Why is maximum clique often easy in practice?

Jose L. Walteros*
Department of Industrial and Systems Engineering
University at Buffalo
josewalt@buffalo.edu

Austin Buchanan
School of Industrial Engineering & Management
Oklahoma State University
buchanan@okstate.edu

Abstract

To this day, the maximum clique problem remains a computationally challenging problem. Indeed, despite researchers' best efforts, there exist unsolved benchmark instances with one thousand vertices. However, relatively simple algorithms solve real-life instances with millions of vertices in a few seconds. Why is this the case? Why is the problem apparently so easy in many naturally occurring networks? In this paper, we provide an explanation. First, we observe that the graph's clique number ω is very near to the graph's degeneracy d in most real-life instances. This observation motivates a main contribution of this paper, which is an algorithm for the maximum clique problem that runs in time polynomial in the size of the graph, but exponential in the gap $g := (d+1) - \omega$ between the clique number ω and its degeneracy-based upper bound $d+1$. When this gap g can be treated as a constant, as is often the case for real-life graphs, the proposed algorithm runs in time $O(dm) = O(m^{1.5})$. This provides a rigorous explanation for the apparent easiness of these instances despite the intractability of the problem in the worst case. Further, our implementation of the proposed algorithm is actually practical—competitive with the best approaches from the literature.

Keywordsmaximum clique; fixed-parameter tractable; fpt; vertex cover; degeneracy; k-core; parameterized below degeneracy; kernelization; parameterized complexity.

1 Introduction

To this day, the maximum clique problem remains a computationally challenging problem. Indeed, despite researchers' best efforts, there exist unsolved benchmark instances with $n = 1,000$ vertices. Examples from the 2nd DIMACS Implementation Challenge (DIMACS_2) are given in Table 1, none of which were solved in a 4-hour time limit in the expository paper of Prosser (2012). Another set of notoriously hard instances arise from error-correcting codes (Sloane 2017). For example, the instance 2dc.2048 took 300 CPU days (in 2015) to solve with a branch-and-bound algorithm, and the instance 1dc.1024 took 200 CPU days (in 2005) to solve using Lovász theta SDP bounds.

In contrast, relatively simple algorithms, that do not rely on sophisticated SDP bounds or supercomputers, often solve real-life instances with millions of vertices in seconds. Examples from the 10th DIMACS Implementation Challenge (DIMACS_10) and the Stanford Large Network Dataset Collection (SNAP) are given in Table 2. Here, we report the time (in seconds) to solve them using the

*Corresponding author. Phone: 716-645-8876; Fax: 716-645-3302; Address: 413 Bell Hall, Buffalo, NY 14260; Email: josewalt@buffalo.edu

Table 1: Instances from the 2nd DIMACS Implementation Challenge that are not solved in a 4-hour time limit, by the algorithms MCQ1, MCSa1, and MCSb1, as per Prosser (2012).

Graph	n	m	ω	$d+1$	g	Time
brock800_1	800	207,505	23	488	465	>14,400
brock800_2	800	208,166	24	487	463	>14,400
brock800_3	800	207,333	25	484	459	>14,400
hamming10-4	1,024	434,176	40	849	809	>14,400
johnson32-2-4	496	107,880	16	436	420	>14,400
keller5	776	225,990	27	561	534	>14,400
keller6	3,361	4,619,898	≥59	2,691	≤2,632	>14,400
MANN_a81	3,321	5,506,380	≥1,100	3,281	≤2,181	>14,400
p_hat700-3	700	183,010	≥62	427	≤365	>14,400
p_hat1000-3	1,000	371,746	≥68	610	≤542	>14,400
p_hat1500-2	1,500	568,960	≥65	505	≤440	>14,400
p_hat1500-3	1,500	847,244	≥94	930	≤836	>14,400

single-threaded version of our implementation. As we will see, these times are competitive with recent exact approaches, such as Rossi et al. (2015), Verma et al. (2015), Buchanan et al. (2014).

Table 2: Some real-life graphs from DIMACS_10 and SNAP. Solve times are reported for our implementation using 1 thread.

Graph	n	m	ω	$d+1$	g	Time
coAuthorsDBLP	299,067	977,676	115	115	0	0.04
web-NotreDame	325,729	1,090,108	155	156	1	0.07
coPapersCiteseer	434,102	16,036,720	845	845	0	0.17
coPapersDBLP	540,486	15,245,729	337	337	0	0.20
web-BerkStan	685,230	6,649,470	201	202	1	0.25
eu-2005	862,664	16,138,468	387	389	2	0.50
in-2004	1,382,908	13,591,473	489	489	0	0.47
wiki-Talk	2,394,385	4,659,565	26	132	106	36.92
uk-2002	18,520,486	261,787,258	944	944	0	15.72

Why? Why is it that such small instances, like the 700-vertex graph p_hat700-3 and the 1,024-vertex graph 1dc.2014, are left unsolved even after hours of computation, using carefully designed exact algorithms that exploit decades of theory? At the same time, why are we able to solve very large real-life graphs, like the 18,520,486-vertex graph uk-2002, in a matter of seconds?

To answer these questions, consider the instances in Table 2. Empirically, these large, real-life instances have a small *clique-core gap*, which we define as $g := (d+1) - \omega$. (Here, ω is the graph's clique number, and d is the graph's degeneracy, also known as the *k-core number*.) Indeed, the clique-core gap is *at most two* on all but one of these easy instances. Similarly, Rossi et al. (2015) plot the ratio $\omega/(d+1)$ for many real-life graphs and note that it is often close to 1, particularly for collaboration and web graphs. In contrast, the challenging instances given in Table 1 have large clique-core gaps—never less than 365.

