Enumerating cliques and their relaxations: sequential and parallel algorithms

PhD School of the Department of Computer Science
at Sapienza - University of Rome.

Dottorato di Ricerca in Informatica – XXX Ciclo

Candidate

Renan Leon Garcia
ID number 1688741

Thesis Advisor

Prof. Irene Finocchi

Dedicated to
my father

Abstract

In this thesis we study subgraph enumeration problems. We provide an extensive literature review of subgraph enumeration, considering different problems associated to cliques, clique relaxations, and other kind of subgraphs. We then devise algorithms for the problems of enumerating k-cliques (i.e., complete subgraphs on k nodes) and one of their relaxations, called k-diamonds (i.e., cliques of size k with one missing edge).

For the first problem we present simple and fast multicore parallel algorithms for counting the number of k-cliques in large undirected graphs, for any small constant $k \geq 4$. Clique counting is important in a variety of network analytics applications. Differently from existing solutions, which mainly target distributed memory settings (e.g., MapReduce), the proposed algorithms work on off-the-shelf shared-memory multicore platforms.

The effectiveness of our approaches is assessed through an extensive experimental analysis on a variety of real-world graphs, considering different clique sizes and scalability on different numbers of cores. The experimental results show that the proposed parallel algorithms largely outperform the running times of the highly optimized sequential solution and gracefully scale to non-trivial values of k even on medium/large graphs. For instance, computing hundreds of billions of cliques for rather demanding Web graphs and social networks requires about 15 minutes on a 32-core machine. Moreover, the running times of the multicore algorithm are competitive – and in some cases much faster than – the state-of-the-art distributed solutions based on MapReduce. As a by-product of the experimental analysis, we also compute the exact number of k-cliques with at most 20 nodes in many real-world networks from the SNAP repository.

For the second problem, we first devise a sequential algorithm for counting the number of k-diamonds in large undirected graphs, for any small constant $k \geq 4$. The algorithm can compute the number of k-diamonds using $O(\sqrt{m})$ extra work with respect to the clique-counting problem. A parallel extension of the sequential algorithm is then proposed, developing a MapReduce-based approach. This algorithm achieves the same local and total space usage of the state-of-the-art MapReduce algorithm for k-cliques, and uses $O(\sqrt{m})$ extra local and global work.

Acknowledgments

Contents

List of figures xii

List of tables xiii

1 Introduction 1
- 1.1 Original contributions . . . 4
- 1.2 Organization of the thesis . . . 6

2 Graph theoretical preliminaries 7
- 2.1 Graph terminology . . . 7
- 2.2 Total order . . . 8
- 2.3 Theoretical properties . . . 8

3 Computational models 11
- 3.1 The Fork/join shared-memory model . . . 11
- 3.2 The MapReduce framework . . . 12

4 Subgraph enumeration: a review of the literature 15
- 4.1 Specific subgraphs . . . 15
  - 4.1.1 Triangles . . . 15
  - 4.1.2 Cliques . . . 18
  - 4.1.3 Bipartite cliques . . . 22
  - 4.1.4 Sample graphs . . . 24
- 4.2 Frequent subgraphs mining . . . 25
- 4.3 All subgraphs on k vertices . . . 28
- 4.4 Clique relaxations . . . 31
  - 4.4.1 Vertex-based relaxations . . . 31
  - 4.4.2 Edge-based relaxations . . . 33
  - 4.4.3 Density-based relaxations . . . 34

5 Enumerating cliques in parallel 39
- 5.1 A general parallel framework . . . 39
  - 5.1.1 Algorithm . . . 39
  - 5.1.2 Main implementation choices . . . 40
  - 5.1.3 Analysis . . . 42
- 5.2 Experimental setup . . . 44
  - 5.2.1 Benchmarks . . . 44

5.2.2 Sequential clique counting: algorithm L+N 45
5.2.3 Platform 45
5.3 Engineering algorithm CountClique 45
5.3.1 Code optimizations 46
5.3.2 Candidate intersection and subset enumeration: sequential or parallel? 46
5.4 Experimental results 47
5.4.1 A bird's eye view 48
5.4.2 Scalability analysis 50
5.4.3 Counting small cliques 52
5.4.4 Counting larger cliques 53
5.4.5 Multithreading or MapReduce? 55

6 Enumerating diamonds: a sequential approach 59
6.1 Diamond classification 59
6.2 Algorithm 60
6.3 Analysis 62
6.4 Extension to k-diamonds 64

7 Enumerating diamonds in parallel 67
7.1 Setting up the computation 67
7.2 Listing 4-diamonds in triangle space 69
7.3 Extension to k-diamonds. 72
7.4 Trading space for parallelism 75

8 Conclusions and open problems 79
8.1 Future work 80

Appendix A 109

List of figures

2.1 Relation between k-diamonds and h-cliques, for $h \leq k-1$. 7
2.2 Examples of induced and non-induced subgraphs. 8
2.3 An undirected graph $\overline{G}$ and its corresponding directed graph G. . . . 9

3.1 Fork/join generic model. 11
3.2 Example of word counting algorithm in MapReduce. 13

4.1 Organization of Chapter 4. 16

5.1 Example of t-expansion as described in Claim 1. 40
5.2 Fork/join algorithm for computing the number of k-cliques by t-expansions. 41
5.3 A graph G and its fork trees for $t = 1$ and $t = 2$, assuming $k = 5$. . . 43
5.4 Clique growth rate: ratio q_k/q_{k-1} for $k \in [4, 7]$ on the different benchmarks. 44
5.5 A comparison of the algorithms for $k = 5$. 49
5.6 A comparison of the algorithms for $k = 7$. 49
5.7 Running times of `L+N` and of the five parallel variants when counting cliques of size $k \in [5, 7]$, using a number of cores ranging from 2 to 32, on the input dataset `asSkitter`. 50
5.8 Scalability of the parallel algorithms. 51
5.9 Running time of the algorithm as a function of k on 16 cores, for $k \leq 7$. 52
5.10 Increasing k on a selection of medium-size benchmarks. 54
5.11 In-depth comparison of `L+N` and `1by1` on `locGowalla` for large values of k. 54
5.12 Counting k-cliques on (a) `amazon`, (b) `citPat`, and (c) `comYoutube` for $k \leq 17$ ($q_k = 0$ for larger values of k as shown in 8.1). 55
5.13 Counting k-cliques on (a) `socPokec`, (b) `webGoogle`, and (c) `socPokec` for $k \leq 20$. 56

6.1 Classification of k-diamonds, for $k = 4$, based on the degree in D of the two smallest nodes. 60
6.2 A sequential algorithm for counting (and listing) 4-diamonds. 61

7.1 Listing triangles in MapReduce 68
7.2 MapReduce code for counting 4-diamonds in triangle space 69
7.3 Reduce 3 code for counting k-diamonds in triangle space 72

7.4 MapReduce code for case 3 76

List of tables

1.1 k-cliques in real-world networks: number n of nodes, number m of edges, and numbers q_k of cliques for $k \in [3, 7]$. 2
1.2 A selection of subgraph structures. 3

2.1 Summary of notation. 9

5.1 High-neighborhood statistics. 47
5.2 Running times (mins:secs) of four sequential/parallel variants for $t \in [2, 3]$ across different datasets and different values of k, using 32 cores. 48
5.3 Running times (mins:secs) of `1by1` and of the MapReduce algorithms analyzed in [98]. 57

6.1 Roles played by nodes a, b, c, and d used in the pseudocode of Figure 6.2 with respect to the six diamond types described in Figure 6.1. 61

7.1 Roles played by nodes a, b, c and d in Figure 7.2 according to the six diamond types described in Figure 6.1. 70
7.2 Analysis of `PDC` algorithm 77

Chapter 1

Introduction

Social networks represent interactions and collaboration between people or groups of people. A social network can be regarded as a graph, where the nodes (vertices) are the people and the edges (arcs) are the pairwise relations between them. Considering that many real-world systems in nature and society can be represented by networks [209], social network analysis (SNA) has received significant attention. One of the central concepts in SNA is the notion of *cohesive* subgraphs, that are subsets of nodes related to each other by relatively strong, direct, intense, frequent, or positive ties [321]. These subgraphs are attractive because their members tend to exhibit similar characteristics [103].

The problem of counting – and possibly listing – all the occurrences of a small pattern subgraph in a given graph has a long history. The first papers date back to the '70s, but there has been a renewed interest in the last few years in connection with the growth of network analytics applications. In particular, the enumeration of small dense subgraphs has been the subject of many recent works. Modern real-world networks have indeed a large number of nodes and sparse connections, but due to locality of relationships are locally very dense, i.e., contain an enormous number of small dense subgraphs that can be exploited for a variety of tasks such as spam and fraud detection [105], social networks analysis [238], link classification and recommendation [301], and the discovery of patterns in biological networks [246].

The focus of this thesis is on listing *k-cliques* (i.e., complete subgraphs of k nodes) and one of their relaxations, called *k-diamonds* (i.e., k-cliques with 1 missing edge) in large-scale networks, considering small values of k. The problem of counting and enumerating k-cliques is an important building block in numerous graph mining algorithms, e.g., [115, 299, 257]. In Table 1.1 we show the number of k-cliques for $k \leq 7$ on a selection of datasets from the SNAP repository [170], a general purpose, high-performance system for the analysis and the manipulation of large networks. Besides a graph mining library, SNAP also provides a collection of more than 50 medium-size and large real-world datasets. As shown in Table 1.1, graphs with millions of edges can easily have hundreds of billions of k-cliques. This sheer size makes the design of enumeration algorithms very challenging. Even for triangle counting, which is the simplest, non-trivial version of the problem ($k = 3$), exact centralized processing algorithms cannot typically scale to massive graphs. Workarounds proposed in the literature to speed up the computation are mostly

Table 1.1. k-cliques in real-world networks: number n of nodes, number m of edges, and numbers q_k of cliques for $k \in [3, 7]$.

	$n = q_1$	$m = q_2$	q_3	q_4	q_5	q_6	q_7
amazon	403 394	3 387 388	3 986 507	4 420 994	3 606 466	2 193 997	988 617
citPat	3 774 768	16 518 948	7 515 023	3 501 071	3 039 636	3 151 595	1 874 488
comYoutube	1 134 890	2 987 624	3 056 386	4 986 965	7 211 947	8 443 803	7 959 704
locGowalla	196 591	950 327	2 273 138	6 086 852	14 570 875	28 928 240	47 630 720
socPokec	1 632 803	22 301 964	32 557 458	42 947 031	52 831 618	65 281 896	83 896 509
webGoogle	875 713	5 105 039	13 391 903	39 881 472	105 110 267	252 967 829	605 470 026
wikiTalk	2 394 385	5 021 410	9 203 519	64 940 189	382 777 822	1 672 701 685	5 490 986 046
webStan	281 903	2 312 497	11 329 473	78 757 781	620 210 972	4 859 571 082	34 690 796 481
hTwitter	456 631	14 855 875	83 023 401	429 733 013	2 170 177 145	11 040 286 581	55 261 342 424
asSkitter	1 696 415	11 095 298	28 769 868	148 834 439	1 183 885 507	9 759 000 981	73 142 566 591
comOrkut	3 072 441	117 185 083	627 584 181	3 221 946 137	15 766 607 860	75 249 427 585	353 962 921 685
webNotreDame	325 729	1 497 134	8 910 005	231 911 102	6 367 609 888	153 998 482 142	3 228 475 265 752
webBerkStan	685 230	6 649 470	64 690 980	1 065 796 916	21 870 178 738	460 155 286 971	9 398 610 960 254
comLiveJ	3 997 962	34 681 189	64 690 980	5 216 918 441	246 378 629 120	10 990 740 312 954	445 377 238 737 777
socLiveJ1	4 847 571	68 993 773	285 730 264	9 933 532 019	467 429 836 174	20 703 476 954 640	-

based on approximation or parallelization techniques.

Approximate counting algorithms (e.g., [217, 227]) typically return estimates strongly concentrated around the true number of cliques and are very accurate in practice, but cannot solve the more general listing problem. Most parallel solutions, on the other side, achieve scalability by exploiting a distributed computing cluster to perform the computation. Though distributed resources can be easily available through the cloud, orchestrating a distributed computation remains challenging and therefore big data systems à la MapReduce [78] are quite often the solution of choice. These systems automatically handle scheduling, synchronization, and fault tolerance issues, but might not naturally support iterative computations and incur large overheads due to communication costs, workload balancing, and I/O operations. Triangles, for instance, can be efficiently enumerated in very large graphs using MapReduce [285], but traditional multicore algorithms [278] or even well-engineered sequential approaches [214] could remain preferable depending on the graph size and on the number of subgraphs to be listed.

The state of the art for k-clique enumeration is less understood with respect to triangles. The running time of fast and practical sequential algorithms [59] is $O(\alpha(\overline{G})^{k-2}m)$, where $\alpha(\overline{G})$ is the arboricity of the graph: this is $O(m^{k/2})$ in the worst case, since $\alpha(\overline{G}) = O(\sqrt{m})$. In a parallel setting, previous works focused on MapReduce, with two different algorithms presented in [4] and [98], respectively. The design of traditional multicore algorithms for counting k-cliques has not been properly addressed in the literature. Hence, differently from triangles, it is not clear whether and to what extent it is possible to count k-cliques on a single multicore machine with tens of cores and adequate memory.

Considering that every pair of nodes must be connected by an edge in a clique, in many real-world systems the concept of clique is often too rigid [12] or presents modeling disadvantages [102]. Requiring the existence of all possible edges may prove to be rather restrictive for many applications, where the interaction between members of the group could be sometimes achieved through intermediaries, without the need to be directly connected [226]. Furthermore, due to possible errors in data collection or interpretation, the absence or presence of an edge is not known with certainty in many problems involving application of clique-detection algorithms [223].

Table 1.2. A selection of subgraph structures.

Structure	Name	Definition	References
$k = 4$	k-clique	Complete subgraph on k nodes.	This thesis, [59, 98, 46]
$k = 4$	k-diamond	Clique of size k with one missing edge.	This thesis
$k = 2$	k-core	Largest subgraph in which all nodes have degree at least k.	[32, 187, 94]
$k = 2$	k-plex	Connected subgraph in which each node may miss at most k neighbors in the subgraph (including itself).	[319, 66, 69]
$k = 4$	k-truss	Connected subgraph in which each edge is incident to at least $k - 2$ triangles	[62, 127, 114]
$s = 2$	s-clique	Connected subgraph in which the length of the shortest path (number of edges in the original graph) between each pair of nodes is at most s. Notice that the grey node is not included in the 2-clique.	[142, 195, 34]
$k = 2$	k-club	Connected subgraph in which the length of the shortest path (number of edges in the subgraph) between each pair of nodes is at most k.	[183, 14, 26]
$\gamma = 0.6$	γ-clique	Connected subgraph S in which the number of edges is at least $\gamma\binom{\|S\|}{2}$.	[300], [184], [232]

To overcome these issues, alternative subgraph structures have been considered in the literature, relaxing the definition of clique. In Table 1.2 we show a collection of subgraph structures, as well as their definition and some of the main works addressing them. Those structures can be classified into three main groups according

to the clique property which is relaxed: vertex-based, edge-based, and density-based relaxations. Vertex-based relaxations, such as k-core [264, 191] and k-plex [265], relax the degree of vertices in a clique. For example, in a 2-core on 4 vertices, each vertex must have degree at least 2, while in a clique of size 4 all vertices must have degree equal to 3. Edge-based relaxations consist of relaxing a property of the edges. The k-diamond structure addressed in this thesis can be included in this group, because there is exactly one missing edge in a “clique” of size k. Another example of edge-based relaxation is the k-truss [62], considering that the number of triangles incident to each edge must be at least $k - 2$. At last, density-based relaxations are focused on a global characteristic, being not directly related to a vertex or an edge property. For instance, the γ-clique structure [3, 228] is classified as density-based relaxation, where its density is relaxed by decreasing the total number of edges. The s-clique [181, 201] and k-club [201] structures can also be considered density-based relaxations, since the length of the shortest path between every two nodes is relaxed.

Besides many previous works focused on cliques and their relaxations, problems dealing with different kind of subgraphs have been widely studied in the literature. For instance, the problem of listing specific sample subgraphs, where the objective is to enumerate all instances of a specific input sample subgraph in a large undirected graph [4]. Furthermore, instead of enumerating all occurrences of a single structure, counting or listing a collection of subgraphs have been extensively studied, such as the problem of listing all subgraphs on k nodes [199] or frequent subgraphs [6].

The focus of this thesis is on k-clique and k-diamonds listing problems. In particular, we design parallel k-clique enumeration algorithms, as well as a sequential and a distributed algorithm for k-diamond enumeration.

1.1 Original contributions

In this section we describe the main contributions of this thesis: a literature review of the subgraph enumeration, multicore clique enumeration algorithms, a sequential diamond enumeration algorithm and a distributed diamond enumeration algorithm.

A literature review of the subgraph enumeration. The large number of subgraphs structures addressed in the literature makes the summary of main works a rather challenging task. The number of articles to be considered becomes even larger when we address different problems related to each subgraph structure, e.g., listing all instances, finding the maximum subgraph, enumerating all maximal subgraphs, exact counting, and approximate counting. Furthermore, there are many other works in the literature addressing problems that deal with enumeration of all subgraphs of a fixed size and with the frequent subgraph mining problem. In this thesis we provide an extensive literature review of the subgraph enumeration, which considers different problems associated to several subgraph structures and group of subgraphs.

Multicore clique enumeration algorithms. Previous works addressing the problem of enumerating triangles or k-cliques propose either sequential or distributed solutions. A first natural question is whether we always need a cluster for listing k-cliques. To shed some light on this question, in this thesis we develop simple and

fast multicore parallel algorithms for listing k-cliques in large undirected graphs, for any small constant $k \geq 4$. The proposed algorithms, based on nodes' neighborhood intersection, are work-optimal in the worst case, i.e., can list all k-cliques on a graph with m edges in $O(m^{k/2})$ work. They exploit a fork/join programming style, which resembles divide-and-conquer, and are suitable to be implemented in shared-memory settings using any languages/libraries supporting dynamic multithreading. In order to tradeoff between the number of parallel task operations and the synchronization overhead due to the number of tasks, they can be naturally parameterized according to the number t of neighborhood intersections performed by each task.

The parallel algorithms were implemented and engineered in a light-weight Java framework for fork/join parallelism [168]. The effectiveness of the implementations is assessed through an extensive experimental analysis on a variety of real-world graphs from the SNAP repository [170], considering different clique sizes and scalability on different numbers of cores.

To the best of our knowledge, multicore algorithms addressing the same k-clique enumeration problem were not proposed so far in the literature. Hence, we provide an experimental analysis comparing the proposed multicore solutions with the highly optimized sequential algorithm. The experimental results show that one of our parallel algorithms largely outperforms the running times of the state-of-the-art sequential solution and gracefully scales to non-trivial values of k even on medium/large graphs. For instance, computing hundreds of billions of cliques for rather demanding Web graphs and social networks from the SNAP repository requires about 15 minutes on a 32-core machine. For moderate values of k the running times are competitive with – and in some cases much faster than – state-of-the-art distributed solutions based on MapReduce [4, 98].

As a by-product of the experimental analysis, it is computed the exact number of k-cliques in many real-world networks from the SNAP repository for $k \in [4, 20]$ and analyzed their distribution. These k-clique numbers, differently from the number of triangles and of other small subgraphs on a few nodes, were not available in the literature before our study.

The parallel implementations, as well as the implementation of the state-of-the-art sequential algorithm, are publicly available on bitbucket at the URL `https://bitbucket.org/renanleong/parallelcliquecounting`.

Sequential and distributed diamond enumeration algorithms. The problem of listing k-diamonds has not been properly addressed so far in the literature: k-diamonds are only considered in a few works that deal with different subgraph structures, such as listing all subgraphs on k nodes and listing all instances of a specific sample subgraph.

In this thesis we address the problem of listing all occurrences of k-diamonds in large undirected graphs, for any small constant $k \geq 4$. Moreover, we develop and analyze a sequential algorithm addressing this problem. The proposed approach can compute all k-diamonds in $O(m^{(k+1)/2})$ time, which is $O(\sqrt{m})$ larger than the time required by the state-of-the-art algorithm for computing k-cliques.

Besides the sequential algorithm, we show that our approach is amenable to parallelization, describing a MapReduce-based approach for k-diamond enumeration.

The MapReduce algorithm extends the strategy of the sequential solution to perform in parallel. It can compute all k-diamonds in $O(m^{(k+1)/2})$ total work and $O(m^{3/2})$ total space. The local work and space are $O(m^{k/2})$ and $O(m)$, respectively. Comparing to the state-of-the-art MapReduce algorithm for k-cliques, the k-diamond algorithm achieves the same local and total space usage, and requires only $O(\sqrt{m})$ more local and total work.

1.2 Organization of the thesis

This thesis consists of three main parts. The first part covers Chapter 4 and presents a literature review of the subgraph enumeration. The second part covers Chapter 5 and describes parallel shared-memory algorithms for the k-clique enumeration problem, as well as an experimental analysis. The third part covers Chapters 6 and 7, where we propose the state-of-the-art sequential and MapReduce algorithms for the k-diamond enumeration problem. In addition, Chapter 2 provides the basic notation, definitions, and properties that will be useful throughout the thesis. Chapter 3 describes the computational models exploited by the proposed algorithms. In Chapter 8 we provide concluding remarks and highlight some interesting open problems.

Chapter 2

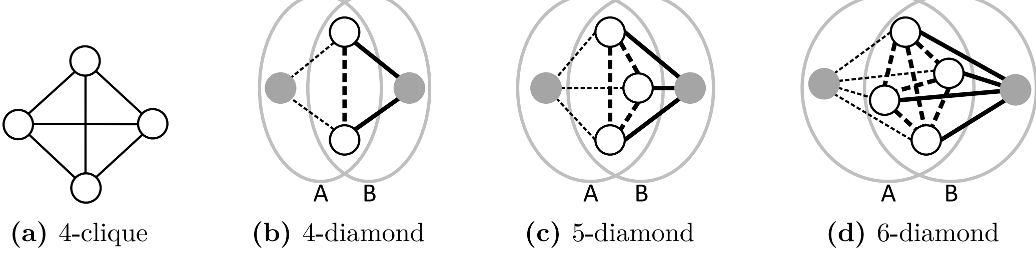
Graph theoretical preliminaries

In this chapter is introduced basic notation and graph-theoretical properties that will be used throughout the thesis.

2.1 Graph terminology

In this section is described the basic graph notation. Let $\overline{G} = (\overline{V}, \overline{E})$ be a simple undirected graph without self loops. For each node $u \in \overline{V}$, let $d(u)$ and $\Gamma(u)$ denote its degree and its neighborhood in $\overline{G}$, respectively (u is not included). Given an integer $k \geq 1$, let q_k and d_k denote the number of k-cliques (i.e., complete subgraphs on k nodes) and the number of k-diamonds in $\overline{G}$, respectively.

A k-diamond is a clique of size k with one missing edge. It 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., one $(k-2)$-clique). Examples are given in Figure 2.1, where edges in one $(k-1)$-clique are dashed (set A), edges in the other $(k-1)$-clique are bold (set B), and the $(k-2)$-clique in their intersection is both dashed and bold (set $A \cap B$). In each k-diamond, the endpoints of the missing edge are grey (notice they have degree $k-2$).



(a) 4-clique **(b)** 4-diamond **(c)** 5-diamond **(d)** 6-diamond

Figure 2.1. Relation between k-diamonds and h-cliques, for $h \leq k-1$.

2.2 Total order

In this section is described the total order technique. It consists of applying the total order $\prec$ over the nodes of the undirected graph $\overline{G}$ to build a directed graph G. This strategy works as follows:

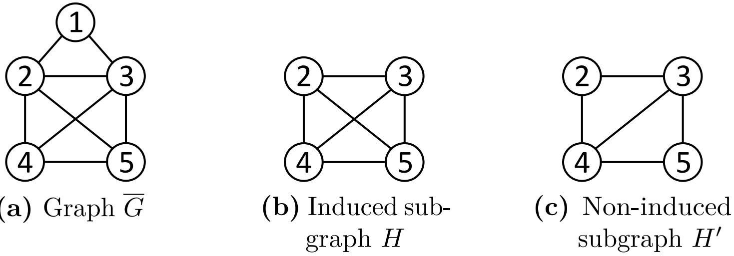
$$u \prec v \text{ if and only if } d(u) < d(v) \text{ or } d(u) = d(v) \text{ and } u < v$$

assuming nodes to have comparable and unique labels. Order $\prec$ implicitly defines a directed graph $G = (V, E)$ as follows: $V = \overline{V}$ and $E = \{(u, v) \in \overline{E}$ such that $u \prec v$ in $\overline{G}\}$. Throughout the thesis, n and m are used to denote the number of nodes and the number of edges in G, respectively, i.e., $n = |V|$ and $m = |E|$. 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, $\Gamma^-(u) = \Gamma(u) \backslash \Gamma^+(u)$ is the set of neighbors v of u such that $v \prec u$.

The total order $\prec$ allows the algorithms to avoid the very high degree nodes. For example, a nodes $u \in \overline{G}$ have degree less or equal m, while the same node $u \in G$ (after applying the total order) have degree less or equal $\sqrt{m}$.

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

Examples of induced and non-induced subgraphs are given in Figure 2.2. Consider the undirected graph $\overline{G} = (\overline{V}, \overline{E})$ in Figure 2.2a. The graph H in Figure 2.2b is an induced subgraph of $\overline{G}$, since $H \subset \overline{G}$ and, for each pair of nodes $u, v \in V(H)$, $(u, v) \in E(H)$ if and only $(u, v) \in \overline{E}$. However, the graph H' in Figure 2.2c is a non-induced subgraph of $\overline{G}$. Although $H' \subset \overline{G}$, the pair of nodes 2 and 5 are not adjacent in H', while they are an edge in $\overline{G}$. Therefore, H' is a non-induced subgraph of $\overline{G}$.



(a) Graph $\overline{G}$ **(b)** Induced subgraph H **(c)** Non-induced subgraph H'

Figure 2.2. Examples of induced and non-induced subgraphs.

In Table 2.1 is summarized the key notation used throughout the thesis.

2.3 Theoretical properties

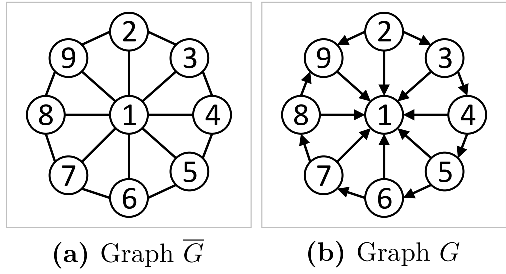
In this section we describe two useful properties that will be crucial in the analysis of the algorithms proposed in the thesis.

Table 2.1. Summary of notation.

Notation	Meaning
$\overline{G} = (\overline{V}, \overline{E})$	Undirected graph $\overline{G}$ composed of nodes set $\overline{V}$ and edges set $\overline{E}$
$d(u)$	Degree of node u in $\overline{G}$
$\Gamma(u)$	Neighborhood of u in $\overline{G}$
$\prec$	Total order
$u \prec v$	$d(u) < d(v)$ or $d(u) = d(v)$ and $u < v$
$G = (V, E)$	Directed graph G composed of nodes set V and edges set E
n	Number of nodes in G, $n = \|V\|$
m	Number of edges in G, $m = \|E\|$
$\Gamma^+(u)$	Set of neighbors v of u in G such that $u \prec v$
$\Gamma^-(u)$	Set of neighbors v of u in G such that $v \prec u$
$G(u)$	Subgraph induced by the neighborhood $\Gamma(u)$ of a node u
$G^+(u)$	Subgraph induced by the high-neighborhood $\Gamma^+(u)$ of a node u
$G^+(u, v)$	Subgraph induced by $\Gamma^+(u) \cap \Gamma^+(v)$ of a pair of nodes u and v
q_k	Number of k-cliques in $\overline{G}$
d_k	Number of k-diamonds in $\overline{G}$

Lemma 1. *[98] Let G be the directed graph with m edges obtained from an undirected graph $\overline{G}$ according to the total order $\prec$. For each node u in G, $|\Gamma^+(u)| \leq 2\sqrt{m}$.*

As an example, Figure 2.3 shows an undirected graph $\overline{G}$ 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 the directed graph G because $v \prec 1$ for each other node v. Hence, thanks to the preprocessing we can get rid of all the high degree nodes (also called heavy hitters in the literature [171]). Preprocessing the graph to eliminate heavy hitters is simpler and can reduce memory consumption with respect to other approaches, such as the triangle counting algorithm presented in [171], which works on the undirected graph and needs two distinct subroutines to count triangles all of whose nodes have high degree and triangles containing at least a low degree node.



(a) Graph $\overline{G}$ **(b)** Graph G

Figure 2.3. An undirected graph $\overline{G}$ and its corresponding directed graph G.

Lemma 2. *Let n and m be respectively the numbers of nodes and edges of an undirected graph $\overline{G}$ without isolated nodes. Then it holds: $\sqrt{m}/2 \leq n \leq 2m$.*

Proof. The upper and the lower bound on n are proved separately. Since there are no isolated nodes in $\overline{G}$, the degree of each node is at least one and thus $n = \sum_{u \in V} 1 \leq \sum_{u \in V} d(u) = 2m$. Now consider the directed graph G obtained from $\overline{G}$ according to the total order. Notice that $m = \sum_{u \in V} |\Gamma^{+}(u)|$ and that, for each node u in G, $|\Gamma^{+}(u)| \leq 2\sqrt{m}$ by Lemma 1. Hence, $m = \sum_{u \in V} |\Gamma^{+}(u)| \leq \sum_{u \in V} 2\sqrt{m} = 2n\sqrt{m}$. This implies that $n \geq \sqrt{m}/2$. □

Chapter 3

Computational models

In this chapter we discuss the models of computation exploited by the algorithms proposed in the thesis to parallelize their execution. In Section 3.1 is described the Fork/join framework, which is exploited by the clique enumeration algorithms in Chapter 5 to run in a shared-memory platform. MapReduce framework is presented in Section 3.2, being exploited by the diamond enumeration algorithm in Chapter 7 to execute in a distributed platform.

3.1 The Fork/join shared-memory model

The idea of implementing parallel processing in a multiprocessor system design was introduced in 1963 by [70], called fork and join system calls. The origins of how fork/join strategy was designed are investigated by [213]. Fork/join parallelism is a programming style that hinges upon divide-and-conquer: problems are recursively split into subtasks until they are small enough to solve using a simple sequential method, where the subtasks are solved in parallel. The final result is computed only upon completion of the subtasks.

According to [168], the fork/join parallelism is among to simplest and most effective design techniques to achieve a good parallelism. Furthermore, in [168] is described a generic model of the fork/join strategy, which is shown in Figure 3.1.

Algorithm 1. FORK/JOIN

```
1: function SOLVE(Problem problem)
2:     if problem is small then
3:         directly solve problem                          ▷ Sequential method
4:     else
5:         split problem into independent subproblems
6:         for each subproblem s do
7:             fork a new subtask SOLVE(s)
8:         join all subtasks                               ▷ Aggregate partial results
9:         return combined result
```

Figure 3.1. Fork/join generic model.

The `fork` command starts new parallel fork/join subtasks, while the `join` command forces the current task not to proceed until all forked subtasks have completed. The execution of the fork/join algorithm can be represented as a tree, where the root starts the computation of the problem and the leaves are the base case (subtasks small enough to solve sequentially).

Standard threads are typically too heavyweight to support most fork/join programs and cannot scale to large number of tasks. Hence the implementation of frameworks and libraries that provide lightweight support for the fork/join programming style have been proposed, most notably OpenMP, Java Fork/join, Intel Cilk Plus, Intel Thread Building Blocks and the Task Parallel Library for `.NET`.

Clique enumeration algorithms introduced in Chapter 5 exploit the Java Fork/Join framework, which is available in Java since version 7 [168]. The lightweight threads provided by this framework, called `ForkJoinTasks`, are small enough that even millions of them should not hinder performance. Moreover, the framework implements a very efficient scheduler, based on work stealing, with low practical overhead and theoretically optimal expected-time guarantees. The efficiency of scheduling fork/join tasks with work stealing is analyzed from a theoretical and a practical perspective by [41] and [168], respectively. Moreover, in [224] is investigated how the Java Fork/Join model is applied in practice, primarily by identifying best-practice patterns and ant-patterns.

The Fork/join clique counting algorithm introduced in Chapter 5 is theoretically analyzed according to the three performance metrics:

- **Work:** the total work spent by the algorithm during the computation.
- **Memory space:** the total amount of space used by the algorithm during the computation.
- **Span:** the length of the longest chain of sequential instructions.

3.2 The MapReduce framework

MapReduce is a distributed framework originally developed at Google [78]. Considering the increasing size of large data sets, this framework has emerged as an easy-to-program, reliable, and distributed parallel computing paradigm to process these massive quantities of available data [143].

The basic unit of information in the MapReduce programming paradigm is a $\langle key; value \rangle$ pair, where *key* and *value* are binary strings. Therefore, the input of any MapReduce algorithm is a set of $\langle key; value \rangle$ pairs. A MapReduce program is composed of consecutive rounds, where each round is divided into three consecutive phases: map, shuffle and reduce.

In the map phase, $\langle key; value \rangle$ pairs are arbitrarily distributed among mappers and a programmer-defined map function is applied to each pair. Basically, a mapper μ takes a single $\langle key; value \rangle$ pair as input, and produces a set of intermediate $\langle key; value \rangle$ pairs. Mappers are stateless and process each input pair independently from the others. Hence, different inputs for the map phase can be processed by different machines. The shuffle phase is seamless to the programmer, being executed

automatically. In this stage, intermediate $\langle key; value \rangle$ pairs emitted by the mappers are grouped by *key*. All pairs with the same *key* are then sent to the same reducer. In the reduce phase, each reducer ρ takes all *values* associated with a single *key* and process them by executing a programmer-defined reduce function. Since the reducers have access to all the *values* with the same *key*, the reduce phase can start only after the execution of all mappers have been completed.

Besides introducing and describe the MapReduce programming model, in [78] is also presented an example of MapReduce algorithm for counting the number of occurrences of each word in a large collection of documents. The pseudocode is given in Figure 3.2.

Algorithm 2. WordCount

```
Map 1: input ⟨docName; contents⟩
  for each word w in contents do
      emit⟨w; 1⟩

Reduce 1: input ⟨word; listCounts⟩
  result ← 0
  for each number n in listCounts do
      result ← result + n
  emit ⟨word; result⟩
```

Figure 3.2. Example of word counting algorithm in MapReduce.

Computational model. A model of efficient computation using MapReduce paradigm is proposed in [143], called MapReduce Class ($\mathcal{MRC}$). Considering that MapReduce is designed to compute massive data sets, the proposed model limits the number of machines and memory per machine to be substantially sublinear in the size of the input. However, this model allows each machine to perform sequential computations in time polynomial in the size of the input. In [143] is also pinpointed the critical aspects of efficient MapReduce algorithms:

- **Memory:** the input of any mapper or reducer should be sublinear with respect to the total input size. It allows to exclude trivial algorithms that first map the whole input to a single reducer, and then solve the problem sequentially.
- **Machines:** the total number of machines available should be substantially sublinear in the data size. According to [143], an algorithm requiring n^3 machines, where n is the input size, will not be practical in the near future.
- **Time:** both the map and the reduce function should run in polynomial time with respect to the original input length in order to ensure the efficiency.

Besides the three guiding principles, the model also requires programs to be composed of at most a polylogarithmic number of rounds, since shuffling is a time consuming operation, and to have a total memory usage that grows substantially less than quadratically with respect to the input size. Therefore, algorithms following these conditions belong to the $\mathcal{MRC}$.

