamond classification

Given a diamond with node set \( D \), we hinge upon the total order \( \prec \) to decide which node is responsible for counting \( D \). In our sequential algorithm, each diamond is counted by the node \( x \in D \) such that \( x \prec y \) for all nodes \( y \in D \setminus x \).

Given a \( k \)-diamond \( D \) and a node \( v \in D \), we denote by \( d_{0}(v) \) its degree in \( D \), i.e., \( d_{0}(v) = |\Gamma^+(v) \cap D| \). Notice that \( d_{0}(v) \) must be either \( k - 1 \) or \( k - 2 \). In the remainder of this section, we also denote by \( x \) and \( y \) the two smallest nodes in \( D \), respectively, with \( x \prec y \). One of the three following conditions must hold, depending on the position of \( x \) and \( y \) in the diamond:

- Case 1: \( d_{0}(x) = k - 1 \);
- Case 2: \( d_{0}(x) = k - 2 \) and \( d_{0}(y) = k - 1 \);
- Case 3: \( d_{0}(x) = k - 2 \) and \( d_{0}(y) = k - 2 \).

A \( k \)-diamond \( D \) belongs to case 1 if and only if its smallest node \( x \) is adjacent to all the other \( k - 1 \) diamond nodes. Otherwise, it must necessarily be \( d_{0}(x) = k - 2 \), since a \( k \)-diamond only contains nodes of degree \( \geq k - 2 \). In the latter case (i.e. when there is a node in \( D \) that is not a neighbor of \( x \)), we use the second smallest node \( y \) to classify \( D \). Namely, \( D \) belongs to case 2 if and only if \( y \) is adjacent to all nodes in \( D \setminus \{y\} \), and thus \( x \) and \( y \) are also adjacent. Otherwise, if \( d_{0}(y) = k - 2 \), nodes \( x \) and \( y \) are the endpoints of the unique missing edge and the diamond belongs to case 3.

Example. Assuming \( k = 4 \), in a directed graph with edge orientations following the total order \( \prec \), there are six possible distinct 4-diamond types, which are classified as shown in Figure 3. The smallest node \( x \) of each diamond is highlighted in bold, while the second smallest node \( y \), if needed, is dashed. We call \( u \) and \( v \) the
3.2. Counting 4-diamonds

In this section, we present an algorithm that explores the high-neighborhoods of nodes to spot each 4-diamond, which is then counted—exactly once—by its smallest node. Consider the classification described in Section 3.1 and a diamond with node set \( D = \{x, y, u, v\} \) such that \( x \prec y \prec u \prec v \).

(i) If \( D \) belongs to case 1, we can find it by exploring only the high-neighborhood of \( x \), since \( y, u, v \in \Gamma^+(x) \).

(ii) Otherwise, we need to explore the high-neighborhoods of at least two nodes:

(a) If \( D \) belongs to case 2, \( y \in \Gamma^+(x) \) and \( u, v \in \Gamma^+(y) \). Therefore, we can find \( D \) by progressively exploring the high-neighborhoods of \( x \) and \( y \).

(b) If \( D \) belongs to case 3, nodes \( x \) and \( y \) are not adjacent. In this case we work by triangle augmentation: after computing all triangles \( \{x, u, v\} \) such that \( x \prec u \prec v \), we try to augment them with nodes \( y \in V \) such that \( x \prec y \), counting a diamond only if \( u, v \in \Gamma^+(y) \).

The pseudocode of the algorithm is shown in Figure 4. Notice that \( a, b, c \) and \( d \) can represent different nodes with respect to the configurations shown in Figure 3, depending on their position within the diamond. Node \( a \) always coincides with the smallest diamond node, being thus responsible for counting the diamond itself. The algorithm receives the directed graph \( G \), preprocessed as described in Section 2, and returns the number \( d_4 \) of 4-diamonds in \( G \), computed as follows:

- Diamonds from case 2 are counted at lines 3–7–10 (formal proof in Lemma 3). For each directed edge \( (a, b) \in E \), at line 8, we list all pairs of high-neighbors of \( b \), i.e., nodes \( c, d \in \Gamma^+(b) \), connected by an edge (as in the previous case, the edge orientation is not relevant). Assuming that \( (c, d) \in E^a \), we then count the diamond only if \( (a, d) \) exists in \( G \), but \( a \) and \( c \) are not connected. Notice that we check for the connectivity between \( a \) and \( d \) in \( E \), instead of \( E^a \), as \( a \prec b \) (line 7), \( b \prec d \) (line 8), and \( \prec \) is a transitive relation.

- Diamonds from case 3, where the two smallest nodes are not adjacent, are counted at lines 3, 7 and 11–15 (formal proof in Lemma 3). We first list all triangles \( \{a, b, c\} \subseteq G \) such that \( a \prec b \prec c \) (lines 3, 7, 11 and 12). Then, for each other graph node \( d \) such that \( a \prec d \), the diamond is counted only if \( d \) is not connected to \( a \), but has outgoing directed edges to both \( b \) and \( c \). With these checks, it is guaranteed that \( a \) and \( d \) are the pair of non-adjacent nodes (with \( a \prec d \)) as well as the smallest diamond nodes according to the total ordering.

3.3. Analysis

We first formally prove the correctness of the algorithm.

Lemma 3. Algorithm SeqD4 counts each 4-diamond exactly once.