Is this pattern more than just a coincidence? Must hard instances have a large clique-core gap? Put differently, are instances with small clique-core gap always easy to solve? In this paper, we answer these questions in the affirmative.

To prove this, we provide an algorithm for the maximum clique problem whose running time is bounded by $1.28^g n^{O(1)}$, where the polynomial function of the number n of vertices does not depend on the clique-core gap g. When the clique-core gap can be treated as a constant, as is often the case for real-life graphs, the proposed algorithm runs in time $O(dm) = O(m^{1.5})$, where m refers to the number of edges. Furthermore, the largest instance in Table 2, called uk-2002, falls into a class of instances that are solved by our algorithm in linear time. This provides a rigorous explanation for the apparent easiness of these instances despite the intractability of the maximum clique problem in the

worst case. As a bonus, our implementation is actually practical and competitive with state-of-the-art approaches for large, real-life graphs, like those of Verma et al. (2015) and Rossi et al. (2015), that run quickly in practice but lack worst-case time bounds. Our implementation is publicly available at: `https://github.com/jwalteros/dOmega`.

Our Contributions. In Section 2, we review previous work on the maximum clique problem and discuss concepts that will be employed in our approach, such as graph degeneracy, minimum degree orderings, and the parameterized complexity of the minimum vertex cover problem. In Section 3, we provide an algorithm that, when given a graph G and a nonnegative integer p, answers the question: "Does G have a clique of size $d + 1 - p$?" We analyze its running time when applied to this decision problem and also when used as a subroutine for solving the maximum clique problem. In Section 4, we present computational results over sixty instances arising from different real-world applications that are often used to test graph algorithms. Fifty of these sixty instances were also considered by Verma et al. (2015) to test their own approach for maximum clique. We show that fourteen of these instances can be solved by our approach in linear time, allowing us to exclude them in further tests. We also compare our running times with those of Verma et al. (2015), Rossi et al. (2015), and Buchanan et al. (2014) and demonstrate that our approach is practical and competitive. In Section 5, we provide further insights into why our approach can run relatively quickly even when g is moderately large. In Section 6, we conclude and offer insights and roadblocks for other possible parameterizations for the maximum clique problem and for the minimum vertex coloring problem.

2 Background

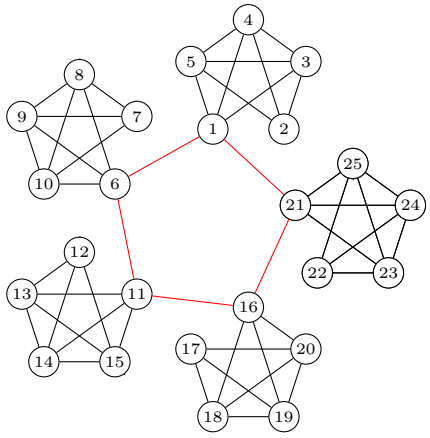
In this section, we set the stage by reviewing concepts like graph degeneracy and k-cores. We discuss how they have been used to solve the maximum clique problem, particularly when the input is a large, sparse graph. Then, we review topics from parameterized complexity used in our approach, including kernelization and fixed-parameter tractable (fpt) algorithms for vertex cover.

We consider a simple graph $G = (V, E)$ with vertex set V and edge set $E \subseteq \binom{V}{2}$. The number of vertices and edges of G are denoted by $n := |V|$ and $m := |E|$, respectively. The subgraph of G induced by the vertex subset $S \subseteq V$ is denoted by $G[S] := \left(S, \binom{S}{2} \cap E\right)$. The complement of G is denoted $\overline{G} = (V, \overline{E})$, where $\overline{E} = \binom{V}{2} \setminus E$. The neighborhood of a vertex $v \in V$ in G is denoted $N_G(v) := \{w \in V \mid \{v, w\} \in E\}$, and the degree of v is $\deg_G(v) := |N_G(v)|$. The minimum and maximum degrees of G are denoted by $\delta(G)$ and $\Delta(G)$, respectively. A graph G is r-regular if $r = \delta(G) = \Delta(G)$. A clique $C \subseteq V$ is a subset of pairwise adjacent vertices, i.e., every $\{i, j\} \in \binom{C}{2}$ belongs to E. The size of a largest clique in G is denoted $\omega(G)$. A clique of largest possible size is called a maximum clique. When the graph G in question is clear, the subscript or argument G is often dropped from the aforementioned notations. When k is a positive integer, we let $[k] := \{1, 2, \ldots, k\}$.

2.1 Caveman Graphs

Figure 1 provides a connected *caveman* graph, which will be used for illustrative purposes throughout this paper. The class of connected caveman graphs were introduced by Watts (1999) in the context of social network analysis. They are generated by taking a disjoint collection of cliques ("caves") and rewiring one edge within each clique to a nearby clique (e.g., replacing $\{6, 7\}$ by $\{6, 1\}$), creating a cycle. There are ten maximum cliques in this graph, including the set $\{2, 3, 4, 5\}$.

Figure 1: An example of a connected caveman graph.



2.2 Degeneracy, k-Cores, and MD Orderings

A common preprocessing procedure for the maximum clique problem is based on k-cores, see the peeling procedure of Abello et al. (1999). The idea is as follows. Suppose there is a known lower bound L on the clique number. Then, a vertex in a *maximum* clique will neighbor (at least) $L-1$ vertices in said clique, and so its degree in the original graph will also be at least $L-1$. Importantly, a vertex with degree less than $L-1$ cannot belong to a maximum clique and can be safely deleted from the instance. This deletion lowers the degrees of its neighbors, making them candidates for deletion, and so this degree check can be performed iteratively. At the end, what remains is the k-core of the graph, where $k = L-1$. The name k-core comes from Seidman (1983). For many real-life graphs, this preprocessing is especially helpful given the large numbers of low-degree vertices and the fact that it can be implemented to run in linear time (Matula & Beck 1983).