Based on the $\mathcal{MRC}$, the MapReduce-based diamond enumeration algorithm introduced in Chapter 7 is theoretically analyzed according to the four performance metrics:

- **Global work:** the total time spent by all mappers and reducers.
- **Local work:** the time spent by each mapper and each reducer.
- **Global space:** the amount of space used by all mappers and reducers.
- **Local space:** the amount of space used by each mapper and each reducer, considering input size and the working space.

Chapter 4

Subgraph enumeration: a review of the literature

In this chapter we discuss the works related to the problems addressed in this thesis. As described in Chapter 2, this research considers simple undirected static graphs, excluding specific graph classes (e.g., dynamic graphs). Therefore, in this chapter is discussed works that tackle problems in simple undirected static graphs.

Works are presented according to the problems addressed by them. The problem of counting or listing specific subgraphs is tackled by works described in Section 4.1. Frequent subgraphs mining is the aim of the works discussed in Section 4.2. In Section 4.3 are discussed the works addressing the problem of counting or listing all subgraphs of a fixed size k, while different clique relaxations problems are discussed in Section 4.4.

The organization of this chapter is shown in Figure 4.1.

4.1 Specific subgraphs

In this section we discuss the works tackling the problem of counting or listing specific subgraphs. Triangles are the aim of the works described in Section 4.1.1. Works in Section 4.1.2 address cliques, which are a generalization of triangles. In Section 4.1.3 are discussed the works addressing the problem of bipartite counting cliques in bipartite graphs. Sample graphs are tackled by works in Section 4.1.4.

4.1.1 Triangles

The triangle listing problem consists of enumerating all instances of triangles in an undirected input graph $\overline{G}$ exactly once. Triangle listing is very well studied, due to its large number of applications in network analysis. Several exact algorithms exist, most of which date back to the '80s. A simple and practical edge-searching strategy was presented by Chiba and Nishizeki [59]. Given an input graph $\overline{G}$ with n nodes and m edges, their approach requires $O(\alpha(\overline{G})m)$ time to enumerate all triangles in $\overline{G}$, where $\alpha(\overline{G}) = O(\sqrt{m})$ is the arboricity of the graph. This running time is optimal in the worst case.

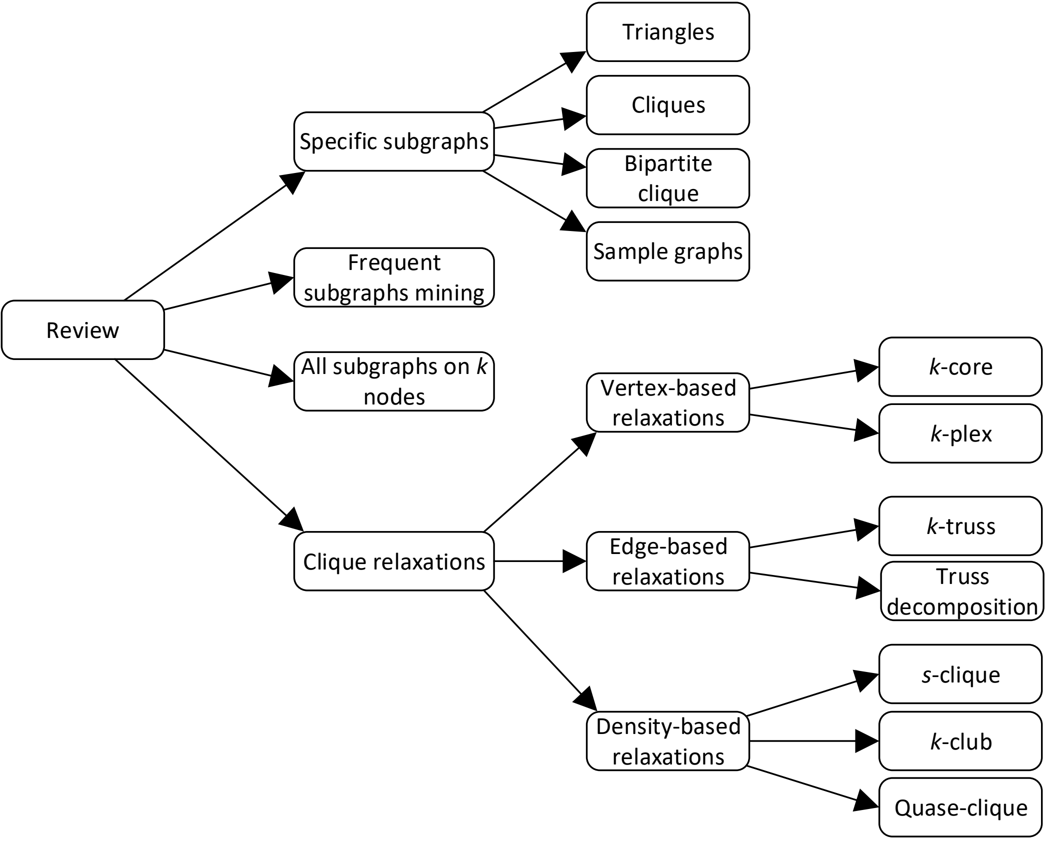


Figure 4.1. Organization of Chapter 4.

Another work-optimal solution, called Heavy Hitters, is described in [171]. It is based on the idea of listing separately triangles whose all vertices have high degree and triangles containing at least a low degree vertex. This strategy was used by Alon et al. [16] in a previous result for finding and counting simple cycles of a given length. Neighborhood intersection techniques were continuously studied, iterating over graph nodes [261] or graph edges [132]. An extensive experimental analysis of these approaches, that have been all cast into a common framework, is given in [214]. In their analysis the variant called L+N shows the best performance.

When graphs are too large to fit in main memory, the problem can be solved in the external memory model [216, 72]. Considering that triangle listing requires to access the neighbors of the neighbor of a vertex, they may appear in any position of the graph. Thus, random access to the graph stored on disk is needed, which can incur prohibitively large I/O cost. Based on that argument, an I/O-efficient algorithm is proposed in [60], partitioning the input graph into a set of subgraphs that fit into memory and listing the triangles in each local subgraph. An analysis and improvement of I/O complexity is described in [71]. A different solution to graphs too large is proposed in [164], which consists of compressing the input graph and listing all triangles without decompressing it.

Focused on speeding up the enumeration of triangles, parallel approaches have been proposed in different models of computation. Suri and Vassilvitskii [285]

introduced two MapReduce algorithms for listing triangles, dubbed NodeIterator++ and Partition. The former uses $O(m^{3/2})$ global space and works in two rounds, generating all possible paths of length two and checking which paths can be closed to form a triangle. The latter divides the input graph into subgraphs that are then processed using a sequential listing algorithm. A new graph partitioning based algorithm is proposed in [220], which classifies the triangles into three types to reduce duplication. Generalizations of Partition approach, that take into account memory constraints, are described in [221]. In [106] is introduced a distributed algorithm with provable CPU, I/O, memory, and network bounds. MapReduce-based algorithms for listing triangle were also proposed in [222, 329, 346, 149].

Other works address the triangle listing problem on single-node multicore platforms, extending node iterator to compute clustering coefficients [112], parallelizing the compact forward algorithm [278] presented in [167] or exploiting hash maps [291]. An in-memory parallel solution is proposed in [268], combining two existing approaches: EdgeIterator [262] and NodeIterator++ [285]. Two MPI-based algorithms are given in [25], exploiting the overlapping partitions strategy to achieve a very fast algorithm and non-overlapping strategy to achieve a space-efficient algorithm.

A bit batch-based algorithm is described in [240], where the list of neighbors are represented by bit vectors. Matrix multiplication approaches performed in parallel for listing triangle are the topic of [29], while index-based method is presented in [241]. Implementations exploring the large numbers of cores in a GPU are proposed in [113] and [39].

Although triangle enumeration has been widely studied in the last years, counting the exact number of triangles in a graph is enough in many network analysis applications. Linear algebra approaches for triangle counting problem are given in [180] and [325], which compute in parallel the number of triangles using OpenMP architecture and KokkosKernels implementation, respectively. MPI-based algorithms are described in [22, 24, 23].

A comparison and analysis of GPU implementations for triangle counting is given in [310]. Other parallel algorithms were proposed focusing on specific frameworks, such as Nvidia CUDA in [233] and Graphulo library in [130]. Voegele et al. [308] describe CPU and GPU implementations of triangle counting algorithm. A scalable framework designed for GPUs is proposed in [127], combining a 2-D graph partition strategy with a binary search based intersection. In [141] are analyzed and discussed four different approaches, which are generalized into a common framework implemented in a distributed shared-memory setting.

Besides parallelism, approximation has been also exploited to speed up triangle counting algorithms on massive graphs. It consists of estimating as accurately as possible the number of triangles in an input graph $\overline{G}$. A variety of triangle counting algorithms are proposed in the literature [217, 157], achieving different accuracy guarantees. A wedge sampling approach is given in [267], being posteriorly implemented in MapReduce programming model in [156] to parallelize the computation. In [303] is proposed a hybrid estimate algorithm that exploits edge and wedge sampling strategies.

In [158] is described an approximation approach that explores two existing strategies: the sampling algorithm in [302] and the partitioning method presented

in [16]. A *sublinear-time* approach is introduced by Eden et al. [87], where the algorithm is given query access to the graph. However, in [266] is described a simpler sublinear-time algorithm that achieves the same theoretical running time of the approach presented in [87]. The first approximate triangle counting algorithm using only polylogarithmic queries is introduced in [37].

A hybrid parallel algorithm is proposed in [315], which combines the advantages of NodeIterator and EdgeIterator solutions introduced by Alon et al. [16]. Focused on a different parallel model, in [236] is described a multicore approximation algorithm based on edge-iteration [132]. A few works, such as [33, 135, 312], focus on the data stream model.

Since approximate triangle counting problem is widely addressed in the literature, comparative works have been proposed. A detailed analysis of random sampling algorithms is given in [327], providing an experimental and analytical comparison of existing approaches. In [52] is introduced a random framework for expressing and analyzing approximate triangle counting. A more generic review of triangle counting problem in large network is given in [9], which includes exact and approximate approaches.

4.1.2 Cliques

The clique counting problem consists of computing all instances of cliques on k nodes in an undirected input graph $\overline{G}$ exactly once. It can be extended to the clique enumeration problem, which consists of listing all k-cliques in $\overline{G}$. Clique counting/listing problem can be considered a generalization of the triangle counting/listing problem, since a triangle is a k-clique of size three, i.e., $k = 3$.

Chiba and Nishizeki [59] extend their triangle listing algorithm K3 to compute cliques on k nodes, named COMPLETE. Basically, the algorithm computes iteratively the nodes' neighborhood until it finds the k-cliques. It can list all k-cliques in a graph $\overline{G}$ in $O(\alpha(\overline{G})^{k-2}m)$ time, where m is the number of edges in $\overline{G}$ and $\alpha(\overline{G}) = O(\sqrt{m})$ is the arboricity of the graph. This theoretical running time is optimal in the worst case. Although the Heavy Hitters [171] approach was initially proposed to address the triangles listing problem, it can be easily extended to k-cliques, yielding optimal running time $O(m^{k/2})$.

As shown in Chapter 1, graphs with millions of edges can easily have hundreds of billions of k-cliques. Even for the problem of counting triangles (k-cliques with 3 nodes), exact centralized processing algorithms cannot typically scale to massive graphs. Based on that argument, parallel approaches have been proposed in the literature, providing theoretical and experimental analysis. Finocchi et al. [98] introduce two MapReduce algorithms for counting the exact and the approximate number of k-cliques, respectively. The strategy of the exact approach is to split the whole input graph in many subgraphs, and then count the cliques in each subgraph independently. This approach is work-optimal in the worst case, listing the number of k-cliques in $O(m^{3/2})$ total space and $O(m^{k/2})$ work. Moreover, it can be easily adapted to the enumeration problem. A different one-round MapReduce-based algorithm is given in [4], designed to enumerate subgraph instances (including cliques). An experimental analysis of the MapReduce approaches is carried out in [98]. Focused on shared-memory settings, in [73] is proposed an extension of the

algorithm COMPLETE [59] to run in parallel.

Approximation methods have also been exploited for counting k-cliques in massive graphs. In [134] is described a randomized algorithm to approximate the number of k-cliques in a graph, providing an experimental analysis of k-cliques for k ranging from 5 to 10. Another approximation algorithm is proposed in [88], which runs in sublinear time. This approach estimates the number of k-cliques in a graph when given query access to the graph. In [107] is proposed an algorithm to estimate the distribution of clique sizes, a variant of the clique counting problem. It is computed from a probability sample of nodes obtained from the input graph. Another variant of the clique counting problem is introduced in [299], called *k-clique densest subgraph problem*. It consists of finding the k-clique with maximum average degree. Moreover, an approximation algorithm is proposed tackling the introduced clique problem.

Maximal cliques. Instead of enumerate all cliques of a constant size k, several works in the literature have addressed the *maximal* clique problem. It consists of listing all *maximal* cliques in an input graph $\overline{G}$, i.e., all cliques $Q \subseteq \overline{G}$ such that the nodes in Q are not all contained within any other larger clique in $\overline{G}$.

A classical worldwide studied algorithm for listing maximal cliques is presented by Bron and Kerbosch [46]. The algorithm consists of a recursive backtracking procedure exploiting a branch-and-bound technique to cut off branches that can not complete a clique. Bron and Kerbosch's algorithm works with three arguments: a partial clique, a set of candidate vertices and a set of non-candidate vertices. Basically, the algorithm takes each vertex from the candidate set and tries to add it in the partial clique. The non-candidate set is used to store vertices that would lead to a clique already listed, avoiding duplication. An adaptation of Bron-Kerbosch algorithm is given in [18], using a complement matrix to list all maximal cliques. A parallel algorithm, called GP, is described in [126], which exploits the binary graph partitioning to find the cliques. Moreover, a parallel hybrid approach is proposed, combining the characteristics of GP and Bron-Kerbosch [46] algorithms. Both introduced approaches are implemented on MapReduce and experimentally analyzed, showing that the hybrid approach achieves the best results.

Tomita et al. [297] use algorithm engineering techniques to design a depth-first algorithm for enumerating all maximal cliques, where the output is generated in a tree-like form. Furthermore, the authors prove that the worst case time complexity of their algorithm is $O(3^{n/3})$ in a graph with n nodes, which is optimal considering that there exist up to $3^{n/3}$ maximal cliques. The enumeration of maximal cliques is also addressed in [185], and implementation issues are presented in [304]. In [292] is described an extensive analysis of the branch-and-bound depth-first algorithms for *maximum* (i.e., the largest) and *maximal* cliques introduced in [297, 295, 293, 296, 298]. In addition, the author presents a new algorithm for enumerating maximal cliques. The depth-first strategy is exploited by Li [176] to introduce a new approach, which implements several pruning methods to improve its performance.

Yu and Liu [335] design a novel structure, called CANDIDATE MAP, to hold all candidate cliques during the construction of maximal cliques. Moreover, a new linear time algorithm is proposed exploiting the introduced structure. The first sublinear

space and bounded delay approach for listing maximal cliques is described in [67]. A theoretical comparison with previous works shows that the proposed approach can be competitive with the state-of-the-art when suitably implemented, ensuring small space and bounded delay.

Different models of parallel computation have been proposed for the maximal clique problem. In [172] is described a shared-memory multicore algorithm, consisting entirely of data-parallel operations. Based on previous branch-and-bound solution, San Segundo et al. [249] propose a new algorithm combining bit-parallelism with a simple greedy pivot selection. A distributed approach based on a decomposition strategy is proposed in [64], providing experimental evidences of the efficiency and scalability of the algorithm. Another distributed approach is proposed in [332], which is built on a *share-nothing* style. In this strategy, each node in the architecture is completely independent and self-sufficient. Focused on MapReduce framework, parallel algorithms for listing maximal cliques are given in [286, 55, 319].

Maximum clique. Another variant of the clique listing problem is the *maximum* clique problem. It consists of listing the largest clique in a graph, i.e., a clique with maximum number of vertices. An extensive review of the literature is given in [328], providing a comprehensive description of exact and heuristic methods proposed before the date of publication of their work. After that, a new *selective coloring* heuristic for listing the maximum clique is introduced in [255], which is based on approximate vertex coloring. A different solution is described in [74], combining the concept of quantum computing [121] and evolutionary algorithm.

San Segundo et al. [253] propose a branch-and-bound approach that can prune the search space using an infra-chromatic upper bound. The bound is improved in [263], and the experimental results report that the new bound is significantly better than the state-of-the-art algorithms. Another branch-and-bound solution is given in [137], combining incremental MaxSAT reasoning [175] and an efficient preprocessing procedure. The branch-and-bound strategy is also exploited in [296], introducing a new approximate coloring technique to speed up the computation. Improvements for the algorithm proposed in [296] are described in [294]. Based on the argument that branch-and-bound algorithms present a discrepancy between the theoretical and experimental results, Zurge and Carmo [347] present a review those approaches. They show that a broad class of proposed solutions display sub-exponential average running time behavior. Furthermore, the authors introduce a structured methodology for the experimental analysis of the algorithms for the maximum clique problem.

Seeking to scale to larger graphs, parallel solutions have been proposed in the literature. A shared-memory approach is described in [243], which combines a branch-and-bound strategy with aggressive pruning technique. The authors provide the implementation details, analysis and performance evaluation of the introduced algorithm. In [251] is presented a bit-parallel approach, which exploits a novel compressed representation of the bit-encoded adjacent matrix applied in [254, 252]. Bit-parallel approaches are also proposed in [250, 248].

A distributed graph partitioning algorithm is proposed in [117]. The authors introduce a new graph partition method and design an algorithm based on Apache

Hadoop platform [19] (an open source implementation of the MapReduce programming model). A more recent open source distributed framework, called Apache Spark [284], was used in [92] to implement a new parallel approach. The algorithm exploits the multi-phase partitioning strategy, enabling iterative in-memory processing of graphs.

Maximum weight clique. The problem of listing the maximum weight clique is an extension to weighted graph of the maximum clique problem. It consists of enumerating the clique with largest weight in an input graph. Considering that the weight can be associated to vertices and/or edges, the following variants have been considered in the literature:

- **Maximum vertex weight clique:** Given a graph $\overline{G} = (\overline{V}, \overline{E})$ such that each vertex $v \in \overline{V}$ is associated with a positive integer weight $w(v)$, find a clique Q which maximizes $\sum_{v \in Q} w(v)$.
- **Maximum edge weight clique:** Given a graph $\overline{G} = (\overline{V}, \overline{E})$ such that each edge $\{x, y\} \in \overline{E}$ is associated with a positive integer weight $w(\{x, y\})$, find a clique Q which maximizes $\sum_{x,y \in Q} w(\{x, y\})$.
- **Maximum total weight clique:** Given a graph $\overline{G} = (\overline{V}, \overline{E})$ such that each vertex $v \in \overline{V}$ and each edge $\{x, y\} \in \overline{E}$ are associated with a positive integer weight $w(v)$ and $w(\{x, y\})$, respectively, find a clique Q which maximizes $\sum_{v \in Q} w(v) + \sum_{x,y \in Q} w(\{x, y\})$.

The problem of finding the maximum vertex weight clique is addressed in [173], introducing a new algorithm that exploits the maximum satisfiability (MaxSAT) techniques. MaxSAT reasoning is also exploited in [96] to introduce a new solution. In [317] is applied the *Binary Quadratic Programming* model to the maximum vertex weight clique problem, which is combined to the tabu search approach proposed in [318]. Heuristic methods are described in [316], being exploited to implement local search algorithms. In [48] is proposed an approach that can scale to massive graphs, which interleaves between clique construction and graph reduction. In the reduction phase, the graph is reduced by removing some vertices that are impossible to be in any clique of the optimal weight. A new solution is introduced in [120], which is based on *satisfiability* (SAT) technique combined to the CONFLICT-DRIVEN CLAUSE LEARNING algorithm [189, 205].

In [344] is described a new operator to be used in a local search approach. It is implemented in a tabu search algorithm to assess the usefulness of the proposed operator. Shimizu et al. [276] use algorithm engineering techniques to design a new branch-and-bound approach for listing the maximum vertex weight clique. Their method is divided in two phases: precomputation and branch-and-bound. Basically, in the first phase the weights of cliques in many subgraphs are calculated and stored, while in the second phase the problems are divided in small subproblems, pruning the unneeded ones. Another brunch-and-bound approach is given in [138], which is specially designed for large graphs. The algorithm implements a preprocessing phase to determine an initial vertex ordering and reduce the graph size, while the

incremental vertex-weighted splitting phase reduces the number of branches. More recent branch-and-bound algorithms are given in [174, 128].

Seeking to parallelize the computation, a distributed tabu search algorithm is proposed in [153]. In [89] is described a parallel metaheuristic for the maximum vertex weight clique problem. The proposed method is an optimization of the ant colony strategy, being implemented using Message Passing Interface (MPI). Nogueira and Pinheiro [211] introduce a new CPU-GPU local search heuristic, which is implemented to run in a CPU-only architecture and a hybrid CPU-GPU platform. The experimental analysis shows that the hybrid implementation performed better, achieving an average speed up of 12 times over the CPU-only implementation.

Focused on *maximum edge weight clique problem*, a recent review of approaches is described in [125], providing a description of mathematical optimization formulations and solution approaches. Although the techniques proposed to find the maximum vertex weight clique can be widely used to solve similar problems, they may fail to tailor the local search to specific structures. Based on that argument, in [182] is described an approach specialized to solve the *maximum edge weight clique problem*. The algorithm exploits the Deterministic Tournament Selections (DTS) strategy [198] to choose the edges with large weight, minimizing the search space. In [124] is described a quadratic optimization formulation approach, analyzing the characteristics of the proposed method in terms of local and global optimality. Li et al. [177] introduce three new heuristics and a local search algorithm to find the maximum edge weight clique.

The problem of listing the *maximum total weight clique* is address in [95]. The authors develop a new local search algorithm, providing an extensive experimental evaluation in large sparse vertex-weight and edge-weight graphs.

4.1.3 Bipartite cliques

A bipartite graph $\overline{G} = (\overline{U} \cup \overline{V}, \overline{E})$ is a graph with two distinguished disjoint sets of nodes $\overline{U}$ and $\overline{V}$, such that edges in $\overline{E}$ connect nodes in $\overline{U}$ to nodes in $\overline{V}$, but two nodes within the same set are not adjacent. Bipartite clique, also known as *biclique*, is an induced bipartite subgraph $Q \subseteq \overline{G}$ where every node in $\overline{U}(Q) \subseteq \overline{U}$ is connected to every node in $\overline{V}(Q) \subseteq \overline{V}$ by edges in $\overline{E}(Q) \subseteq \overline{E}$. Given an input bipartite graph $\overline{G}$ and two integer t and z, the problem of counting – and possibly listing – bicliques consists of counting all bipartite cliques of size $t + z$ with t nodes in $\overline{U}$ and z nodes in $\overline{V}$. In [256] is described an algorithm for counting *butterflies*, i.e., complete 2×2 bicliques. The proposed approach is a randomized algorithm to approximate the number of butterflies in a graph with provable guarantee on accuracy.

Butterflies are exploited by [259] to list dense subgraphs in a bipartite graph and detect the relations among them. The proposed approach is a peeling algorithm designed to enumerate maximal k-tips (vertex-based subgraph) and k-wings (edge-based subgraph). Given a subgraph $S = (U \cup V, E)$ such that U and V are the sets of disjoint nodes and E is the set of edges, S is a k-tip only if each node $u \in U$ is contained in at least k butterflies, and each pair of nodes $u, v \in U$ is connected by series of butterflies. By similar arguments, S is a k-wing only if every edge $(u, v) \in E$ is contained in at least k butterflies, and each pair of edges $(u_1, v_1), (u_2, v_2) \in E$ is connected by series of butterflies.

Besides the problem of counting all bicliques of fixed size, the maximal biclique problem has also been addressed in the literature. It consists of enumerating bicliques containing nodes that are not properly contained within any other larger biclique. Two algorithms for sparse bipartite graphs are presented in [185]. Given an input graph $\overline{G}$ with n nodes and m edges, the first approach runs in $O(\triangle^3)$ time and $O(n+m)$ space, while the second one runs in $O(\triangle^2)$ time and $O(n+m+N \cdot \triangle)$ space, where $\triangle$ is the maximum degree of $\overline{G}$ and N is the number of all maximal bicliques in $\overline{G}$. In [146] is described a depth-first strategy to enumerate maximal bicliques. The proposed approach starts with singleton nodes and expands each one by adding adjacent nodes to them one by one using a depth-first search.

A distributed approach is introduced in [208]. The parallel algorithm is designed for the MapReduce framework and implemented using the Apache Hadoop platform [19]. The main strategy of the algorithm is to cluster the input graph into smaller size subgraphs and process them in parallel. A variation of bicliques is proposed in [11], called c-isolated bicliques. In this case, both parts of the biclique are constrained with respect to their outgoing edges. Hence, it is designed an algorithm for listing all c-isolated *maximal* bicliques in a given bipartite graph, which is inspired by the technique for listing c-isolated cliques introduced in[133].

Focused on listing only the largest biclique, the maximum biclique enumeration problem has also been studied in the literature. In [271] are proposed scale reduction techniques for the maximum biclique problem. Those techniques can be combined with an exact algorithm to solve this problem optimality on large sparse networks. Furthermore, the proposed scale reduction techniques can also be applied for the maximum *edge* biclique problem, which consists of finding the biclique with largest number of edges. The maximum *edge* biclique problem is also addressed in [269], where is proposed a probabilistic algorithm based on the Monte Carlo subspace clustering method.

A special case of the maximum biclique problem focused on balanced bicliques has also been addressed, called the maximum *balanced* biclique problem. Given a bipartite graph $\overline{G} = (\overline{U} \cup \overline{V}, \overline{E})$, this problem consists of listing the largest biclique $Q \subseteq \overline{G}$ such that both disjoint nodes set of Q have the same number of nodes, i.e., $|\overline{U}(Q)| = |\overline{V}(Q)|$. In [195] is described how techniques from branch-and-bound algorithms for the maximum clique problem can be adapted to find the maximum balanced biclique. The proposed branch-and-bound approach is improved in [345], where is applied a new Upper Bound Propagation procedure inspired by [283]. An evolutionary algorithm with structure mutation is given in [336]. Moreover, a mutation operator is proposed to enhance the exploration during the local search process. In [342] is proposed a tabu search method combined with a graph reduction to find the maximum balanced biclique. The proposed approach employs the Constraint-Balanced Tabu Search algorithm to explore the search space and two bound-based dedicated reduction techniques to shrink progressively the given graph.

An extension to bipartite subgraph of the bipartite clique problem is studied in [287], called maximum weighted induced bipartite subgraph problem (WIBSP). Given a bipartite graph $\overline{G}$ and non-negative weights for the nodes, this problem consists of finding an induced subgraph $S \subseteq \overline{G}$ with the largest total weight. In the study, it is shown that the WIBSP can be reduced to the weight independent set problem. Thus, non-trivial approximation and exact algorithm for the WIBSP

can be obtained using the reduction and results about the weight independent set problem.

4.1.4 Sample graphs

Given a sample graph S of size k, the problem of listing sample graphs consists of enumerating all instances of S in a large undirected graph $\overline{G}$. This problem is also known as subgraph or motif listing.

Enumeration algorithm. The PARTITION algorithm for listing triangles proposed by Suri and Vassilvitskii [285] has been extended for listing subgraphs in [4]. The extended algorithm is a parallel one-round MapReduce approach for listing all non-induced instances of a subgraph S in a graph $\overline{G}$, designed based on the computation of multiway joins [5]. MapReduce-based framework is also proposed in [165], exploiting the edge-based join to allow multiple edges to join in each round. The input/output complexity of the problem when graphs do not fit in the main memory is the focus of [279], introducing two external memory algorithms. The first is a deterministic algorithm that exploits a matched independent set technique, while the second is a randomized algorithm exploiting the random coloring technique [216].

In [275] is described a distributed approach, called PSGL, for listing non-induced subgraphs, which exploits the divide-and-conquer strategy. The proposed algorithm is based on graph processing paradigm [186] and BULK SYNCHRONOUS PARALLEL [305], expanding the partial subgraph instances by data vertices in parallel until all instances are found. Another distributed approach based on MapReduce framework is described in [150], named TWINTWIGJOIN. Experimental evaluations are performed comparing TWINTWIGJOIN and the PSGL algorithm [275], showing that the introduced approach achieved the best results. In [166] is described a distributed algorithm called SEED, which is compared to TWINTWIGJOIN [150]. SEED returns the solution in a generalized join framework, avoiding the constraints in TWINTWIGJOIN. Experimental results show that SEED outperforms TWINTWIGJOIN. Shared-memory platform is exploited in [152]. The authors parallelize the sequential approach RI [43] and propose an improved version called RI-DS.

Works addressing the subgraph enumeration problem listed in this section can enumerate all non-induced instances of a given subgraph S in a large graph $\overline{G}$. However, the problem of listing all induced instances of a given subgraph S in $\overline{G}$ can not be solved by the described approaches.

Exact counting algorithm. The problem of counting, instead of listing, all instances of S in $\overline{G}$ has also been addressed in the literature. It consists of counting all instances of a given subgraph S on k nodes in a large graph $\overline{G}$, being not necessary to *touch* all instances of S during the process. In [258] is described an algorithm for counting independent instances of a specific subgraph in probabilistic biological network. A parallel algorithm for counting the number of occurrences of a given graph S on six nodes in a large graph $\overline{G}$ is given in [144]. The proposed approach is a GPU implementation of a distributed algorithm described in [40].

Approximate counting algorithms. Approximate approaches have been introduced in the literature seeking to speed up the running time. In [108] is proposed an estimating subgraph algorithm, employing the egocentric approach described in [321]. The algorithm exploits the probability sample of egocentric networks to estimate the subgraph frequency. A parallel approximate solution is given in [53], which implements the color-coding technique [15]. Another parallel approximate solution is described in [36], being implemented on parallel graph processing framework Pregel [186].

Detecting induced subgraphs. Given a sample graph S and a large graph $\overline{G}$, this problem consists of detecting whether S exists in $\overline{G}$, i.e., if $\overline{G}$ contains at least 1 *induced* instance of S. The problem of detecting a subgraph S of size 4 in a large graph $\overline{G}$ is addressed in [324], introducing a general randomized framework. Although the algorithm can detect the existence or not of a subgraph S in a large graph $\overline{G}$, it cannot list all induced instances of S in $\overline{G}$. The detection of small induced subgraph is also addressed in [99], which can find induced subgraph of fixed size k in a large graph $\overline{G}$.

4.2 Frequent subgraphs mining

In this section are discussed the works focused on mining frequent subgraphs. This problem consists of finding all subgraphs that appear frequently in a graph or collection of small graphs according to a given frequency threshold τ. Therefore, a subgraph is frequent if it has at least τ appearances in the graph.

In [239] is given a survey on frequent subgraph mining algorithm. The survey is focused on reviewing few existing scalable techniques to find frequent subgraphs in a collection of graphs or in a single large graph. A novel framework, called GraMi, for frequent subgraph mining in a single large graph is presented in [93]. The algorithm models the frequency evaluation as a *constraint satisfaction problem* (CSP). GraMi stores only the templates of frequent subgraphs and, at each iteration, solves the CSP until finds the minimal set of appearances that are enough to evaluate the subgraph frequency. This process is repeated by extending the subgraphs until no more frequent subgraphs can be found. In [206] is proposed a new version of GraMi. It is a hybrid algorithm that combines approximate structural graph matching and semantic graph matching.

A genetic framework is described in [85], which is an extension of [83] and [84]. The authors implement two instances of the framework: a breadth-first order and a pattern-growth approach. In [2] is proposed an algorithm that exploits the mapping sets method. This strategy allows to eliminate the isomorphism computation during the search for frequent subgraphs. An optimization for frequent subgraph mining approaches is given in [81]. The proposed technique, called filtration, exploits the property of repeating edges of subgraphs, allowing to eliminate the non-frequent subgraphs during the computation without calculating the exact support value. The filtration method is general purpose, which can be applied to any algorithm designed to address the frequent subgraphs mining.

Focused on a single large graph, in [80] is presented a brief survey of frequent

subgraph mining algorithms. The review describes the type of techniques used in each work and their respective phases. A more general survey is given in [289], focusing on algorithms for mining frequent subgraphs in different types of graphs. Moreover, the survey discusses distinct mining strategies and the types of frequent subgraphs produced by the previous works.

In [10] is proposed a new approach focused on real-time frequent subgraph mining. The algorithm can avoid redundant and false pattern checking and reduce costly isomorphism checking by indexing subgraphs with custom data structures. Demetrovics et al. [79] describes an approach that can solve the subgraph isomorphism in polynomial time in some settings. The proposed algorithm is based on canonical labeling strategy, Random Access Machine (RAM) model [260], and the *A prior-based* approach. The binary search and the isomorphism test procedures run in $O(\log n)$ and $O(\log |C_k^i|)$, respectively, where n is the number of nodes in the input graph and $|C_k^i|$ is the number of subgraphs candidates on k nodes. In [97] are proposed two hybrid bi-objective evolutionary solutions. The first method, called GASLS, combines genetic algorithm and stochastic local search, while the second approach, named GAVNS, combines genetic algorithm and variable neighborhood search. Experimental evaluation is performed, and the algorithm GASLS showed the best results. An approach for frequent subgraph mining in multigraphs is proposed in [131]. The algorithm implements a set of pruning rules to swiftly transverse the search space for multigraph pattern extraction.

Considering that graph mining is computationally very hard, parallel solutions for frequent subgraph mining have been proposed in the literature. A parallel algorithm on GPUs is given in [147], analyzing major challenges for GPU-based subgraph mining. The proposed approach is investigated in [290], where the algorithm in [147] is extensively analyzed and implemented using the Nvidia CUDA framework [212]. In [307] is described a parallel multicore approach, exploiting the gSpan strategy [333]. Another in-memory algorithm is described in [274], which executes on current multicore and multiprocessor machines. The proposed approach incorporates a fast heuristic with high-performance concurrent data structure in order to accelerate the detection and counting subgraphs. A parallel frequent subgraph mining algorithm is proposed in [334], adopting the *master-slave* parallel mode. In this case, master processor generates frequent subtrees and distributes them to slave processing nodes, dealing with the expansion of frequent subgraph edge and the isomorphism identification. The proposed approach can compute frequent subgraphs in $O(n^2 \cdot 2^n / k)$ time, where n is the number of nodes in the input large graph and k is the number of slave processors. The distributed programming framework Pregel [186] is exploited in [340] to introduce an algorithm for single massive graph. Moreover, the authors describe two optimizations processes applied in the proposed approach to reduce the communication cost and distribution overhead.

MapReduce framework is exploited in [179], presenting a two-step *filter-and-refinement* approach. The algorithm partitions the graph (or collection of graphs) among worker nodes in the *filter* step, hence each worker determines a set of locally frequent subgraphs. Then, the union of all local candidates is processed in the *refinement* step, storing only globally frequent subgraphs. Another MapReduce-based approach is proposed in [38], which is implemented using the Apache Hadoop platform [19]. Basically, the algorithm constructs and retains all patterns in a

partition that has a non-zero support in the map phase, and then it decides whether a pattern is frequent by aggregating its support in the reduce phase. In [317] is presented a two rounds MapReduce approach. The algorithm mines the locally frequent subgraphs in the first round, and then calculates the global frequency of each candidate subgraph in the second round. Talukder and Zaki [288] focus on a single massive graph that is too large to fit in the local memory at each compute node. Therefore, they propose a hybrid solution that exploits both thread-based parallelism within each compute node and distributed computation across multiple compute nodes. Their approach can scale to a massive graph with over a billion nodes and four billion edges. A more recent MapReduce-based approach is given in [229]. It exploits the breadth-first search strategy to iteratively extract frequent subgraphs. Moreover, new frequent subgraphs are generated without performing any isomorphism test, which is costly and imperative in existing approaches.