Proof. Let \( D \) be a diamond with node set \( \{a, b, c, d\} \). The algorithm iterates over each node \( a \in V \) at line 3, exploring \( a \)'s high-neighborhood \( \Gamma^+(a) \) in order to identify and count 4-diamonds where \( a \) is the smallest node. In Table 1, we present the roles played by nodes \( \{a, b, c, d\} \) for each of the diamond types given in Section 3.1: notice that \( a \) always coincides with the smallest node \( x \). We now show that diamonds are listed by algorithm SeqD4 according to that classification:

(i) Lines 3–6: each triple of nodes \( \{b, c, d\} \subseteq \Gamma^+(a) \) such that \( (b, c) \in E^a \), \( (b, d) \in E^a \), and \( (c, d) \not\in E \) (test at line 5) corresponds to a diamond belonging to case 1. If the test succeeds, nodes \( a \) and \( b \) have indeed degree 3, while \( c \) and \( d \) are the endpoints of the unique missing edge. The checks performed by the algorithm guarantee that \( a \) is smallest node and that \( c \prec d \). Hence, depending on node \( b \), only three total orderings are possible between the four nodes: \( a \prec b \prec c \prec d \), \( a \prec c \prec b \prec d \) and \( a \prec c \prec d \prec b \). These orderings correspond to the three configurations 1a, 1b, and 1c shown in Figure 3 (see also the corresponding columns in Table 1).

(ii) Lines 7–10: each triple of nodes \( \{b, c, d\} \subseteq \Gamma^+(a) \) and \( (c, d) \in \Gamma^+(b) \) such that \( (a, d) \in E \), \( (c, d) \in E^a \) and \( (a, c) \not\in E^a \) (test at line 9) corresponds to a diamond belonging to case 2. Indeed, the checks guarantee that \( a \) is the smallest node \( x \), with degree 2 in the diamond, and \( b \) is the second smallest node \( y \), with degree 3 (see also Table 1). Hence, checking if \( (c, d) \in E^a \) corresponds to check if \( (c, d) \in E \) when \( c \prec d \), or \( (d, c) \in E \) when \( d \prec c \). The two total orderings \( a \prec b \prec c \prec d \) and \( a \prec b \prec d \prec c \) correspond to the two configurations 2d and 2e shown in Figure 3.
Figure 4. A sequential algorithm for counting (and listing) 4-diamonds.

Table 1. Roles played by nodes a, b, c and d used in the pseudocode of Figure 4 with respect to the six diamond types described in Figure 3, where x < y < u < v.

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1b</th>
<th>1c</th>
<th>2d</th>
<th>2e</th>
<th>3f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>b</td>
<td>y</td>
<td>u</td>
<td>u</td>
<td>y</td>
<td>y</td>
<td>u</td>
</tr>
<tr>
<td>c</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>u</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>d</td>
<td>v</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

(iii) Lines 11–15: each pair of nodes b ∈ Γ+(a) and c ∈ Γ+(a) \ Γ+(b) (see lines 7 and 11–12, respectively) corresponds to a triangle abc such that a < b < c. The second smallest diamond node is then chosen from set V \ \{a, b, c\} at line 13. A diamond is counted at line 15 only if b, c ∈ Γ+(d), but d is not a neighbor of a. These constraints yield a 4-diamond where only one total ordering is possible, i.e. a < d < b < c, which corresponds to configuration 3f in Figure 3. The test a < d at line 13 is needed to guarantee that the diamond is counted only once, i.e. from the point of view of its smallest node a.

Since each configuration shown in Figure 3 is considered exactly once by algorithm SeqD4 and any 4-diamond must belong to one of the three cases, the overall count is correct.

Theorem 1. Given a graph G with n nodes and m edges, algorithm SeqD4 correctly enumerates 4-diamonds of G in O(n^3/2) time.

Proof. We analyze the running time of algorithm SeqD4 by considering separately the three cases described in Section 3.1 and referring to Lemma 3 for its correctness:

- Case 1: the time spent to enumerate all nodes a ∈ V and all nodes b ∈ Γ+(a) at lines 3 and 4 is \(\sum_{a \in V} |\Gamma+(a)| = m\). Since c and d are also high-neighbors of a and |Γ+(a)| ≤ \(2\sqrt{m}\) by Lemma 1, the time to enumerate these node pairs at line 4 is \(O((\sqrt{m})^3) = O(m)\). Hence, counting all case 1 diamonds requires \(O(m^2)\) time, assuming that node adjacency tests at line 5 can be implemented in constant time (using, e.g. a hash table).

- Case 2: this is similar to case 1, except that c and d are high-neighbors of b and |Γ+(b)| ≤ \(2\sqrt{m}\) by Lemma 1. Overall, we still obtain \(O(m^2)\) time to list case 2 diamonds.

- Case 3: the algorithm first lists all triangles of G. The number of appropriate lists is counted at line 11 is \(\frac{1}{2} \binom{\sqrt{m}}{3} \cdot \sum_{a \in V} |\Gamma+(a)| = \frac{1}{2} \binom{\sqrt{m}}{3} \cdot m / \sqrt{m}\) by Lemma 1. Since the last node d is chosen from V among nodes larger than a in the total ordering, line 13 requires \(O(n)\) time. Hence, all diamonds from case 3 are counted in \(O(nm^{3/2})\) time.

Summing up the previous bounds, counting all case 1 and case 2 diamonds requires time \(O(m^2)\). In total, the running time is \(O(n^2m^{3/2})\), because \(n = \Omega(\sqrt{m})\) by Lemma 2.

Note that the \(O(m^2)\) bound achieved in cases 1 and 2 matches the worst-case running time for k-clique enumeration [23]. However, the running time of algorithm SeqD4 is dominated by case 3: while optimal on dense graphs, the bound is \(O(\sqrt{m})\) larger than k-clique counting if the graph is sparse.