Definition 1 (k-core) *For an integer k, the k-core of a graph G is the maximum subgraph G' of G satisfying $\delta(G') \geq k$ (if one exists).*

As an example, the connected caveman graph depicted in Figure 1 is itself the graph's 3-core, since every vertex has degree at least 3. The graph itself is also the 1-core and the 2-core.

Similar ideas can also be used to find an *upper bound* on the clique number. Specifically, suppose that the graph has *no* k-core, then there can be no clique of $k+1$ vertices. For example, it can be observed that our caveman graph has no 4-core, and so it has no clique of 5 vertices, thus $\omega \leq 4$.

The largest integer k for which there exists a k-core is sometimes called the (highest) k-core number of the graph (Bader & Hogue 2003), which we will denote by d. The reason for using the notation d is that the k-core number is equivalent to the graph invariant called degeneracy, as defined by Lick & White (1970).[1]

Definition 2 (degeneracy) *A graph is said to be d-degenerate if every subgraph (with at least one vertex) has a vertex of degree at most d. The degeneracy of a graph is the smallest value of d such that it is d-degenerate.*

As we have just argued, a degeneracy-based upper bound on the clique number is $\omega \leq d+1$, and Table 2 given in the introduction illustrates that this bound is often very strong on real-life graphs.

[1] Also see the equivalent or nearly equivalent notions of coloring number (Erdös & Hajnal 1966) (not to be confused with chromatic number), the Szekeres-Wilf number (Szekeres & Wilf 1968), width (Freuder 1982), and linkage (Kirousis & Thilikos 1996).

Fortunately, computing this bound is an easy task. Indeed, an algorithm of Matula & Beck (1983) finds it in linear time by computing a vertex ordering satisfying Lemma 1.

Lemma 1 (Lick & White (1970)) *A graph is d-degenerate if and only if it admits a vertex ordering $(v_1, v_2, \ldots, v_n)$ in which each vertex v_i has at most d neighbors after it in the ordering, i.e., $|N(v_i) \cap \{v_i, v_{i+1}, \ldots, v_n\}| \leq d$, for every $i \in [n]$.*

The algorithm by Matula & Beck (1983) works by iteratively removing a vertex of minimum degree (in the remaining graph) and appending it to the ordering. The ordering that results is not only a degeneracy ordering, but also a *minimum degree ordering*, which is the name given by Nagamochi (2010). In our proposed algorithm, we employ a minimum degree ordering.[2]

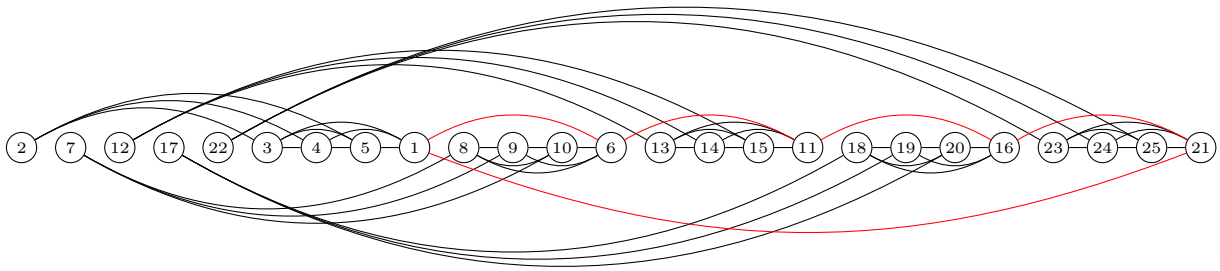
Definition 3 (MD ordering) *A minimum degree (MD) ordering of a graph $G = (V, E)$ is a vertex ordering $(v_1, v_2, \ldots, v_n)$ such that*

$$\deg_{G[S_i]}(v_i) = \delta(G[S_i])$$

for every $i \in [n]$, where $S_i := \{v_i, v_{i+1}, \ldots, v_n\}$.

As an example, Figure 2 gives an MD ordering of the connected caveman graph from Figure 1. Here, each vertex has at most 3 neighbors after it in the ordering, and so the graph is 3-degenerate (and 4-degenerate and 5-degenerate and so on), and the graph's degeneracy is $d = 3$. So, by our upper bound $\omega \leq d + 1$ we can argue that the clique number of our connected caveman graph is *at most* 4. Since the clique number is indeed 4, the clique-core gap $g := (d + 1) - \omega$ here is zero. It can be observed that *all* caveman graphs constructed by the procedure of Watts (1999) have $g = 0$.

Figure 2: An MD ordering of the connected caveman graph from Figure 1, which is obtained by iteratively removing a vertex of minimum degree and appending it to the ordering. Here, $d = 3$ and so $\omega \leq 4$.



The idea behind k-core preprocessing for the maximum clique problem is to remove low-degree *vertices* that cannot be part of an optimal solution. A similar procedure can be used to remove "low-degree" *edges*. Namely, when given a lower bound L on the clique number, one can safely and iteratively remove edges $\{i, j\}$ whose endpoints have fewer than $L - 2$ common neighbors. This leads to the essentially equivalent notions of k-truss (Cohen 2008), k-community (Verma et al. 2015), and

[2] MD orderings are closely related to degeneracy orderings, width orderings (Freuder 1982), and smallest-last orderings (Matula & Beck 1983), and they have been used in many clique algorithms, with explicit worst-case guarantees on their running time (Eppstein et al. 2013, Buchanan et al. 2014, Manoussakis 2014) and without guarantees (Carraghan & Pardalos 1990, Prosser 2012).

triangle-core (Rossi 2014).[3] By iteratively removing an edge $\{i, j\}$ whose endpoints have a *minimum* number $|N(i) \cap N(j)|$ of common neighbors, one can compute the k-truss of a graph, for any k, in time $O(m^{1.5})$, as first described by Wang & Cheng (2012). This gives rise to the notion of community degeneracy c, which is the largest value c for which there exists a subgraph in which the endpoints of each edge have at least c neighbors in common. The associated upper bound on the maximum clique number is $\omega \leq c + 2$. Conceivably, one could extend the approach that we propose in this paper to develop an algorithm whose running time is polynomial in the size of the graph, but exponential in the gap between ω and $c+2$. However, the polynomial part of its running time, $O(m^2)$, would become too much of a hindrance when attempting to solve very large instances (with hundreds of millions of edges), so we will not take that route in this paper.