A different distributed approach is given in [230]. It utilizes the features provided by distributed in-memory dataflow systems such as Apache Spark [284] or Apache Flink [49]. Apache Spark is also exploited in [234], where is proposed a parallel frequent subgraph mining algorithm in a single large graph. The authors introduce a heuristic search strategy and three optimizations for the support computing operation. In [1] is described a novel parallel frequent subgraph mining system for a single large graph. The approach is divided in two phases: approximate and exact. In the approximate phase are identified subgraphs that are frequent with high probability, while in the exact phase is computed the exact solution based on the results of the approximation phase. Mohamed et al. [200] adapts the FSG approach [163] to a parallel version. The proposed solution is based on the parallelism model in cloud system by using *HoriVertical* partition.

Maximal frequent subgraphs mining. Due to the extremely large space needed to mine all frequent subgraphs, the *maximal* frequent subgraphs mining problem has been addressed in the literature. It consists of listing all maximal frequent subgraphs, i.e., frequent subgraphs that are not properly contained within any other larger subgraph. The computational complexity of this problem is the topic of [151]. This work provides an extensive analysis of the maximal frequent subgraphs enumeration considering the effect of three different parameters: possible restrictions on the class of graphs, a fixed bound on the threshold, and a fixed bound on the number of desired answers. An algorithm for listing maximal frequent subgraphs in a single graph is described in [100]. The proposed approach uses inexact matching strategy, which allows identifying maximal patterns with structural differences, in vertices and edges, respect to their occurrences. Focused on parallel platform, in [235] is proposed a multi-threaded algorithm, which is a parallelization of the Mule approach [161]. The algorithm decides whether a subgraph is maximally frequent by exploiting a depth-first search strategy, listing each frequent subgraph exactly once. A recent review of the frequent subgraph mining is given in [207]. An overview of different algorithms from a bioinformatics perspective is presented, as well as their potential biomedical application.

4.3 All subgraphs on k vertices

In this section are discussed the works addressing the problem of counting/listing subgraphs of fixed size. Given a large graph $\overline{G}$ and an integer k, this problem consist of counting – and possibly listing – all subgraphs on k nodes in $\overline{G}$. This problem is known in the literature as subgraphs, graphlets or motifs counting/enumeration problem.

Enumeration algorithms. A notable tool for listing all subgraphs on k nodes is given in [323], called FANMOD. The tool implements a novel algorithm described in [322]. It can enumerate all subgraphs up to eight nodes (i.e., $k \leq 8$), exporting the results to a variety of machine- and human-readable file formats. In [188] is described an algorithm for 4-node subgraphs, named RAGE. Although the proposed approach is based on non-induced subgraphs, the authors show how to calculate the count of induced subgraphs given the non-induced count. An experimental analysis is performed comparing the algorithm RAGE and the tool FANMOD [323], showing that RAGE achieved the best results. Connected induced subgraphs are addressed in [193]. The authors describe an algorithm that enumerates all connected induced subgraphs of a given size k, combining a local search tree strategy and a conventional depth-first approach. In [68] is tackled the problem of listing connected subgraphs with bounded girth in directed graph. The girth of a graph is defined as the length of its shortest cycle. The authors propose two algorithms for listing induced and non-induced subgraphs, respectively. Both proposed approaches run in $O(n|S|)$ time and use $O(n^3)$ space, where n is the number of nodes in the input graph $\overline{G}$ and S is the set of all solutions.

Focused on parallel platforms, distributed and multicore solutions have been proposed in the literature for listing subgraphs of fixed size. In [20] is described a parallel multicore version of the sequential algorithm FaSE [219]. This approach exploits the network-centric strategy, which is combined with a dynamic load balancing scheme in the parallel version to speed up the computation. Another multicore algorithm is given in [273], which enumerates all induced subgraphs using edges instead of vertices. This strategy leads to a load-balanced enumeration approach efficiently executed on current multicore and multiprocessor machine. In [197] is described a multithread algorithm. The proposed approach is an extension of [196] that can enumerate all induced subgraphs up to 6 nodes. Distributed MapReduce-based solution is given in [272]. It exploits a heuristic method for subgraph isomorphism detection, being implemented on the Apache Hadoop framework [19].

The problem of enumerating bipartite subgraphs in an undirected simple graph $\overline{G}$ is addressed in [320]. The bipartite subgraph enumeration problem tackled consists of, for a given graph $\overline{G}$ and a constraint R, listing all bipartite subgraphs of $\overline{G}$ exactly once. In the work are proposed two algorithms for listing all bipartite *induced* subgraphs and all bipartite subgraphs, respectively. The first approach can compute all induced subgraphs in a bipartite graph with degeneracy k in $O(k)$ time per solution, while the second solution can compute all subgraphs in $O(1)$ per solution.

Exact counting algorithms. Although the enumeration problem has several applications in the complex network analysis, the problem of counting, instead of

listing, all subgraphs on k nodes has also been addressed in the literature. It consists of counting the number of subgraphs on k nodes without the need of enumerating each one of them.

A combinatorial method for counting subgraphs, called ORCA, is described in [122]. It exploits a system of equations introduced by [154] to compute all subgraphs up to 5 nodes. Hence, given an input graph $\overline{G}$ on n nodes and m edges, the algorithm can count all subgraphs on 4 nodes in $O(m \cdot d + T_4)$ time and $O(m + n)$ space, where d is the maximal node degree and T_4 is the time needed to enumerate all 4-cliques. Subgraphs on 5 nodes are counted in $O(m \cdot d^2 + T_5)$ time and $O(m \cdot d)$ space, where T_5 is the time needed to enumerate all 5-cliques. A generalization of ORCA for counting subgraph of any size is given in [123]. In [196] is proposed an algorithm for counting subgraphs of size $k+2$ based on the set of induced subgraphs on k nodes. This approach can compute 3, 4 and 5-node subgraphs in directed graphs in $O(\alpha(\overline{G})m)$, $O(m^2)$ and $O(nm^2)$ time, respectively, where $\alpha(\overline{G})$ is the arboricity of the input graph. Ortmann and Brandes [215] present an algorithm for counting all induced and non-induced subgraphs of size 4 on a per-node and per-edge basis. This solution combines a system of equations and the algorithms K3, C4 and COMPLETE proposed by Chiba and Nishizeki [59], requiring $O(\alpha(\overline{G})^2 m)$ time for counting all 4-node subgraphs.

In [231] is proposed an algorithm, called ESCAPE, to enumerate all 5-node subgraphs. Its main strategy is to cut a subgraph into smaller ones, and using counts of smaller subgraphs to get larger counts. The proposed approach counts all 5-node subgraphs by listing four specific subgraphs, which three of them have less then 5 nodes. Given an undirected graph $\overline{G}$ with n nodes and m edges, the algorithm first construct the degree ordered directed graph $G^{\rightarrow}$ by orienting all edges in $\overline{G}$. Then, it counts all *connected* 5-node subgraphs in $\overline{G}$ in $O(W(\overline{G}) + D(\overline{G}) + DP(G^{\rightarrow}) + DBP(G^{\rightarrow}) + m + n)$ time and $O(n + m + T(\overline{G}))$ space, where $W(\overline{G})$ is the time required to count all wedges, $D(\overline{G})$ is the time required to count all diamonds, $DP(G^{\rightarrow})$ is the time required to count all directed 3-paths, and $DPB(G^{\rightarrow})$ is the time required to count all directed bipyramids. A general purpose tool for detection and analysis of subgraphs is described in [17], which is a Java library, designed for extensibility and sustainability.

Parallel solutions for counting, without listing, all subgraphs of size k have also been proposed in the literature. A multicore algorithm is given in [21]. The proposed approach exploits the *g-trie* data structure [242], which is implemented using Pthreads [210]. Ahmed et al. [7] propose a parallel multicore approach for counting connected and disconnected subgraphs of size 3 and 4. For each edge, the algorithm counts a few subgraphs, which are used to obtain the exact counts of others in constant time by combinatorial arguments. Given a graph $\overline{G}$ with n nodes and m edges, their algorithm counts all subgraphs on 3 and 4 nodes in $O(m \cdot \triangle)$ and $O(m \cdot \triangle \cdot T_{max} + m \cdot \triangle \cdot S_{max})$ time, respectively, where $\triangle$ is the maximum degree in $\overline{G}$, T_{max} is the maximum number of triangles incident to an edge, and S_{max} is the maximum number of stars incident to an edge.

Another shared-memory multicore approach is described in [77]. It can count all subgraphs on 3, 4 and 5 nodes considering all possible edge orbits of a subgraph. It allows the approach to enumerate 4 out of 8 subgraphs on 4 nodes, and 14 out of 32 subgraphs on 5 nodes, being the number of remaining subgraphs obtained in

constant time by combinatorial calculation. The proposed approach can count all connected 5-node subgraphs in $O((T^{max} + N_u^{max} + N_v^{max})^3)$ time, where T^{max} is the largest value of $|T| = \Gamma(u) \cap \Gamma(v)$, while N_u^{max} and N_v^{max} represent the largest value of $|N_u| = \Gamma(u) \backslash T$ and $|N_v| = \Gamma(v) \backslash T$, respectively. CPU and GPU parallelizations are the topic of [244]. Inspired by the parallel CPU-based algorithm PGD [7], the authors introduce three different parallel approaches: single-GPU, multi-GPU and hybrid CPU-GPU. A distributed MapReduce-based solution is given in [86], being implemented using Apache Spark [284]. The main strategy of the algorithm is to apply an adaptive time threshold and an efficient work-sharing mechanism to dynamically do load balancing between the workers.

Approximate counting algorithms. Besides enumeration and exact counting algorithms, approximate solutions have also been proposed in the literature for counting subgraphs on k nodes. The main objective of the approximate approaches is to estimate as accurately as possible the number of all subgraphs on k nodes in an input large graph $\overline{G}$.

Two sampling methods are described in [311] for estimating subgraphs statistics. The first one can estimate the number of connected induced subgraphs on k nodes for any value of k, while the second approach can jointly count the number of subgraphs on $k-1$, k, and $k+1$ for any $k \geq 4$. Focused on 4-node subgraphs, in [136] is described a *path sampling* approach. The proposed algorithm approximates the number of 4-node subgraphs applying a novel technique of *3-path sampling* and a special pruning scheme. In [237] is proposed an approximate algorithm, called GRAFT, to count approximately all k-node subgraphs for $k \leq 5$. GRAFT counts the subgraphs by iterating over a random subset of edges of the input graph. Hence, the lower the sampling factor, the faster the algorithm runs. However, the higher the sampling factor, the better the accuracy of GRAFT. Subgraphs on 4 and 5 nodes are tackled in [313]. The proposed method can sample and count 4- and 5-node subgraphs, providing unbiased estimators of subgraph frequencies. Two approximate algorithms based on random walk strategy are proposed in [247]. They exploits the MONTE CARLO MARKOV CHAIN sampling method over the candidate subgraph space.

A general framework to estimate subgraph statistics of any size is presented in [56], which exploits the random walk strategy. Furthermore, two optimization techniques are proposed to improve the accuracy of the framework. Another random walk based approach is given in [57], which generates samples by leveraging consecutive steps of the random walk as well as by observing neighbors of visited nodes. In [119] is described a sampling algorithm for counting connected and disconnected subgraphs up to 8 nodes. A new sampling method for counting subgraphs on 5 nodes is given in [314], while sampling methods for subgraphs of any size are described in [58, 118]. In [218] is proposed a Monte Carlo sampling algorithm. It can simultaneously sample all subgraphs of size up to k nodes.

Bressan et al. [45] provide a theoretical and experimental comparison of two popular approaches for counting approximately the number of subgraphs: MONTE CARLO MARKOV CHAIN (MC) and COLOR CODING (CC) [15]. Although MC method is very efficient in terms of space, a carefully engineered version of CC approach presented the best results. In [155] are studied two of the most widely used sampling

schemes: subgraph and neighborhood sampling. The subgraph sampling strategy consists of sampling each node independently with probability p and observes the subgraph induced by the sample nodes, while the neighborhood sampling method additionally observes the edges between the sampled nodes and their neighbors.

Focused on parallel computation, in [8] is described an exact counting framework for subgraphs on k nodes. The exact approach is combined with a statistical unbiased estimation framework with provable error bounds for computing local subgraphs statistics approximately. A distributed algorithm for k-node subgraphs counting is proposed in [281]. It is a distributed-memory version of the shared-memory algorithm FASCIA described in [280], which exploits the color coding strategy. Subgraphs on 3 nodes are tackled in [90], describing a distributed solution implemented in GraphLab PowerGraph framework [110]. Moreover, the concept of *edge pivoting* is introduced, being exploited by the parallel solution. The *edge pivoting* strategy is also exploited in [91], describing a distributed approach for counting approximately the number of 4-node subgraphs. The proposed approach is a distributed message-passing scheme, which is implemented using GraphLab PowerGraph framework [110]. Another distributed approach is given in [192]. Two techniques are also introduced, *multi-phased* sampling and *cost-aware* sampling, reducing the query time on large graphs with less than 1% relative error. In [245] is proposed an unbiased subgraph estimation framework. It can be implemented in both shared and distributed memory architecture, presenting an effective accuracy with less than 1% relative error.

4.4 Clique relaxations

In this section are discussed the works addressing clique relaxation problems. Considering that a k-clique (small complete subgraphs on k nodes) is too restrictive in common real-life scenarios, different types of relaxations equally useful have been introduced in the literature. The origins of clique relaxation concepts are discussed in [225], providing a brief overview of mathematical programming formulations for different clique relaxation problems. According to the characteristics of their problems, the works discussed in this section are presented in three main clique relaxation categories: vertex-based relaxations in Section 4.4.1, edge-based relaxations in Section 4.4.2 and density-based relaxations in Section 4.4.3.

4.4.1 Vertex-based relaxations

In this section are discussed the works addressing the vertex-based clique relaxations. Those problems consist of listing small subgraphs S of size s such that the degree of every vertex in S is relaxed according to a given threshold $k \leq s - 1$. In this category is included the problem of listing k-core and k-plex subgraphs.

***k*-core.** Given an input graph $\overline{G}$, the k-core of $\overline{G}$ is the largest induced subgraph in which every vertex has degree at least k. Hence, the problem of k-core decomposition of the graph is to find all the k-cores of the graph. A study of patterns and anomalies related to k-cores is presented in [277]. It introduces three empirical patterns that govern k-cores or degeneracy across a wide variety of real-world graphs, providing practical uses of those patterns.

In [148] is described an extensive analysis of previous proposed approaches for k-core decomposition, which aims to explore whether k-core decomposition on large networks can be computed using a consumer-grade PC. Several implementations have been evaluated, and the Batagelj and Zaversnik algorithm [32] implemented on Webgraph framework [42] presented the best performance. An experimental analysis of Batagelj and Zaversnik algorithm [32] is performed in [75]. Moreover, it is introduced a new multicore approach, called ParK. Another multicore algorithm is given in [139], called PKC. The proposed approach can reduce the synchronization overhead and creates a smaller graph to process high degree vertices. An experimental analysis is carried out comparing the performance of PKC with Batagelj and Zaversnik algorithm [32], ParK [75] and the distributed approach MPM [202], showing that the PKC achieved the best performance.

Focused on distributed platform, in [187] is described a parallel solution for k-core decomposition, being implemented on Apache Spark [284]. The proposed approach is based on "think like a vertex" paradigm, which is an iterative execution framework provided by Pregel [186]. A MapReduce-based algorithm for approximate k-core decomposition is introduced in [94]. It is a sketching technique based on edge sampling strategy for computing a $1 - \epsilon$-approximate k-core for all k simultaneously.

The problem of (k, r)-core is addressed in [339]. It is a variant of k-core which consists of finding cohesive subgraphs on social networks considering both user engagement and similarity perspective. Therefore, given an attributed graph $\overline{G}$, a connected subgraph S in $\overline{G}$ is a (k, r)-core if and only if it satisfies both structure and similarity constraints. Several algorithms are proposed to enumerate all maximal (k, r)-cores and find the maximum (k, r)-core, being evaluated using four real-world datasets.

***k*-plex.** Given an input large graph $\overline{G}$ and a positive integer k, a k-plex is a subgraph S in $\overline{G}$ such that every vertex u in S is a neighbor of at least $|S| - k$ vertices in S. In other words, each vertex in a k-plex S can miss at most k neighbors in S. A clique is a special case of a k-plex, where 1-plex of size s correspond to a clique on s vertices.

Based on the definition of k-plex, the problem of enumerating all *maximal* k-plexes has been proposed in the literature. Algorithm engineering techniques are used in [35] to design two algorithms for listing maximal k-plexes and maximal *connected* k-plexes, respectively. The algorithms are based on the work described in [63], being applied a method to optimize the generic approach. Focused on densely connected k-plexes for non-small k, in [337] is proposed an algorithm for enumerating maximal k-plexes that can avoid small k-plexes and non-dense medium k-plexes. Densely connected k-plexes are also the target of [338], where is described an algorithm for enumerating densely connected k-plexes in networks. In [66] is introduced a solution for listing all *large* k-plexes, i.e. all k-plexes non-smaller than an input integer value i. The main idea of the algorithm is to find large k-plexes by looking in the neighborhood of cliques of a size that depends on k and i. A generalization of sequential algorithm for maximal clique enumeration to handle maximal k-plex enumeration is described in [319]. Parallel computation is exploited in [69], where is proposed a distributed algorithm able to find larger k-plexes of very

large graphs in just a few minutes.

Besides maximal k-plex enumeration, the problem of listing *maximum* k-plex has also been addressed in the literature. It consists of enumerating the largest k-plex in an input graph $\overline{G}$ for all possible value of k, i.e., listing all maximum k-plexes. Maximal and maximum enumeration solutions are proposed in [65]. The main idea is to enumerate all maximal k-plexes and then selecting the largest ones. Combinatorial algorithms are proposed in [194] focused on maximum k-plexes, introducing heuristics and exact approaches. Another combinatorial algorithm is described in [204], which improves the depth-bounded search tree approach introduced by [159].

A study of maximum k-plexes is given in [31], where the problem is formulated as a binary integer program. Moreover, a branch-and-cut framework is implemented based on classes and valid inequalities and facets introduced in the study. In [104] is proposed an exact algorithm that can deal with large-scale graphs. Several graph reduction methods and heuristic strategies are introduced, which are integrated into a branch-and-bound search algorithm. Structural properties of maximum k-plexes are investigated in [330]. It is also described a branch-and-bound algorithm, being theoretically and experimentally evaluated. In [343] is proposed a tabu search approach to enumerate maximum k-plexes in very large networks. The algorithm exploits two transformation operators to locate high-quality solutions and a frequency-driven perturbation operator to escape and search beyond the identified local optimum.

The maximum k-plex problem is also addressed in edge-weight graphs, which is known as maximum edge-weight k-plex problem (Max-EkPP). Given a graph $\overline{G}$ such that each edge is associated with a positive integer weight, this problem consists of listing k-plexes with largest total weight of edges. The Max-EkPP was introduced in [190], where the author also provides a linear programming formulation. The first heuristic method for the Max-EkPP is proposed in [111]. It is based on the variable neighborhood search metaheuristic, implementing a objective function which takes into account the degree of every edge.

4.4.2 Edge-based relaxations

In this section are discussed the works addressing the edge-based clique relaxations. In this category is included the problem of listing k-truss subgraphs and the truss decomposition problem.

***k*-truss.** The cohesive subgraph k-truss was introduced by Cohen [62], being considered an extension of the clique. It consists of a connected subgraph S such that each edge in S is incident to at least $k-2$ triangles. Notice that a clique of order k is a k-truss. In [62] is also proposed the problem of listing maximal k-truss, i.e., k-truss that is not a proper subgraph of another larger k-truss. In [127] is proposed a preliminary version of an approach for maximal k-truss, being implemented in a multithread platform. Focused on GPU computation, in [114] is proposed an approach using algorithm engineering techniques for finding both the k-truss of the graph for a given k and the maximal k-truss using a dynamic graph formulation. Although the authors present an implementation for the NVIDIA GPU, the proposed approach is architecture independent.

Based on the concept of k-truss, in [341] is introduced a novel community model, called weighted k-truss community. In addition to the characteristics of the k-truss problem, this model takes the edge weight into consideration. To address the new community model, it is described an algorithm to find *top-r* weighted k-truss communities. Given an undirected edge-weighted graph $\overline{G}$ and parameters r and k, this problem consists of listing all the top-r weighted k-truss communities with largest weights. The strategy of the algorithm is to remove iteratively the smallest-weight edge from the maximal k-truss to generate all weighted k-truss communities one by one. The proposed approach requires $O(m^{3/2})$ time, where m is the number of edges in the input graph. In [145] is described a linear-time algorithm for listing top-r k-truss communities in an undirected unweighted graph $\overline{G}$. Given a graph $\overline{G}$, a vertex $v \in \overline{G}$ and integers r and k, this problem consists of finding top-r k-truss communities containing v.

A novel dense subgraph model combining k-core and k-truss is proposed in [178], called k-core-truss. It is based on a new concept of important edges. Therefore, the k-core-truss of a graph $\overline{G}$ is the largest subgraph S in $\overline{G}$ such that every edge has the importance value at least k in S.

Truss decomposition. Given a graph $\overline{G}$, the truss decomposition consists of finding all largest k-trusses of $\overline{G}$ for all values of k. In [309] is described an in-memory algorithm for truss decomposition. It achieves the same worst-case complexity of the in-memory triangle listing approach [285], requiring $O(m^{3/2})$ time and $O(m+n)$ space on graphs with n vertices and m edges. Probabilistic graphs are the topic of [129], being designed a dynamic programming approach.

Parallel solutions have also been proposed for truss decomposition. In [140] is introduced a multicore approach for large sparse graph. It is a level-synchronous parallelization of the sequential approach for k-truss decomposition described in [309]. Another multicore solution is proposed in [331], which applies a new optimization method to achieve a better parallelization. In [282] is proposed a shared-memory parallel algorithm based on peeling. It breaks a serial approach into several bulk-synchronous parallel steps and then uses a *multi-stage peeling* method. Collaborative CPU+GPU approaches for triangle counting and truss decomposition are given in [76]. Moreover, it is described a comparison of the memory management schemes offered by Nvidia CUDA [212] and NVLink [101]. In [54] is proposed a distributed algorithm for k-truss decomposition. It is designed in the MapReduce framework based on the previous work [61].

Bounds and algorithms for k-truss are discussed in [47]. Moreover, two new approaches are proposed. The first is inspired by [309], using linear memory and requiring $O(m^{3/2})$ time. The second is a matrix multiplication method, which avoids enumerating all the triangles in the input graph and can achieve running times significantly below $m^{3/2}$.

4.4.3 Density-based relaxations

In this section are discussed the works addressing problems related to density-based clique relaxations. In this category are included s-clique, k-club, quasi-clique, bipartite clique and bipartite subgraphs.

***s*-clique.** Given an input graph $\overline{G}$, let $\text{dist}_{\overline{G}}(u,v)$ be the length of the shortest path (number of edges) between nodes u and v in $\overline{G}$. A subgraph $S \subseteq \overline{G}$ is a s-clique only if, for all pair of nodes $u, v \in S$, it holds that $\text{dist}_{\overline{G}}(u,v) \leq s$. Notice that, for $k = 1$, a k-clique is a clique (complete subgraph), where all pair of nodes in S are adjacent. A s-clique is also known in the literature as n-clique or k-clique.

Exact and approximation algorithms for the s-clique problem are proposed in [142]. The exact approach is based on branch-and-bound strategy, which can list all s-cliques. The approximation algorithm produces s-clique with 2-approximation. It can produce all or top-k s-cliques in polynomial delay. Another polynomial delay algorithm is proposed in [34] for listing all *maximal* s-cliques, i.e., s-cliques S such that they are not properly contained within any other larger s-clique. Furthermore, several variants of the well-know Bron-Kerbosch algorithm [46] for maximal clique enumeration are introduced. In [195] is described an approach for enumerating the *maximum* s-clique, i.e., the largest s-clique. The proposed algorithm is an adaptation of the maximum clique algorithm described in [296] with an introduced lazy global domination rule.

***k*-club.** Given a subgraph S, let $\text{diam}_S(u,v)$ be the length of the shortest path (number of edges) between nodes u and v in S. Given a positive integer k, a k-club is a subgraph S in $\overline{G}$ such that $\text{diam}_S(u,v) \leq k$ for all pairs of nodes $u, v \in S$. In other words, S is a k-club if and only if the shortest path between every two nodes u, v in S is composed of at most k edges. Notice that a 1-club is a clique, where all pair of nodes are connected. Although the definitions of s-clique and k-club are similar, $\text{dist}_{\overline{G}}(u,v)$ and $\text{diam}_S(u,v)$ have different definitions. In a s-clique $S \subseteq \overline{G}$, each pair of nodes $u, v \in S$ is connected by a path of length at most s ($\text{dist}_{\overline{G}}(u,v) \leq s$), where that path may use any node in $\overline{G}$. However, in a k-club $S' \subseteq \overline{G}$, each pair of nodes $u, v \in S'$ is connected by a path of length at most k ($\text{diam}_{S'}(u,v) \leq k$) such that all nodes in the path are also in the k-club. Based on the definition of k-club, the problem of listing the k-club with *maximum* cardinality for a given integer k has been addressed in the literature.

A combinatorial branch-and-bound algorithm is described in [183]. It combines a distance coloring based upper-bounding scheme and a bounded enumeration based lower-bounding routine for enumerating the largest k-club. In [51] is described a new heuristic approach and a dynamic data structure that maintains the k-neighborhood for each node of a graph. Moreover, four tricks for the implementation of the branch-and-bound algorithms are presented in [44, 183]. A new variable neighborhood search heuristic is described in [270]. It is incorporated into the branch-and-bound algorithm introduced by [183] to create a new exact approach for the maximum k-club listing problem.

An study of maximum k-club enumeration in different graphs is given in [109]. The authors prove that, for a given graph $\overline{G}$ and an integer k, a maximum k-club in $\overline{G}$ can be computed in polynomial time when $\overline{G}$ is a chordal bipartite, a strongly chordal or a distance heredity graph. Moreover, on a superclass of these graphs, polynomial-time algorithm can be obtained when k is odd. In [13] is described a comparative study of models for the maximum k-club problem. A linear programming relaxation standpoint is used to compare integer formulations

proposed in the literature. Besides the comparison of existing approaches, it is described two enhanced compact formulations for $k = 3$.

Heuristic methods for listing the maximum k-club are described in [14]. All proposed heuristics are composed of two phases, combining simple construction schemes and exact optimization of restricted integer models. In [326] are proposed two exact methods. The problem of listing the maximum k-club is encoded as an instance of the *partial MAX-SAT* problem, where two *MAX-SAT* formulations are designed to tackle it. Graph decomposition and model decomposition techniques are combined in [203] to prove that the maximum k-club problem can be solved optimally on large-scale graphs. The proposed method combines a simple relaxation based on necessary conditions with canonical hypercube cuts described in [30].

Approximation polynomial-time approaches for the maximum k-club problem are given in [28, 26]. The approximation ratio of [28] is $O(n^{1/2})$ and $O(n^{2/3})$ for any *even* $k \geq 2$ and any *odd* $k \geq 3$, respectively, where n is the number of nodes in the input graph. However, the algorithm proposed in [26] achieves an optimal approximation ratio of $O(n^{1/2})$ for any $k \geq 2$. Implementation and experimental evaluation of both algorithms [28, 26] are given in [27]. Focused on $k = 2$, in [160] is investigated the algorithmic complexity for three variants of well-connected 2-clubs: robust, hereditary, and "connected" 2-clubs. To address this problem, it is designed an exact combinatorial algorithm, being experimentally evaluated on real-world graphs. A variation of the maximum k-club listing problem is described in [50], called maximum triangle k-club problem. For a given input graph $\overline{G}$ and an integer k, this problem consists of listing the maximum k-club induced subgraph S such that all nodes in S belong to at least one triangle in S. To tackle the proposed problem, integer programming formulations are introduced, which are stated in different variable spaces.

Quasi-clique. Given a graph $\overline{G}$ and an induced subgraph $S \subseteq \overline{G}$, let $e[S]$ be the number of edges in the induced subgraph S. According to [3], given a threshold γ such that $0 < \gamma \leq 1$, a subgraph $S \subseteq \overline{G}$ is a γ-clique (quasi-clique) only if S is connected and $e[S] \geq \gamma\binom{|S|}{2}$. Based on this definition, the problem of enumerating the *maximum* quasi-clique in an input large graph has been addressed in the literature, which consists of listing the largest quasi-clique.

A combinatorial branch-and-bound approach for the maximum quasi-clique problem is described in [184]. The proposed algorithm is a classical depth-first search based on the method described by [162]. Mixed integer programming formulations are introduced in [306], designing four different solutions. In [232] is proposed a biased random-key genetic algorithm for the maximum quasi-clique problem, where two variants of the proposed approach are implemented using two alternative decoders.

A variation of the quasi-clique problem is proposed in [300], called optimal quasi-clique. This problem consists of finding the *best* quasi-clique, i.e., a subgraph $S \subseteq \overline{G}$ that maximized the function $f_\gamma(S) = e[S] - \gamma\binom{|S|}{2}$. Furthermore, two algorithms for extracting optimal quasi-cliques are introduced: greedy and heuristic algorithms. In [169] is addressed the query-driven maximum quasi-clique problem. It consists of finding the largest quasi-clique containing a given query node set. To solve this problem, the notion of *core tree* is proposed to organize dense subgraphs recursively,

as well as three refinement operations are presented to optimize a solution. Another variant of the quasi-clique problem is addressed in [223], called degree-based quasi-clique problem. Given a graph $\overline{G}$ and an induced subgraph $S \subseteq \overline{G}$, instead of considering the edge density $e[S]$, this problem considers the degree of nodes in S. Hence, given a threshold γ, S is a degree-based quasi-clique only if S is connected and the degree of any node in S is at least $\gamma(|S| - 1)$. Two exact approaches are proposed for finding the *maximum* degree-based quasi-clique: a branch-and-bound based solution and a degree decomposition algorithm.

Chapter 5

Enumerating cliques in parallel

In this chapter we introduce a new parallel clique enumeration algorithm. The proposed approach is described and theoretically analyzed in Section 5.1. Experimental setup is given in Section 5.2. In Section 5.3 is described the implementation of the proposed algorithm, while experimental results are given in Section 5.4.

5.1 A general parallel framework

In this section is described a work-efficient parallel algorithm to compute k-cliques. Though the focus is on counting, the algorithm can be easily adapted to the listing problem. We assume that the input undirected graph has been preprocessed as described in Chapter 2: this is a standard preliminary step that can be performed efficiently in parallel (see, e.g., [278]), independently of k. In Section 5.1.1 is described the clique counting algorithm. Main implementation choices are given in Section 5.1.2, while the analysis of the introduced algorithm is presented in Section 5.1.3.

5.1.1 Algorithm

The proposed approach can be regarded as an extension – to cliques with an arbitrary number of nodes and to a parallel setting – of the FORWARD algorithm for triangle counting described in [167]. It exploits the basic idea of t-expansion, implemented through neighborhood intersection for the sake of efficiency.

In a t-expansion, t new nodes are added to an existing clique on h nodes.

Claim 1. *Let h and t be two positive integer values. Let $U_h = \{u_1 \dots u_h\} \subseteq V$ be a set of h nodes of G such that $(u_i, u_j) \in E$ for each $i < j$. Let $W_t = \{w_1 \dots w_t\} \subseteq V$ be a set of t nodes of G such that:*

1. *$(w_i, w_j) \in E$ for each $i < j$, with $i, j \in [1, t]$;*
2. *$w_i \in \Gamma^+(u_1) \cap \dots \cap \Gamma^+(u_h)$ for each $i \in [1, t]$.*

Then the graph induced by $U_h \cup W_t$ in $\overline{G}$ is a clique on $h + t$ nodes.

Figure 5.1 illustrates t-expansion for $h = 3$ and $t = 2$. Edges are directed according to the total order $\prec$ introduced in Chapter 2. Nodes in U_3 are $\{u_1, u_2, u_3\}$

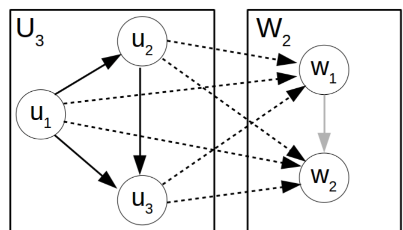


Figure 5.1. Example of t-expansion as described in Claim 1.

such that $u_2 \in \Gamma^+(u_1)$ and $u_3 \in \Gamma^+(u_1) \cap \Gamma^+(u_2)$. The dashed arrows from U_3 to W_2 represent condition 2 of Claim 1: since no arrow is missing, all nodes of W_2 are adjacent to nodes of U_3 in $\overline{G}$. Moreover, since $w_2 \in \Gamma(w_1)$ due to condition 1, the union of the two sets forms a 5-clique in the undirected graph $\overline{G}$ with nodes $\{u_1, u_2, u_3, w_1, w_2\}$, where $u_1 \prec u_2 \prec u_3 \prec w_1 \prec w_2$.

Our approach is to start from small cliques and expand them by progressively adding t new nodes until a clique of size k is obtained. This is implemented as follows. Assuming that the intersection $C = \Gamma^+(u_1) \cap \ldots \cap \Gamma^+(u_h)$ has been previously computed, all subsets of t nodes are taken from C. Notice that condition 2 is automatically satisfied for each node in any of these subsets. Then, if the t subset nodes also satisfy condition 1, the computation is forked passing to the new thread the intersection $C \cap \Gamma^+(w_1) \cap \ldots \cap \Gamma^+(w_t)$.

The core of the algorithm, based on the fork/join paradigm, is shown in Figure 5.2. The `fork` command is used to start a new thread and compute the next iteration. The first invocation is CountClique(k, t, G, V, 0), where k and t are the clique and the clique expansion size, respectively (assuming $k \geq 2$ and $t < k$), G is the directed graph, V contains candidate nodes (at the beginning, all graph nodes), and 0 is the current clique size. At lines 3 to 6 we check if we need less then t nodes to complete k-cliques: in this case we increase the number q of cliques by the number of subsets W of size $k - h$, with nodes in W satisfying condition 1.

If we are not in a base step, we compute lines 7 to 15. At lines 8 to 11, for each possible set $W \subseteq C$ of size t satisfying condition 1, a new thread τ is started. Candidate nodes C' processed by τ result from the intersection between C and the neighborhoods of all nodes in W (if $h = 0$, i.e., at the first invocation of CountClique, $C = V$ is not taken into account in the intersection). At lines 12 to 15, when all forked computations are done (i.e., each spawned thread has completed its execution and joined its parent), the partial results are summed.

We remark that the algorithm does not explicitly check for condition 2 of Claim 1 thanks to the use of a data structure – the neighborhoods' intersection set C – that explicitly stores candidates: if a node $v \in C$, then v is guaranteed to be in the high-neighborhood of all the previously selected nodes.

5.1.2 Main implementation choices

A variety of aspects related to the algorithm implementation have not been specified in Section 5.1.1. In particular, we discussed neither subset computation nor neighborhood intersection algorithms, which are addressed below.