3.4. Extension to k-diamonds

Algorithm SeqDk, described in this section, generalizes the approach of Section 3.2 to the enumeration of k-diamonds, for any \(k \geq 4\). Only a few changes to algorithm SeqD4 are sufficient to deal with larger values of k. We refer to Figure 4 to describe such changes. Consider a k-diamond D with smallest node x:

(i) In case 1, \(d_0(x) = k-1\). Once the smallest node a = x has been fixed, the algorithm lists all subsets \(S \subseteq \Gamma+(a)\) of size \(k-1\) (instead of triples as in line 4 of Figure 4). The check at line 6 is changed so that a diamond is counted only if S contains a (k - 3)-clique Q and a pair of non-adjacent nodes c, d ∈ S\{Q\} such that c and d are adjacent to all nodes in Q and c < d.

(ii) In case 2, \(d_0(x) = k-2\), while the second smallest node y ∈ \(\Gamma+(x)\) has degree k − 1. After listing all directed edges (a, b) ∈ E (lines 3 and 7), for each \(\Gamma+(x)\) ∈ \(\Gamma+(y)\) the algorithm counts a diamond at line 10 only if there is exactly one node in Q not adjacent to a.

(iii) In case 3, \(d_0(x) = d_2(y) = k-2\) and the two smallest nodes x and y are thus not adjacent. The algorithm finds D by computing a (k − 2)-clique Q in \(\Gamma+(y)\) as follows: after fixing a node a ∈ V (line 3), at lines 7 and 11 it lists all (k − 2)-cliques Q in \(\Gamma+(x)\) and \(\Gamma+(y)\) as follows. Then, for each node \(d \in V \setminus Q\{u\}\), such that a < d, the diamond is counted at line 15 only if Q \(\subseteq \Gamma+(d)\).

Theorem 2. Given a graph G with n nodes and m edges, algorithm SeqDk enumerates all k-diamonds of G in \(O(nm^{k-3/2})\) time.
Proof. We separately consider the three cases of Section 3.1, proving that algorithm SeqDK enumerates diamonds from cases 1 and 2 in $O(m^{3/2})$ time, and diamonds from case 3 in $O(nm^{(k-1)/2})$ time:

(i) Case 1: the time spent to enumerate all nodes $a \in V$ and their neighborhood subsets of size $k-1$ is $\sum_{\alpha \in \Gamma^+(a)} (\Gamma^+(a))^{-1} = \sum_{\alpha \in \Gamma^+(a)} |\Gamma^+(a)|^{-1} = O(nm^{k-1})$, where we used the fact that $|\Gamma^+(a)| \leq 2\sqrt{m}$ by Lemma 1 and that $|\Gamma^+(a)| = m$.

(ii) Case 2: the time required for listing all directed edges $(a,b)$ is $O(m)$ and the time to enumerate subsets of $\Gamma^+(b)$ of size $k-2$ is $\binom{m}{k-2}$, which is at most $(2\sqrt{m})^{k-2} = O(m^{k-2})$ by Lemma 1. In total, this yields $O(m \cdot m^{k-2/2}) = O(m^{k/2})$ time.

(iii) Case 3: the algorithm lists all nodes $a \in V$ and cliques $Q \in \Gamma^+(a)$ of size $k-2$ in $\sum_{\alpha \in \Gamma^+(a)} (\Gamma^+(a))^{-1} = O(nm^{k-1/2})$ time. All nodes $d \in V$ are listed in $O(n)$ time. Hence, the running time for listing all diamonds from case 3 is $O(nm^{k-1/2})$.

Since $n = \Omega(\sqrt{m})$ by Lemma 2, the running time is dominated by the bound of case 3 (in more details, algorithm SeqDK requires in total $O(m^{3/2})$ and $O(nm^{k-1/2})$ time on dense and sparse graphs, respectively).

The correctness follows by extending the arguments given in Lemma 3 to values of $k \geq 4$. Indeed, although the number of valid edge orientations between the nodes of a $k$-diamond (i.e. the number of possibilities akin to the configurations shown in Figure 3) increases as a function of $k$, the classification into three main cases, being solely based on the degree of the two smallest diamond nodes, remains valid. Algorithm SeqDK, similarly to SeqD4, exploits this diamond classification and the total order $\prec$ to identify a unique node responsible for counting each $k$-diamond, which is thus taken into account exactly once.

4. A parallel implementation

In this section, we adapt algorithms SeqD4 and SeqDK to a parallel setting. We use a vertex-centric approach, splitting the input undirected graph $G^v$ into many induced subgraphs $G(x)$, for each $x$, and counting diamonds in each subgraph independently. Similarly to the sequential approach, the parallel algorithms use the total order $\prec$ and the diamond classification of Section 3.1 to decide which node is responsible for counting a $k$-diamond.

At a high level, our strategy consists of listing all triangles in graph $G^v$ exactly once and use them to create induced subgraphs $G(x)$ for each node $x \in V^v$. Thus, we compute locally the number of diamonds for which $x$ is responsible. We begin by setting up the computation in Section 4.1. This procedure will be used by our MapReduce algorithms as a first step for counting diamonds. For a better comprehension, we first focus on 4-diamonds in Section 4.2. Then, we present an algorithm for counting $k$-diamonds in Section 4.3.

4.1. Setting up the computation: triangle listing

The computation of triangles has been widely studied in the literature. For the sake of completeness, we give in Figure 5 pseudocode of Algorithm ListTri, which is based on results described in [26, 27]. Triangles can be listed in two rounds.

4.1.1. Round 1. High-neighborhood computation

Considering that the input graph $G^v = (V^v, E^v)$ is undirected, we apply the total order $\prec$ over the nodes $a \in V^v$ to compute the high-neighborhood of each node. For each edge $(a, b)$, mappers emit the pair $(a, b)$ only if $a < b$, allowing the reducer with key $a$ to aggregate all nodes $b \in \Gamma^+(a)$.