As a final remark, the k-core and k-truss preprocessing procedures for solving the maximum clique problem can be thought of as quick and specialized variants of probing (Savelsbergh 1994).

2.3 Maximum Clique in Graphs with Millions of Vertices

In this section, we review previous approaches for solving the maximum clique problem, particularly for sparse graphs with millions of vertices. The interested reader is directed to the survey of Bomze et al. (1999) for more information about the maximum clique problem more generally and to the expository computer implementation paper of Prosser (2012).

Abello et al. (1999) were perhaps the first to try to solve the maximum clique problem in a graph with millions of vertices. They considered a multigraph (allowing parallel edges) with > 53 million vertices and > 170 million edges obtained from AT&T call data. In its largest (connected) component of 45 million vertices, they found a clique of size 30 using the Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic, but were unable to prove its optimality. A key step in their approach is the k-core procedure which they called "peeling." No running times were given, and the AT&T call data multigraph was never made publicly available.

Years later, Verma et al. (2015) and Rossi et al. (2015) independently worked on this problem and gave detailed computational experiments for publicly available instances.[4] Like Abello et al. (1999), both Verma et al. (2015) and Rossi et al. (2015) first use a heuristic to find an initial lower bound and then apply the k-core peeling procedure. At this point, Verma et al. (2015) may apply k-community peeling as well, and if necessary, send the remaining (small) graph to the maximum clique solver of Östergård (2002). In contrast, Rossi et al. (2015) implement their own branch-and-bound algorithm which, among other things, prunes the search tree based on colorings.

The approaches of Verma et al. (2015) and Rossi et al. (2015) work rather well given the sizes of the graphs. Indeed, the largest instance considered by Verma et al. (2015) has 18.5 million vertices and was solved in 2.5 minutes. Rossi et al. (2015) report solving an instance having 66 million vertices in 20 minutes. Despite their success for solving large instances, these two approaches lack nontrivial worst-case running times, but the easiness of their instances can be partially attributed to their sparsity. Indeed, Eppstein et al. (2013) show that all maximal cliques can be enumerated in time $O(d(n-d)3^{d/3})$, implying that a maximum clique can be found in the same time. Later, Buchanan et al. (2014) exploit the same idea to solve the maximum clique problem in time $O(nm+n2^{d/4})$ relying on a subroutine of Robson (2001).[5] With slight modifications, Manoussakis (2014) improves the time bound to $O((n-d)2^{d/4})$.

[3] A subgraph G' of G is called a k-community if the endpoints of every edge $e \in E(G')$ have at least k common neighbors in G'.

[4] Anurag Verma presented preliminary results as early as the INFORMS Annual Meeting in Charlotte, NC in November 2011. The resulting paper was submitted to a journal in July 2012, but was never posted to a preprint website. Ryan Rossi posted his group's preprint to arXiv in October 2012.

[5] Buchanan et al. (2014) also give an algorithm whose running time is polynomial in the number of vertices, but exponential in the "k-truss number" or "community degeneracy."

Most previous algorithms from the maximum clique literature are designed to solve small benchmark instances, like those from the 2nd DIMACS implementation challenge (DIMACS_2). Since these graphs tend to be small and dense, most approaches use an adjacency matrix to store the graph and thus require $\Omega(n^2)$ space. Indeed, this is true for challenge solvers like Cliquer (Östergård 2002), MCR (Tomita & Kameda 2007), MCS (Tomita et al. 2010), MAXCLQ (Li & Quan 2010), and BBMCX (San Segundo et al. 2015). Thus, they would quickly exceed available memory when applied to the large, real-life graphs coming from the 10th DIMACS implementation challenge or the SNAP database. Moreover, these challenge solvers do not bother to reduce the size of the instance through peeling because peeling is ineffective on DIMACS_2 instances. Another important difference is that challenge solvers tend to use sophisticated pruning strategies that can be quite time consuming, especially when applied to large, real-life graphs. For these reasons, challenge solvers are ill-suited for the real-life instances that we consider (Verma et al. 2015).

Another notable class of approaches is based on variants of the Lovász theta bound (Lovász 1979, Knuth 1994). The conventional approach for computing this bound is to solve a semidefinite program (SDP) with an $n \times n$ matrix variable X, as well as constraints $X(i,j) = 0$ for edges $\{i,j\} \in E$ and another constraint $\text{trace}(X) = 1$. Solving this SDP would require $\Omega(n^2 + m^2)$ space using a solver like CSDP (Borchers 1999, 2017). More recent developments, like those of Lieder et al. (2015), would still require $\Omega(n^2)$ space to store the dual variable. For these reasons, SDP-based bounding mechanisms have been limited to instances with a few thousand vertices (Wilson 2009, Lieder et al. 2015, Sloane 2017) and are ill-suited for the instances that we consider.