Algorithm 3. COUNTCLIQUE

```
1: function CountClique(clique size k, clique expansion size t, directed graph G = (V, E),
                        candidates C = V ∩ Γ(u_1) ∩ ... ∩ Γ(u_h), current clique size h)
2:     q ← 0                                                    ▷ Local number of cliques
3:     if h + t ≥ k then
4:         for each subset W ⊆ C of size k − h do
5:             if nodes in W are mutually adjacent then         ▷ Condition 1 in Claim 1
6:                 q ← q + 1
7:     else
8:         for each W = {w_1 ... w_t} ⊆ C of size t do
9:             if nodes in W are mutually adjacent then         ▷ Condition 1 in Claim 1
10:                C' ← C ∩ Γ(w_1) ∩ Γ(w_2) ∩ ... ∩ Γ(w_t)       ▷ Neighborhoods' intersection
11:                fork CountClique(k, t, G, C', h + t)
12:        for each thread τ forked at line 11 do               ▷ Aggregate partial clique counts
13:            join τ
14:            let q' be the number of k-cliques computed by τ
15:            q ← q + q'
16:    return q
```

Figure 5.2. Fork/join algorithm for computing the number of k-cliques by t-expansions.

Subset enumeration. The number of subsets of size t to be enumerated at line 8 is $\binom{|C|}{t}$. When this is large, it may be worth to enumerate subsets in parallel in order to minimize span. To this aim, COUNTCLIQUE is combined with an adaptation of the parallel subset enumeration algorithm SUBSETLIM introduced in [82]. The SUBSETLIM approach enumerates subsets in lexicographic order: subset positions in the ordering are used as indexes to divide subset into groups that are then distributed among different threads. In the implementation, the maximum group size σ is fixed to 10 thousands.

COUNTCLIQUE, when combined with SUBSETLIM, works as follows. Consider set C' obtained by neighborhood intersection at line 10 in Figure 5.2: instead of forking a unique thread with input C' at line 11, the number of t-size subsets in C' is computed and divided into γ groups each containing at most σ subset indexes (following the lexicographic order). Then, γ new threads are forked, one per group. When the execution of each spawned thread reaches line 8, it will enumerate at most σ subsets of C', which is its input set of candidates.

For a simple example, consider the set of nodes $C' = \{1, 2, 3, 4, 5, 6\}$, $t = 3$ and $\sigma = 5$. In this case, the number of subsets of size 3 in C' is $\binom{6}{3} = 20$. Considering that the maximum group size σ is 5, four new threads are forked. Each thread receives as input the set C' and the index of the first subset to be enumerated: $thread1 = 1$, $thread2 = 6$, $thread3 = 11$, and $thread4 = 16$. Hence, the subsets of C' are enumerated by the spawn threads as follows:

- *thread*1: {1,2,3}, {1,2,4}, {1,2,5}, {1,2,6}, {1,3,4}.
- *thread*2: {1,3,5}, {1,3,6,}, {1,4,5}, {1,4,6}, {1,5,6}.
- *thread*3: {2,3,4}, {2,3,5}, {2,3,6}, {2,4,5}, {2,4,6}.
- *thread*4: {2,5,6}, {3,4,5}, {3,4,6}, {3,5,6}, {4,5,6}.

Neighborhood intersection. Intersections are computed using a merge-style approach (nodes are kept ordered by their labels both in the neighborhoods and in the candidate sets). If subsets are enumerated sequentially, node selection at line 8 and set intersection at line 10 can be also performed incrementally. A first node w_1 is selected in C, computing the intersection $C^1 = C \cap \Gamma^+(w_1)$. The next node w_2 is selected in C^1, computing $C^2 = C^1 \cap \Gamma^+(w_2)$. This procedure, repeated until $C^t = C^{t-1} \cap \Gamma^+(w_t)$, guarantees that $w_1 \prec w_2 \prec \ldots \prec w_t$: each node belongs to the high-neighborhood of the previously selected nodes. We call this the *progressive intersection heuristic*. Since each intersection decreases the number of candidates, node selection and set intersection are likely to be faster. In addition, besides avoiding the test for condition 2, the progressive intersection heuristic allows the algorithm not to check condition 1 as well: any selected node $w_x \in C^{x-1}$ is thus adjacent to all previously selected nodes.

With parallel subset enumeration, this improvement is not applicable: since subsets are divided into groups and represented by indexes, all subset nodes must be selected from C before computing any intersection.

5.1.3 Analysis

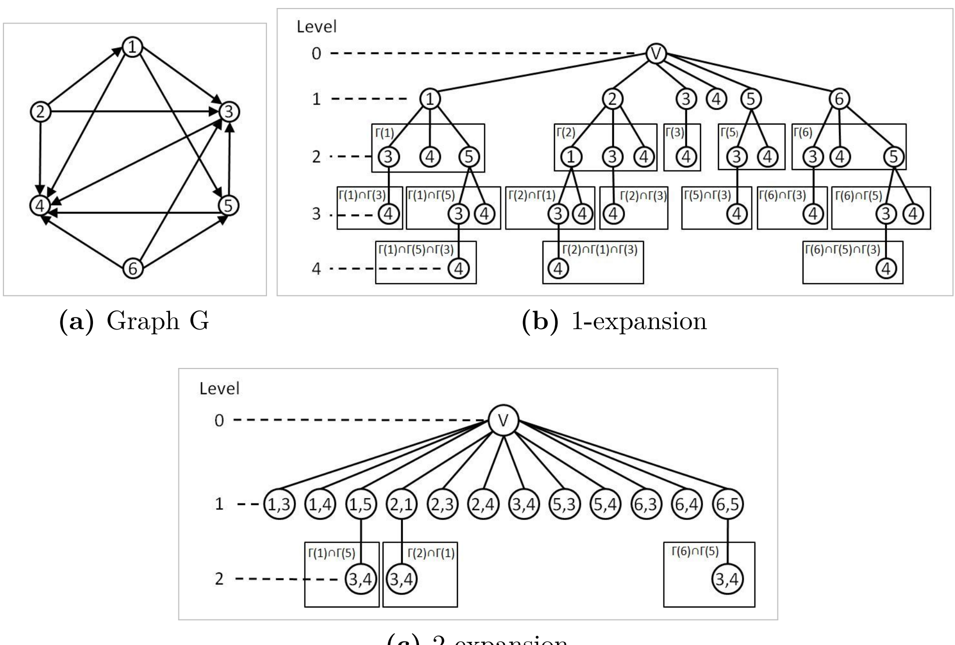
The computation can be conveniently represented on a *fork tree*, where each branch works in parallel. At the root node (level 0), all nodes in V are candidates. Tree nodes at level 1 correspond to all the $\binom{|V|}{t}$ subsets of V of size t. Each subset $\{w_1, \ldots, w_t\}$ associated with a fork tree node W of level 1 has a candidate set C obtained as the intersection between the neighborhoods $\Gamma^+(w_1) \cap \ldots \cap \Gamma^+(w_t)$. Similarly, the children of W at level 2 correspond to all the $\binom{|C|}{t}$ subsets of C of size t. A similar reasoning can be repeated downto the last level $\lceil \frac{k}{t} \rceil - 1$. Examples of fork trees for $t = 1$ and $t = 2$ are shown in Figure 5.3. A square box surrounding one or more tree nodes is labeled with the candidate set C of their (common) parent.

Given any tree node W at level ℓ, consider the $t \cdot \ell$ graph nodes whose neighborhoods have been intersected along the path from the root to W. These nodes form a $(t \cdot \ell)$-clique. For instance, nodes $\{1, 5\}$ and $\{3, 4\}$ in Figure 5.3c yield the 4-clique $\{1, 3, 4, 5\}$. Moreover, since $\Gamma^+(1) \cap \Gamma^+(3) \cap \Gamma^+(4) \cap \Gamma^+(5) = \emptyset$, no clique of size at least 5 containing these four nodes can exist in $\overline{G}$.

Consider a leaf at the maximum depth of the fork tree. If $(k \mod t) = 0$, then the leaf corresponds to a $(k-t)$-clique, which is extended to a k-clique in the base step (lines 3 - 6) by adding t nodes satisfying conditions of Claim 1. Conversely, if $k \% t \neq 0$, the number of nodes added to the clique in the base step is smaller than t (see the test at line 3).

The proposed algorithm is work-optimal [59], i.e., it can list all k-cliques of a graph with m edges in time $O(m^{k/2})$. Moreover, its span and memory usage are also analyzed. Among the many implementation variants, the analysis is focused on the case $t = 1$, assuming that forks and set intersections are done sequentially.

Work analysis. By Lemma 1, at any time during the execution the set of candidates contains at most $2\sqrt{m}$ nodes, since $2\sqrt{m}$ is the maximum size of the neighborhood of any node. Since candidates are obtained as intersections of node neighborhoods, throughout the computation, C can only get smaller than this bound.



(a) Graph G **(b)** 1-expansion

(c) 2-expansion

Figure 5.3. A graph G and its fork trees for $t = 1$ and $t = 2$, assuming $k = 5$.

This implies that the degree of any fork tree node is at most $2\sqrt{m}$ because $t = 1$. The only exception is the tree root: $C = V$ at the beginning of the execution and thus the root has n children.

It is not difficult to see that the number of nodes at a level $\ell \geq 2$ of the fork tree is $O(m \cdot \sqrt{m}^{\ell-2}) = O(m^{\ell/2})$. Indeed, the number of nodes at level 2 is $\sum_{v \in V} |\Gamma^+(v)| = m$ and the degree of any internal node is upper bounded by $2\sqrt{m}$, as observed above.

The number of operations done by the thread corresponding to a leaf is dominated by line 4 and is $O(\sqrt{m})$, because $|C| \leq 2\sqrt{m}$ (the check at line 5 takes $O(|W|^2) = O(1)$ time since t and k are constant). The time spent on each internal node (different from the root) is instead $O(m)$, because line 8 enumerates $O(\sqrt{m})$ sets and the intersection at line 9 requires $O(\sqrt{m})$ time.

The total work is achieved summing the work done by the nodes at each level of the fork tree. Using the bounds given above, since the deepest leaves are at level $k-1$, the time spent on these leaves turns out to be $O(\sqrt{m} \cdot m^{(k-1)/2}) = O(m^{k/2})$. Similarly, the time spent at any level $\ell \leq k-2$ is $O(m \cdot m^{\ell/2})$, which can be upper bounded by $O(m \cdot m^{(k-2)/2}) = O(m^{k/2})$. Since we have at most $k-1$ levels and k is a constant, the total work is $O(m^{k/2})$.

Span analysis. Consider any path in the fork tree. The time spent on the root (lines 8-9) is given by $\sum_{w \in V} |\Gamma^+(w)| = O(m)$. Any other node has degree $\leq |C| = O(\sqrt{m})$ and therefore lines 8-10 take time $O(\sqrt{m} \cdot \sqrt{m}) = O(m)$. Any leaf requires time $O(\sqrt{m})$. Summing up along the path, whose length is $k-1$, yields span $O(k \cdot m)$.

Memory space analysis. As stated above, the number of nodes at level $\ell \geq 2$ is $O(m^{\ell/2})$. Hence, the total number of nodes from level 2 downto the last level is $\sum_{x=2}^{k-1} m^{x/2}$, which is $\Theta(m^{(k-1)/2})$. Since the space usage per node is $O(\sqrt{m})$ (due to the upper bound on the number of candidates), the total memory usage is $O(m^{k/2})$. The contribution due to the first three levels is upper bounded by the contribution of the bottom levels. The memory consumption thus matches the worst-case number of k-cliques, which is $\Theta(m^{k/2})$ [59].

5.2 Experimental setup

In this section is described the experimental platform, deferring implementations details on algorithm CountClique to Section 5.3. Benchmarks used to evaluate the algorithms are described in Section 5.2.1. The state-of-the-art sequential clique counting algorithm in the experimental analysis is presented in Section 5.2.2. The description of the platform where the experiments are performed is given in Section 5.2.3

5.2.1 Benchmarks

The algorithms are evaluated on publicly available real-world networks from the SNAP graph library [170], preprocessing all graphs so that they are undirected (if an edge (u, v) appears in the graph, we also add edge (v, u) if it does not already exist). We selected a set of 15 instances of different sizes and growth rates as function of k (see Table 1.1). The results are obtained on eight social networks (`comYoutube`, `locGowalla`, `socPokec`, `wikiTalk`, `hTwitter`, `comOrkut`, `comLiveJ`, `socLiveJ1`), four Web graphs (`webGoogle`, `webNotreDame`, `webStan`, `webBerkStan`), a citation network among US Patents (`citPat`), an Internet topology graph (`asSkitter`), and a product co-purchasing network (`amazon`). The exact numbers of k-cliques on these datasets, for $k \in [1, 7]$, are shown in Table 1.1. In Figure 5.4 is analyzed the clique growth rate by plotting the ratio q_k/q_{k-1} as a function of k. Benchmarks are divided into three groups, based on their clique growth rate: smaller than 3, in $[3, 10]$, and larger than 10, respectively. Notice that some graphs of the last group – most notably `comLiveJ` and `socLiveJ1` – exhibit very steep rates.

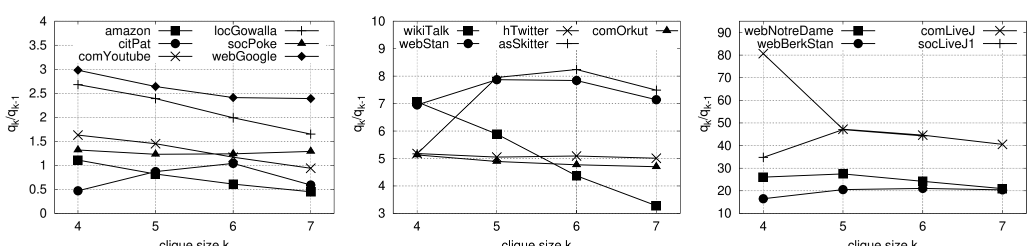


Figure 5.4. Clique growth rate: ratio q_k/q_{k-1} for $k \in [4, 7]$ on the different benchmarks.

5.2.2 Sequential clique counting: algorithm L+N

As a baseline to understand and quantify the acceleration factor that our parallel algorithms can achieve, a highly optimized sequential implementation is used, called `L+N`, taken from [98]. `L+N` was implemented in Java and extends the state-of-the-art sequential implementation for triangle enumeration presented in [214]. It is an intersection-based algorithm that iterates over all edges, intersecting the neighborhoods of the endpoints of each edge to find nodes that complete a triangle. To this aim, during the computation, nodes are associated with a counter: when processing a node u, the counters of all nodes in the neighborhood of u are set to 1. Each node v in the neighborhood of u also increases the counters of its own neighbors. Hence, triangles including edge (u, v) are formed by nodes whose counter becomes equal to 2. `L+N` extends this approach to k-cliques, where the counter of the node that completes a clique must be equal to $k - 1$.

The counter-based implementation seems difficult to parallelize due to concurrency issues in the management of counters: since the counter of the node completing a clique must be $k - 1$, if different threads are allowed to write counters during the execution, they must be properly synchronized. The proposed parallel algorithms overcome this issue by using lists of candidates, instead of counters, and maintaining only nodes that are adjacent to all the candidates. This technique, which proved useful also in a sequential setting [304], might result in larger space usage, but can be more easily parallelized avoiding concurrency issues.

5.2.3 Platform

The experiments have been performed on an Oracle Hotspot VM with 4 AMD Opteron 8-Core 6272 2.1Ghz processors (32 overall cores), 64 GB of RAM (2 GB per core), and Java version 1.7.

5.3 Engineering algorithm CountClique

The implementation of algorithm CountClique was carefully engineered considering a variety of subset enumeration procedures, set intersection strategies, and data structures. Hence, the most effective implementation choices are reported in the thesis, focusing on five variants. Three of them (called `1by1`, `2by2` and `3by3`) use sequential subset enumeration for $t = 1$, 2, and 3, respectively. Two variants (called `2by2P` and `3by3P`) use parallel enumeration for $t = 2$ and 3 (when $t = 1$, the number $|C|$ of subsets to be enumerated is typically rather small, making parallel enumeration unworthy).

All the implementations have been realized within the Java Fork/Join framework [168]. Besides the progressive intersection heuristic, code optimizations are applied, whose are described in Section 5.3.1. A preliminary experimental analysis aimed at identifying the most promising implementation choices is given in Section 5.3.2, which discusses whether it is convenient to use parallel subroutines for both candidate intersection and subset enumeration.

5.3.1 Code optimizations

Clique expansion check. Whenever a node w such that $|\Gamma^+(w)| < k - h$ is selected at line 8, w is immediately discarded before computing any intersection. Similarly, in the progressive intersection heuristic, if $|C'| < k - h$ after each intersection, C' is dropped without forking any new thread. Incremental clique expansion tests are instead impossible with parallel enumeration, where an entire subset must be generated before checking that a clique can grow up to size k from the current set of candidates.

Bounded forks. With sequential subset enumeration and $t \geq 2$, at the first tree level we need to bound the number of forks, which could be extremely large because $C = V$. In order to avoid exceeding the Fork/join task queue capacity during the execution, when an upper bound is reached, the algorithm stops forking until previously spawned tasks have been completed. According to preliminary tests, a bound of one million appears to be rather effective, yielding a total number of tasks close to queue capacity on the platform.

For parallel enumeration variants, 1-expansion is applied at the first level of the fork tree and t-expansion thereafter. When the graph to be analyzed is sparse, as in the case of social networks (see also Table 1.1), many of the $\binom{|V|}{t}$ fork tree nodes would be indeed associated to subsets of t graph nodes that are not mutually adjacent (and cannot be thus expanded to a k-clique). The use of 1-expansion at the first level reduces the number of root children to $|V|$, spending less time forking worthless tasks. This optimization does not need to be applied at lower levels, where we observed that the number of children becomes manageable (candidates in C decrease after each intersection).

5.3.2 Candidate intersection and subset enumeration: sequential or parallel?

Parallel merging. A first natural question is whether set intersection, which uses a merge-style approach, should be performed sequentially or in parallel. Since a parallel approach might be worth if sets to be intersected are large enough, in Table 5.1 we analyze, for each benchmark, the degree of nodes with non-empty high-neighborhood. Even graphs with a few million nodes present an average high-neighborhood size of just a few dozens neighbors. In agreement with power laws in degree distribution observed in real networks, there is a very large gap between the maximum and the average sizes: the percentage of nodes with degree larger than max/2, for instance, is close to 0 on most benchmarks. On such small sets, parallel intersection does not pay off. Hence, sequential merging is considered to implement all variants of the CountClique algorithm.

Parallel subset enumeration. Differently from set intersection, which involves only small sets, a large number of subsets is likely to be generated at line 8 of algorithm CountClique. To understand if parallel enumeration can prove useful, in Table 5.2 we compare the running times, for $t = 2$ and 3, when subsets are generated either sequentially or in parallel.

Table 5.1. High-neighborhood statistics.

Graph	Nodes with non-empty high-neighborhood	Max high-neighborhood size	Avg high-neighborhood size	% larger than Max/2	% larger than Max/4
amazon	398 879	24	6.13	0.115	59.720
citPat	3 718 682	77	4.44	0.021	0.428
comYoutube	1 120 769	164	2.67	0.016	0.318
locGowalla	195 500	84	4.86	0.487	3.534
socPokec	1 632 214	96	13.66	1.297	20.125
webGoogle	863 740	93	5.00	0.008	0.122
wikiTalk	2 391 620	340	1.95	0.013	0.084
webStan	276 003	103	7.22	0.596	3.899
hTwitter	456 408	377	27.41	0.179	4.466
asSkitter	1 685 838	231	6.58	0.023	0.372
comOrkut	3 072 280	535	38.14	0.314	2.165
webNotreDame	322 769	155	3.38	0.215	0.598
webBerkStan	676 928	201	9.82	0.274	2.547
comLiveJ	3 990 338	524	8.69	0.012	0.114
socLiveJ1	4 834 939	687	8.86	0.003	0.097

For $t = 2$, the parallel variant `2by2P` performs slightly better than its sequential counterpart `2by2` on small graphs and for small values of k. However, `2by2` becomes considerably faster as the graph size and the clique size increase. The same behavior can be observed for $t = 3$, where the difference of performance is even more noticeable (see, e.g., `socLiveJ1` for $k = 5$).

The parallel subset enumeration algorithm has shown very good speedups with respect to sequential approaches when tested in isolation. Interestingly, when applied as a subroutine of CountClique, parallel enumeration does not show the same benefits. An in-depth analysis of this phenomenon revealed that this is largely due to the incremental clique expansion checks performed by the sequential variants: when k is larger, sequential incremental checks make it possible to enumerate far less subsets than with parallel implementations, improving substantially the overall running time.

Overall, there is no clear winner: parallel or sequential subset enumeration might be preferable depending on the value of k and on the specific benchmark. Hence, results obtained with both the implementations are reported in the experimental evaluation.

5.4 Experimental results

In this section are summarized the main experimental findings. We remark that, to the best of our knowledge, multicore algorithms addressing the same k-clique enumeration problem were not proposed so far in the literature. Therefore, we provide an experimental analysis comparing the proposed multicore solutions with the state-of-the-art sequential algorithm. Notice that the running times of the sequential algorithm are used as a baseline with the aim to understand and quantify the

Table 5.2. Running times (mins:secs) of four sequential/parallel variants for $t \in [2, 3]$ across different datasets and different values of k, using 32 cores.

Graph	k	2by2	2by2P	3by3	3by3P
amazon	4	0:05	0:04	0:04	0:04
	5	0:05	0:04	0:05	0:04
	6	0:05	0:04	0:05	0:04
	7	0:04	0:04	0:04	0:04
comYoutube	4	0:06	0:05	0:06	0:05
	5	0:06	0:05	0:09	0:06
	6	0:06	0:05	0:11	0:06
	7	0:06	0:05	0:08	0:06
socPokec	4	1:02	0:42	0:54	0:50
	5	1:07	0:42	1:18	0:54
	6	1:08	0:41	1:15	0:56
	7	1:09	0:43	1:11	0:54
	14	0:57	0:53	0:57	1:12
	20	0:48	0:46	0:47	1:00
wikiTalk	4	0:14	0:09	0:17	0:21
	5	0:11	0:12	0:38	0:43
	6	0:21	0:46	0:51	0:53
	7	1:00	1:21	1:13	2:57
hTwitter	4	0:40	0:26	1:01	1:37
	5	1:02	0:43	2:35	2:26
	6	1:28	2:25	3:53	3:29
	7	4:48	8:02	6:05	17:24
comOrkut	4	12:16	5:49	15:23	-
	5	14:01	7:41	36:23	-
	6	13:32	23:42	38:02	-
	7	42:16	51:25	60:29	-
webBerkStan	4	0:27	0:16	0:31	0:21
	5	1:08	1:29	4:20	4:25
	6	17:18	48:22	18:58	28:48
socLiveJ1	4	3:06	2:25	6:07	8:14
	5	22:16	54:40	37:41	80:37

Graph	k	2by2	2by2P	3by3	3by3P
citPat	4	0:50	0:43	0:46	0:47
	5	0:47	0:44	0:47	0:44
	6	0:45	0:43	0:44	0:42
	7	0:42	0:40	0:44	0:41
locGowalla	4	0:03	0:02	0:02	0:01
	5	0:02	0:01	0:05	0:02
	6	0:02	0:02	0:05	0:02
	7	0:02	0:02	0:05	0:02
	14	0:04	0:05	0:05	0:07
	20	0:02	0:03	0:02	0:04
webGoogle	4	0:11	0:09	0:09	0:08
	5	0:14	0:08	0:24	0:10
	6	0:12	0:10	0:25	0:10
	7	0:12	0:12	0:25	0:13
webStan	4	0:07	0:04	0:05	0:04
	5	0:08	0:06	0:23	0:09
	6	0:14	0:29	0:35	0:25
	7	1:13	2:27	1:21	4:52
asSkitter	4	0:24	0:18	0:30	0:24
	5	0:31	0:23	1:06	0:43
	6	0:53	1:29	1:44	1:19
	7	4:10	5:55	5:06	13:42
	8	26:49	57:35	30:18	115:20
webNotreDame	4	0:04	0:04	0:06	0:04
	5	0:11	0:23	0:47	1:00
	6	3:43	12:19	3:48	9:00
	7	75:56	176:23	74:51	427:30
comLiveJ	4	2:21	1:41	4:20	3:50
	5	12:50	28:13	22:02	39:14

acceleration factor that our parallel algorithms can achieve by using the parallelism.

Throughout the experiments three main independent variables are changed: input dataset, number of cores, and clique size k. After a bird's eye view on the outcomes of the proposed study (Section 5.4.1), Section 5.4.2 addresses the scalability of CountClique on different numbers of cores. Section 5.4.3 and Section 5.4.4 focus on the computation of cliques with at most or more than 7 nodes, respectively: 7 is the maximum value of k considered in previous experimental studies, most of which focus on $k = 4$ and 5. Hence, as a by-product of the experimental analysis, we also compute the exact number of k-cliques with at most 20 nodes in many real-world networks from the SNAP repository. A comparison with state-of-the-art distributed solutions is given in Section 5.4.5.

5.4.1 A bird's eye view

In order to assess the efficiency of the proposed parallel algorithms, as a first experiment cliques of moderate size are counted across different datasets using a number of cores (8) that can be easily matched on modern commodity hardware.

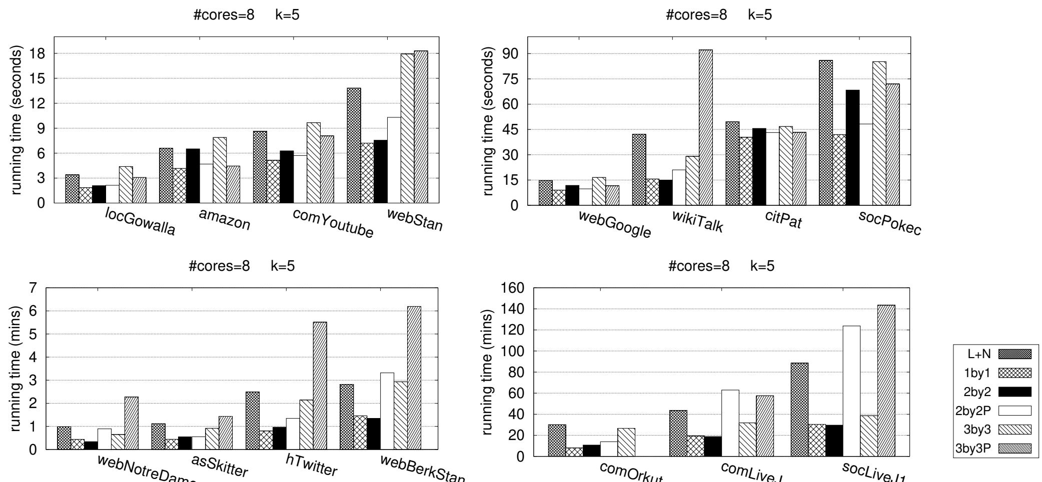


Figure 5.5. A comparison of the algorithms for $k = 5$.

The outcome of such experiments is shown in Figure 5.5 and Figure 5.6, reporting on the running times of L+N and of the five parallel variants described in Section 5.3 when counting cliques of 5 and 7 nodes, respectively.

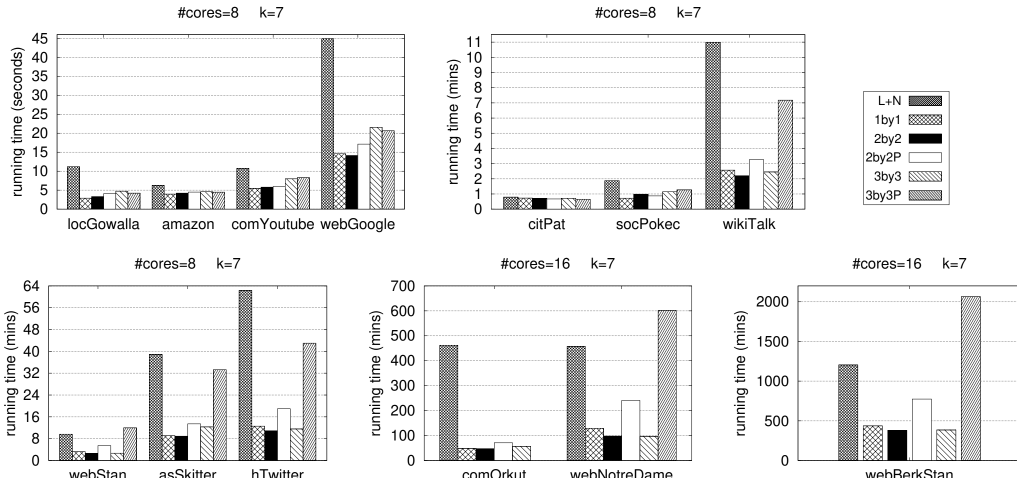


Figure 5.6. A comparison of the algorithms for $k = 7$.

At a first glance, it is possible to observe that the performance of the parallel algorithms changes significantly depending on the value of t and on the specific dataset. The variants for $t = 3$ (3by3 and 3by3P, last two columns in the histograms) appear to be the least competitive: they can be even slower than L+N, especially on small graphs for which the number of k-cliques is not large (see also Table 1.1). The variants of $t = 2$ perform better, but still struggle to beat L+N in a few graphs for $k = 5$: e.g., 2by2 on amazon and citPat and 2by2P on comLiveJ and socLiveJ1 in Figure 5.5. On the other hand, 1by1 clearly shows the benefits of parallelism and is always significantly faster than the optimized sequential algorithm. Its speedup, though largely affected by k and by the dataset characteristics, in this experiment is

as large as 9.5 on `comOrkut` for $k = 7$ on 16 cores.

Overall, the experiment suggests that computing intersections of a relatively large number of neighborhoods in order to reduce the number of tasks does not pay off substantially in the Fork/join framework. Additional tests for values of $t > 3$ have confirmed this performance trend. This is not completely unexpected: the larger is t, the smaller is the number of nodes in the fork tree but, at the same time, the higher is the running time at each node (see Section 5.1). In practice, since each fork tree node is mapped to a distinct `ForkJoinTask` in the Java Fork/join framework, small values of t yield many short tasks, while larger values result into fewer but longer tasks. It is well known [116] that the Java Fork/join framework is extremely efficient at handling even very large numbers of tasks. Conversely, long tasks are more difficult to be scheduled optimally and could thus harm the scalability of the framework. The results confirm this behaviour, showing that it is mostly convenient using $t \leq 2$. In particular, algorithm `1by1` seems well worth of being used even on a relatively small number of cores.

5.4.2 Scalability analysis

In this section the scalability of the algorithms on different numbers of cores is studied. As a typical outcome, Figure 5.7 shows the running times when counting cliques of size $k \in [5, 7]$ on the `asSkitter` dataset, using a number of cores that ranges from 2 to 32.

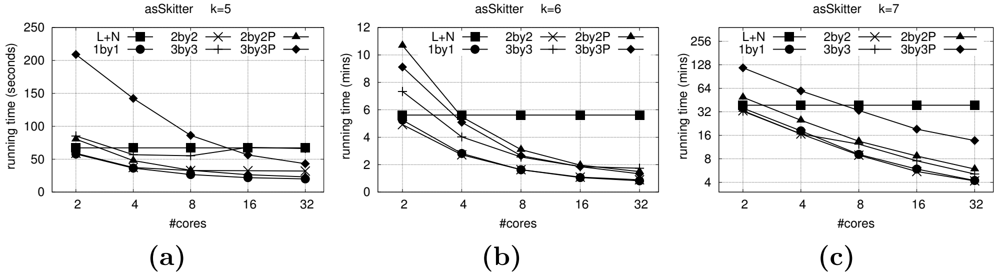


Figure 5.7. Running times of `L+N` and of the five parallel variants when counting cliques of size $k \in [5, 7]$, using a number of cores ranging from 2 to 32, on the input dataset `asSkitter`.

Considering that the state-of-the-art algorithm (`L+N`) is *sequential*, its running time does not change. On a few cores, the least efficient parallel solutions can have a running time comparable to, or even slower than, `L+N` (e.g., `2by2P` and `3by3P` on 2 and 4 cores). As previously stated, `L+N` and CountClique perform the same asymptotic work, which is $O(m^{k/2})$ in the worst case. However, due to the overhead of thread synchronization and to the fact that the implementation of `L+N` is very well engineered, it is not surprising that the sequential approach has smaller constant factors hidden by the asymptotic notation. Conversely, when the number of cores increases, speedups are more tangible. Actually, using at least 4 cores the parallel approach starts to pay off.

Not surprisingly, the scalability of the parallel algorithms is better for the largest

values of k, i.e., when the clique counting problem becomes more computationally demanding. With $k = 7$, for instance, the speedup of `1by1` is about 9.2 on 32 cores (notice the logarithmic scale on the y axis in Figure 5.7c). Comparable speedup trends can be observed for $k = 6$. On the other hand, when $k = 5$, all the running times are very small – less than 1 minute for `1by1` – and the advantages over `L+N` are limited (the speedup of `1by1` is only about 3.3 on 32 cores).

Figure 5.8 illustrates the scalability of the algorithms on the other datasets. It is reported only one experiment per benchmark, focusing on the largest – and most difficult – value of k considered for each dataset.

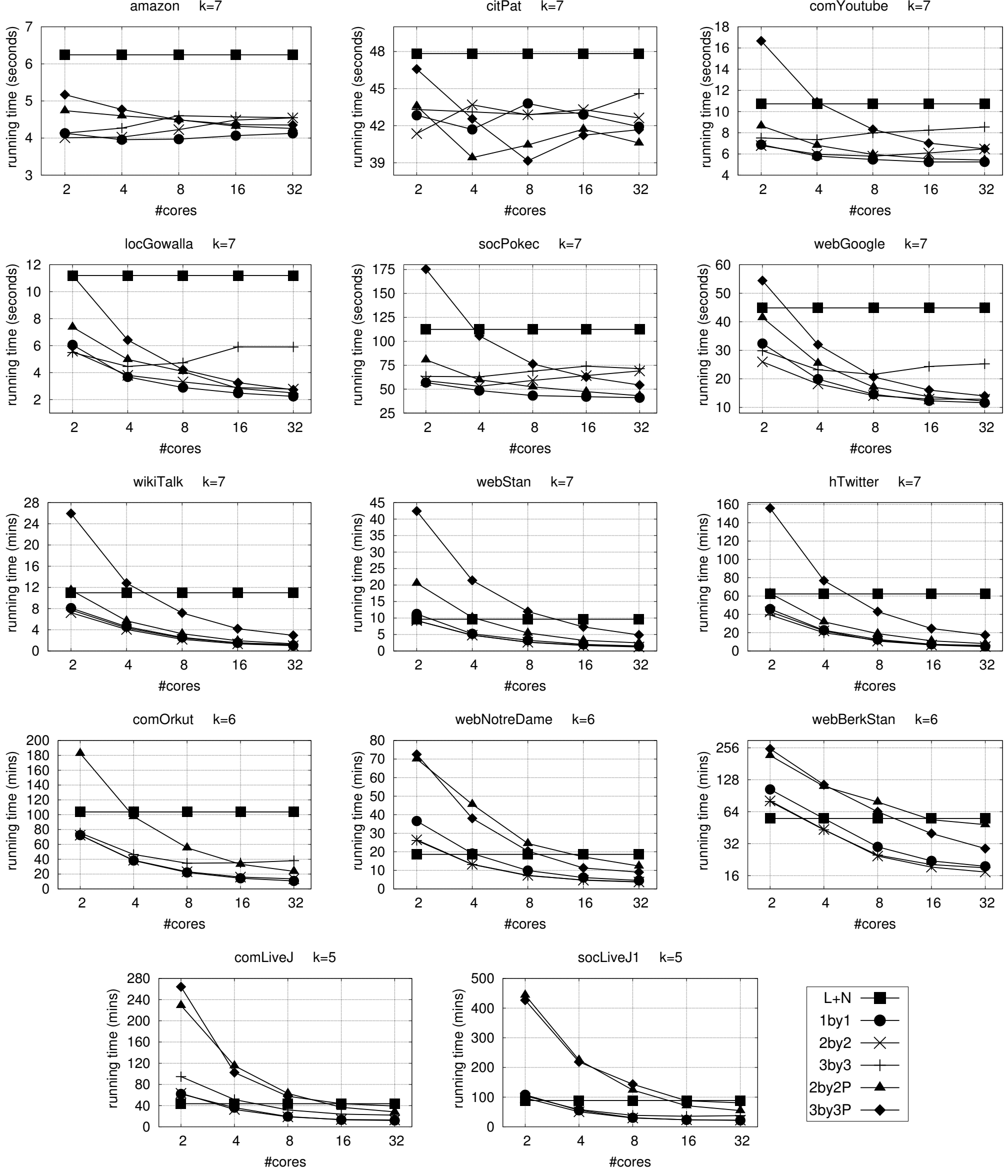


Figure 5.8. Scalability of the parallel algorithms.