4.1.2. Round 2. Finding triangles

In this round we discover all triangles by exploring the high-neighborhood of nodes. Map instance with input $(a; \Gamma^+(a))$ emits a pair $(b, c; a)$ for each pair of nodes $b, c \in \Gamma^+(a)$ such that $b < c$. Besides the output of round 1, mappers in round 2 are fed with the original set of edges and emit a pair $(a, b, c)$ for each edge $(a, b, c) \in E^v$ such that $a < b$. Hence, reduce instance with key $(b, c)$ can check whether $(b, c)$ is an edge by looking for the symbol $\$ among its value. Furthermore, this instance receive the set of nodes $a$ such that $a < b < c$, where each node $a$ completes a triangle together with nodes $b$ and $c$.

Akin to the analyses in [26, 27], the performance of Algorithm ListTri can be summarized as follows:

Lemma 4. Consider a graph $G$ with $n$ nodes and $m$ edges. Algorithm ListTri enumerates all triangles in $G$ exactly once using $O(m^{3/2})$ total space and $O(m^{3/2})$ work.

Mappers and reducers use $O(m)$ local space, and their local running time is $O(m)$.

Proof. The total space usage in round 1 is $O(m)$. Map 2 instances produce key-value pairs of constant size, whose total number is upper bounded by $\binom{V^v}{3}$, which is at most $2\sqrt{m} \cdot \binom{\sqrt{m}}{3} = O(m^{3/2})$ by Lemma 1. Reduce 2 instances emit all triangles in $G^v$, using $O(m^{3/2})$ space. Hence, the total space usage of ListTri is $O(m^{3/2})$. Focus on local space. Map 1 instances use constant memory. The input of any reduce 1 instance is $O(\sqrt{m})$ by Lemma 1, as well as, any map 2 instance receives $O(\sqrt{m})$ input edges. Consider a reduce 2 instance with key $(b, c)$. The input of this instance is the set of nodes $a \in V^v$ such that $a \in \Gamma^-(b) \cap \Gamma^+(c)$, which is at most $O(n)$. Since $n \leq O(\sqrt{m})$ by Lemma 2, the local space usage of mappers and reducers is $O(n)$.

Following the same reasoning, the running time of each map instance in the two rounds is $O(1)$ and $O(m)$, respectively, while reduce instances require $O(\sqrt{m})$ and $O(n)$ time. The total work of Round 1 is $O(m)$. The cost of the mappers on round 2 is $\sum_{\alpha \in \Gamma^+(a)} (\Gamma^+(a))^{-1} = O(m^{3/2})$. Since the number of triangles in $G^v$ is upper bounded by $\sum_{\alpha \in \Gamma^+(a)} (\Gamma^+(a))$, the total work of reduce 2 remains $O(m^{3/2})$, concluding the proof of the total running time claim.

The algorithm enumerates triangles by computing all pair of nodes $b, c \in \Gamma^+(a)$ for each node $a \in V^v$. Moreover, a triangle $a, b, c$ is listed only if $a < b < c$ and $(a, b, c) \in E^v$, ensuring that each triangle is enumerated exactly once.

4.2. Listing 4-diamonds in triangle space

As described in Section 2, a 4-diamond can be represented by two triangles with one common edge (see Figure 1b). Based on that, we introduce an algorithm, called ParD4, to compute all 4-diamonds in parallel using triangle space, i.e. total space $O(m^{3/2})$. In this approach, the ListTri algorithm is used to enumerate all triangles in the first two rounds. Those triangles are needed to reconstruct the induced subgraphs in round 3. Consider a diamond $D = \{x, y, u, v\}$ such that $x < y < u < v$. According to the diamond classification described in Section 3.1, the node responsible for counting $D$ is defined as follows:

- Case 1: the smallest node $x$ is responsible for counting $D$.
- Case 2: the second smallest node $y$ is responsible for counting $D$. 


ListTri (undirected graph $G^u = (V^u, E^u)$)

Map 1: input $\langle (a, b); \emptyset \rangle$
1: if $a < b$ then
2: emit $(a; b)$

Reduce 1: input $\langle a; \Gamma^+(a) \rangle$
1: if $|\Gamma^+(a)| > 2$ then
2: emit $(a; \Gamma^+(a))$

Map 2: input $\langle a; \Gamma^+(a) \rangle$ or $\langle (a, b); \emptyset \rangle$
1: if input of type $\langle (a, b); \emptyset \rangle$ and $a < b$ then
2: emit $\langle (a, b); b \rangle$
3: if input of type $\langle (a, \Gamma^+(a)) \rangle$ then
4: for each $b, c \in \Gamma^+(a)$ s.t. $b < c$ do
5: emit $\langle (b, c); a \rangle$

Reduce 2: input $\langle (b, c); \{a_1, \ldots, a_t\} \cup \emptyset \rangle$
1: if input contains $\emptyset$ then
2: emit $\langle (b, c); \{a_1, \ldots, a_t\} \rangle$

Map 3: input $\langle (b, c); \{a_1, \ldots, a_t\} \rangle$
for each $a \in \{a_1, \ldots, a_t\}$ do
3: emit $(a; (b, c)), (b; (a, c)), (c; (a, b))$

Reduce 3: input $\langle a; G(a) \rangle$
4: $d_4 \leftarrow 0$
5: Let $\Gamma^+(u)$ be the set of nodes $u \in G(a)$ s.t. $a < u$
6: for each $(b, c, d) \subseteq \Gamma^+(u)$ s.t. $c < d$ do
7: if $(b, c), (b, d) \in E^u$ and $(c, d) \notin E$ then
8: $d_4 \leftarrow d_4 + 1$
9: $\triangleright$ Case 1

for each $(b, c) \in G(a)$ s.t. $b < a < c$ do
for each $d \in \Gamma^+(a) \setminus \{c\}$ do
10: if $(c, d) \in E^u$ and $(b, d) \notin E$ then
11: $d_4 \leftarrow d_4 + 1$
12: $\triangleright$ Case 2

for each $d \in G(a) \setminus \{b\}$ s.t. $b < d < a$ do
13: if $(d, c) \in E^u$ and $(b, d) \notin E$ then
14: $d_4 \leftarrow d_4 + 1$
15: $\triangleright$ Case 3
16: emit $(a; d_4)$

Figure 5. Listing triangles in MapReduce.