2.4 The Connection to Vertex Cover

The approach that we propose in this paper takes advantage of the well-known relationship between the maximum clique problem and the minimum vertex cover problem:

$$\omega(G) = \alpha(\overline{G}) = n - \tau(\overline{G}), \tag{1}$$

where α and τ denote the independence and vertex cover numbers, respectively. That is, to compute $\omega(G)$, one can simply subtract $\tau(\overline{G})$ from n.

However, it is not computationally viable to use this relationship directly. First, most very large, real-life graphs are incredibly sparse, meaning that $\overline{G}$ would be too large to store in computer memory. For example, the complement of the graph wiki-Talk has nearly 3 trillion edges. Second, even if one could construct and store $\overline{G}$, there is no reason to think that solving for $\tau(\overline{G})$ would be any easier than computing $\omega(G)$ directly.

Instead, we will: (1) solve a (parameterized) vertex cover problem, not on $\overline{G}$ directly, but on specially-constructed *small* graphs $\overline{G}[S]$; and (2) exploit the fact that the vertex cover problem is fixed-parameter tractable (fpt) with respect to solution size. Below, we review existing kernelization and fpt algorithms that will be used as subroutines in our approach.

2.4.1 Kernelization Algorithms for Vertex Cover

Kernelization is a key tool in our approach. In the context of parameterized complexity, *kernelization* refers to a certain type of preprocessing. When given an input (G,k) to a parameterized decision problem (with parameter k), it runs in polynomial time and returns another instance (G',k') such that

- (G,k) is a "yes" instance if and only if (G',k') is a "yes" instance;
- the size of the instance (G',k') is bounded by some computable function of k.

Note that the second condition ensures that the instance's size is bounded as a function of the parameter k *only*; it does not depend on the size of G.

Below, we discuss two popular kernels for vertex cover. For more on kernelization, we refer the reader to textbooks by Cygan et al. (2015) and Downey & Fellows (1999), and to the computational experiments—specifically for vertex cover—by Abu-Khzam et al. (2004).

Buss kernel. The kernelization given by Sam Buss (Buss & Goldsmith 1993) relies on the observation that vertices of sufficiently high degree must belong to every k-vertex cover. Namely, if a vertex $v \in V$ has degree greater than k, then v belongs to every k-vertex cover (otherwise, all of its $> k$ neighbors must be chosen to cover the edges incident to v). This leads to the Buss kernel (following the presentation of Balasubramanian et al. (1998)):

1. let $S := \{v \in V(G) \mid \deg_G(v) > k\}$;
2. if $|S| > k$, return "no"; otherwise let $k' := k - |S|$.
3. $G' \leftarrow G - S \cup I$, where I is the set of isolated vertices in $G - S$;
4. if $|E(G')| > kk'$, return "no";
5. else return the *kernel* (G', k').

The graph G' from the kernel has at most $2kk'$ vertices, since G' has at most kk' edges and no isolated vertices. The rule used in step 4 is safe, as follows. We are interested in k'-vertex covers of G', and each vertex of G' has degree at most k. Hence, these k' vertices can cover at most kk' edges. This entire kernelization can be implemented to run in time $O(kn)$ when G is represented via adjacency lists (Buss & Goldsmith 1993, Chen et al. 2001), making it very useful in practice (Akiba & Iwata 2016).

Nemhauser-Trotter kernel. Another notable kernel for vertex cover that we will use follows by results of Nemhauser & Trotter Jr (1975). As they observed, the natural linear programming (LP) relaxation for vertex cover gives half-integral (basic) solutions.

$$\min \sum_{v \in V} x_v \tag{2}$$

$$x_i + x_j \geq 1 \qquad \{i, j\} \in E \tag{3}$$

$$0 \leq x_v \leq 1 \qquad v \in V. \tag{4}$$

That is, there always exists an optimal solution $x^* \in \mathbb{R}^n$ to this LP such that $x^* \in \left\{0, \frac{1}{2}, 1\right\}^n$. Moreover, there is a minimum vertex cover S of G such that

$$V_1 \subseteq S \subseteq V_1 \cup V_{\frac{1}{2}}.$$

where $V_i = \{v \in V \mid x^*_v = i\}$ for $i \in \left\{0, \frac{1}{2}, 1\right\}$.

Chen et al. (2001) observed that this leads to the following kernelization procedure. If the optimal objective of the LP is greater than k, then obviously (G, k) is a "no" instance. Otherwise, the Nemhauser-Trotter kernel (G', k') has $G' = G - V_0 - V_1$ and $k' = k - |V_1|$. It can be seen that G' has at most $2k$ vertices, beating the $2k^2$ bound given by the Buss kernel.[6] We note that the Nemhauser-Trotter kernel need not use LP algorithms. Indeed, such an x^* can be found in time $O(m\sqrt{n})$ using the

[6] In some sense, both kernels are "optimal" in that there is no (polytime) kernel for vertex cover with $O(k^{2-\epsilon})$ edges, unless coNP⊆NP/poly (Dell & Van Melkebeek 2014). However, the Nemhauser-Trotter kernel is also believed to be optimal with respect to the number of vertices. Namely, Corollary 3.13 of Chen et al. (2007) states that if there exists a polytime algorithm for constructing a kernel *that is a subgraph of the original graph* with $(2 - \epsilon)k$ vertices, then P=NP.

Hopcroft-Karp algorithm (after a suitable transformation to an instance of bipartite matching (Bar-Yehuda & Even 1985)). By applying the Buss kernel *before* finding x^*, we get a runtime of $O(kn+k^3)$, as was observed by Chen et al. (2001). Further, Iwata et al. (2014) give a linear-time post-processing procedure that, when applied after the Hopcroft-Karp algorithm, finds an optimal LP solution x^* that maximizes the number of coordinates x_i^* that are integral. In our implementation, we will employ all of these kernelization techniques.