Overall, the results are consistent with those observed on `asSkitter`: `1by1` is the fastest variant and shows similar scalability trends. Scalability, however, appears

to depend on the benchmark characteristics and is substantially better for the most demanding datasets. For instance, `amazon` and `citPat`, which have a relatively small number of cliques, appear not to take much advantage of parallelism: the curves in Figure 5.8 are rather flat. Conversely, on `hTwitter` and `comOrkut` `1by1` gets a speedup about 13 and 14, respectively, on 32 cores, starting from a running time close to `L+N` on 2 cores.

5.4.3 Counting small cliques

In this section we focus on small cliques with at most 7 nodes. Figure 5.9 reports on the results obtained for thirteen benchmarks on 16 cores (q_6 and q_7 could not be computed for `comLiveJ` and `socLiveJ1`).

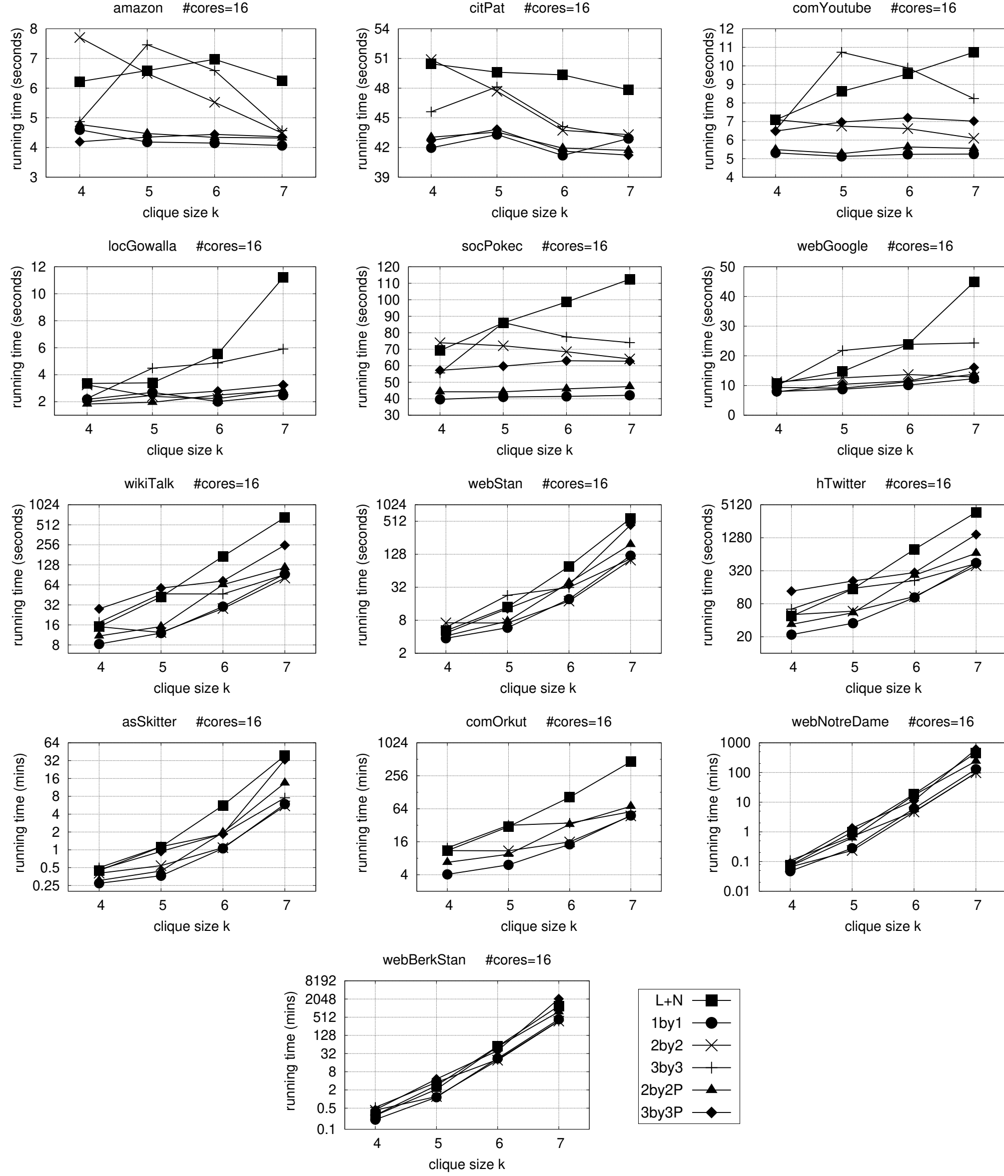


Figure 5.9. Running time of the algorithm as a function of k on 16 cores, for $k \leq 7$.

As previously stated, L+N has a theoretical running time $O(m^{k/2})$, where m is the number of edges in the graph and k is the clique size, and this is optimal in the worst case. Hence, the wall clock times measured for L+N should increase exponentially with k on worst-case instances. Ideally, if L+N could be perfectly parallelized, for moderate numbers of cores it would still be expect an exponential growth of the wall clock time when increasing the clique size, but mitigated by a linear scaling factor proportional to the number of cores.

On webNotreDame and webBerkStan, the running time of all the algorithms has a sharp exponential trend with respect to k (notice the logarithmic scale on the y-axis). As shown in Table 1.1, the number of cliques in webNotreDame and webBerkStan grows considerably with k and the performance is thus likely to be close to the worst case. Except for considerably small benchmarks (e.g., amazon, citPat and comYoutube), the exponential trend is still visible in the other graphs, but mitigated for the parallel algorithms when $k = 4$ and 5, since the algorithms can take advantage of a relatively smaller number of k-cliques.

5.4.4 Counting larger cliques

The last set of experiments considers cliques of larger sizes, up to 20 nodes. This is the first analysis in the literature of the exact number of k-cliques for $k > 7$. In Appendix A is shown the number of k-cliques for $k \in [8, 20]$ in 10 real-world networks from the SNAP graph library [170]. The experiments are performed running both 1by1, which is fastest variant of CountClique, and its sequential competitor L+N across small and medium-size benchmarks for $k \in [4, 20]$, aborting long executions after a time limit of 48 hours (2880 minutes).

Parallelism clearly shows its benefits as the clique size k increases. Figure 5.10 shows the running times on webGoogle, webStan, hTwitter, and asSkitter using 32 cores. On each benchmark, the number of k-cliques for the largest plotted value of k is in the order of *trillions*. For $k \leq 7$, the proposed parallel solution does not present any noticeable difference with respect to L+N. However, it becomes extremely faster as k increases: e.g., webGoogle for $k = 15$, where L+N and 1by1 require almost 26 hours and 132 minutes, respectively. Overall, 1by1 can often compute the number of k-cliques for values of k that L+N cannot afford to solve. On webGoogle, for instance, L+N exceeded the 48 hours time limit for all $k > 15$.

Optimizations applied to algorithm CountClique revealed to be crucial to speed up its execution. To support this claim, in Figure 5.11 is compared L+N with 1by1, running the parallel algorithm both on 1 and on 32 cores, on benchmark locGowalla for $k \in [4, 20]$. The raw running times is presented in Figure 5.11a. The speedup of 1by1 with respect to L+N is shown in Figure 5.11b. The number of node neighborhoods explored by each algorithm is presented in Figure 5.11c. And k-clique distribution on locGowalla is given in Figure 5.11d.

It is remarkable that 1by1, even using 1 core, is faster than L+N when k is large enough ($k > 7$). L+N appears to be very sensitive to the number of k-cliques, as can be noticed by comparing the trend of its running time in Figure 5.11a with the clique distribution in Figure 5.11d. 1by1 is instead less affected, especially on 32 cores, and the difference of their running times becomes larger and larger as q_k increases. To explain this behaviour, notice that 1by1 explores far less neighborhoods than

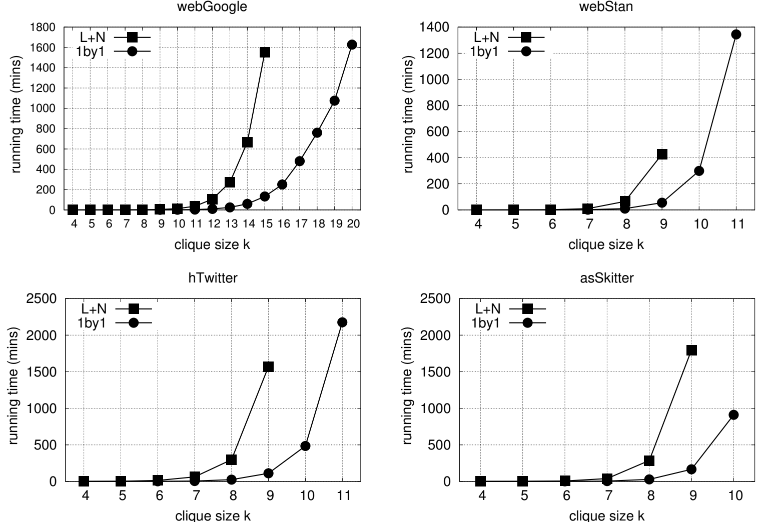


Figure 5.10. Increasing k on a selection of medium-size benchmarks.

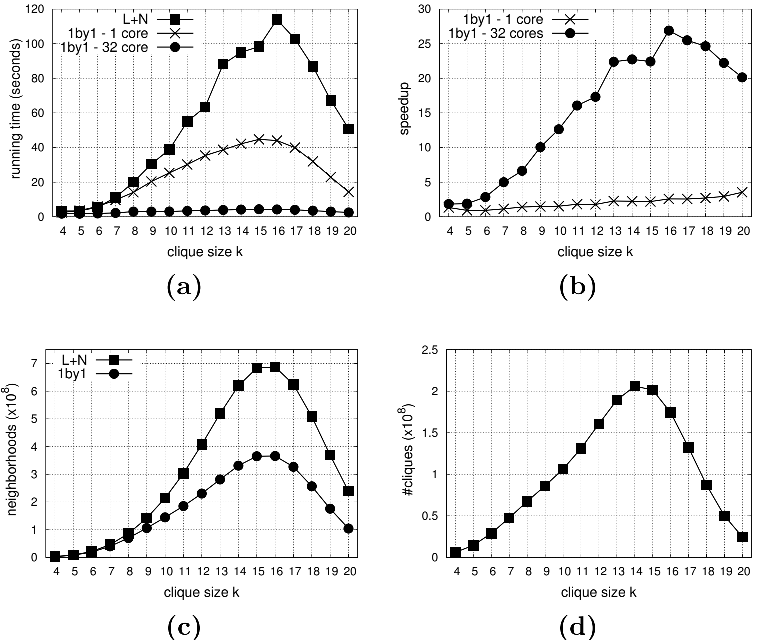
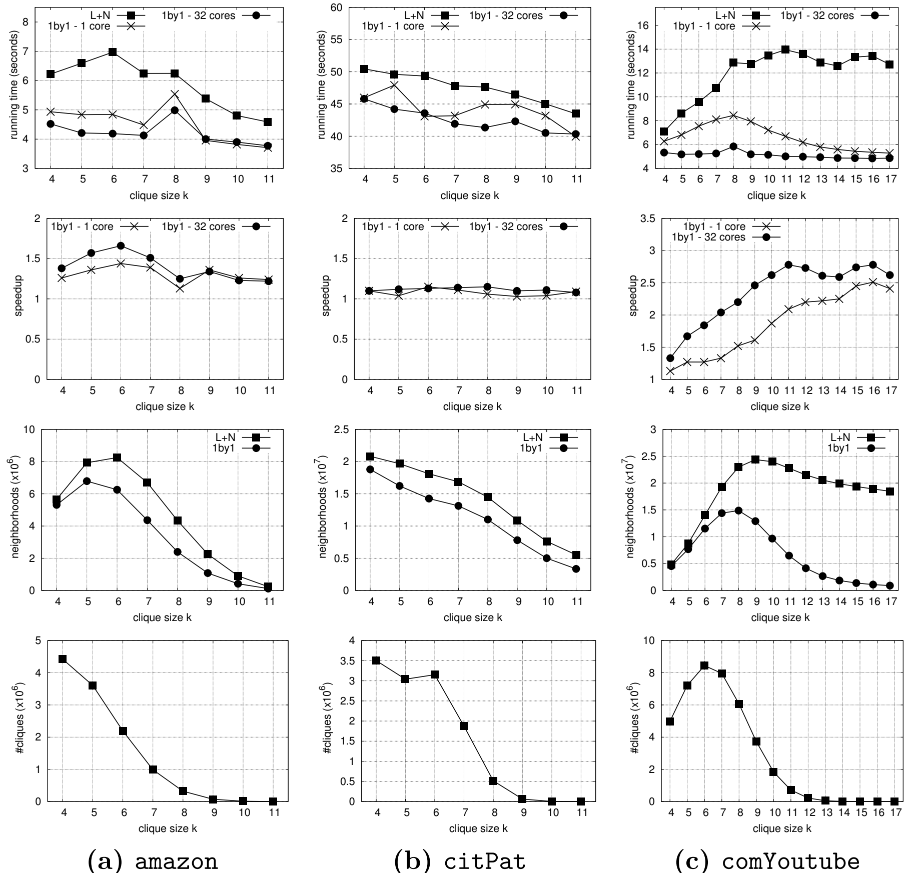


Figure 5.11. In-depth comparison of `L+N` and `1by1` on `locGowalla` for large values of k.

`L+N`, as shown in Figure 5.11c. Thanks to the progressive intersection and expansion check heuristics, it can skip set of candidates that can not complete a k-clique since early stages of the execution. `L+N`, with its counter-based implementation, can not instead perform similar optimizations. These unique properties of `1by1`, combined with parallelism, make it possible to achieve very large speedups on `locGowalla`: e.g., 26.8 on 32 cores for $k = 16$.

Figure 5.12 and Figure 5.13 confirm the results reported for `locGowalla` on



(a) `amazon` **(b)** `citPat` **(c)** `comYoutube`

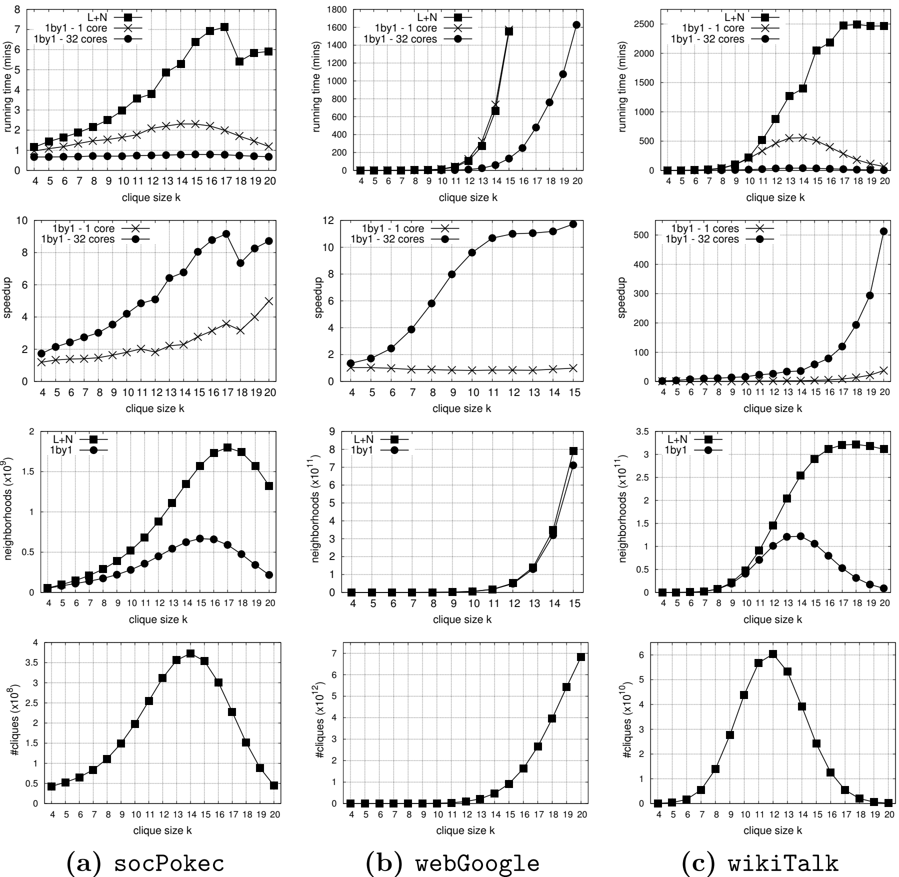
Figure 5.12. Counting k-cliques on (a) `amazon`, (b) `citPat`, and (c) `comYoutube` for $k \leq 17$ ($q_k = 0$ for larger values of k as shown in 8.1).

other benchmarks. In Figure 5.12 are considered three rather small graphs: `amazon`, `citPat`, and `comYoutube`. Since on these benchmarks the number of k-cliques becomes 0 even for moderate values of k (see Appendix A), the results reported here are obtained when $q_k \geq 1$. Even if `1by1` always outperforms `L+N`, Figure 5.12 confirms that the benefits of parallelism are limited on small graphs with a relatively small number of cliques (notice that q_k is "only" proportional to 10^6 on these benchmarks).

In Figure 5.13 are considered three medium-size graphs: `socPokec`, `webGoogle`, and `wikiTalk`. The charts exhibit the same behavior observed for `locGowalla` in Figure 5.11. Graph `wikiTalk` (Figure 5.13c) is a prime example of the utility of the progressive intersection and the clique expansion checks heuristics: even the sequentialized execution of `1by1` on a single core is 37 times faster than `L+N`. A truly parallel execution on 32 cores blows up the speedup by an additional 14 factor, resulting in a running time which is 523 times faster than `L+N`.

5.4.5 Multithreading or MapReduce?

A natural question is how the proposed algorithms compare to other parallel solutions. The current state-of-the-art approaches for parallel k-clique counting hinge upon



(a) `socPokec` **(b)** `webGoogle` **(c)** `wikiTalk`

Figure 5.13. Counting k-cliques on (a) `socPokec`, (b) `webGoogle`, and (c) `socPokec` for $k \leq 20$.

MapReduce [78], a distributed programming framework targeted at large clusters of commodity machines. The two clique counting algorithms that are most promising from a theoretical point of view, presented in [4] and [98], respectively, have been experimentally analyzed in [98].

From a methodological point of view, a comparison of the multicore approach with cluster-based MapReduce algorithms is not straightforward, since the underlying computational platforms are very different. Hence, the analysis is limited to a few spot observations.

In Table 5.3 is shown, for a common selection of benchmarks, the running times achieved by `1by1` on 32 cores and by the two MapReduce algorithms described in [4, 98] and experimentally analyzed in [98]. It should be noticed that the MapReduce experiments in [98] have been carried out using a 16-node cluster on Amazon Web Services (AWS). Each node was configured for providing 4 cores (based on Intel Xeon E5-2670 v2 Sandy Bridge processors) and 7.5 GB of main memory. Overall, the cluster was therefore more powerful than our platform (see Section 5.2) with respect to both main memory (120 GB vs 64 GB) and number of cores (64 vs 32). The comparison is thus largely unfair to `1by1`. Nonetheless, when looking at raw running times, even graphs with billions of cliques can be processed by `1by1` on

a single machine within reasonable wall clock times and the performance of the multicore algorithm is comparable – or even faster – than the MapReduce solution on a variety of datasets and for several values of k. MapReduce is the solution of choice when the dataset or the number of cliques are extremely large. In these cases, the maximum level of parallelism that can be achieved on a single node machine is bounded by physical limits beyond which multicore approaches cannot scale. However, for several intermediate benchmarks (medium/large datasets or moderate values of k), using MapReduce appears to be overkill and the multithreading approach can be largely preferable, even on a less powerful platform.

Table 5.3. Running times (mins:secs) of `1by1` and of the MapReduce algorithms analyzed in [98].

	$k = 4$			$k = 5$			$k = 6$			$k = 7$		
	1by1	FFF	AFU	1by1	FFF	AFU	1by1	FFF	AFU	1by1	FFF	AFU
citPat	**0:45**	3:11	3:11	**0:44**	3:13	2:18	**0:43**	3:13	2:19	**0:41**	3:09	2:24
comYoutube	**0:05**	2:39	1:41	**0:05**	2:34	1:33	**0:05**	2:36	1:39	**0:05**	2:38	1:49
locGowalla	**0:02**	3:04	1:21	**0:01**	3:02	1:30	**0:02**	3:04	1:24	**0:02**	3:03	1:30
socPokec	**0:41**	4:02	2:29	**0:40**	4:13	2:39	**0:41**	4:15	2:51	**0:42**	4:09	3:02
webGoogle	**0:07**	2:43	1:27	**0:08**	2:43	1:32	**0:09**	2:40	1:40	**0:11**	2:40	1:52
webStan	**0:03**	2:29	1:27	**0:05**	2:37	2:06	**0:15**	2:36	4:00	**1:31**	2:05	14:12
asSkitter	**0:18**	3:17	2:59	**0:23**	3:18	5:34	**0:51**	3:14	25:30	4:27	**4:12**	>40
comOrkut	**4:07**	23:08	20:17	**6:04**	23:10	>50	**12:50**	23:22	>50	42:25	**28:08**	–
webBerkStan	**0:11**	3:01	2:53	**1:01**	3:08	8:24	16:54	**4:56**	>30	365:13	**50:17**	–
comLiveJ	**2:05**	5:24	4:06	12:16	**6:13**	14:02	–	**41:22**	>170	–	–	–
socLiveJ1	**1:44**	6:43	5:10	21:43	**7:51**	23:35	–	**86:34**	>180	–	–	–

Chapter 6

Enumerating diamonds: a sequential approach

In this chapter we introduce our sequential algorithm for counting the number of k-diamonds in a graph with n nodes and m edges in $O(nm^{(k-1)/2})$ time. Though the focus is on counting, the algorithm can be easily adapted to the listing problem. To the best of our knowledge, the k-diamond structure was not properly addressed in the literature before our study. Therefore, the state-of-the-art algorithms for k-diamond enumeration do not exist so far in the literature

Assuming that the input undirected graph has been preprocessed as described in Chapter 2, we first introduce a structural classification of k-diamonds based on node degrees and edge orientations in the directed graph G (Section 6.1). We then describe (Section 6.2) and analyze (Section 6.3) an algorithm to compute the number of 4-diamonds. The extension to $k > 4$ is addressed in Section 6.4.

6.1 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 its node with smallest degree in $\overline{G}$, i.e., 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 $d_D(v)$ its degree in D, i.e., $d_D(v) = |\Gamma(v) \cap D|$. Notice that $d_D(v)$ must be either $k-1$ or $k-2$. In the remainder of this chapter, 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: $d_D(x) = k-1$;
- Case 2: $d_D(x) = k-2$ and $d_D(y) = k-1$;
- Case 3: $d_D(x) = k-2$ and $d_D(y) = k-2$.

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 $d_D(x) = k-2$,

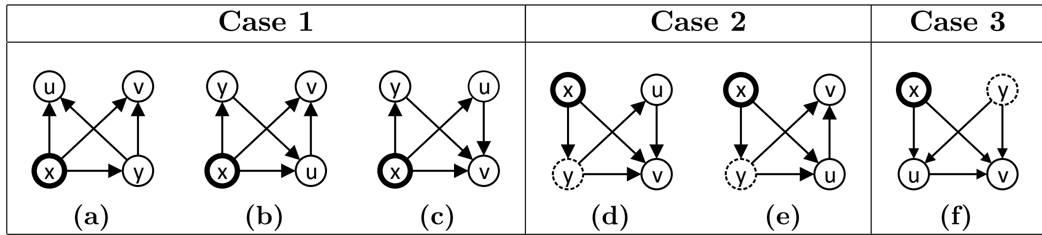


Figure 6.1. Classification of k-diamonds, for $k = 4$, based on the degree in D of the two smallest nodes.

since a k-diamond only contains nodes of degree $\geq k - 2$. In this case, when x is not adjacent to one node in D, 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 $d_D(y) = k - 2$, 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 6.1. 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 other two diamond nodes, assuming without loss of generality that $u \prec v$.

Consider first the structure of diamonds belonging to case 1. Since $d_D(x) = 3$, the second smallest node y must be a neighbor of x and, by definition of x and y, it must be $x \prec y$. Node y, in turn, can have degree either 3 or 2 in D. The orientations of edges must obey the ordering constraints assumed so far (i.e., $x \prec y \prec u \prec v$), yielding the diamond shown in Figure 6.1a when $d_D(y) = 3$. On the other hand, when $d_D(y) = 2$, the third smallest node u can have degree either 3 or 2: this results into the two different diamonds shown in Figure 6.1b and Figure 6.1c, respectively.

A similar reasoning can be applied to diamonds classified in case 2, for which $d_D(x) = 2$ and $d_D(y) = 3$: $d_D(u)$ can be either 2 or 3, yielding the two distinct configurations shown in Figure 6.1d and Figure 6.1e, respectively. In case 3, $d_D(x) = d_D(y) = 2$: x and y are thus the endpoints of the unique missing edge and edge orientations induce only one possible structure, shown in Figure 6.1f. $\square$

Although the number of edge orientations between the nodes of a k-diamond (i.e., the number of configurations shown in Figure 6.1) would increase as a function of k, the classifications into three main cases, being solely based on the degree of the two smallest diamond nodes, remains valid: since $x \prec y$ and each node in a k-diamond D has degree $k - 1$ or $k - 2$, the only three possibilities are Cases 1–3 introduced above.

6.2 Algorithm

In this section we focus on counting 4-diamonds. Our algorithm explores the high-neighborhoods of nodes to spot each 4-diamond, which is then counted – exactly once – by its smallest node x. Consider the classification described in Section 6.1

and a diamond with node set $D = \{x, y, u, v\}$ such that $x \prec y \prec u \prec v$:

- If D belongs to case 1, we can find it by exploring only the high-neighborhood of x, since $y, u, v \in \Gamma^+(x)$.
- Otherwise, we need to explore the high-neighborhoods of at least two nodes:
  - 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.
  - If D belongs to case 3, nodes x and y are not adjacent. Hence, 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 $u, v \in \Gamma^+(y)$.

```
 1: function SeqD4(Directed graph G = (V, E))
 2:     d4 ← 0
 3:     for each a ∈ V do
 4:         for each b, c, d ∈ Γ+(a) such that c ≺ d do
 5:             if (b, c) ∈ Ē and (b, d) ∈ Ē and (c, d) ∉ E then
 6:                 d4 ← d4 + 1                                        ▷ Case 1
 7:         for each b ∈ Γ+(a) do
 8:             for each c, d ∈ Γ+(b) do
 9:                 if (a, d) ∈ E and (c, d) ∈ Ē and (a, c) ∉ E then
10:                     d4 ← d4 + 1                                    ▷ Case 2
11:             for each c ∈ Γ+(b) do
12:                 if (a, c) ∈ E then              ▷ Triangle abc has been found
13:                     for each d ∈ V\{a, b, c} such that a ≺ d do
14:                         if (d, b) ∈ E and (d, c) ∈ E and (a, d) ∉ E then
15:                             d4 ← d4 + 1                            ▷ Case 3
16:     return d4
```

Figure 6.2. A sequential algorithm for counting (and listing) 4-diamonds.

Table 6.1. Roles played by nodes a, b, c, and d used in the pseudocode of Figure 6.2 with respect to the six diamond types described in Figure 6.1.

	1a	1b	1c	2d	2e	3f
a	x	x	x	x	x	x
b	y	u	v	y	y	u
c	u	y	y	u	v	v
d	v	v	u	v	u	y

The pseudocode of the algorithm is shown in Figure 6.2, where a, b, c, and d can represent different nodes with respect to the configurations shown in Figure 6.1, depending on their position within the diamond. In Table 6.1 we present the roles played by these nodes for each diamond type. Note that a always coincides with the smallest node x, being thus responsible for counting the diamond itself. The algorithm receives the directed graph G, preprocessed as described in Chapter 2, and returns the number d_4 of 4-diamonds in G, computed as follows:

- Diamonds from case 1 are counted at lines 3–6 (formal proof in Lemma 3). For each node $a \in V$, we list all subsets $\{b, c, d\} \subseteq \Gamma^+(a)$ such that $c \prec d$. A diamond is found only if edges (b, c) and (b, d) exist, but nodes c and d are not adjacent. Notice that the test at line 5 involves $\overline{E}$, i.e., the direction of edges (b, c) and (b, d) is not relevant, only their existence is: this can be easily implemented on the directed input graph G by checking if $(b, c) \in E$ **or** $(c, b) \in E$, i.e., if $b \in \Gamma^+(c)$ **or** $c \in \Gamma^+(b)$ (the checks on (b, d) are similar).

- Diamonds from case 2 are counted at lines 3 and 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 $(c, d) \in \overline{E}$, 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 $\overline{E}$, 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.

6.3 Analysis

We first 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 (as a consequence, a always coincides with x in Table 6.1). We now show that diamonds are listed by the algorithm according to the classification given in Section 6.1. Namely:

- Lines 3–6: each triple of nodes $b, c, d \in \Gamma^+(a)$ such that $(b, c) \in \overline{E}$, $(b, d) \in \overline{E}$, and $(c, d) \notin E$ (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 the smallest node and that $c \prec d$. Hence, depending on the position of 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 $a \prec c \prec d \prec b$. It is not difficult to see that these correspond to the three configurations 1a, 1b, and 1c shown in Figure 6.1.

- Lines 7–10: each triple of nodes $b \in \Gamma^+(a)$ and $c, d \in \Gamma^+(b)$ such that $(a, d) \in E$, $(c, d) \in \overline{E}$, and $(a, c) \notin \overline{E}$ (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 6.1). Hence, checking if $(c, d) \in \overline{E}$ corresponds to verify if $(c, d) \in E$ when $c \prec d$, or $(d, c) \in E$ when $d \prec c$. It is not difficult to see that 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 6.1.

- Lines 11–15: each pair of nodes $b \in \Gamma^+(a)$ and $c \in \Gamma^+(a) \cap \Gamma^+(b)$ (see lines 7 and 11–12, respectively) corresponds to a triangle $\widehat{abc}$ such that $a \prec b \prec c$. The second smallest diamond node is then chosen from set $V \setminus \{a, b, c\}$. A diamond is counted at line 15 only if $b, c \in \Gamma^+(d)$, but d is not a neighbor of a. It is not difficult to see that these constraints yield a 4-diamond where only one total ordering is possible: $a \prec d \prec b \prec c$. This corresponds to configuration 3f in Figure 6.1. The test $a \prec d$ at line 13 is needed to guarantee that the diamond will be counted only once, i.e., from the point of view of its smallest node a.

In summary, each configuration in each of the three cases shown in Figure 6.1 is considered exactly once by algorithm SeqD4. Since any 4-diamond must belong to one of the three cases, it follows that the overall count (updated at line 6 for Case 1 diamonds, line 10 for Case 2 diamonds, and line 15 for Case 3 diamonds) is correct. □

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

Proof. We separately analyze the running time of algorithm SeqD4 in the three cases of Section 6.1.

In case 1, the time spent to enumerate all nodes $a \in V$ and all nodes $b \in \Gamma^+(a)$ at lines 3 and 4 is $\sum_{a \in V} |\Gamma^+(a)| = m$. Since c and d are also high-neighbors of a and $|\Gamma^+(a)| \leq 2\sqrt{m}$ by Lemma 1, the time to enumerate these node pairs at line 4 is $(2\sqrt{m})^2 = O(m)$. Hence, counting all case 1 diamonds requires $O(m^2)$ time, assuming that node adjacency tests can be implemented in constant time (worst-case or expected, depending on the available graph data structure).

Diamonds from case 2 are counted by exploring progressively the neighborhood of their two smallest nodes. Based on the analysis of case 1, it is not difficult to see that the algorithm spends the same amount of time for listing all diamonds from case 2. All pair of smallest nodes a and b are listed at lines 3 and 7, respectively, in $O(m)$ time. Since $\Gamma^+(b) \leq 2\sqrt{m}$ by Lemma 1, nodes c and d are enumerated in $2\sqrt{m} \cdot 2\sqrt{m} = O(m)$. Therefore, the algorithm count all diamonds from case 2 in $O(m \cdot m) = O(m^2)$.

Focused on diamonds from case 3, the algorithm first lists all triangles in G at lines 3, 7 and 11. This procedure requires $\sum_{a \in V} |\Gamma^+(a)| \cdot |\Gamma^+(b)|$ time, which is at most $2\sqrt{m} \cdot \sum_{a \in V} |\Gamma^+(a)| = O(m^{3/2})$ by Lemma 1. Since the last node d is chosen from V, line 13 requires $O(n)$ time. Hence, all diamonds from case 3 are counted in $O(nm^{3/2})$ time.

Summing up the analysis above, all diamonds are counted in $O(m^2 + nm^{3/2})$ time. Since

$$\sqrt{m}/2 \leq n \leq 2m$$

by Lemma 2, the running time for counting the diamonds is $O(nm^{3/2})$ time, being $O(m^2)$ time on dense graphs and $O(m^{5/2})$ time on sparse graphs.

By Lemma 3, each diamond is counted exactly once by its smallest node (according to $\prec$), proving the correctness of the algorithm. □

As described above, our SeqD4 algorithm counts diamonds from cases 1 and 2 in $O(m^{3/2})$ time. This bound matches the work required by the state-of-the-art algorithm $\texttt{FFF}_k$ [98] for listing all 4-cliques in a graph. However, the work of our algorithm is dominated by the running time for listing diamonds from case 3, requiring $O(nm^{3/2})$. This bound is n times larger than the $\texttt{FFF}_k$ algorithm. Since $\sqrt{m}/2 \leq n \leq 2m$ by Lemma 2, SeqD4 matches the bound of $\texttt{FFF}_k$ algorithm on dense graphs and it is $O(\sqrt{m})$ larger than $\texttt{FFF}_k$ on sparse graphs. The extra work is due to the need to verify the existence of exactly one pair of non-adjacent nodes in a diamond, which is worse on diamonds classified as case 3. Hence, SeqD4 guarantees that 4-cliques and C4 (4-cliques with two missing edges) are not counted during the computation.

6.4 Extension to k-diamonds

Based on the analysis of SeqD4 algorithm introduced in Section 6.3, a first natural question is whether we can extend the algorithm to k-diamonds keeping the running time $O(\sqrt{m})$ times larger than the state-of-the-art algorithm for counting k-cliques. Therefore, in this section we describe an extension of the algorithm SeqD4 to compute all diamonds on k nodes in a graph for $k \geq 4$, we call this approach SeqDk.

As described in section 6.1, although the number of diamond configurations increases as function of k, the number of diamond cases remains the same. Hence, based on the SeqD4 algorithm in Figure 6.2, we describe how to compute the k-diamonds in each of the three classifications with a few changes in the pseudocode.