Figure 6. MapReduce code for counting 4-diamonds in triangle space.

Table 2. Roles played by nodes $a, b, c, d$ in Figure 6 according to the six diamond types described in Figure 3.

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1b</th>
<th>1c</th>
<th>2d</th>
<th>2e</th>
<th>3f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>y</td>
<td>y</td>
<td>u</td>
</tr>
<tr>
<td>b</td>
<td>y</td>
<td>u</td>
<td>u</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>c</td>
<td>u</td>
<td>y</td>
<td>y</td>
<td>v</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>d</td>
<td>v</td>
<td>v</td>
<td>u</td>
<td>u</td>
<td>v</td>
<td>y</td>
</tr>
</tbody>
</table>

- Case 3: the third smallest node $u$ is responsible for counting $D$.

The pseudocode of round 3 of Algorithm ParD4 is given in Figure 6. Nodes $a, b, c$ and $d$ in the reduce function can represent different nodes from Figure 3, according to the 4-diamond classification. The roles played by $a, b, c$ and $d$ for each 4-diamond configuration are given in Table 2. Notice that node $a$ is responsible for counting the 4-diamond in the reduce function of Figure 6 and plays the role of $x$ in case 1, $y$ in case 2 and $u$ in case 3.

The strategy of ParD4 is to count all 4-diamonds in each induced subgraph $G(a)$ for each $a \in V^u$. This is implemented as follows. All triangles in $G^u$ are listed in rounds 1 and 2 using Algorithm ListTri. Consider a map 3 instance with key $(b, c)$. This instance receives a list of nodes $a \in V$ where each node $a$ completes a triangle with edge $(b, c)$. For each triangle $abc$ such that $a < b < c$, the instance emits three tuples $(a; (b, c)), (b; (a, c))$ and $(c; (a, b))$. After shuffling, the reduce 3 instance with key $a$ receives as input a list of edges between the nodes in $\Gamma(a)$, i.e. the subgraph $G(a)$ induced by the neighborhood of $a$. Akin to our sequential algorithm, reduce instances count each diamond according to its classification, computing locally the number of diamonds for which $a$ is responsible. Notice that $G(a)$ is composed of nodes in $\Gamma(a) = \Gamma^+(a) \cup \Gamma^-(a)$. However, $G(a)$ is a directed graph, i.e. given a pair of nodes $b, c \in \Gamma(a)$, $(b, c)$ is an edge of $G(a)$ only if $(b, c) \in E^u$ and $b < c$.

We now analyze Algorithm ParD4.

Lemma 5. Each 4-diamond is counted exactly once by algorithm ParD4.

Proof. For each triangle listed in the first two rounds, map 3 instances emit all possible combinations of a node and an edge, creating three tuples. The reduce 3 instance with key $a$ receives a set of edges $(b, c)$ such that $b \in \Gamma(a), c \in \Gamma(a), b < c$ and $(b, c)$ is an edge in $E^u$. This set of edges represents the directed induced subgraph $G(a)$.

Each 4-diamond is counted by the reduce function according to the classification described in Section 3.1. Considering a diamond $D$ with node set $\{a, b, c, d\}$, we separately analyze the correctness of Algorithm ParD4 in the three cases:

(i) Case 1: each triple of nodes $b, c$ and $d$ selected in $\Gamma^+(a)$ such that $c < d$ represents a possible 4-diamond $D$ belonging to case 1, where the smallest node $a$ in $D$ has degree 3. If the checks $(b, c), (b, d) \in E^u$ and $(c, d) \notin E$ succeed, node $b$ is the second smallest node with $d_0(b) = 3$, while nodes $c$ and $d$ are...
the pair of non-adjacent nodes in the diamond. Considering that \( c < d \) and \( a < b, c, d \), only three total orderings are possible between the four nodes: \( a < b < c < d \), \( a < c < b < d \) and \( a < c < d < b \). It is not difficult to see that these orderings correspond to the three configurations 1a, 1b and 1c shown in Figure 3. Notice that, although \( G(a) \) is a directed graph, the test \((b, c), (b, d) \in E^d \) implies the undirected edges \( E^d \). Hence, checking if \((b, c) \in E^d \) corresponds to check if \((b, c) \in E \) when \( b < c \), or \((c, b) \in E \) when \( c < b \) (the checks on \((b, d) \) are similar).

(ii) Case 2: each pair of nodes \( b, c \in G(a) \) such that \( b < a < c \) corresponds to a triangle \( b \hat{c} a \). Each node \( d \in \Gamma^+(c) \cap \Gamma^+(a) \) together with triangle \( b \hat{c} a \) can correspond to a 4-diamond \( D \) belonging to case 2. Hence, in \( D \), the smallest node \( b \) has degree 2 and the second smallest node \( a \) has degree 3. The checks \((c, d) \in E^a \) and \((b, d) \notin E \) guarantee that \( b < a < c, d \) and nodes \( b \) and \( d \) are the endpoints of the unique missing edge. Considering the total order between nodes \( c \) and \( d \), it is not difficult to see that the two possible orderings \( b < a < c < d \) and \( b < a < d < c \) correspond to the two configurations 2d and 2e shown in Figure 3.