Pulleyblank (1979) gives characterizations of those graphs for which the all-half vector is the unique optimal LP solution, as well as a proof that this occurs for "almost all graphs." Indeed, more than 93% of graphs on 50 vertices admit the all-half vector as the unique optimal LP solution. For graphs on 100 vertices, it is more than 99.9999986%, meaning that the post-processing procedure described by Iwata et al. (2014) can be helpful only for relatively small graphs. Nevertheless, due to its low overhead and the frequent occurrence of small graphs in the subproblems of our approach, our implementation includes this procedure.

Other preprocessing rules. There are a number of other reduction rules for vertex cover (Chen et al. 2001, Downey & Fellows 1999, Cygan et al. 2015). For example, crown reduction gives a kernel of at most $3k$ vertices, cf. the recent development of Li & Zhu (2018) ensuring a $2k$ kernel based on crown decomposition. Some other reduction rules have no worst-case guarantees but can be helpful. For example, one can safely select the neighbor of a leaf vertex to be part of a minimum vertex cover. Or, consider a degree-two vertex v with neighbors u and w. If u neighbors w, one can safely select u and w but not v. If u does not neighbor w, the *vertex folding* operation can be applied to reduce the parameter k by one without branching (Chen et al. 2001), which is also employed in our implementation.

2.4.2 FPT Algorithms for Vertex Cover

An algorithm is said to be fixed-parameter tractable (fpt) if its runtime is bounded by $f(k)n^{O(1)}$, where n is the input size, k is the parameter of choice, and f is a function that depends only on k (Downey & Fellows 1999). A parameterized problem is fpt if it admits an fpt algorithm. The k-vertex cover problem is *the* classical fpt problem. For example, a simple bounded search tree algorithm solves k-vertex cover in time $2^k n^{O(1)}$ (Cygan et al. 2015). Simple arguments reduce the exponential term to 1.4656^k. We describe both of these approaches here. There is also a more complex approach due to Chen et al. (2010) that reduces the runtime to $O(1.2738^k + kn)$, but we do not describe it in detail here.

Bounded search tree. A simple fpt algorithm **vc1** for k-vertex cover is as follows, where the input is a graph G and an integer k. It operates on the fact that, in any vertex cover, a vertex v—or all of its neighbors $N(v)$—must belong to it.

vc1(G, k)

1. if $k < 0$, return "no";
2. if $|E(G)| = 0$, return "yes";
3. pick a vertex $v \in V(G)$ with $|N_G(v)| \geq 1$;
4. return **vc1**$(G - v, k - 1) \vee$ **vc1**$(G - N[v], k - |N_G(v)|)$;

Notice that in the last step, the parameter in the recursive calls has decreased by at least one. Thus, the depth of the recursion is (at most) $k + 1$. Since there are two recursive calls, the number of vertices in the search tree is fewer than 2^{k+2}. The time spent in steps 1, 2, and 3 is $O(n)$. The

subproblems in step 4 can be created in linear time. This gives a total runtime of $O(2^k(n+m))$. This can be reduced to $O(kn+2^k k^2)$ by applying the Buss kernel beforehand (Balasubramanian et al. 1998). Another improvement can be gained by *interleaving* kernelization within the search tree. For example, if we added the step:

$$\text{if } |E(G)| > ck^2, \text{ then replace } (G,k) \text{ by its Buss kernel}$$

just prior to step 3, this would reduce the time to $O(kn+2^k)$, where $c \geq 1$ is a user-specified constant (Niedermeier & Rossmanith 2000).

Branching on vertices of degree ≥ 3. To improve the runtime even further, we must tackle the base of the exponential term, which can be done through the following insight. If a graph G has maximum degree $\Delta(G)$ at most 2, then it is the disjoint union of cycle and path graphs. In this case, we can compute a minimum vertex cover in time $O(n)$ and return the appropriate yes/no response. Otherwise, we can branch on a vertex of degree *at least three*. This drastically reduces the search space.

vc2(G,k)

1. if $k<0$, return "no";
2. if $|E(G)| > ck^2$, replace (G,k) by its Buss kernel (possibly returning "no");
3. if $\Delta(G) \leq 2$, run the polynomial-time algorithm for vertex cover, and return appropriate yes/no;
4. pick a vertex $v \in V(G)$ with $|N_G(v)| \geq 3$;
5. return **vc2**$(G-v, k-1) \vee$ **vc2**$(G-N[v], k-|N_G(v)|)$;

This improves the running time to $O(kn+1.4656^k)$. The term 1.4656 comes from solving the recurrence $T(k) = T(k-1)+T(k-3)$. In practice, it is natural to branch on vertices of degree larger than three (say, on a vertex of maximum degree), since this reduces the parameter more quickly. Indeed, this has performed well in experiments (Akiba & Iwata 2016), so we will do this as well.

2.4.3 Limitations of FPT

While these fpt algorithms for vertex cover are important subroutines in our approach, we emphasize that it is *not* computationally viable to apply them to $\overline{G}$ directly. For example, consider the graph wiki-Talk, which has degeneracy $d = 131$. Suppose that we wanted to test whether this 2,394,385-vertex graph has a clique of size $d+1 = 132$. Using the relationship $\tau(\overline{G}) = n - \omega(G)$, this would be equivalent to testing whether the complement of wiki-Talk (which has nearly 3 trillion edges) has a vertex cover of size $k = n - 132 = 2,394,253$. The kernelization and fpt algorithms that we just reviewed would be no match for this instance.

3 A New Algorithm for Maximum Clique

In this section, we provide an algorithm **main** that, when given a graph G and a nonnegative integer p, determines whether G has a clique of size $d+1-p$. In other words, it determines whether the clique-core gap is at most p. In the pseudocode, we use the concepts of right-neighborhood and right-degree, which are defined below.