- Case 1: Consider a diamond D on k nodes such that x is its smallest node. In this case, $d_D(x) = k - 1$ and node a in Figure 6.2 plays the role of the smallest node x. The algorithm lists all nodes in V at line 3. Them, all subsets S of size $k - 1$ in $\Gamma^+(a)$ are listed at line 4. A diamond is counted at line 6 only if there is a clique $Q \subseteq S$ such that $|Q| = k - 3$ and a pair of nodes $b, c \in S \backslash Q$ such that $b \prec c$, b and c are adjacent to all nodes in Q and $(b, c) \notin E$.

- Case 2: Considering that D is classified as case 2, the smallest x has $k - 2$ neighbors in D, while the second smallest node y has $k-1$, moreover $y \in \Gamma^+(x)$. In this case, nodes a and b in the pseudocode play the roles of nodes x and y, respectively. The algorithm lists all edges $(a, b) \in E$ such that $a \prec b$ at lines 3 and 7. Then, for each clique $Q \in \Gamma^+(b)$ on $k - 2$ nodes, the algorithm counts the diamond at line 10 only if there is exactly one node in Q not adjacent to a.

- Case 3: Assuming that D belongs to case 3, its two smallest nodes x and y, respectively, have $k - 2$ neighbors in D, consequently they are not adjacent. In this case, nodes x and y are represented by nodes a and d in the pseudocode. The algorithm finds D by computing a clique Q of size $k - 2$ such that $Q \subseteq \Gamma^+(a) \cap \Gamma^+(d)$. This is implemented as follows. The algorithm first lists

all nodes $a \in V$ at line 3. At lines 7 and 11, all cliques Q of size $k-2$ in $\Gamma^+(a)$ are listed. Then, for each node $d \in V \backslash \{a\} \cup Q$ such that $a \prec d$, the diamond is counted at line 15 only if $Q \subseteq \Gamma^+(d)$.

Theorem 2. *The algorithm SeqDk enumerates all diamonds from cases 1, 2 and 3 exactly once in $O(m^{k/2})$, $O(m^{k/2})$ and $O(nm^{(k-1)/2})$ time, respectively. Therefore, the overall running time of SeqDk is $O(nm^{(k-1)/2})$.*

Proof. Diamonds classified as case 1 are counted by exploring the neighborhood of their smallest node a, listing $k-1$ nodes in $\Gamma^+(a)$. Hence, the algorithm enumerates all diamonds in $\sum_{a \in V} \binom{|\Gamma^+(a)|}{k-1}$, which is at most $(2\sqrt{m})^{k-2} \cdot \sum_{a \in V} |\Gamma^+(a)| = O(m^{k/2})$ by Lemma 1.

For diamonds from case 2, the algorithm computes all pair of nodes $(a, b) \in E$ such that $a \prec b$ and lists $k-2$ nodes in $\Gamma^+(b)$. The time required for listing all edges (a, b) in E is $O(m)$, while the time to enumerate $k-2$ nodes in $\Gamma^+(b)$ is $\binom{|\Gamma^+(b)|}{k-2}$, which is at most $(2\sqrt{m})^{k-2} = O(m^{(k-2)/2})$ by Lemma 1. Therefore, the total work for computing all nodes from case 2 is $O(m \cdot m^{(k-2)/2}) = O(m^{k/2})$.

Considering diamonds from case 3, the algorithm lists all nodes $a \in V$ and cliques $Q \in \Gamma^+(a)$ of size $k-2$ in $\sum_{a \in V} \binom{|\Gamma^+(a)|}{k-2}$ time, which is at most $(2\sqrt{m})^{k-3} \cdot \sum_{a \in V} |\Gamma^+(a)| = O(m^{(k-1)/2})$. All nodes $d \in V$ are enumerated in $O(n)$ time. Hence, the running time for listing all diamonds from case 3 is $O(nm^{(k-1)/2})$. Considering that $n = \Omega(\sqrt{m})$ by Lemma 2, the running time of `SeqDk` is dominated by diamonds from case 3, requiring $O(m^{k/2})$ time on dense graphs and $O(m^{(k+1)/2})$ time on sparse graphs.

SeqDk uses the same properties exploited by SeqD4 for counting 4-diamonds, including the diamond classification and the total order $\prec$ to decide which node is responsible for counting each diamond. Hence, `SeqDk` counts all k-diamonds exactly once. □

Chapter 7

Enumerating diamonds in parallel

In this chapter we adapt to MapReduce framework the algorithms SeqD4 and SeqDk for computing 4-diamonds and k-diamonds in parallel, respectively. The main idea is to split the input undirected graph $\overline{G}$ in many induced subgraphs $G(x)$ for each $x \in \overline{V}$. Thus, we count the diamonds in each subgraph independently. Following the strategy exploited by the sequential approaches, given a diamond D on k nodes, the MapReduce algorithms use the total order $\prec$ and the diamond classification described in Section 6.1 to decide which node is responsible for counting D.

At a high level, our strategy consists of listing all triangles in a graph $\overline{G}$ exactly once and use them to create subgraphs $G(x)$ induced by the neighborhood $\Gamma(x)$ for each node $x \in \overline{V}$. Then, we compute locally the number of diamonds for which x is responsible. We begin by setting up the computation in Section 7.1. This procedure will be used by our MapReduce algorithms as a first step for counting diamonds. For a better comprehension, we first describe an algorithm to compute all 4-diamonds in Section 7.2. We then present two algorithms for counting k-diamonds in Sections 7.3 and 7.4.

7.1 Setting up the computation

The computation of triangles can be regarded as an adaptation to diamonds of the $\texttt{FFF}_k$ algorithm described in [98]. The pseudocode is shown in Figure 7.1. It works in two rounds as follows.

Round 1. High-neighborhood computation. Considering that the input graph $\overline{G} = (\overline{V}, \overline{E})$ is undirected, we apply the total order $\prec$ over the nodes $a \in \overline{V}$ to compute the high-neighborhood of each node. For each edge (a, b), mappers emit the pair $\langle a; b \rangle$ only if $a \prec b$, allowing the reducer with key a to aggregate all nodes $b \in \Gamma^+(a)$.

Round 2. Finding triangles. In this round we discover all triangles by exploring the high-neighborhood of nodes. Map instance with input $\langle a; \Gamma^+(a) \rangle$ emits a pair

Algorithm 4. ListTri (undirected graph $\overline{G} = (\overline{V}, \overline{E})$)

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

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

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

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

Figure 7.1. Listing triangles in MapReduce

$\langle (b, c); a \rangle$ for each par of nodes $b, c \in \Gamma^+(a)$ such that $b \prec c$. Besides the output of round 1, mappers in round 2 are fed with the original set of edges and emit a pair $\langle (a, b); \$ \rangle$ for each edge $(a, b) \in \overline{E}$ such that $a \prec b$. 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 receives the set of nodes a such that $a \prec b \prec c$, where each node a completes a triangle together with nodes b and c.

Theorem 3. *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 $O(n)$ 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 $\sum_{a \in V} \binom{|\Gamma^+(a)|}{2}$, which is at most $2\sqrt{m} \cdot \sum_{a \in V} |\Gamma^+(a)| = O(m^{3/2})$ by Lemma 1. Reduce 2 instances emit all triangles in $\overline{G}$, using $O(m^{3/2})$ space. Hence, the total space usage of ListTri is $O(m^{3/2})$. Focused 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 $a \in \overline{V}$ such that $a \in \Gamma^-(b) \cap \Gamma^-(c)$, which is at most $O(n)$. Since $n \geq \sqrt{m}/2$ 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 $\sum_{a \in V} \binom{|\Gamma^+(a)|}{2} = O(m^{3/2})$. Since the number of triangles in $\overline{G}$ is upper bounded by $\sum_{a \in V} \binom{|\Gamma^+(a)|}{2}$, 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 $a \in \overline{V}$. Moreover, a triangle $\widehat{a, b, c}$ is listed only if $a \prec b \prec c$ and $(a, b), (a, c), (b, c) \in \overline{E}$, ensuring that each triangle is enumerated exactly once. □

Algorithm 5. ParD4 (undirected graph $\overline{G} = (\overline{V}, \overline{E})$)

```
Map 3: input ⟨(b, c); {a_1, ..., a_t}⟩
    for each a ∈ {a_1, ..., a_t} do
        emit ⟨a; (b, c)⟩, ⟨b; (a, c)⟩ and ⟨c; (a, b)⟩

Reduce 3: input ⟨a; G(a)⟩
  d_4 ← 0
  Let Γ+(a) be the set of nodes u ∈ G(a) s.t. a ≺ u
  for each {b, c, d} ⊆ Γ+(a) s.t. c ≺ d do
      if (b, c), (b, d) ∈ Ē and (c, d) ∉ E then
          d_4 ← d_4 + 1                          ▷ Case 1
  for each (b, c) ∈ G(a) s.t. b ≺ a ≺ c do
      for each d ∈ Γ+(a)\{c} do
          if (c, d) ∈ Ē and (b, d) ∉ E then
              d_4 ← d_4 + 1                      ▷ Case 2
      for each d ∈ G(a)\{b} s.t. b ≺ d ≺ a do
          if (d, c) ∈ E and (b, d) ∉ E then
              d_4 ← d_4 + 1                      ▷ Case 3
  emit ⟨a; d_4⟩
```

Figure 7.2. MapReduce code for counting 4-diamonds in triangle space

7.2 Listing 4-diamonds in triangle space

As described in Chapter 2, a 4-diamond can be represented by two triangles with one common edge (see Figure 2.1b). Based on that, we introduce an algorithm, called ParD4, to compute all 4-diamonds in parallel using triangle space, i.e., $O(m^{3/2})$. 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 $x \prec y \prec u \prec v$. According to the diamond classification described in Section 6.1, 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.
- Case 3: the third smallest node u is responsible for counting D.

We present the pseudocode of the algorithm ParD4 in Figure 7.2. Nodes a, b, c and d in reduce 3 phase can represent different nodes from Figure 6.1, according to the classification of their 4-diamond. The roles played by those nodes in each 4-diamond configuration are given in Table 7.1. Notice that node a always plays the role of the responsible node for counting its diamond.

The strategy of ParD4 is to count all 4-diamonds in each induced subgraph $G(a)$ for each node $a \in \overline{V}$. This is implemented as follows. All triangles in $\overline{G}$ are listed in rounds 1 and 2 by 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 $\widehat{abc}$ such that $a \prec b \prec c$, the instance emits three tuples $\langle a; (b, c)\rangle$, $\langle b; (a, c)\rangle$ and $\langle c; (a, b)\rangle$. After shuffling, reduce 3 instance with key a receives as input a list of edges between the nodes in $\Gamma(a)$. Hence, the subgraph $G(a)$ induced by the neighborhood of a can be reconstructed. Similar to our sequential algorithm, reduce instances count each diamond according to its

Table 7.1. Roles played by nodes a, b, c and d in Figure 7.2 according to the six diamond types described in Figure 6.1.

	1a	1b	1c	2d	2e	3f
a	x	x	x	y	y	u
b	y	u	v	x	x	x
c	u	y	y	v	u	v
d	v	v	u	u	v	y

classification, computing locally the number of diamonds for which a is responsible. Notice that $G(a)$ is composed of nodes in $\Gamma(a)$ which is $\Gamma^-(a) \cup \Gamma^+(a)$. 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 $(b,c) \in \overline{E}$ and $b \prec c$.

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

Proof. The algorithm uses triangles to reconstruct induced subgraphs $G(a)$ for each $a \in \overline{V}$ and explores it to count 4-diamonds for which a is responsible. We remark that node a always plays the role of the responsible node for counting its 4-diamond. Therefore, a is the first, second and third smallest node of 4-diamonds classified as case 1, 2 and 3, respectively.

Rounds 1 and 2 list all triangles in $\overline{G}$. For each triangle, map 3 instances emit all possible combination of a node and an edge, creating three possible tuples. Reduce 3 instance with key a receives a set of edges (b,c) such that $b \prec c$, nodes b and c are in $\Gamma(a)$ and (b,c) is an edge in $\overline{E}$. The set of edges are used to reconstruct the induced graph $G(a)$.

Each 4-diamond is counted in reduce 3 phase according to the classification described in Section 6.1. Consider a diamond D with nodes set $\{a,b,c,d\}$, we analyze the correctness of the algorithm ParD4 in the three cases:

- Case 1: each triple of nodes b, c and d selected in $\Gamma^+(a)$ such that $c \prec d$ represents a possible 4-diamonds belonging to case 1, where its smallest node a has degree 3 in D. If the checks $(b,c),(b,d) \in \overline{E}$ and $(c,d) \notin E$ succeed, node b is the second smallest node with $d_D(b) = 3$, while nodes c and d are the pair of non-adjacent nodes in the diamond. Considering that $c \prec d$ and $a \prec b,c,d$, 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 $a \prec c \prec d \prec b$. It is not difficult to see that these orderings correspond to the three configurations 1a, 1b and 1c shown in Figure 6.1. Notice that, although $G(a)$ is a directed graph, the test $(b,c),(b,d) \in \overline{E}$ involves $\overline{E}$. In this case, checking if $(b,c) \in \overline{E}$ corresponds to check if $(b,c) \in E$ when $b \prec c$, or $(c,b) \in E$ when $c \prec b$ (the checks on (b,d) are similar).

- Case 2: each pair of nodes $b,c \in G(a)$ such that $b \prec a \prec c$ corresponds to a triangle $\widehat{bac}$. Each node $d \in \Gamma^+(a) \backslash \{c\}$ together with triangle $\widehat{bac}$ can correspond to a 4-diamond belonging to case 2. Hence, the smallest node b has degree 2 and the second smallest node a has degree 3. The checks $(c,d) \in \overline{E}$

and $(b, d) \notin E$ guarantee that $b \prec a \prec c, d$ 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 ordering $b \prec a \prec c \prec d$ and $b \prec a \prec d \prec c$ correspond to the two configurations 2d and 2e shown in Figure 6.1.

- Case 3: as described on case 2, each pair of nodes $b, c \in G(a)$ such that $b \prec a \prec c$ corresponds to a triangle $\widehat{bac}$. However, instead of selects $d \in \Gamma^+(a)$, node d is chosen in $G(a)\backslash\{b\}$ such that $b \prec d \prec a$. Hence, 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 its 4-diamond. The checks $(d, c) \in E$ and $(b, d) \notin E$ guarantee that $b \prec d \prec a \prec c$ and nodes b and d are the endpoints of the unique missing edge. The test $b \prec d \prec a$ is needed to ensure that the diamond will be counted only once.

Overall, reduce 3 exploits the properties introduced in the sequential approach described in Section 6.2, proving the correctness of the algorithm. □

Theorem 4. *Given an undirected graph $\overline{G}$ with n nodes and m edges, algorithm* PARD4 *correctly enumerates 4-diamonds in $\overline{G}$ using $O(m^{3/2})$ total space and $O(nm^{3/2})$ work. Mappers and reducers use $O(m)$ local space, and their local running time is $O(nm)$.*

Proof. We focus the analysis of PARD4 on round 3. See Theorem 3 for detailed analysis of rounds 1 and 2.

The total space usage in the first two rounds is $O(m^{3/2})$. Since round 2 emits the triangles in the input graph, map 3 instances uses $O(m^{3/2})$ space. Each triangle is triplicated in map 3 phase. Hence, reduce 3 uses $O(m^{3/2})$ space, concluding the proof of total space claim. Focused on local space, mappers and reducers in rounds 1 and 2 use $O(n)$ space, as well as any map 3 instance. 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 $n \leq 2m$ by Lemma 2).

By similar arguments, the running time of mappers and reducers in rounds 1 and 2 is $O(m)$. Each map instance requires $O(n)$ time. We analyze the local running time of reduce 3 instances in the three cases of Section 6.1 separately. 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, reduce 3 lists all edges $(b, c) \in G(a)$ in $O(m)$ time and all nodes $d \in \Gamma^+(a)$ in $O(\sqrt{m})$ time. Therefore, the running time for listing all 4-diamonds belonging to case 2 is $O(m \cdot \sqrt{m}) = O(m^{3/2})$. Focused on case 3, edges $(b, c) \in G(a)$ are enumerated in $O(m)$ time, while nodes $d \in G(a)$ such that $d \prec a$ are listed in $O(n)$ time. Hence, 4-diamonds from case 3 are counted in $O(nm)$ time. Case 3 dominates the local running time of reduce 3, since $n = \Omega(\sqrt{m})$ by Lemma 2. Therefore, the running time of the mappers and reducers is $O(nm)$. We remark that node adjacency tests can be implemented in constant time (worst-case or expected, depending on the available graph data structure).

The total work of rounds 1 and 2 is $O(m^{3/2})$. Map 3 requires $O(m^{3/2})$. We consider separately the three cases of Section 6.1 to analyze the work of reduce 3.

In case 1, the work required for counting all 4-diamonds is $\sum_{a\in V}\binom{|\Gamma^+(a)|}{3}$, which is at most $2\sqrt{m}\cdot 2\sqrt{m}\cdot\sum_{a\in V}|\Gamma^+(a)| = O(m^2)$ by Lemma 1. Since $G(a)\leq m$, 4-diamonds from case 2 are counted in $O(m\cdot\sum_{a\in V}|\Gamma^+(a)|) = O(m^2)$. Focused on 4-diamonds from case 3, reduce 3 lists all triangles $\widehat{bac}$ in $O(m^{3/2})$ time, and, for each node $d\in G(a)$, it verifies whether edge (c,d) exists and nodes b and d are not adjacent in constant time. Therefore, 4-diamonds from case 3 are counted in $O(nm^{3/2})$ time, dominating the total work of algorithm ParD4.

By Lemma 4, each 4-diamond is counted exactly once by the reducer associated to its responsible node (according to the total order $\prec$ and diamond classification), proving the correctness of the algorithm. □

7.3 Extension to k-diamonds.

In this section we describe an extension to k-diamonds for $k\geq 4$ of the algorithm ParD4, called ParDk. This approach can compute all k-diamonds using triangle space $O(m^{3/2})$ and $O(nm^{(k-1)/2})$ work, achieving the running time required for computing k-diamonds sequentially by algorithm SeqDk described in Section 6.4.

Based on the diamond classification (Section 6.1), we describe how to compute all k-diamonds in each of the three distinct cases with a few changes in the pseudocode of ParD4. We remark that, although the number of diamond configurations shown in Figure 6.1 would increase as a function of k, the classifications into three main cases remains valid.

Considering that ParDk is an extension to k-diamonds of ParD4, it exploits the strategies introduced in Section 7.2: algorithm ListTri to enumerate all triangles in the first two rounds and map 3 phase from ParD4. Hence, map 3 emits each triangle in three different formats, allowing reduce 3 instances to reconstruct the induced subgraph $G(a)$ for each $a\in\overline{V}$. The changes to compute k-diamonds are concentrated in the third reduce. We present the reduce 3 pseudocode of algorithm ParDk in Figure 7.3.

Algorithm 6. ParDk (undirected graph $\overline{G}=(\overline{V},\overline{E})$, diamond size k)

```
Reduce 3: input ⟨a; G(a)⟩
  d_k ← 0
  Let Γ+(a) be the set of nodes u ∈ G(a) s.t.
  a ≺ u
  t ← k − 3
  for each b, d ∈ Γ+(a) s.t. b ≺ d do
     for each Q_t ⊆ Γ+(a)\{b, d} do
        if Q_t ⊂ Γ(b)∩Γ(d) and (b, d)∉ E then
           d_k ← d_k + 1                    ▷ Case 1
  t ← k − 4
  for each Q_t ⊂ Γ+(a) do
     for each (b, c)∈ G(a)\Q_t s.t. b≺a≺c do
        if Q_t ⊆ Γ+(b) ∩ Γ+(c) then
           for each d ∈ Γ+(a)\Q_t ∪ {c} do
              if {c}∪Q_t⊂Γ(d) and (b, d)∉ E
              then
                 d_k ← d_k + 1              ▷ Case 2
           for each d ∈ G(a)\{b} s.t.
           b ≺ d ≺ a do
              if {c}∪Q_t⊂Γ+(d) and (b, d)∉E
              then
                 d_k ← d_k + 1              ▷ Case 3
  emit ⟨a; d_k⟩
```

Figure 7.3. Reduce 3 code for counting k-diamonds in triangle space

The strategy of ParDk is to count all k-diamonds in each induced subgraph $G(a)$

for each node $a \in \overline{V}$. This is done as follows. Similar to algorithm PARD4, after reconstructing the induced subgraph $G(a)$ for each node $a \in \overline{V}$ in the reduce 3 phase, the instance with key a counts each k-diamond in $G(a)$ according to its classification, computing locally the number of k-diamonds for which a is responsible.

The responsible node for counting its k-diamond follows the same rules from algorithm PARD4. In the pseudocode, node a always plays the role of the responsible node for counting its k-diamond. Hence, a is the smallest node of k-diamonds from case 1, second smallest in case 2, and third smallest in case 3.

Lemma 5. *Each k-diamond is counted exactly once by algorithm* PARDK.

Proof. By Theorems 3 and 4, the first two rounds correctly enumerates all triangles in $\overline{G}$ and map 3 emits each triangle in three different formats. Hence, we focus the proof of correctness on last reduce phase. Reduce 3 instance with key a receives a set of edges (b, c) such that $b \prec c$, nodes b and c are in $\Gamma(a)$ and (b, c) is an edge in $\overline{E}$. This set of edges is used to reconstruct the induced graph $G(a)$.

Let Q_t denote a clique of size t with nodes $\{u_1, \ldots, u_t\}$ such that $u_1 \prec u_2 \prec \ldots \prec u_t$. Each clique Q_t is enumerated exactly once according to the total order. Consider a diamond D on k nodes. We analyze the correctness of the algorithm PARDK in the three classifications described in Section 6.1.

- Case 1: each pair of nodes $b, d \in \Gamma^+(a)$ such that $b \prec d$ together with a clique $Q_t \subseteq \Gamma^+(a) \backslash \{b, d\}$ correspond to a possible k-diamond belonging to case 1, where its smallest node a has degree $k-1$ in D. If the checks $Q_t \in \Gamma(b) \cap \Gamma(d)$ and $(b, d) \notin E$ succeed, node a and all nodes in Q_t have indeed degree $k-1$, while nodes b and c 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_t correspond to a unique k-diamond configuration from case 1.

- Case 2: each clique Q_t of size $k-4$ together with a pair of nodes $b, c \in G(a) \backslash Q_t$ such that $b \prec a \prec c$ and a node $d \in \Gamma^+(a) \backslash Q_t \cup \{c\}$ corresponds to a possible k-diamond belonging to case 2. In this case, the smallest node b has $d_D(b) = k-2$, while the second smallest node a has $d_D(a) = k-1$. The check $Q_t \subseteq \Gamma^+(b) \cap \Gamma^+(c)$ guarantees that all nodes in Q_t are high-neighbors of nodes b and 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, it is not difficult to see that the total order between the node d and nodes $Q_t \cup \{c\}$ correspond to a unique k-diamond configuration from case 2.

- Case 3: the enumeration of k-cliques from case 3 is very similar to k-cliques from case 2. The main difference between these two processes is the enumeration of the last node d in the k-diamond. Therefore, each clique Q_t of size $k-4$ together with a pair of nodes $b, c \in G(a) \backslash Q_t$ such that $b \prec a \prec c$ correspond to a clique of size $k-1$, composed of nodes $\{b, a, c\} \cup Q_t$. In this case, the last node d is enumerated in $G(a) \backslash \{b\}$ such that $b \prec d \prec a$. 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. Hence, the smallest node b has degree 2, as well as the second smallest node d. Only one diamond configuration can exists in case 3, where nodes b and d are the endpoints of the unique missing edge and the total order over the nodes is $b \prec d \prec a \prec c \prec Q_t$. □

Theorem 5. *Let $\overline{G}$ be an undirected graph with n nodes and m edges. The algorithm* ParDk *counts the number of k-diamonds in $\overline{G}$ exactly once using $O(m^{3/2})$ total space and $O(nm^{(k-1)/2})$ work. Mappers and reducers use $O(m)$ local space, and their local running time is $O(nm^{(k-2)/2})$.*

Proof. Considering that ParDk uses rounds 1 and 2 from algorithm ListTri (Figure 7.1) and map 3 phase from algorithm ParD4 (Figure 7.2), we focus the proof on reduce 3 phase. Detailed analysis of first two rounds and map 3 phase is provided in Theorems 3 and 4, respectively.

The total space usage in the rounds 1 and 2 is $O(m^{3/2})$. Map 3 receives all triangles in the input graph and triplicates each triangle, using $O(m^{3/2})$ space. Considering that each triangle is triplicated in map 3, the space used by reduce 3 instances remains $O(m^{3/2})$. Focused on local space, mappers and reducers in the first two rounds use $O(m)$ space, as well as any map 3 instance. Based on the output of map 3, $G(a)$ can be represented by a list of edges. Therefore, any reduce 3 instance uses at most $O(m)$ space. Thus, the local space of mappers and reducers is $O(m)$ (recall $n \leq 2m$ by Lemma 2).

The running time of mappers and reducers in rounds 1 and 2 is $O(m)$. Each map 3 instance requires $O(n)$ time. We analyze the local running time of reduce 3 instances in the three cases of Section 6.1 separately. Consider a reduce 3 instance with key a. The local running time for counting k-diamonds belonging to case 1 is $\binom{|\Gamma^+(a)|}{k-1}$, which is at most $O(m^{(k-1)/2})$ by Lemma 1. In case 2, cliques $Q_t \subset \Gamma^+(a)$ of size $k-4$ are listed in $\binom{|\Gamma^+(a)|}{k-4} = O(m^{(k-4)/2})$ time. Edges $(b,c) \in G(a)$ requires $O(m)$ time, since $G(a)$ is composed of $O(m)$ edges. Each node $d \in \Gamma^+(a)$ is listed in $O(\sqrt{m})$. Hence, the local running time for counting k-diamonds belonging to case 2 is $O(m^{(k-4)/2} \cdot m \cdot \sqrt{m}) = O(m^{(k-1)/2})$. The difference between cases 2 and 3 is the enumeration of the last node d. Hence, cliques $Q_t \subset \Gamma^+(a)$ of size $k-4$ and edges $(b,c) \in G(a)$ are listed in $O(m^{(k-4)/2})$ and $O(m)$ time, respectively. Nodes $d \in G(a)$ such that $d \prec a$ require $O(n)$ time, since $\Gamma^-(a) \leq O(n)$. Therefore, k-diamonds classified as case 3 are counted in each reduce 3 instance in $O(m^{(k-4)/2} \cdot m \cdot n) = O(nm^{(k-2)/2})$ time, dominating the local work of algorithm ParDk.

Applying the same analysis on total work, k-diamonds from case 1 are counted in $\sum_{a \in V} \binom{|\Gamma^+(a)|}{k-1}$, which is at most $O(m^{(k-2)/2} \cdot \sum_{a \in V} |\Gamma^+(a)|) = O(m^{k/2})$ by Lemma 1. For k-diamonds classified as case 2, the total work required is $O(m^{k/2})$. Cliques $Q_t \subset \Gamma^+(a)$ requires $\sum_{a \in V} \binom{|\Gamma^+(a)|}{k-4} = O(m^{(k-3)/2})$, edges (b,c) are listed in $O(m)$ time, while nodes $d \in \Gamma^+(a)$ requires $O(\sqrt{m})$ time. Following the same reasoning for k-diamonds from case 3, listing cliques Q_t, edges (b,c) and nodes d such that $d \prec a$ requires $O(m^{(k-3)/2})$, $O(m)$ and $O(n)$ time, respectively. Hence, the work for listing all k-diamonds belonging to case 3 is $O(m^{(k-3)/2} \cdot m \cdot n) = O(nm^{(k-1)/2})$, dominating the total work of algorithm ParDk. □

7.4 Trading space for parallelism

Algorithm ParDk described in Section 7.3 enumerates all k-diamonds from cases 1 and 2 in $O(m^{(k-1)/2})$ local running time, while k-diamonds from case 3 are listed in $O(nm^{(k-2)/2})$. Considering that k-diamonds belong to case 3 demand more work for counting than k-diamonds from cases 1 and 2, a first natural question is whether we can reduce the local work of case 3 diamonds to be equivalent or even smaller than the local work of k-diamonds from cases 1 and 2, increasing the parallelism. Hence, in this section we introduce an algorithm to compute all k-diamonds belonging to case 3 using $\max\{O(n^2), O(m^{(k-2)/2})\}$ local work. We call this approach ParDkC3.

Our strategy is to combine the algorithms ParDk and ParDkC3 to create a new approach called PDC, where ParDk counts k-diamonds belonging to cases 1 and 2 while ParDkC3 counts k-diamonds classified as case 3. Thus, algorithm PDC can compute all k-diamonds in an input graph using $O(m^{(k-1)/2})$ local work for $k \geq 5$ and $\max\{O(n^2), O(m^{3/2})\}$ for $k = 4$ (formal proof in Theorem 6).

The new approach ParDkC3 uses in the first two rounds the algorithm ListTri described in Section 7.1 to compute all triangles in a graph. Hence, we introduce a new round 3 for computing all k-diamonds belonging to case 3. Consider a k-diamond D such that a_i and a_j are the two smallest nodes and $a_i \prec a_j$. In this approach, the pair of smallest nodes a_i and a_j is responsible for counting D. Notice that the two smallest nodes in a k-diamond belonging to case 3 are not adjacent (see Section 6.1).

We present the pseudocode of ParDkC3 in Figure 7.4. The strategy of the algorithm is to count the k-diamonds from case 3 in parallel for each pair of non-adjacent nodes in $\overline{V}$ with at least $k-2$ neighbors in common. For each pair of nodes $a_i, a_j \in \overline{V}$, the algorithm computes locally the number of $(k-2)$-cliques in the subgraph $G^+(a_i, a_j)$ induced by the high-neighborhoods $\Gamma^+(a_i) \cap \Gamma^+(a_j)$. This is implemented as follows. Map 3 instances receive all triangles from Round 2 (see Figure 7.1). Consider a map 3 instance with key (b, c), it receives a list of nodes $a \in \overline{V}$ such that each node a completes a triangle together with edge (b, c), where $a \prec b \prec c$. For each pair of triangles $\widehat{a_ibc}$ and $\widehat{a_jbc}$ such that $a_i \prec a_j$, the instance emits a tuple with the common edge (b, c) as value and the nodes a_i and a_j as key. Besides the output of round 2, mappers in round 3 are fed with the original set of edges and emit a pair $\langle (a, b); \$ \rangle$ for each edge $(a, b) \in \overline{E}$ such that $a \prec b$. Hence, reduce 3 instance with key (a_i, a_j) can check whether the nodes a_i and a_j are adjacent by looking for the symbol \$ among its value. If (a_i, a_j) is an edge, the algorithm skip it. Otherwise, the algorithm counts the number of $(k-2)$-cliques in $G^+(a_i, a_j)$. Notice that each $(k-2)$-clique completes a k-diamond together with nodes a_i and a_j.

Lemma 6. *Each k-diamond belonging to case 3 is counted exactly once by algorithm* ParDkC3.

Consider a case 3 diamond $D = \{a_i, a_j, b, c, d\}$ such that $a_i \prec a_j \prec b \prec c \prec d$ and $(a_i, a_j) \notin E$. By Theorem 3, all triangles in $\overline{G}$ are enumerated in rounds 1 and 2. Map 3 instance with key (b, c) receives from round 2 a list of nodes $a \in \overline{V}$ such that $a \prec b \prec c$ and $\widehat{abc}$ is a triangle. For each pair of nodes a_i and a_j such that $a_i \prec a_j$, the instance emits (a_i, a_j) as key and the edge (b, c) as value. The same procedure is applied in map 3 instances with keys (b, d) and (c, d). Hence, reduce 3 instance

Algorithm 7. ParDkC3 (undirected graph $\overline{G} = (\overline{V}, \overline{E})$, diamond size k)

```
Map 3: input ⟨(b,c); {a_1,...,a_t}⟩ or ⟨(a,b); ∅⟩
  if input of type ⟨(a,b); ∅⟩ and a ≺ b then
    emit ⟨(a,b); $)⟩
  if input of type ⟨(b,c); {a_1,...,a_t}⟩ then
    for each a_i, a_j ∈ {a_1,...,a_t} s.t. a_i ≺ a_j do
      emit ⟨(a_i,a_j); (b,c)⟩

Reduce 3: input ⟨(a_i,a_j); G^+(a_i,a_j) ∪ $⟩
  d_k ← 0
  t ← k − 2
  if input does not contain $ then
    for each Q_t ⊆ G^+(a_i,a_j) do
      d_k ← d_k + 1
  emit ⟨(a_i,a_j); d_k⟩
```

Figure 7.4. MapReduce code for case 3

with key (a_i, a_j) receives the list of edges $\{(b,c),(b,d),(c,d)\} \subseteq \Gamma^+(a_i) \cap \Gamma^+(a_j)$. The algorithm verifies if (a_i, a_j) is an edge in $\overline{E}$ by looking for symbol \$ among its value. If a_i and a_j are not adjacent, each unique $(k-2)$-clique in $G^+(a_i, a_j)$ completes a k-diamond with nodes a_i and a_j. Therefore, the triangle $\widehat{bcd} \in G^+(a_i, a_j)$ completes the 5-diamond D. Considering that the two smallest node a_i and a_j are not adjacent, the algorithm verifies in map 3 phase if $a_i \prec a_j$, thus D can only be counted by the pair of nodes a_i, a_j in this ordering. Moreover, each $(k-2)$-clique in $G^+(a_i, a_j)$ is enumerated following the total order $\prec$, being listed exactly once. In general, the algorithm lists all non-adjacent pair of nodes a_i and a_j such that $|\Gamma^+(a_i) \cap \Gamma^+(a_j)| \geq 2$ and counts the number of cliques Q_t of size $k-2$ in $G^+(a_i, a_j)$, guaranteeing that each k-diamond from case 3 is counted exactly once. □

Theorem 6. *Let G be a graph with n nodes and m edges. The algorithm* PDC*, which combines the algorithms* ParDk *and* ParDkC3*, counts the number of k-diamonds in $\overline{G}$ exactly once using $O(nm^{3/2})$ total space and $O(nm^{(k-1)/2})$ work. Mappers and reducers use $O(m)$ local space, and their local running time is $O(m^{(k-1)/2})$ for $k \geq 5$ and* $\max\{O(n^2), O(m^{3/2})\}$ *for $k = 4$.*

Proof. We begin by proving the bounds of algorithm ParDkC3. Map 3 receives $O(m^{3/2})$ triangles from round 2 (see Theorem 3). For each triangle $\widehat{a_i bc}$ and for each $a_j \in \{a_1, \ldots, a_t\} \backslash \{a_i\}$, map 3 instances emit $\langle (a_i, a_j); (b, c) \rangle$. Since $|\{a_1, \ldots, a_t\}| \leq n$, map 3 instances emit $O(nm^{3/2})$ tuples of constant size. For each pair of non-adjacent nodes in $\overline{G}$, reduce 3 emits the number of k-diamonds for which these nodes are responsible, using $O(n^2)$ space. Hence, the total space usage of ParDkC3 is $O(nm^{3/2})$. Focused on local space, any map 3 instance receives $O(n)$ nodes. Considering that $G^+(a_i, a_j)$ can be represented by a list of edges, reduce 3 instances receive $O(m)$ edges, concluding the proof of local space claim.

By similar arguments, the local running time of map 3 instances is $\binom{n}{2} = O(n^2)$. Any reduce 3 instance runs on graphs of at most $\sqrt{m}$ nodes and require $O(m^{(k-2)/2})$ time for computing all local cliques Q_t of size $k-2$. The total work of map 3 is $O(nm^{3/2})$, while reduce 3 instances require $\sum_{a_i \in V} \sum_{a_j \in V} \binom{|\Gamma^+(a_i) \cap \Gamma^+(a_j)|}{k-2}$ time, which is at most $O(m^{(k-3)/2} \cdot \sum_{a_i \in V} \sum_{a_j \in V} |\Gamma^+(a_i) \cap \Gamma^+(a_j)|) = O(nm^{(k-1)/2})$ by Lemma 1.