(iii) Case 3: as in case 2, each pair of nodes \( b, c \in G(a) \) such that \( b < a < c \) corresponds to a triangle \( b \hat{c} a \). Node \( d \) is then chosen in \( G(a) \setminus \{b\} \) so that \( b < d < a \). Hence, in \( D \), the smallest node \( b \) has degree 2, as well as the second smallest node \( d \). In this case, the third smallest node \( a \) is responsible for counting the 4-diamond. The checks \((d, c) \in E \) and \((b, d) \notin E \) guarantee that \( b < d < a < c \) and nodes \( b \) and \( d \) are the endpoints of the unique missing edge. We note that the test \( b < d < a \) is needed to ensure that the diamond is counted only once.

This proves the correctness of the algorithm. \( \square \)

**Theorem 3.** Given an undirected graph \( G^a \) with \( n \) nodes and \( m \) edges, algorithm ParD4 correctly enumerates 4-diamonds in \( G^a \) using \( O(m^{1/2}) \) total space and \( O(mn^{1/2}) \) work. Mappers and reducers use \( O(m) \) local space, and their local running time is \( O(nm) \).

**Proof.** The correctness of the algorithm follows by Lemma 5. Rounds 1 and 2 are analyzed in Lemma 4 and are compliant with the bounds in the theorem. We now focus on space and running time of round 3.

Total space. Round 2 emits the triangles in the input graph and thus map 3 instances use \( O(m^{1/2}) \) space (see Lemma 4). Each triangle is then replicated by the map 3 function. Hence, reduce 3 instances also use \( O(m^{1/2}) \) space in total.

Local space. Map 3 instances use \( O(n) \) local space. Considering that the subgraph \( G(a) \) can be represented by a list of edges, any reduce 3 instance uses at most \( O(m) \) space. Thus, the local space of mappers and reducers is \( O(m) \) (recall that \( n \leq 2m \) by Lemma 2).

Local running time. Each map 3 instance requires \( O(nm) \) time. We now separately analyze the running time of reduce 3 instances in the three cases of Section 3.1. Consider a reduce 3 instance with key \( a \). In case 1, 4-diamonds are counted in \( O(m^{3/2}) \) time, since \( \{b, c, d \} \in \Gamma^+(a) \) and \( \Gamma^+(a) \leq 2 \sqrt{m} \) by Lemma 1. For 4-diamonds from case 2, the reduce function lists all edges \((b, c) \in G(a) \) in \( O(m) \) time and all nodes \( d \in \Gamma^+(a) \) in \( O(\sqrt{m}) \) time. The overall running time is therefore \( O(m^{3/2}) \). In case 3, edges \((b, c) \in G(a) \) are enumerated in \( O(m) \) time, while nodes \( d \in G(a) \) such that \( d < a \) are listed in \( O(n) \) time. Hence, 4-diamonds from case 3 are counted in \( O(mn) \) time. Case 3 dominates the local running time of reduce 3 instances, since \( n = \Omega(\sqrt{m}) \) by Lemma 2. Therefore, the local running time of the mappers and reducers is \( O(nm) \), assuming that node adjacency tests can be implemented in constant time.

Work. Map 3 instances spend constant time per triangle and thus require \( O(m^{3/2}) \) time in total. We now separately consider the three cases of Section 3.1 to analyze the work of reduce 3 functions. In case 1, 4-diamonds can be counted in time proportional to \( \sum_\nu \gamma(\nu) \). By Lemma 1, this is at most \( (2 \sqrt{m})^2 \sum_\nu \gamma(\nu) = O(m^2) \). Since edges in \( G(a) \) are at most \( m \), 4-diamonds from case 2 are counted in \( O(m \sum_\nu \delta(\nu)) = O(m^2) \). In case 3, the reduce function lists all triangles \( b \hat{a} c \) in \( O(m^{3/2}) \) time, and, for each node \( d \in G(a) \), checks in constant time whether edge \((c, d) \) exists and nodes \( b \) and \( d \) are not adjacent. Therefore, 4-diamonds from case 3 can be counted in \( O(nm^{3/2}) \) time, dominating the total work of algorithm ParD4.

### 4.3. Extension to k-diamonds

In this section, we describe an extension of the algorithm ParD4, called ParDk, to k-diamonds, for \( k \geq 4 \). This approach computes all k-diamonds using \( O(m^{1/2}) \) space and \( O(mn^{1/2}) \) work, matching the sequential running time of algorithm SeqDk (see Section 3.4).

We modify algorithm ParD4 to compute all k-diamonds in each of the three distinct cases of Section 3.1. We remark that, although the number of diamond configurations shown in Figure 3 would increase as a function of \( k \), the classification into three main cases remains valid.

With respect to algorithm ParD4, the only changes to compute k-diamonds are in the reduce 3 function, whose pseudocode is given in Figure 7. This function, for each key \( a \in V^a \), is given the induced subgraph \( G(a) \). Similarly to ParD4, it counts each k-diamond in \( G(a) \) according to its classification. The responsible node for counting a k-diamond follows the same rules of algorithm ParD4. In the pseudocode, node \( a \) always plays the role of the node responsible for counting and is the smallest node of a k-diamond in case 1, the second smallest in case 2 and the third smallest in case 3. As shown in Figure 7, algorithm ParDk hinges upon the enumeration of t-cliques for \( t = k - 3 \) and \( k - 4 \).