Definition 4 (right-degree, right-neighborhood) *If the vertices of a graph $G=(V,E)$ are ordered $(v_1, v_2, \ldots, v_n)$, then the right-neighborhood of a vertex v_i is defined as the set $N(v_i) \cap \{v_i, v_{i+1}, \ldots, v_n\}$ and the right-degree of v_i, denoted* $\text{rdeg}(v_i)$*, is the size of its right-neighborhood.*

To illustrate these definitions, consider Figure 2 in which the first vertex in the ordering is $v_1 = 2$, its right-degree is $\text{rdeg}(2) = 3$, and its right-neighborhood is $\{3, 4, 5\}$.

The pseudocode for our proposed algorithm is as follows.

main(G, p)

1. compute an MD ordering $(v_1, v_2, \ldots, v_n)$ and degeneracy d of G;
2. let $D = \{v_i \in V \mid i \leq n - d,\ \text{rdeg}(v_i) \geq d - p\}$;
3. for $v_i \in D$ do
   (a) construct $\overline{G}[V_i]$, where V_i is the right-neighborhood of v_i;
   (b) if $\overline{G}[V_i]$ has a vertex cover of size $q_i := |V_i| + p - d$, return "yes";
4. construct $\overline{G}[V_f]$, where $V_f = \{v_f, \ldots, v_n\}$ and $f := n - d + 1$;
5. if $\overline{G}[V_f]$ has a vertex cover of size $q_f := p - 1$, return "yes";
6. return "no."

3.1 Illustration of the Algorithm

To illustrate the algorithm, let G be the connected caveman graph given in Figure 1 and suppose that the task is to determine whether it has a clique of size $d + 1 - p$, where $p = 0$. The MD ordering to be computed in step 1 is not unique, but suppose that the one depicted in Figure 2 is identified. The set D defined in step 2 will be given by all vertices v_i whose right-degree is at least $d - p = 3 - 0 = 3$, which is the set

$$D = \{v_1, v_2, v_3, v_4, v_5, v_6, v_{10}, v_{14}, v_{18}, v_{22}\} = \{2, 7, 12, 17, 22, 3, 8, 13, 18, 23\}.$$

Then the for-loop in step 3 is reached, and entered with, say, $v_i = v_1 = 2$. In this case, $V_i = \{3, 4, 5\}$ and the graph $\overline{G}[V_i]$ is constructed. This graph is the edgeless graph on three vertices, and so it *does* have a vertex cover of size $q_i = |V_i| + p - d = 3 + 0 - 3 = 0$ (i.e., the empty set), and so the algorithm returns "yes." And, indeed the vertex cover $\emptyset$ of $\overline{G}[V_i]$ corresponds to the independent set $\{3, 4, 5\}$ of $\overline{G}[V_i]$, which corresponds to the clique $\{3, 4, 5\}$ of $G[V_i]$, which corresponds to the 4-vertex clique $\{2, 3, 4, 5\}$ of G. It can be observed that the algorithm runs in linear time for this graph. The same can be said for *all* caveman graphs constructed as per Watts (1999).

3.2 Analysis of the Algorithm

To analyze the algorithm's running time, we will use the following lemma, which is essentially due to Manoussakis (2016), although he considers the graphs $G[V_i]$ instead of their complements $\overline{G}[V_i]$.

Lemma 2 (Manoussakis (2016)) *The graphs $\overline{G}[V_1]$, $\overline{G}[V_2]$, ..., $\overline{G}[V_f]$ can be constructed in time and space $O((n - d + 1)d^2)$ using an adjacency list representation of G.*

Though not explicitly mentioned by Manoussakis (2016), his procedure can generate each graph $\overline{G}[V_i]$ on the fly in time $O(d^2)$, and the additional space requirement for our purposes is $O(d^2)$. Note that $d^2 = O(m)$ since $2m \geq d(d + 1)$.

Theorem 1 *The algorithm* **main** *correctly determines whether a graph G with degeneracy d has a clique of size $d + 1 - p$ in time $O((n - d)(1.28^p + d^2))$ and space $O(m + \text{poly}(d))$.*

Proof. First we analyze the running time and space. Steps 1 and 2 take $O(m+n)$ time and space by the MD ordering algorithm of Matula & Beck (1983), and each graph $\overline{G}[V_i]$ can be constructed in time $O(d^2)$, essentially by Manoussakis (2016). Finally, the remaining nontrivial steps are 3(b) and 5 in which we ask if $\overline{G}[V_i]$ has a vertex cover of size q_i. Denote by $T(n,k)$ the time to check whether an n-vertex graph has a vertex cover of size k. Then the total time is

$$O\left((m+n)+(|D|+1)d^2+\sum_{v_i\in D}T(|V_i|,q_i)\right). \tag{5}$$

Chen et al. (2010) show that $T(n,k)=O(1.2738^k+kn)$ using space polynomial in n. Since the MD ordering is also a degeneracy ordering, we have $|V_i|\le d$. Then, since $q_i := |V_i|+p-d \le p$, we have

$$T(|V_i|,q_i)=O(1.2738^{q_i}+q_i|V_i|)=O(1.2738^p+pd)$$

using space polynomial in d. Thus, since $|D|\le n-d$ and $1\le n-d$ and

$$m=\sum_{i=1}^{n-d}|V_i|+|E(G_f)|\le (n-d)d+\binom{d}{2}\le (n-d+1)d^2,$$

the running time (5) is bounded by

$$O\left((n-d+1)d^2+(n-d+1)(1.2738^p+pd)\right)=O\left((n-d)(d^2+1.2738^p)\right).$$

Finally we prove correctness. Suppose G has a clique $C\subseteq V$ of size $d+1-p$, and let v_i be its earliest vertex in the MD ordering. If $i\le n-d$, then the graph $G[V_i]$ has a clique $C\setminus\{v_i\}$ of size $d-p$, implying $\overline{G}[V_i]$ has an independent set of size $d-p$, meaning $\overline{G}[V_i]$ has a vertex cover of size $|V_i|+p-d=q_i$. Thus, the algorithm will return "yes." And, if $i>n-d$, then the graph $G[V_f]$ has a clique of size $d+1-p$ implying $\overline{G}[V_f]$ has an independent set of size $d+1-p$, meaning that $\overline{G}[V_f]$ has a vertex cover of size $|V_f|-(d+1-p)=p-1=q_f$, in which case the algorithm will return "yes."