Comparing the bounds of algorithm ParDk for counting k-diamonds from cases 1

and 2 with the bounds of algorithm ParDkC3 for k-diamonds classified as cases 3, we can assume that algorithm PDC counts all k-diamonds in a graph using $O(nm^{3/2})$ total space, $O(m)$ local space, and $O(nm^{(k-1)/2})$ work.

Focused on local running time, for $k \geq 5$, reduce 3 dominates the local running time of the algorithm. Since $n \leq 2m$ by Lemma 2, $O(n^2) \leq O(m^{(k-1)/2})$ for $k \geq 5$. However, for $k = 4$, the local running time of PDC can be dominated by map 3 phase if the number of nodes n in the input graph is large enough. By Lemma 2, $\sqrt{m} \leq n \leq 2m$. Therefore, if $n \leq m^{3/4}$, the local work of map 3 is $O(n^2) \leq O(m^{3/2})$. In this case, reduce 3 dominates the local work of the algorithm, being $O(m^{3/2})$. On the other hand, if $n > m^{3/4}$, the local work is dominated by map 3, being $O(n^2) > O(m^{3/2})$. Therefore, the local running time for $k = 4$ is $\max\{O(n^2), O(m^{3/2})\}$. We present the bounds of the algorithm `PCD` for each MapReduce phase in Table 7.2.

Algorithm ParDk counts all k-diamonds from cases 1 and 2 exactly once by Lemma 5, while ParDkC3 counts all k-diamonds from case 3 exactly once by Lemma 6, proving the correctness of the algorithm PDC. □

Table 7.2. Analysis of `PDC` algorithm

	MR phase	**Global space**	**Local space**	**Global work**	**Local work**
ListTri	M1	$O(m)$	$O(1)$	$O(m)$	$O(1)$
	R1	$O(m)$	$O(\sqrt{m})$	$O(m)$	$O(\sqrt{m})$
	M2	$O(m^{3/2})$	$O(\sqrt{m})$	$O(m^{3/2})$	$O(m)$
	R2	$O(m^{3/2})$	$O(n)$	$O(m^{3/2})$	$O(n)$
ParDk	M3	$O(m^{3/2})$	$O(n)$	$O(m^{3/2})$	$O(n)$
	R3 - cases 1, 2	$O(m^{3/2})$	$O(m)$	$O(m^{k/2})$	$O(m^{(k-1)/2})$
ParDkC3	M3 - case 3	$O(nm^{3/2})$	$O(n)$	$O(nm^{3/2})$	$O(n^2)$
	R3 - case 3	$O(nm^{3/2})$	$O(m)$	$O(nm^{(k-1)/2})$	$O(m^{(k-2)/2})$

Chapter 8

Conclusions and open problems

In this thesis were addressed subgraph enumeration problems in large-scale undirected graphs. We focused on two different structures: k-cliques and one of their relaxations, dubbed k-diamonds.

As the first contribution of this thesis, in Chapter 4 we presented an extensive literature review of subgraph enumeration, discussing the main previous works which address different problems related to k-cliques, clique relaxations, and other groups of subgraphs.

Focusing on k-clique enumeration, in Chapter 5 we proposed parallel shared-memory algorithms. Our clique enumeration approaches, based on nodes' neighborhood intersection, can list all k-cliques in a graph with m edges in $O(m^{3/2})$ work, which is optimal in the worst case. Considering that, to the best of our knowledge, the state-of-the-art multicore algorithms for listing k-cliques do not exist so far in the literature, we compare the proposed approaches with the state-of-the-art sequential and distributed algorithms, respectively. The experimental analysis shows that the our parallel algorithms can largely outperform the running times of a highly optimized sequential solution, especially for the most demanding datasets, and gracefully scale to non-trivial values of k even on medium and large scale graphs with huge numbers of k-cliques. The `1by1` variant provides on average the best performance and appears to be the algorithm of choice in the setting considered in the experiments. Moreover, a performance comparison of `1by1` with distributed solution shows that the multicore algorithm is competitive, and in some cases much faster than, state-of-the-art distributed solutions based on MapReduce. As a by-product of the experimental analysis, it was computed the exact number of k-cliques in many real-world networks from the SNAP repository for $k \in [4, 20]$ and analyzed their distribution. These number of k-cliques were not available in the literature for $k \geq 8$.

The k-diamond structure was not properly addressed in the literature before our study. Therefore, the state-of-the-art algorithms for k-diamond enumeration do not exist so far in the literature. In Chapter 6, we addressed the problem of listing all instances of k-diamonds in a large undirected graph, for a given integer $k \geq 4$, presenting a sequential algorithm that can list all k-diamonds in a graph with m edges in $O(m^{(k+1)/2})$ time.

An extension to parallel computation has been described in Chapter 7. Our parallel approach is based on the MapReduce framework, and can compute all

k-diamonds in a graph with m edges in $O(m^{(k+1)/2})$ total work, $O(m^{k/2})$ local work, $O(m^{3/2})$ total space, and $O(m)$ local space. Comparing the bounds of the sequential and the MapReduce algorithms for k-diamonds with the bounds of the state-of-the-art k-clique enumeration approaches, the proposed solutions require $O(\sqrt{m})$ extra work.

8.1 Future work

The subgraph enumeration problems addressed in this thesis could be extended along different directions. In the following we describe a few directions that would be a natural continuation of this work.

- Considering that the work required by the proposed algorithm for listing k-diamonds is $O(\sqrt{m})$ larger than the work required by the state-of-the-art algorithm for listing k-cliques, a first natural question is whether k-diamonds can be computed using the same work for computing k-cliques, or proving a stronger lower bound, should the problem be intrinsically more difficult.

- A k-diamond is a clique of size k with exactly 1 missing edge. The definition of k-diamond can be extended to (k, s)-diamonds, consisting of cliques on k nodes with s missing edges. Hence, listing (k, s)-diamonds can be an interesting research line.

- The state-of-the-art MapReduce algorithms presented in [98] use as a subroutine the non-distributed clique counting algorithm `L+N`. Hence, it would be interesting to understand whether any performance gains in MapReduce solutions can be obtained by replacing `L+N` with `1by1`, locally exploiting the multiple cores likely to be available at single cluster nodes.

- Since memory appears to be heavily used in the proposed implementations, it would be also helpful to analyze whether different multithreading libraries and programming languages (e.g., Cilk) could provide any performance improvements.

- The problem of listing/counting k-cliques or k-diamonds in large undirected graphs should be finally extended to other types of input large graphs, especially bipartite and directed graphs.

Bibliography

[1] Abdelhamid, E., Abdelaziz, I., Kalnis, P., Khayyat, Z., and Jamour, F. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, (2017), 716. doi:10.1109/SC.2016.60.

[2] Abedijaberi, A. and Leopold, J. FSMS: A frequent subgraph mining algorithm using mapping sets. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **9729** (2016), 761. doi:10.1007/978-3-319-41920-6_58.

[3] Abello, J., Resende, M. G., and Sudarsky, S. Massive quasi-clique detection. pp. 598–612. Springer (2002).

[4] Afrati, F. N., Fotakis, D., and Ullman, J. D. Enumerating subgraph instances using map-reduce. *Proceedings - International Conference on Data Engineering*, (2013), 62. doi:10.1109/ICDE.2013.6544814.

[5] Afrati, F. N. and Ullman, J. D. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, **23** (2011), 1282.

[6] Agrawal, R., Srikant, R., et al. Fast algorithms for mining association rules. vol. 1215, pp. 487–499 (1994).

[7] Ahmed, N. K., Neville, J., Rossi, R. A., and Duffield, N. Efficient graphlet counting for large networks. *Proceedings - IEEE International Conference on Data Mining, ICDM*, **2016-Janua** (2015), 1. doi:10.1109/ICDM.2015.141.

[8] Ahmed, N. K., Willke, T. L., and Rossi, R. A. Estimation of local subgraph counts. *Proceedings - IEEE International Conference on Big Data*, (2016), 586. doi:10.1109/BigData.2016.7840651.

[9] Al Hasan, M. and Dave, V. S. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, **8** (2018), e1226.

[10] Alam, T., Zahin, S. A., Samiullah, C., and Farhan, A. An Efficient Approach for Mining Frequent Subgraphs. *International Conference on Pattern Recognition and Machine Intelligence*, (2017).

[11] Alamgir, Z., Karim, S., and Husnine, S. Linear-time algorithm for generating c-isolated bicliques. *International Journal of Computer Mathematics*, **94** (2017), 1574. doi:10.1080/00207160.2016.1226498.

[12] Alba, R. D. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, **3** (1973), 113.

[13] Almeida, M. T. and Carvalho, F. D. An analytical comparison of the LP relaxations of integer models for the k-club problem. *European Journal of Operational Research*, **232** (2014), 489. doi:10.1016/j.ejor.2013.08.004.

[14] Almeida, M. T. and Carvalho, F. D. Two-phase heuristics for the k-club problem. *Computers and Operations Research*, **52** (2014), 94. doi:10.1016/j.cor.2014.07.006.

[15] Alon, N., Yuster, R., and Zwick, U. Color-coding. *Journal of the ACM (JACM)*, **42** (1995), 844.

[16] Alon, N., Yuster, R., and Zwick, U. Finding and Counting Given Length Cycles. *Algorithmica (New York)*, **17** (1997), 209.

[17] Andersen, A. and Kim, W. NemoLib: A Java Library for Efficient Network Motif Detection. (2017). doi:10.1007/978-3-319-59575-7.

[18] Antoro, S. C., Sugeng, K. A., and Handari, B. D. Application of Bron-Kerbosch algorithm in graph clustering using complement matrix. *AIP Conference Proceedings*, **1862** (2017), 30158. doi:10.1063/1.4991245.

[19] Apache Software Foundation. Apache Hadoop (2014). Available from: http://hadoop.apache.org/.

[20] Aparicio, D., Paredes, P., and Ribeiro, P. A scalable parallel approach for subgraph census computation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **8806** (2014), 194.

[21] Aparicio, D. O., Ribeiro, P. M. P., and Silva, F. M. A. D. Parallel subgraph counting for multicore architectures. *IEEE International Symposium on Parallel and Distributed Processing with Applications*, **1** (2014), 34. doi:10.1109/ISPA.2014.14.

[22] Arifuzzaman, S., Khan, M., and Marathe, M. Patric: A parallel algorithm for counting triangles in massive networks. *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, (2013), 529.

[23] Arifuzzaman, S., Khan, M., and Marathe, M. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. *IEEE International Conference on Big Data*, (2015), 1839.

[24] Arifuzzaman, S., Khan, M., and Marathe, M. A space-efficient parallel algorithm for counting exact triangles in massive networks. *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, (2015), 527.

[25] Arifuzzaman, S., Khan, M., and Marathe, M. Distributed-memory parallel algorithms for counting and listing triangles in big graphs. *arXiv preprint arXiv:1706.05151*, (2017).

[26] Asahiro, Y., Doi, Y., Miyano, E., and Shimizu, H. Optimal approximation algorithms for maximum distance-bounded subgraph problems. pp. 586–600. Springer (2015).

[27] Asahiro, Y., Kubo, T., and Miyano, E. Experimental Evaluation of Approximation Algorithms for Maximum Distance-Bounded Subgraph Problems. *8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS)*, (2016), 892. `doi:10.1109/SCIS-ISIS.2016.0193`.

[28] Asahiro, Y., Miyano, E., and Samizo, K. Approximating maximum diameter-bounded subgraphs. pp. 615–626. Springer (2010).

[29] Azad, A., Buluç, A., and Gilbert, J. Parallel triangle counting and enumeration using matrix algebra. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, (2015), 804.

[30] Balas, E. and Jeroslow, R. Canonical cuts on the unit hypercube. *SIAM Journal on Applied Mathematics*, **23** (1972), 61.

[31] Balasundaram, B., Butenko, S., and Hicks, I. V. Clique Relaxations in Social Network Analysis: The Maximum k -Plex Problem. *Operations Research*, **59** (2011), 133. `doi:10.1287/opre.1100.0851`.

[32] Batagelj, V. and Zaversnik, M. An o(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, (2003).

[33] Becchetti, L., Boldi, P., Castillo, C., and Gionis, A. Efficient semi-streaming algorithms for local triangle counting in massive graphs. *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, (2008), 16.

[34] Behar, R. and Cohen, S. Finding All Maximal Connected s-Cliques in Social Networks. (2018). `doi:10.5441/002/edbt.2018.07`.

[35] Berlowitz, D., Cohen, S., and Kimelfeld, B. Efficient Enumeration of Maximal k-Plexes. *Proceedings of the ACM International Conference on Management of Data*, (2015), 431. `doi:10.1145/2723372.2746478`.

[36] Bhatia, V. and Rani, R. Ap-FSM: A parallel algorithm for approximate frequent subgraph mining using Pregel. *Expert Systems with Applications*, **106** (2018), 217. doi:10.1016/j.eswa.2018.04.010.

[37] Bhattacharya, A., Bishnu, A., Ghosh, A., and Mishra, G. Triangle estimation using polylogarithmic queries. *arXiv preprint arXiv:1808.00691*, (2018).

[38] Bhuiyan, M. A. and Al Hasan, M. An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, **27** (2015), 608. doi:10.1109/TKDE.2014.2345408.

[39] Bisson, M. and Fatica, M. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems*, **28** (2017), 3501.

[40] Björklund, A. and Kaski, P. How proofs are prepared at camelot. *CoRR*, **abs/1602.01295** (2016). doi:10.1145/2933057.2933101.

[41] Blumofe, R. D. and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, **46** (1999), 720.

[42] Boldi, P. and Vigna, S. The webgraph framework i: compression techniques. pp. 595–602. ACM (2004).

[43] Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, **14** (2013), S13. doi:10.1186/1471-2105-14-S7-S13.

[44] Bourjolly, J.-M., Laporte, G., and Pesant, G. An exact algorithm for the maximum k-club problem in an undirected graph. *European Journal of Operational Research*, **138** (2002), 21.

[45] Bressan, M., Chierichetti, F., Kumar, R., Leucci, S., and Panconesi, A. Motif Counting Beyond Five Nodes. *TKDD*, **12** (2018). doi:10.1145/3186586.

[46] Bron, C. and Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, **16** (1973), 575.

[47] Burkhardt, P., Faber, V., and Jul, C. O. Bounds and algorithms for k-truss. (2018), 0.

[48] Cai, S. and Lin, J. Fast solving maximum weight clique problem in massive graphs. *IJCAI International Joint Conference on Artificial Intelligence*, **2016-Janua** (2016), 568.

[49] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, **36** (2015).

[50] Carvalho, F. D. and Almeida, M. T. The triangle k-club problem. *Journal of Combinatorial Optimization*, **33** (2017), 814. doi:10.1007/s10878-016-0009-9.

[51] Chang, M. S., Hung, L. J., Lin, C. R., and Su, P. C. Finding large k -clubs in undirected graphs. *Computing*, **95** (2013), 739. doi:10.1007/s00607-012-0263-3.

[52] Chehreghani, M. H. A framework for description and analysis of sampling-based approximate triangle counting algorithms. *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, (2016), 80.

[53] Chen, L., Peng, B., Ossen, S., and Vullikanti, A. High-Performance Massive Subgraph Counting using Pipelined Adaptive-Group Communication. (2018).

[54] Chen, P.-l., Chou, C.-k., and Chen, M.-s. Distributed Algorithms for k-truss Decomposition. *IEEE International Conference on Big Data*, (2014).

[55] Chen, Q., Fang, C., Wang, Z., Suo, B., Li, Z., and Ives, Z. G. Parallelizing Maximal Clique Enumeration Over Graph Data. *International Conference on Database Systems for Advanced Applications*, **9643** (2016), 249.

[56] Chen, X., Li, Y., Wang, P., and Lui, J. C. S. A General Framework for Estimating Graphlet Statistics via Random Walk. (2016). doi:10.14778/3021924.3021940.

[57] Chen, X. and Lui, J. C. Mining graphlet counts in online social networks. *Proceedings - IEEE International Conference on Data Mining, ICDM*, (2017), 71. doi:10.1109/ICDM.2016.99.

[58] Chen, Y. and Chen, Y. An Efficient Sampling Algorithm for Network Motif Detection. *Journal of Computational and Graphical Statistics*, **8600** (2017), 0. doi:10.1080/10618600.2017.1391696.

[59] Chiba, N. and Nishizeki, T. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, **14** (1985), 210.

[60] Chu, S. and Cheng, J. Triangle listing in massive networks and its applications. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, (2011), 672.

[61] Cohen, J. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, **11** (2009), 29.

[62] Cohen, J. D. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, (2008), 1. doi:noDOI,noURL.

[63] Cohen, S., Kimelfeld, B., and Sagiv, Y. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *Journal of Computer and System Sciences*, **74** (2008), 1147.

[64] Conte, A., De Virgilio, R., Maccioni, A., Patrignani, M., and Torlone, R. Finding all maximal cliques in very large social networks. *Advances in Database Technology - EDBT*, **2016-March** (2016).

[65] Conte, A., Firmani, D., Mordente, C., Patrignani, M., and Torlone, R. Fast Enumeration of Large k-Plexes. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, **10** (2017), 115. doi:10.1145/3097983.3098031.

[66] Conte, A., Firmani, D., Mordente, C., and Torlone, R. Cliques are Too Strict for Representing Communities : Finding Large k-plexes in Real Networks. (2018).

[67] Conte, A., Grossi, R., Marino, A., and Versari, L. Sublinear-Space Bounded-Delay Enumeration for Massive Network Analytics: Maximal Cliques. *ICALP 2016: the 43rd International Colloquium on Automata, Languages, and Programming*, **55** (2016), 148:1.

[68] Conte, A., Kurita, K., Wasa, K., and Uno, T. Listing acyclic subgraphs and subgraphs of bounded girth in directed graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **10628 LNCS** (2017), 169. doi:10.1007/978-3-319-71147-8_12.

[69] Conte, Alessio and De Matteis, Tiziano and De Sensi, Daniele and Grossi, Roberto and Marino, Andrea and Versari, L. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery \& Data Mining*, (2018), 1272. doi:10.1145/3219819.3220093.

[70] Conway, M. E. A multiprocessor system design. *Proceedings of the November 12-14, 1963, fall joint computer conference*, (1963), 139.

[71] Cui, Y., Xiao, D., Cline, D. B., and Loguinov, D. Improving i/o complexity of triangle enumeration. *Data Mining (ICDM), 2017 IEEE International Conference on*, (2017), 61.

[72] Cui, Y., Xiao, D., and Loguinov, D. On efficient external-memory triangle listing. *IEEE Transactions on Knowledge and Data Engineering*, (2018).

[73] Danisch, M., Balalau, O., and Sozio, M. Listing k-cliques in Sparse Real-World Graphs *. (2018). doi:10.1145/3178876.3186125.

[74] Das, P. P. and Khan, M. H. A. Solving Maximum Clique Problem using a novel Quantum-inspired Evolutionary Algorithm. *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, (2015), 1.

[75] Dasari, N. S., Desh, R., and Zubair, M. ParK : An Efficient Algorithm for k -core Decomposition on Multicore Processors. *IEEE International Conference on Big Data*, (2014), 9.

[76] Date, K., Feng, K., Nagi, R., Xiong, J., Kim, N. S., and Hwu, W. M. Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition on the Minsky architecture: Static graph challenge: Subgraph isomorphism. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, (2017). doi:10.1109/HPEC.2017.8091042.

[77] Dave, V. S., Ahmed, N. K., and Hasan, M. A. E-clog: Counting edge-centric local graphlets. *2017 IEEE International Conference on Big Data (Big Data)*, (2017), 586.

[78] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, **51** (2008), 107. doi:10.1145/1327452.1327492.

[79] Demetrovics, J., Quang, H. M., Ank, N. V., and Thi, V. D. An optimization of closed frequent subgraph mining algorithm. *Cybernetics and Information Technologies*, **17** (2017), 3. doi:10.1515/cait-2017-0001.

[80] Dhiman, A. and Jain, S. K. Frequent subgraph mining algorithms for single large graphs - A brief survey. *Proceedings - 2016 International Conference on Advances in Computing, Communication and Automation, ICACCA 2016*, (2016). doi:10.1109/ICACCA.2016.7578886.

[81] Dhiman, A. and Jain, S. K. Optimizing Frequent Subgraph Mining for Single Large Graph. *Procedia Computer Science*, **89** (2016), 378. doi:10.1016/j.procs.2016.06.085.

[82] Djokic, B., Miyakawa, M., Sekiguchi, S., Semba, I., and Stojmenovic, I. Parallel algorithms for generating subsets and set partitions. vol. 450 of *Lecture Notes in Computer Science*, pp. 76–85. Springer (1990).

[83] Douar, B., Liquiere, M., Latiri, C., and Slimani, Y. Fgmac: Frequent subgraph mining with arc consistency. pp. 112–119. IEEE (2011).

[84] Douar, B., Liquiere, M., Latiri, C., and Slimani, Y. Graph-based relational learning with a polynomial time projection algorithm. pp. 98–112. Springer (2011).

[85] Douar, B., Liquiere, M., Latiri, C., and Slimani, Y. LC-mine: a framework for frequent subgraph mining with local consistency techniques. *Knowledge and Information Systems*, **44** (2015), 1. doi:10.1007/s10115-014-0769-4.

[86] Eddin, A. N. and Ribeiro, P. Scalable subgraph counting using MapReduce. *Proceedings of the Symposium on Applied Computing - SAC '17*, (2017), 1574. doi:10.1145/3019612.3019744.

[87] Eden, T., Levi, A., Ron, D., and Seshadhri, C. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, **46** (2017), 1603.

[88] Eden, T., Ron, D., and Seshadhri, C. On Approximating the Number of k-cliques in Sublinear Time. (2017). doi:10.1145/3188745.3188810.

[89] El Baz, D., Hifi, M., Wu, L., and Shi, X. A parallel ant colony optimization for the maximum-weight clique problem. *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, (2016), 796.

[90] Elenberg, E. R., Shanmugam, K., Borokhovich, M., and Dimakis, A. G. Beyond Triangles: A Distributed Framework for Estimating 3-profiles of Large Graphs. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (2015), 229. doi:10.1145/2783258.2783413.

[91] Elenberg, E. R., Shanmugam, K., Borokhovich, M., and Dimakis, A. G. Distributed Estimation of Graph 4-Profiles. *CoRR*, **abs/1510.0** (2015), 483. doi:10.1145/2872427.2883082.

[92] Elmasry, A., Khalafallah, A., and Meshry, M. A scalable maximum-clique algorithm using Apache Spark. *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, (2017). doi:10.1109/AICCSA.2016.7945631.

[93] Elseidy, M., Abdelhamid, E., and Skiadopoulos, S. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proceedings of the VLDB Endowment*, **7** (2014), 517. doi:10.14778/2732286.2732289.

[94] Esfandiari, H., Lattanzi, S., and Mirrokni, V. Parallel and Streaming Algorithms for K-Core Decomposition. *Proceedings of the 35th International Conference on Machine Learning*, **80** (2018), 1396.

[95] Fan, Y., Ma, Z., Su, K., Li, C., Rao, C., Liu, R.-H., and Latecki, L. J. Efficient Local Search for Maximum Weight Cliques in Large Graphs. *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, (2017), 1099. doi:10.1109/ICTAI.2017.00168.

[96] Fang, Z., Li, C. M., and Xu, K. An exact algorithm based on MaxSAT reasoning for the maximum weight clique problem. *Journal of Artificial Intelligence Research*, **55** (2016), 799.

[97] Farhi, S. and Boughaci, D. Two bi-objective hybrid approaches for the frequent subgraph mining problem. *Applied Soft Computing Journal*, **72** (2018), 291. doi:10.1016/j.asoc.2018.07.058.

[98] Finocchi, I., Finocchi, M., and Fusco, E. G. Clique Counting in MapReduce: Algorithms and Experiments. *ACM Journal of Experimental Algorithmics*, **20** (2015).

[99] Floderus, P., Lingas, A., and Lundell, E.-M. DETECTING AND COUNTING SMALL PATTERN GRAPHS *. (2015). doi:10.1137/140978211.

[100] Flores-Garrido, M., Carrasco-Ochoa, J. A., and Martínez-Trinidad, J. F. Mining maximal frequent patterns in a single graph

using inexact matching. *Knowledge-Based Systems*, **66** (2014), 166. doi: 10.1016/j.knosys.2014.04.040.

[101] Foley, D. Nvlink, pascal and stacked memory: Feeding the appetite for big data. *Nvidia. com*, (2014).

[102] Freeman, L. C. The sociological concept of" group": An empirical test of two models. *American journal of sociology*, **98** (1992), 152.

[103] Friedkin, N. E. Structural cohesion and equivalence explanations of social homogeneity. *Sociological Methods & Research*, **12** (1984), 235.

[104] Gao, J., Chen, J., Yin, M., Chen, R., and Wang, Y. An Exact Algorithm for Maximum k -Plexes in Massive Graphs. (2017), 1449.

[105] Gibson, D., Kumar, R., and Tomkins, A. Discovering large dense subgraphs in massive graphs. pp. 721–732 (2005).

[106] Giechaskiel, I., Panagopoulos, G., and Yoneki, E. Pdtl: Parallel and distributed triangle listing for massive graphs. *Parallel Processing (ICPP), 2015 44th International Conference on*, (2015), 370.

[107] Gjoka, M., Smith, E., and Butts, C. Estimating clique composition and size distributions from sampled network data. *Proceedings - IEEE INFOCOM*, (2014), 837.

[108] Gjoka, M., Smith, E., and Butts, C. T. Estimating Subgraph Frequencies with or without Attributes from Egocentrically Sampled Data. (2015).

[109] Golovach, P. A., Heggernes, P., Kratsch, D., and Rafiey, A. Cliques and clubs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **7878 LNCS** (2013), 276. doi:10.1007/978-3-642-38233-8_23.

[110] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: distributed graph-parallel computation on natural graphs. vol. 12, p. 2 (2012).

[111] Grbi, M., Kartelj, A., Jankovi, S., Mati, D., and Filipovi, V. Variable neighborhood search for partitioning sparse biological networks into the maximum edge-weighted k-plexes. Tech. rep. (2018).

[112] Green, O., Munguía, L., and Bader, D. A. Load balanced clustering coefficients. pp. 3–10 (2014).

[113] Green, O., Yalamanchili, P., and Munguía, L.-M. Fast triangle counting on the gpu. Tech. rep. (2014).

[114] Green, O., et al. Quickly finding a truss in a haystack. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, (2017), 1. doi:10.1109/HPEC.2017.8091038.

[115] Gregori, E., Lenzini, L., and Mainardi, S. Parallel k-clique community detection on large-scale networks. *IEEE Trans. Parallel Distrib. Syst.*, **24** (2013), 1651.

[116] Grossman, D. *A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency* (2015).

[117] Guo, J., Zhang, S., Gao, X., and Liu, X. Parallel graph partitioning framework for solving the maximum clique problem using Hadoop. *2017 IEEE 2nd International Conference on Big Data Analysis, ICBDA 2017*, (2017), 186. doi:10.1109/ICBDA.2017.8078804.

[118] Han, G. and Sethu, H. Waddling random walk: Fast and accurate mining of motif statistics in large graphs. *Proceedings - IEEE International Conference on Data Mining, ICDM*, (2017), 181. doi:10.1109/ICDM.2016.120.

[119] Hasan, A., Chung, P. C., and Hayes, W. Graphettes: Constant-time determination of graphlet and orbit identity including (possibly disconnected) graphlets up to size 8. *PLoS ONE*, **12** (2017). doi:10.1371/journal.pone.0181570.

[120] Hebrard, E., Katsirelos, G., and Toulouse, D. Conflict Directed Clause Learning for Maximum Weighted Clique Problem. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, (2018), 1316.

[121] Hey, T. Quantum computing: an introduction. *Computing & Control Engineering Journal*, **10** (1999), 105.

[122] Hočevar, T. and Demšar, J. A combinatorial approach to graphlet counting. *Bioinformatics*, **30** (2014), 559. doi:10.1093/bioinformatics/btt717.

[123] Hočevar, T. and Demšar, J. Combinatorial algorithm for counting small induced graphs and orbits. *PLoS ONE*, **12** (2017). doi:10.1371/journal.pone.0171428.

[124] Hosseinian, S., Fontes, D. B., and Butenko, S. A nonconvex quadratic optimization approach to the maximum edge weight clique problem (2018). doi:10.1007/s10898-018-0630-5.

[125] Hosseinian, S., Fontes, D. B., Butenko, S., Nardelli, M. B., Fornari, M., and Curtarolo, S. The maximum edge weight clique problem: Formulations and solution approaches. vol. 130, pp. 217–237 (2017). doi:10.1007/978-3-319-68640-0_10.

[126] Hou, B., Wang, Z., Chen, Q., Suo, B., Fang, C., Li, Z., and Ives, Z. G. Efficient Maximal Clique Enumeration Over Graph Data. *Data Science and Engineering*, **1** (2017), 219. doi:10.1007/s41019-017-0033-5.

[127] Hu, Y., Kumar, P., Swope, G., and Huang, H. H. Trix: Triangle counting at extreme scale. *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, (2017), 1.

[128] Hua Jiang, Chu-Min Li, Yanli Liu, F. M. A Two-Stage MaxSAT Reasoning Approach for the Maximum Weight Clique Problem. *AAAI Conference on Artificial Intelligence*, (2018), 1338.

[129] Huang, X., Lu, W., and Lakshmanan, L. V. Truss Decomposition of Probabilistic Graphs. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD*, (2016), 77. doi:10.1145/2882903.2882913.

[130] Hutchison, D. Distributed triangle counting in the graphulo matrix math library. *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, (2017), 1.

[131] Ingalalli, V., Ienco, D., and Poncelet, P. Mining Frequent Subgraphs in Multigraphs. *Information Sciences*, (2018), 50. doi:10.1016/j.ins.2018.04.001.

[132] Itai, A. and Rodeh, M. Finding a minimum circuit in a graph. *SIAM J. Comput.*, **7** (1978), 413.

[133] Ito, H. and Iwama, K. Enumeration of isolated cliques and pseudo-cliques. *ACM Transactions on Algorithms (TALG)*, **5** (2009), 40.

[134] Jain, S. and Seshadhri, C. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. *Proceedings of the 26th International Conference on World Wide Web - WWW '17*, **1828** (2017), 441. doi:10.1145/3038912.3052636.

[135] Jha, M., Seshadhri, C., and Pinar, A. title=Triangle finding and listing in CONGEST networks, author=Izumi, Taisuke and Gall, François Le, journal=arXiv preprint arXiv:1705.09061, year=2017. pp. 589–597 (2013).

[136] Jha, M., Seshadhri, C., and Pinar, A. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. (2014). doi:10.1145/2736277.2741101.

[137] Jiang, H., Li, C. M., and Manyà, F. Combining efficient preprocessing and incremental MaxSAT reasoning for MaxClique in Large graphs. *Frontiers in Artificial Intelligence and Applications*, **285** (2016), 939.

[138] Jiang, H., Li, C.-M., and Manya, F. An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs. (2017).

[139] Kabir, H. and Madduri, K. Parallel k-Core decomposition on multicore platforms. *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, (2017), 1482. doi:10.1109/IPDPSW.2017.151.

[140] Kabir, H. and Madduri, K. Shared-Memory Graph Truss Decomposition. *Proceedings - 24th IEEE International Conference on High Performance Computing, HiPC 2017*, **2017-Decem** (2017), 13. doi:10.1109/HiPC.2017.00012.

[141] Kanewala, T. A. and Lumsdaine, A. Distributed , Shared-Memory Parallel Triangle Counting. *Proceedings of the Platform for Advanced Scientific Computing Conference on - PASC '18*, (2018), 1.

[142] Kargar, M. and An, A. Keyword search in graphs: Finding r-cliques. *PVLDB*, **4** (2011), 681. doi:10.14778/2021017.2021025.

[143] Karloff, H., Suri, S., and Vassilvitskii, S. A Model of Computation for MapReduce. *ACM-SIAM Symposium on Discrete Algorithms (SODA10)*, (2010), 11. doi:10.1137/1.9781611973075.76.

[144] Kaski, P. Engineering a Delegatable and Error-Tolerant Algorithm for Counting Small Subgraphs *. Tech. rep. (2018).

[145] Katunka, A. M., Yan, C., Serge, K. B., and Zhang, Z. K-Truss Based Top-Communities Search in Large Graphs. *Proceedings - 5th International Conference on Advanced Cloud and Big Data, CBD 2017*, (2017), 244. doi:10.1109/CBD.2017.49.

[146] Kershenbaum, A., Cutillo, A., Darabos, C., Murray, K., Schiaffino, R., and Moore, J. H. Bicliques in Graphs with Correlated Edges: From Artificial to Biological Networks. (2016). doi:10.1007/978-3-319-31204-0.

[147] Kessl, R., Talukder, N., Anchuri, P., and Zaki, M. J. Parallel Graph Mining with GPUs. *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, **36** (2014), 1.

[148] Khaouid, W., Barsky, M., Srinivasan, V., and Thomo, A. K-core decomposition of large networks on a single PC. *Vldb*, **9** (2015), 13. doi:10.14778/2850469.2850471.

[149] Kim, H., Kim, S., and Min, J.-K. An Efficient Triangle Enumeration on Parallel and Distributed Frameworks. *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, (2018), 545.

[150] Kim, H., Lee, J., Bhowmick, S. S., Han, W.-S., Lee, J., Ko, S., and Jarrah, M. H. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, (2016), 1231. doi:10.1145/2882903.2915209.

[151] Kimelfeld, B. and Kolaitis, P. G. The complexity of mining maximal frequent subgraphs. *Proceedings of the 32nd symposium on Principles of database systems - PODS '13*, (2013), 13. doi:10.1145/2463664.2465222.

[152] Kimmig, R., Meyerhenke, H., and Strash, D. Shared memory parallel subgraph enumeration. *Proceedings - IEEE 31st International Parallel and Distributed Processing Symposium Workshops*, (2017), 519. doi:10.1109/IPDPSW.2017.133.

[153] Kiziloz, H. E. and Dokeroglu, T. A robust and cooperative parallel tabu search algorithm for the maximum vertex weight clique problem. *Computers and Industrial Engineering*, **118** (2018), 54. doi:10.1016/j.cie.2018.02.018.

[154] Kloks, T., Kratsch, D., and Müller, H. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, **74** (2000), 115.

[155] Klusowski, J. M. and Wu, Y. Counting Motifs with Graph Sampling. (2018).

[156] Kolda, T. G., Pinar, A., Plantenga, T., Seshadhri, C., and Task, C. Counting Triangles in Massive Graphs with MapReduce. *Sandia National Laboratories*, **36** (2013), 48.

[157] Kolda, T. G., Pinar, A., and Seshadhri, C. Triadic measures on graphs: The power of wedge sampling. pp. 10–18 (2013).

[158] Kolountzakis, M. N., Miller, G. L., Peng, R., and Tsourakakis, C. E. Efficient Triangle Counting In Large Graphs Via Degree-Based Vertex Partitioning. *Internet Mathematics*, **8** (2012), 161.

[159] Komusiewicz, C., Hüffner, F., Moser, H., and Niedermeier, R. Isolation concepts for efficiently enumerating dense subgraphs. *Theoretical Computer Science*, **410** (2009), 3640.

[160] Komusiewicz, C., Nichterlein, A., Niedermeier, R., and Picker, M. Exact Algorithms for Finding Well-Connected 2-Clubs in Real-World Graphs: Theory and Experiments *. Tech. rep. (2018).

[161] Koyutürk, M., Grama, A., and Szpankowski, W. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, **20** (2004), i200.

[162] Kreher, D. L. and Stinson, D. R. *Combinatorial algorithms: generation, enumeration, and search*, vol. 7. CRC press (1998).