**Lemma 6.** Each k-diamond is counted exactly once by algorithm ParDk.

**Proof.** By Lemma 4 and Theorem 3, the first two rounds correctly enumerate all the triangles in \( G^a \) and map 3 emits each triangle using three different keys. The reduce 3 instance with key \( a \) receives the directed induced graph \( G(a) \), i.e., the set of edges \((b, c) \) such that \( b < c \), nodes \( b \) and \( c \) are in \( \Gamma(a) \) and \( (b, c) \) is an edge in \( E^a \).

Let \( Q_k \) denote a t-clique with nodes \( \{u_1, \ldots, u_t\} \) such that it holds \( u_1 < u_2 < \ldots < u_t \). As shown in Figure 7, each clique \( Q_k \) is enumerated exactly once according to the total order \( < \). We now consider a k-diamond \( D \) and analyze the three cases described in Section 3.1.

(i) Case 1: each pair of nodes \( b, d \in \Gamma^+(a) \) such that \( b < d \) together with a clique \( Q_k \subseteq \Gamma^+(a) \setminus \{b, d\} \) correspond to a possible k-diamond belonging to case 1, where its smallest node \( a \) has degree \( k - 1 \) in \( D \). If the checks \( Q_k \in \Gamma(b) \cap \Gamma(d) \) and \( (b, d) \notin E \) succeed, node \( a \) and all nodes in \( Q_k \) have indeed degree \( k - 1 \), while nodes \( b \) and \( d \) are the endpoints of the unique missing edge. Hence, it is not difficult to see that each ordering between the pair of non-adjacent nodes and the nodes in \( Q_k \) corresponds to a unique k-diamond configuration from case 1.
PARDK (undirected graph \( G^u = (V^u, E^u) \), diamond size \( k \))

**Reduce 3:** input \( \langle a; G(a) \rangle \)

1. \( d_k \leftarrow 0 \)
2. Let \( \Gamma^+(a) \) be the set of nodes \( u \in G(a) \) s.t. \( a \prec u \)
3. \( t \leftarrow k - 3 \)
4. for each \( b, d \in \Gamma^+(a) \) s.t. \( b < d \)
5. \( \quad \) for each \( Q_t \subseteq \Gamma^+(a) \setminus \{b, d\} \)
6. \( \quad \) if \( Q_t \subseteq \Gamma^+(b) \cap \Gamma^+(d) \) and \( (b, d) \not\in E \) then
7. \( \quad \quad d_k \leftarrow d_k + 1 \) \( \text{Case 1} \)
8. \( \quad t \leftarrow k - 4 \)
9. for each \( Q_t \subseteq \Gamma^+(a) \)
10. \( \quad \) for each \( (b, c) \in G(a) \setminus Q_t \) s.t. \( b < a < c \) do
11. \( \quad \quad \) if \( Q_t \subseteq \Gamma^+(b) \cap \Gamma^+(c) \) then
12. \( \quad \quad \quad \) for each \( d \in \Gamma^+(a) \setminus \{Q_t \cup \{c\} \} \)
13. \( \quad \quad \quad \quad \) if \( \{c\} \cup Q_t \subseteq \Gamma^+(d) \) and \( (b, d) \not\in E \)
14. \( \quad \quad \quad \text{then} \)
15. \( \quad \quad \quad \quad d_k \leftarrow d_k + 1 \) \( \text{Case 2} \)
16. \( \quad \quad \) for each \( d \in G(a) \setminus \{b\} \) s.t. \( b < d < a \)
17. \( \quad \quad \quad \) if \( \{c\} \cup Q_t \subseteq \Gamma^+(d) \) and \( (b, d) \not\in E \)
18. \( \quad \quad \text{then} \)
19. \( \quad \quad d_k \leftarrow d_k + 1 \) \( \text{Case 3} \)
20. emit \( \langle a; d_k \rangle \)

**Figure 7.** Reduce 3 code for counting \( k \)-diamonds in triangle space.

(ii) Case 2: each clique \( Q_t \) of size \( k - 4 \) together with a pair of nodes \( b, c \in G(a); Q_t \) such that \( b < a < c \) and a node \( d \in \Gamma^+(a) \setminus (Q_t \cup \{b, c\}) \) correspond to a possible \( k \)-diamond belonging to case 2. In this case, the smallest node \( b \) has \( d_0(b) = k - 2 \), while the second smallest node \( a \) has \( d_0(a) = k - 1 \). The check \( Q_t \subseteq \Gamma^+(b) \cap \Gamma^+(c) \) guarantees that all nodes in \( Q_t \) are high-neighbors of nodes \( b, c \). At this point, a clique of size \( k - 1 \) composed of nodes \( b, a, c \cup Q_t \) have been listed. A \( k \)-diamond is counted only if \( (b, d) \) is not an edge and all nodes in \( Q_t \cup \{c\} \) are adjacent to node \( d \). Hence, each total order between node \( d \) and nodes \( Q_t \cup \{c\} \) corresponds to a unique \( k \)-diamond configuration from case 2.

(iii) Case 3: the enumeration of \( k \)-cliques from case 3 is very similar to case 2, except for the enumeration of the last node \( d \) in the \( k \)-diamond. In more details, each clique \( Q_t \) of size \( k - 4 \) together with a pair of nodes \( b, c \in G(a); Q_t \) such that \( b < a < c \) corresponds to a clique of size \( k - 1 \), composed of nodes \( b, a, c \cup Q_t \). The last diamond node \( d \) is selected in \( G(a) \setminus \{b, c\} \) such that \( b < d < a \). A \( k \)-diamond is counted only if \( (b, d) \) is not an edge and all nodes in \( Q_t \cup \{c\} \) are in the high-neighborhood of \( d \). Overall, the smallest node \( b \) has degree \( k - 2 \), as well as the second smallest node \( d \). Only one diamond configuration can exist in case 3, where nodes \( b, d \) are the endpoints of the unique missing edge and the total order over the diamond nodes is \( b < d < a < c < Q_t \).

**Theorem 4.** Let \( G^u \) be an undirected graph with \( n \) nodes and \( m \) edges. The algorithm PARDK counts the number of \( k \)-diamonds in \( G^u \) exactly once using \( O(m^{k/2}) \) total space and \( O(nm^{k/2-1/2}) \) work. Mappers and reducers use \( O(m) \) local space, and their local running time is \( O(nm^{k/2-1/2}) \).

**Proof.** The analysis of the first two rounds, of the map 3 function, and of the global and local space of the reduce 3 function is the same as in Theorem 3. Hence, we focus on the running time of reduce 3 instances and separately analyze the three cases introduced in Section 3.1.

**Local running time.** Consider a reduce 3 instance with key \( a \). Using an efficient set intersection algorithm, the enumeration of \( (k - 3) \)-cliques in case 1 can be done in time upper bounded by \( \sum_{k, v \in \Gamma^+(v)} (\Gamma^{(k-4)}(v) \cdot 2^{(k-4)2}) \), which is proportional to \( (\Gamma^{(k-4)}(v) \cdot 2^{(k-4)2}) \) by Lemma 1. Similarly, in case 2, cliques \( Q_t \subseteq \Gamma^+(a) \) of size \( k - 4 \) can be listed in \( O(|\Gamma^{(k-4)}(a)|) \) time. Edges \( (b, c) \in G(a) \) and nodes \( d \in \Gamma^+(a) \) can be listed in \( O(m) \) and \( O(\sqrt{m}) \) time, respectively. Hence, the local running time for counting \( k \)-diamonds belonging to case 2 is \( O(m^{k/2} \cdot m \cdot \sqrt{m}) = O(m^{k/2} \cdot \sqrt{m}) \). The only difference between cases 2 and 3 is in the enumeration of the last node \( d \in G(a) \) such that \( d < a \). Which requires \( O(n) \) time. Therefore, \( k \)-diamonds classified as case 3 can be counted in \( O(m^{k/2} \cdot m \cdot n) = O(nm^{k-2/2} \cdot \sqrt{m}) \), dominating the local running time of algorithm PARDK.

**Work.** Applying similar analyses, \( k \)-diamonds from case 1 can be counted in \( \sum_{v \in \Gamma^+(v)} (\Gamma^{(k-4)}(v) \cdot 2^{(k-4)2}) \), which is at most \( O(m^{k/2} \cdot m \cdot \sqrt{m}) = O(m^{k/2} \cdot \sqrt{m}) \) by Lemma 1. In case 2, the total work is \( O(m^{k/2}) \): cliques \( Q_t \subseteq \Gamma^+(a) \), edges \( (b, c) \), and nodes \( d \in \Gamma^+(a) \) can be listed in time \( \sum_{v \in \Gamma^+(v)} (\Gamma^{(k-4)}(v) \cdot 2^{(k-4)2}) \), \( O(m) \), and \( O(\sqrt{m}) \), respectively. By similar arguments, in case 3 cliques \( Q_t \), edges \( (b, c) \), and nodes \( d \) such that \( d < a \) can be listed in \( O(m^{k/2} \cdot \sqrt{m}) \), \( O(m) \), and \( O(n) \) time, respectively. Hence, the work for enumerating all \( k \)-diamonds in case 3 is \( O(m^{k/2} \cdot m \cdot n) = O(nm^{k-2/2}) \), dominating the total work of algorithm PARDK.

**5. CONCLUDING REMARKS**

Motivated by the fact that cliques require full interconnections between nodes and may be too restrictive a model in many real applications, the article introduces and studies the computation of \( k \)-diamonds, a natural and simple relaxation of cliques of size \( k \). We address the enumeration of \( k \)-diamonds both in a sequential and in a parallel setting. We first provide a sequential algorithm for counting the number of \( k \)-diamonds in large graphs, for any \( k \geq 4 \), in \( O(nm^{k-2/2}) \) time. A parallel extension of the sequential algorithm is then proposed, developing a MapReduce-based approach. The algorithm lists the \( k \)-diamonds without repetitions using \( O(m^{k/2}) \) total space and \( O(m^{k-2/2}) \) work. Mappers and reducers use \( O(m) \) local space, and their local running time is \( O(nm^{k-2/2}) \).

Different lines of research remain open from this work. First, both the sequential and parallel algorithms use \( O(\sqrt{m}) \) extra time with respect to the clique-counting problem if the graph is sparse. It would be interesting to understand whether this can...
be improved, matching the bound for k-cliques in all cases. As a second direction, the k-diamonds subgraphs could be further generalized to (k,h)-diamonds: k-cliques with h missing edges, for small values of h. The design and analysis of sequential and parallel algorithms for this case would be much interesting.

Additionally, akin to triangle counting that has been extensively studied in different settings (see, e.g. [41-43] and the references therein), it may be worthwhile to consider the diamond enumeration problem in similar models of computation, such as external memory, streaming, and GPU computing.

Last but not the least, we believe that implementing, engineering and experimentally analyzing the performance of our algorithms, especially on real-world graphs, would be also very useful.

ACKNOWLEDGEMENTS

This work has been partially supported by MUR, the Italian Ministry of University and Research, under PRIN Project n. 2022M5Z778 “NextGRAAL-Next-generation algorithms for constrained GRAPh visuALization”.

DATA AVAILABILITY

No new data were generated or analysed in support of this research.

REFERENCES