[163] Kuramochi, M. and Karypis, G. Frequent subgraph discovery. pp. 313–320. IEEE (2001).

[164] Lagraa, S. and Seba, H. An efficient exact algorithm for triangle listing in large graphs. *Data Mining and Knowledge Discovery*, **30** (2016), 1350.

[165] Lai, L., Qin, L., Lin, X., and Chang, L. Scalable Subgraph Enumeration in MapReduce. Tech. rep. (2015).

[166] Lai, L., Qin, L., Lin, X., Zhang, Y., Chang, L., and Yang, S. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, **10** (2016), 217. doi:10.14778/3021924.3021937.

[167] Latapy, M. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, **407** (2008), 458.

[168] Lea, D. A Java Fork/Join Framework. JAVA '00, pp. 36–43. ACM (2000). ISBN 1-58113-288-3. doi:10.1145/337449.337465.

[169] Lee, P. and Lakshmanan, L. V. S. Query-Driven Maximum Quasi-Clique Search. Tech. rep. (2016).

[170] Leskovec, J. SNAP graph library (2014). Available from: http://snap.stanford.edu/.

[171] Leskovec, J., Rajaraman, A., and Ullman, J. D. pp. 325–383. Cambridge University Press (2014).

[172] Lessley, B., Perciano, T., Mathai, M., Childs, H., and Bethel, E. W. Maximal clique enumeration with data-parallel primitives. *2017 IEEE 7th Symposium on Large Data Analysis and Visualization, LDAV 2017*, **2017-Decem** (2017), 16. doi:10.1109/LDAV.2017.8231847.

[173] Li, C. M., Fang, Z., Li, C.-M., Qiao, K., Feng, X., and Xu, K. Solving Maximum Weight Clique Using Maximum Satisfiability Reasoning. *ECAI*, (2014).

[174] Li, C. M., Liu, Y., Jiang, H., Manyà, F., and Li, Y. A new upper bound for the maximum weight clique problem. *European Journal of Operational Research*, **270** (2018), 66. doi:10.1016/j.ejor.2018.03.020.

[175] Li, C. M. and Quan, Z. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. vol. 10, pp. 128–133 (2010).

[176] Li, H. A new algorithm for enumerating all maximal cliques. *2017 3rd IEEE International Conference on Computer and Communications, ICCC 2017*, **2018-Janua** (2018), 2176. doi:10.1109/CompComm.2017.8322922.

[177] Li, R., Wu, X., Liu, H., Wu, J. U. N., and Yin, M. An Efficient Local Search for the Maximum Edge Weighted Clique Problem. (2018), 10743.

[178] Li, Z., Lu, Y., Zhang, W.-P., Li, R.-H., Guo, J., Huang, X., and Mao, R. Discovering Hierarchical Subgraphs of K-Core-Truss. (2018). doi:10.1007/s41019-018-0068-2.

[179] Lin, W., Xiao, X., and Ghinita, G. Large-scale frequent subgraph mining in MapReduce. *Proceedings - International Conference on Data Engineering*, (2014), 844. doi:10.1109/ICDE.2014.6816705.

[180] Low, T. M., Rao, V. N., Lee, M., Popovici, D., Franchetti, F., and McMillan, S. First look: Linear algebra-based triangle counting without matrix multiplication. *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, (2017), 1.

[181] Luce, R. D. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, **15** (1950), 169.

[182] Ma, Z., Fan, Y., Su, K., Li, C., and Sattar, A. Deterministic Tournament Selection in Local Search for Maximum Edge Weight Clique on Large Sparse Graphs. (2017). doi:10.1007/978-3-319-63004-5.

[183] Mahdavi Pajouh, F. and Balasundaram, B. On inclusionwise maximal and maximum cardinality k-clubs in graphs. *Discrete Optimization*, **9** (2012), 84. doi:10.1016/j.disopt.2012.02.002.

[184] Mahdavi Pajouh, F., Miao, Z., and Balasundaram, B. A branch-and-bound approach for maximum quasi-cliques. *Annals of Operations Research*, **216** (2014), 145. doi:10.1007/s10479-012-1242-y.

[185] Makino, K. and Uno, T. New algorithms for enumerating all maximal cliques. *Algorithm Theory-SWAT*, (2004), 260.

[186] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. pp. 135–146. ACM (2010).

[187] Mandal, A. and Hasan, M. A. A distributed k-core decomposition algorithm on spark. *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, **2018-Janua** (2018), 976. doi:10.1109/BigData.2017.8258018.

[188] Marcus, D. and Shavitt, Y. RAGE - A rapid graphlet enumerator for large networks. *Computer Networks*, **56** (2012), 810. doi:10.1016/j.comnet.2011.08.019.

[189] Marques-Silva, J. P. and Sakallah, K. A. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, **48** (1999), 506.

[190] Martins, P. Modeling the maximum edge-weight k-plex partitioning problem. *arXiv preprint arXiv:1612.06243*, (2016), 1.

[191] Matula, D. W. and Beck, L. L. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, **30** (1983), 417.

[192] Mawhirter, D., Wu, B., Mehta, D., and Ai, C. ApproxG: Fast approximate parallel graphlet counting through accuracy control. *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018*, (2018), 533. doi:10.1109/CCGRID.2018.00080.

[193] Maxwell, S., Chance, M. R., and Koyutürk, M. Efficiently Enumerating All Connected Induced Subgraphs of a Large Molecular Network. Tech. rep. (2014).

[194] Mcclosky, B. and Hicks, I. V. Combinatorial Algorithms for the Maximum k -plex Problem. (2006), 1.

[195] Mccreesh, C. and Prosser, P. An Exact Branch and Bound Algorithm with Symmetry Breaking for the Maximum Balanced Induced Biclique Problem. Tech. rep. (2014).

[196] Meira, L. A., Máximo, V. R., Fazenda, Á. L., and Da Conceiçao, A. F. Acc-motif: accelerated network motif detection. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, **11** (2014), 853.

[197] Meira, L. A., Máximo, V. R., Fazenda, A. L., and da Conceição, A. F. An faster network motif detection tool. *arXiv preprint arXiv:1804.09741*, (2018).

[198] Miller, B. L., Goldberg, D. E., et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, **9** (1995), 193.

[199] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. Network motifs: simple building blocks of complex networks. *Science*, **298** (2002), 824.

[200] Mohamed, M., Refaat, H., Amin, H., and Abdellh, M. PFSG: Parallel Frequent SubGraphs by HoriVertical Partition using Cloud. *International Journal of Informatics and Medical Data Processing (IJIMDP)*, **2** (2017), 46.

[201] Mokken, R. J. Cliques, clubs and clans. *Quality and quantity*, **13** (1979), 161.

[202] Montresor, A., Pellegrini, F. D., and Miorandi, D. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, **24** (2013), 288.

[203] Moradi, E., Balasundaram, · . B., Balasundaram, B. B., Moradi, E., and Balasundaram, B. Finding a maximum k-club using the k-clique formulation and canonical hypercube cuts. *Optimization Letters*, (2015). doi:10.1007/s11590-015-0971-7.

[204] Moser, H., Niedermeier, R., and Sorge, M. Exact combinatorial algorithms and experiments for finding maximum k-plexes. *Journal of Combinatorial Optimization*, **24** (2012), 347. doi:10.1007/s10878-011-9391-5.

[205] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient sat solver. pp. 530–535. ACM (2001).

[206] Moussaoui, M., Zaghdoud, M., and Akaichi, J. POSGRAMI: Possibilistic frequent subgraph mining in a single large graph. *Communications in Computer and Information Science*, **610** (2016), 549. doi:10.1007/978-3-319-40596-4_46.

[207] Mrzic, A., Meysman, P., Bittremieux, W., Moris, P., Cule, B., Goethals, B., and Laukens, K. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining*, **11** (2018), 20. doi:10.1186/s13040-018-0181-9.

[208] Mukherjee, A. P., Xu, P., and Tirthapura, S. Enumeration of Maximal Cliques from an Uncertain Graph. *IEEE Transactions on Knowledge and Data Engineering*, **29** (2017), 543. doi:10.1109/TKDE.2016.2527643.

[209] Newman, M. E. The structure and function of complex networks. *SIAM review*, **45** (2003), 167.

[210] Nichols, B., Buttlar, D., Farrell, J., and Farrell, J. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc." (1996).

[211] Nogueira, B. and Pinheiro, R. G. A CPU-GPU local search heuristic for the maximum weight clique problem on massive graphs. *Computers and Operations Research*, **90** (2018), 232. doi:10.1016/j.cor.2017.09.023.

[212] Nvidia, C. Programming guide (2010).

[213] Nyman, L. and Laakso, M. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, **38** (2016), 84. doi:10.1109/MAHC.2016.34.

[214] Ortmann, M. and Brandes, U. Triangle Listing Algorithms: Back from the Diversion. *Alenex*, (2014), 1.

[215] Ortmann, M. and Brandes, U. Quad Census Computation: Simple, Efficient, and Orbit-Aware. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **9564** (2016), 1. doi:10.1007/978-3-319-28361-6.

[216] Pagh, R. and Silvestri, F. The input/output complexity of triangle enumeration. pp. 224–233. ACM (2014).

[217] Pagh, R. and Tsourakakis, C. E. Colorful triangle counting and a mapreduce implementation. *Inf. Process. Lett.*, **112** (2012), 277.

[218] Paramonov, K. and Sharpnack, J. Estimating Graphlet Statistics via Lifting. Tech. rep. (2018).

[219] Paredes, P. and Ribeiro, P. Towards a faster network-centric subgraph census. pp. 264–271. ACM (2013).

[220] Park, H.-M. and Chung, C.-W. An efficient MapReduce algorithm for counting triangles in a very large graph. *Cikm*, (2013), 539.

[221] Park, H.-M. and Chung, C.-W. MapReduce Triangle Enumeration with Guarantees. *Conference on Information and Knowledge Management*, (2014), 1739.

[222] Park, H.-m., Myaeng, S.-H., and Kang, U. PTE : Enumerating Trillion Triangles On Distributed Systems. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, (2016), 1115.

[223] Pastukhov, G., Veremyev, A., Boginski, V., and Prokopyev, O. A. On maximum degree-based γ-quasi-clique problem: Complexity and exact approaches. *Networks*, **71** (2018), 136. doi:10.1002/net.21791.

[224] Patterns, D. Fork/Join Parallelism in the Wild. (2014), 39.

[225] Pattillo, J., Youssef, N., and Butenko, S. Clique relaxation models in social network analysis. pp. 143–162. Springer (2012).

[226] Pattillo, J., Youssef, N., and Butenko, S. On clique relaxation models in network analysis. *European Journal of Operational Research*, **226** (2013), 9 . doi:https://doi.org/10.1016/j.ejor.2012.10.021.

[227] Pavan, A., Tangwongsan, K., Tirthapura, S., and Wu, K.-L. Counting and sampling triangles from a graph stream. *PVLDB*, **6** (2013), 1870.

[228] Pei, J., Jiang, D., and Zhang, A. On mining cross-graph quasi-cliques. pp. 228–238. ACM (2005).

[229] Peng, Z., Wang, T., Lu, W., Huang, H., Du, X., Zhao, F., and Tung, A. K. Mining frequent subgraphs from tremendous amount of small graphs using MapReduce. *Knowledge and Information Systems*, **56** (2018), 663. doi:10.1007/s10115-017-1104-7.

[230] Petermann, A., Junghanns, M., and Rahm, E. DIMSpan - Transactional Frequent Subgraph Mining with Distributed In-Memory Dataflow Systems. (2017). doi:10.1145/3148055.3148064.

[231] Pinar, A., Seshadhri, C., and Vishal, V. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. *Proceedings of the 26th International Conference on World Wide Web*, (2017), 1431. doi:10.1145/3038912.3052597.

[232] Pinto, B. Q., Ribeiro, C. C., Rosseti, I., and Plastino, A. A biased random-key genetic algorithm for the maximum quasi-clique problem. *European Journal of Operational Research*, **271** (2018), 849. doi:10.1016/j.ejor.2018.05.071.

[233] Polak, A. Counting triangles in large graphs on GPU. *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, (2016), 740.

[234] Qiao, F., Zhang, X., Li, P., Ding, Z., Jia, S., and Wang, H. A Parallel Approach for Frequent Subgraph Mining in a Single Large Graph Using Spark. *Applied Sciences*, **8** (2018), 230. doi:10.3390/app8020230.

[235] Radie, E. E. and Salem, S. A parallel algorithm for mining maximal frequent subgraphs. *Proceedings - 2017 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2017*, **2017-Janua** (2017), 1965. doi:10.1109/BIBM.2017.8217963.

[236] Rahman, M. and Al Hasan, M. Approximate triangle counting algorithms on multi-cores. *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, (2013), 127.

[237] Rahman, M., Bhuiyan, M. A., and Al Hasan, M. Graft: An Efficient Graphlet Counting Method for Large Graph Analysis. *IEEE Transactions on Knowledge and Data Engineering*, **26** (2014), 2466. doi:10.1109/TKDE.2013.2297929.

[238] Rajaraman, A. and Ullman, J. D. *Mining of Massive Datasets*. Cambridge University Press (2012).

[239] Ramraj, T. and Prabhakar, R. Frequent subgraph mining algorithms - A survey. *Procedia Computer Science*, **47** (2014), 197. doi:10.1016/j.procs.2015.03.198.

[240] Rasel, M. K., Elena, E., and Lee, Y. K. Summarized bit batch-based triangle listing in massive graphs. *Information Sciences*, **441** (2018), 1.

[241] Rasel, M. K., Han, Y., Kim, J., Park, K., Tu, N. A., and Lee, Y. K. ITri: Index-based triangle listing in massive graphs. *Information Sciences*, **336** (2016), 1.

[242] Ribeiro, P. and Silva, F. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, **28** (2014), 337.

[243] Rossi, R. A., Gleich, D. F., Gebremedhin, A. H., and Patwary, M. M. A. Parallel Maximum Clique Algorithms with Applications to Network Analysis and Storage. *arXiv preprint arXiv:1302.6256*, (2013).

[244] Rossi, R. A. and Zhou, R. Leveraging Multiple GPUs and CPUs for Graphlet Counting in Large Networks. *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, (2016), 1783. doi:10.1145/2983323.2983832.

[245] Rossi, R. A., Zhou, R., and Ahmed, N. K. Estimation of Graphlet Counts in Massive Networks. *IEEE Transactions on Neural Networks and Learning Systems*, (2018), 1. doi:10.1109/TNNLS.2018.2826529.

[246] Saha, B., Hoch, A., Khuller, S., Raschid, L., and Zhang, X.-N. Dense subgraphs with restrictions and applications to gene annotation graphs. pp. 456–472. Springer (2010).

[247] Saha, T. K. and Al Hasan, M. Finding network motifs using MCMC sampling. *Studies in Computational Intelligence*, **597** (2015), 13. doi:10.1007/978-3-319-16112-9_2.

[248] San Segundo, P., Artieda, J., Batsyn, M., and Pardalos, P. M. An enhanced bitstring encoding for exact maximum clique search in sparse graphs. *Optimization Methods and Software*, **32** (2017), 312. doi:10.1080/10556788.2017.1281924.

[249] San Segundo, P., Artieda, J., and Strash, D. Efficiently enumerating all maximal cliques with bit-parallelism. *Computers and Operations Research*, **92** (2018), 37. doi:10.1016/j.cor.2017.12.006.

[250] San Segundo, P., Lopez, A., Artieda, J., and Pardalos, P. M. A parallel maximum clique algorithm for large and massive sparse graphs. *Optimization Letters*, **11** (2017), 343. doi:10.1007/s11590-016-1019-3.

[251] San Segundo, P., Lopez, A., and Pardalos, P. M. A new exact maximum clique algorithm for large and massive sparse graphs. *Computers and Operations Research*, **66** (2016), 81.

[252] San Segundo, P., Matia, F., Rodriguez-Losada, D., and Hernando, M. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, **7** (2013), 467.

[253] San Segundo, P., Nikolaev, A., and Batsyn, M. Infra-chromatic bound for exact maximum clique search. *Computers and Operations Research*, **64** (2015), 293.

[254] San Segundo, P., Rodríguez-Losada, D., and Jiménez, A. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, **38** (2011), 571.

[255] San Segundo, P. and Tapia, C. Relaxed approximate coloring in exact maximum clique search. *Computers and Operations Research*, **44** (2014), 185.

[256] Sanei-Mehri, S.-V., Sariyüce, A., and Tirthapura, S. Butterfly counting in bipartite networks. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '18*, (2018), 2150. doi:10.1145/3219819.3220097.

[257] Sariyuce, A. E., Seshadhri, C., Pinar, A., and Umit V. Çatalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. pp. 927–937 (2015).

[258] Sarkar, A., Ren, Y., Elhesha, R., and Kahveci, T. A new algorithm for counting independent motifs in probabilistic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **PP** (2018), 1. doi:10.1109/TCBB.2018.2821666.

[259] Sariyüce, A. E. and Pinar, A. Peeling Bipartite Networks for Dense Subgraph Discovery. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining - WSDM '18*, **9** (2018), 504. doi:10.1145/3159652.3159678.

[260] Savage, J. E. *Models of computation*, vol. 136. Addison-Wesley Reading, MA (1998).

[261] Schank, T. *Algorithmic aspects of triangle-based network analysis*. Ph.D. thesis, Universität Karlsruhe (2007).

[262] SCHANK, T. AND WAGNER, D. Finding, counting and listing all triangles in large graphs, an experimental study. pp. 606–609. Springer Berlin Heidelberg (2005).

[263] SEGUNDO, P. S., NIKOLAEV, A., BATSYN, M., AND PARDALOS, P. M. Improved Infra-Chromatic Bound for Exact Maximum Clique Search. *INFORMATICA*, **27** (2016), 463.

[264] SEIDMAN, S. B. Network structure and minimum degree. *Social networks*, **5** (1983), 269.

[265] SEIDMAN, S. B. AND FOSTER, B. L. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, **6** (1978), 139.

[266] SESHADHRI, C. A simpler sublinear algorithm for approximating the triangle count. *arXiv preprint arXiv:1505.01927*, (2015).

[267] SESHADHRI, C., PINAR, A., AND KOLDA, T. G. Triadic Measures on Graphs: The Power of Wedge Sampling. (2012). doi:10.1137/1.9781611972832.2.

[268] SEVENICH, M., HONG, S., WELC, A., AND CHAFI, H. Fast In-Memory Triangle Listing for Large Real-World Graphs. *SNAKDD*, (2014), 2:1.

[269] SHAHAM, E., YU, H., AND LI, X.-L. On finding the maximum edge biclique in a bipartite graph: a subspace clustering approach. *Proceedings of the 2016 SIAM International Conference on Data Mining*, (2016), 315. doi:10.1137/1.9781611974348.36.

[270] SHAHINPOUR, S. AND BUTENKO, S. Algorithms for the maximum k-club problem in graphs. *Journal of Combinatorial Optimization*, **26** (2013), 520. doi:10.1007/s10878-012-9473-z.

[271] SHAHINPOUR, S., SHIRVANI, S., ERTEM, Z., AND BUTENKO, S. Scale reduction techniques for computing maximum induced bicliques. *Algorithms*, **10** (2017). doi:10.3390/a10040113.

[272] SHAHRIVARI, S. AND JALILI, S. Distributed discovery of frequent subgraphs of a network using MapReduce. *Computing*, **97** (2015), 1101. doi:10.1007/s00607-015-0446-9.

[273] SHAHRIVARI, S. AND JALILI, S. Fast parallel all-subgraph enumeration using multicore machines. *Scientific Programming*, **2015** (2015). doi:10.1155/2015/901321.

[274] SHAHRIVARI, S. AND JALILI, S. High-performance parallel frequent subgraph discovery. *Journal of Supercomputing*, **71** (2015), 2412. doi:10.1007/s11227-015-1391-2.

[275] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, (2014), 625. doi:10.1145/2588555.2588557.

[276] Shimizu, S., Yamaguchi, K., Saitoh, T., and Masuda, S. Fast maximum weight clique extraction algorithm: Optimal tables for branch-and-bound. *Discrete Applied Mathematics*, **223** (2017), 120. doi:10.1016/j.dam.2017.01.026.

[277] Shin, K., Eliassi-Rad, T., and Faloutsos, C. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, **54** (2018), 677. doi:10.1007/s10115-017-1077-6.

[278] Shun, J. and Tangwongsan, K. Multicore triangle computations without tuning. *Proceedings - International Conference on Data Engineering*, **2015-May** (2015), 149.

[279] Silvestri, F. Subgraph Enumeration in Massive Graphs. (2014), 1.

[280] Slota, G. M. and Madduri, K. Fast approximate subgraph counting and enumeration. pp. 210–219. IEEE (2013).

[281] Slota, G. M. and Madduri, K. Complex Network Analysis Using Parallel Approximate Motif Counting. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, (2014), 405. doi:10.1109/IPDPS.2014.50.

[282] Smith, S., Liu, X., Ahmed, N. K., Tom, A. S., Petrini, F., and Karypis, G. Truss decomposition on shared-memory parallel systems. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, (2017), 0. doi:10.1109/HPEC.2017.8091049.

[283] Soto, M., Rossi, A., and Sevaux, M. Three new upper bounds on the chromatic number. *Discrete Applied Mathematics*, **159** (2011), 2281.

[284] Spark, A. Apache spark: Lightning-fast cluster computing. *URL http://spark. apache. org*, (2016).

[285] Suri, S. and Vassilvitskii, S. Counting triangles and the curse of the last reducer. *Proceedings of the 20th international conference on World wide web (WWW'11)*, (2011), 607.

[286] Svendsen, M., Mukherjee, A. P., and Tirthapura, S. Mining maximal cliques from a large graph using MapReduce: Tackling highly uneven subproblem sizes. *Journal of Parallel and Distributed Computing*, **79-80** (2015), 104.

[287] Takazawa, Y. and Mizuno, S. On a reduction of the weighted induced bipartite subgraph problem to the weighted independent set problem. Tech. rep. (2018).

[288] Talukder, N. and Zaki, M. J. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery*, **30** (2016), 1024. doi:10.1007/s10618-016-0466-x.

[289] Thomas, S. and Nair, J. J. A survey on extracting frequent subgraphs. *2016 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2016*, (2016), 2290. doi:10.1109/ICACCI.2016.7732394.

[290] Toki, T. and Ozaki, T. Experimental evaluation of a GPU-based frequent subgraph miner using synthetic databases. *Proceedings - 2016 4th International Symposium on Computing and Networking, CANDAR 2016*, (2017), 504. doi:10.1109/CANDAR.2016.70.

[291] Tom, A. S., Sundaram, N., Ahmed, N. K., Smith, S., Eyerman, S., Kodiyath, M., Hur, I., Petrini, F., and Karypis, G. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. pp. 1–7 (2017).

[292] Tomita, E. Efficient algorithms for finding maximum and maximal cliques and their applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **10167 LNCS** (2017), 3. doi:10.1007/978-3-319-53925-6_1.

[293] Tomita, E. and Kameda, T. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, **37** (2007), 95.

[294] Tomita, E., Matsuzaki, S., Nagao, A., Ito, H., and Wakatsuki, M. A Much Faster Algorithm for Finding a Maximum Clique. *Journal of Information Processing*, **25** (2017), 667. doi:10.2197/ipsjjip.25.667.

[295] Tomita, E. and Seki, T. An efficient branch-and-bound algorithm for finding a maximum clique. *Discrete Mathematics and Theoretical Computer Science*, (2003), 278.

[296] Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., and Wakatsuki, M. A simple and faster branch-and-bound algorithm for finding a maximum clique. *WALCOM: Algorithms and Computation*, (2010), 191.

[297] Tomita, E., Tanaka, A., and Takahashi, H. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, **363** (2006), 28. doi:10.1016/j.tcs.2006.06.015.

[298] Tomita, E., Yoshida, K., Hatta, T., Nagao, A., Ito, H., and Wakatsuki, M. A much faster branch-and-bound algorithm for finding a maximum clique. *Frontiers in Algorithmics*, (2016), 215.

[299] Tsourakakis, C. The K-clique Densest Subgraph Problem. *Proceedings of the 24th International Conference on World Wide Web - WWW*, (2015), 1122.

[300] Tsourakakis, C., Bonchi, F., Gionis, A., Gullo, F., and Tsiarli, M. A. Denser than the Densest Subgraph : Extracting Optimal Quasi-Cliques with Quality Guarantees. *Kdd '13*, (2013), 104. doi:10.1145/2487575.2487645.

[301] Tsourakakis, C. E., Drineas, P., Michelakis, E., Koutis, I., and Faloutsos, C. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Netw. Analys. Mining*, **1** (2011), 75.

[302] Tsourakakis, C. E., Kang, U., Miller, G. L., and Faloutsos, C. DOULION: Counting Triangles in Massive Graphs with a Coin. *KDD '09: 15th International Conference on Knowledge Discovery and Data Mining*, (2009), 837.

[303] Turkoglu, D. and Turk, A. Edge-based wedge sampling to estimate triangle counts in very large graphs. *Proceedings - IEEE International Conference on Data Mining, ICDM*, **2017-Novem** (2017), 455.

[304] Uno, T. Implementation issues of clique enumeration algorithm. *Progress in Informatics*, (2012), 25. `doi:10.2201/NiiPi.2012.9.5`.

[305] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM*, **33** (1990), 103.

[306] Veremyev, A., Prokopyev, O. A., Butenko, S., and Pasiliao, E. L. Exact MIP-based approaches for finding maximum quasi-cliques and dense subgraphs. *Computational Optimization and Applications*, **64** (2016), 177. `doi:10.1007/s10589-015-9804-y`.

[307] Vo, B., Nguyen, D., and Nguyen, T. L. A parallel algorithm for frequent subgraph mining. *Advances in Intelligent Systems and Computing*, **358** (2015), 163. `doi:10.1007/978-3-319-17996-4_15`.

[308] Voegele, C., Lu, Y. S., Pai, S., and Pingali, K. Parallel triangle counting and k-truss identification using graph-centric methods. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, (2017), 0.

[309] Wang, J. and Cheng, J. Truss Decomposition in Massive Networks. Tech. rep. (2012).

[310] Wang, L., Wang, Y., Yang, C., and Owens, J. D. A comparative study on exact triangle counting algorithms on the gpu. *Proceedings of the ACM Workshop on High Performance Graph Processing*, (2016), 1.

[311] Wang, P., Lui, J. C. S., Ribeiro, B., Towsley, D., Zhao, J., and Guan, X. Efficiently Estimating Motif Statistics of Large Networks. **9** (2013).

[312] Wang, P., Qi, Y., Sun, Y., Zhang, X., Tao, J., and Guan, X.-H. Approximately Counting Triangles in Large Graph Streams Including Edge Duplicates with a Fixed Memory Usage. *Pvldb*, **10** (2017), 162.

[313] Wang, P., Tao, J., Zhao, J., and Guan, X. Moss: A Scalable Tool for Efficiently Sampling and Counting 4- and 5-Node Graphlets. **1** (2015), 1.

[314] WANG, P., ZHAO, J., ZHANG, X., LI, Z., CHENG, J., LUI, J. C., TOWSLEY, D., TAO, J., AND GUAN, X. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, **30** (2018), 73. `doi:10.1109/TKDE.2017.2756836`.

[315] WANG, W., GU, Y., WANG, Z., AND YU, G. Parallel triangle counting over large graphs. Tech. rep. (2013).

[316] WANG, Y., CAI, S., AND YIN, M. Two efficient local search algorithms for maximum weight clique problem. *AAAI Conference on Artificial Intelligence*, (2016), 805.

[317] WANG, Y., HAO, J. K., GLOVER, F., LÃ¼, Z., AND WU, Q. Solving the maximum vertex weight clique problem via binary quadratic programming. *Journal of Combinatorial Optimization*, **32** (2016), 531.

[318] WANG, Y., LÜ, Z., GLOVER, F., AND HAO, J.-K. Probabilistic grasp-tabu search algorithms for the ubqp problem. *Computers & Operations Research*, **40** (2013), 3100.

[319] WANG, Z., CHEN, Q., HOU, B., SUO, B., LI, Z., PAN, W., AND IVES, Z. G. Parallelizing maximal clique and k-plex enumeration over graph data. *Journal of Parallel and Distributed Computing*, **106** (2017), 79. `doi:10.1016/j.jpdc.2017.03.003`.

[320] WASA, K. AND UNO, T. Efficient enumeration of bipartite subgraphs in graphs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **10976 LNCS** (2018), 454. `doi:10.1007/978-3-319-94776-1_38`.

[321] WASSERMAN, S. AND FAUST, K. *Social network analysis: Methods and applications*, vol. 8. Cambridge university press (1994).

[322] WERNICKE, S. A faster algorithm for detecting network motifs. pp. 165–177. Springer (2005).

[323] WERNICKE, S. AND RASCHE, F. FANMOD: A tool for fast network motif detection. *Bioinformatics*, **22** (2006), 1152. `doi:10.1093/bioinformatics/btl038`.

[324] WILLIAMS, V. V., WANG, J., WILLIAMS, R., AND YU, H. Finding Four-Node Subgraphs in Triangle Time. *Soda*, **111** (2015), 1671. `doi:10.1137/1.9781611973730.111`.

[325] WOLF, M. M., DEVECI, M., BERRY, J. W., HAMMOND, S. D., AND RAJAMANICKAM, S. Fast linear algebra-based triangle counting with KokkosKernels. *IEEE High Performance Extreme Computing Conference, HPEC 2017*, (2017).

[326] WOTZLAW, A. On Solving the Maximum k -club Problem. (2014).

[327] Wu, B., Yi, K., and Li, Z. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering*, **28** (2016), 2013.

[328] Wu, Q. and Hao, J. K. A review on algorithms for maximum clique problems (2015).

[329] Xiao, D., Eltabakh, M., and Kong, X. Bermuda: An efficient mapreduce triangle listing algorithm for web-scale graphs. *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, (2016), 10.

[330] Xiao, M., Lin, W., Dai, Y., and Zeng, Y. A fast algorithm to compute maximum k-plexes in social network analysis. *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, (2017), 919.

[331] Xing, Y., Xiao, N., Lu, Y., Li, R., Yu, S., and Gao, S. Fast Truss Decomposition in Memory. (2017). `doi:10.1007/978-3-319-72395-2_64`.

[332] Xu, Y., Cheng, J., and Fu, A. W. C. Distributed Maximal Clique Computation and Management. *IEEE Transactions on Services Computing*, **9** (2016), 110.

[333] Yan, X. and Han, J. gspan: Graph-based substructure pattern mining. pp. 721–724. IEEE (2002).

[334] Yanshan, H., Ting, W., Jianli, X., and Ming, Z. Parallel frequent subgraph mining algorithm. *Proceedings of the 6th International Conference on Software and Computer Applications - ICSCA '17*, (2017), 198. `doi:10.1145/3056662.3056691`.

[335] Yu, T. and Liu, M. A linear time algorithm for maximal clique enumeration in large sparse graphs. *Information Processing Letters*, **125** (2017), 35.

[336] Yuan, B., Li, B., Chen, H., and Yao, X. A new evolutionary algorithm with structure mutation for the maximum balanced biclique problem. *IEEE Transactions on Cybernetics*, **45** (2015), 1040. `doi:10.1109/TCYB.2014.2343966`.

[337] Zhai, H., Haraguchi, M., Okubo, Y., and Tomita, E. Enumerating Maximal Clique Sets with Pseudo-Clique Constraint. (2015). `doi:10.1007/978-3-319-24282-8`.

[338] Zhai, H., Haraguchi, M., Okubo, Y., and Tomita, E. A fast and complete algorithm for enumerating pseudo-cliques in large graphs. *International Journal of Data Science and Analytics*, **2** (2016), 145. `doi:10.1007/s41060-016-0022-1`.

[339] Zhang, F., Zhang, Y., Qin, L., Zhang, W., and Lin, X. When Engagement Meets Similarity: Efficient (k,r)-Core Computation on Social Networks. (2016). `doi:10.14778/3115404.3115406`.

[340] Zhao, X., Chen, Y., Xiao, C., Ishikawa, Y., and Tang, J. Frequent Subgraph Mining Based on Pregel. *Computer Journal*, **59** (2016), 1113. doi:10.1093/comjnl/bxv118.

[341] Zheng, Z., Ye, F., Li, R. H., Ling, G., and Jin, T. Finding weighted k-truss communities in large networks. *Information Sciences*, **417** (2017), 344. doi:10.1016/j.ins.2017.07.012.

[342] Zhou, Y. and Hao, J.-K. Combining tabu search and graph reduction to solve the maximum balanced biclique problem. Tech. rep. (2017).

[343] Zhou, Y. and Hao, J. K. Frequency-driven tabu search for the maximum s-plex problem. *Computers and Operations Research*, **86** (2017), 65. doi:10.1016/j.cor.2017.05.005.

[344] Zhou, Y., Hao, J. K., and Goëffon, A. PUSH: A generalized operator for the Maximum Vertex Weight Clique Problem. *European Journal of Operational Research*, **257** (2017), 41. doi:10.1016/j.ejor.2016.07.056.

[345] Zhou, Y., Rossi, A., and Hao, J. K. Towards effective exact methods for the Maximum Balanced Biclique Problem in bipartite graphs. *European Journal of Operational Research*, **269** (2018), 834. doi:10.1016/j.ejor.2018.03.010.

[346] Zhu, Y., Zhang, H., Qin, L., and Cheng, H. Efficient MapReduce algorithms for triangle listing in billion-scale graphs. *Distributed and Parallel Databases*, **35** (2017), 149.

[347] Züge, A. P. and Carmo, R. On comparing algorithms for the maximum clique problem. *Discrete Applied Mathematics*, **247** (2018), 1. doi:10.1016/j.dam.2018.01.005.

Appendix A

	amazon	citPat	comYoutube	locGowalla	socPokec	webGoogle	wikiTalk	webStan	hTwitter	asSkitter
q_8	321 706	515 815	6 060 373	67 301 502	111 299 007	1 570 954 476	13 874 385 900	218 217 684 923	265 987 259 636	481 576 204 696
q_9	71 668	61 388	3 721 987	86 198 580	149 174 278	4 464 077 468	27 578 855 848	1 202 688 474 239	1 233 654 381 260	2 781 731 674 867
q_{10}	9 771	2 669	1 833 670	106 264 724	197 987 487	13 006 798 813	43 905 609 021	5 833 322 749 668	5 621 092 966 947	14 217 188 170 569
q_{11}	610	2	717 295	130 982 589	254 990 304	36 314 328 405	56 794 333 067	25 078 452 238 373	25 540 068 647 107	-
q_{12}	0	0	219 257	160 683 420	312 062 412	93 686 488 477	60 380 527 568	-	-	-
q_{13}	0	0	51 221	189 318 194	356 085 446	220 371 755 150	53 228 970 266	-	-	-
q_{14}	0	0	8 857	206 030 765	372 952 314	471 340 759 707	39 167 026 282	-	-	-
q_{15}	0	0	1 068	201 454 150	353 958 854	917 874 929 737	24 163 862 814	-	-	-
q_{16}	0	0	77	174 169 933	301 158 997	1 631 488 953 503	12 531 004 332	-	-	-
q_{17}	0	0	2	131 977 846	227 652 199	2 653 605 642 520	5 466 107 847	-	-	-
q_{18}	0	0	0	87 161 354	151 713 832	3 957 868 133 262	2 003 375 020	-	-	-
q_{19}	0	0	0	49 920 126	88 505 205	5 421 649 112 785	615 026 581	-	-	-
q_{20}	0	0	0	24 649 947	44 865 590	6 827 421 553 633	157 223 565	-	-	-

Number q_k of k-cliques on medium-size graphs for $k \in [8, 20]$.