Now, if the algorithm returns "yes" then it encountered a graph $\overline{G}[V_j]$ with a vertex cover C^* of size q_j. If $j\le n-d$, then $\{v_j\}\cup V_j\setminus C^*$ is a clique in G of size

$$1+|V_{j^*}|-|C^*|=1+(q_{j^*}+d-p)-q_{j^*}=d+1-p.$$

And, if $j^*>n-d$, then $V_f\setminus C^*$ is a clique in G of size $|V_f|-|C^*|=d-q_f=d+1-p$. □

Theorem 2 *When p is a constant, algorithm* **main** *runs in time $O(dm)=O(m^{1.5})$.*

Proof. Let $G'=(V',E')$ be the $(d-p)$-core of G. Denote by $n'=|V'|$ and $m'=|E'|$ the number of vertices and edges of G', respectively. By the sum of vertex degrees in G', we have $2m'\ge(d-p)n'$, so $dn'\le 2m'+pn'$. Note that $|D|\le n'$, so $|D|\le(2m'+pn')/d$. Thus, when p is a constant, the running time (5) of algorithm **main** is bounded by

$$\begin{aligned}
&O\left((m+n)+(|D|+1)d^2+\sum_{v_i\in D}T(|V_i|,q_i)\right)\\
&=O\left((m+n)+\left(\frac{2m'+pn'+d}{d}\right)d^2+|D|(1.2738^p+pd)\right)\\
&=O\left((m+n)+(2m'+pn'+d)d+n'(d)\right)\\
&=O(m+m'd+n'd)=O(dm)=O(m^{1.5}).
\end{aligned}$$

□

Remark 1 (The running time analysis is tight) *There exists an infinite class of graphs for which algorithm* **main** *runs in time* $\Omega(m^{1.5})$*, even when* $p = 0$*.*

The remark holds by the complete bipartite graphs $K_{d,2d}$ which have $n = 3d$ vertices, $m = 2d^2$ edges, and degeneracy d. In an MD ordering, at least d vertices from the larger partition will appear first, and each will have right-degree d. This implies that the set D will have at least d vertices. And, each of these first d subproblems's graphs $\overline{G}[V_1]$, ..., $\overline{G}[V_d]$ are complete, requiring the algorithm to create d edge sets of size $\binom{d}{2}$. This shows a running time of at least $|D|\binom{d}{2} \geq \Omega(d^3) = \Omega(m^{1.5})$.

Remark 2 (A linear-time special case) *If* $p = 0$ *and the* d*-core of* G *is* d*-regular, then* **main** *runs in linear time.*[7]

Proof. Denote by $G' = (V', E')$ the d-core of G with $n' := |V'|$ vertices and $m' := |E'|$ edges. Let $G'_1, \ldots, G'_k$ be the (connected) components of G'. For each component G'_i let $w_i \in V(G'_i)$ be its earliest vertex in the MD ordering. Observe that when w_i is removed from the graph, there will exist a vertex in its component G'_i having degree less than d until all vertices of G'_i have been placed in the MD ordering. As a result, w_i is the only vertex of G'_i that belongs to D. Thus, $D = \{w_i \mid i \in [k]\}$, so $|D| = k$. And, each G'_i has at least $d + 1$ vertices. So, $n' = \sum_{i=1}^{k} |V(G'_i)| \geq k(d + 1)$ implying that $|D| < n'/d$. Thus, by time bound (5), the running time in this situation is

$$
\begin{aligned}
&O\left((m+n) + (|D|+1)d^2 + \sum_{v_i \in D} T(|V_i|, q_i)\right) \\
&= O\left((m+n) + \left(\frac{n'+d}{d}\right) d^2 + |D|(1.2738^p + pd)\right) \\
&= O\left((m+n) + (n'+d)d + n'(d)\right) \\
&= O(m + n'd) = O(m).
\end{aligned}
$$

Here, the last equation follows by $n'd = 2m' \leq 2m$, which holds by the d-regularity of G'. □

We can solve the maximum clique problem in time $O(dm) = O(m^{1.5})$ when $(d+1) - \omega$ is a constant using the following approach. Start with $p = 0$. Apply the algorithm **main** from above. If it returns "yes," then return $\omega = d + 1 - p$. Else, increase p by 1 and repeat until getting a "yes." This proves the following corollary.

Corollary 1 *Let* $g := (d+1) - \omega$ *be the clique-core gap, i.e., the difference between the clique number* ω *and its degeneracy-based upper bound* $d + 1$*. We can compute* ω

1. *in time* $1.28^g \operatorname{poly}(n)$*;*
2. *in time* $O(dm) = O(m^{1.5})$ *when* g *is a constant;*
3. *in polynomial time when* $g = O(\log n)$*.*

As we have seen, many real-life graphs have small clique-core gaps, making Corollary 1 practically relevant. Is there a reasonable explanation for why these graphs have small g? To help answer this question, we can turn to generative graph models, such as those proposed by Barabási & Albert (1999) and Bonato et al. (2009). As noted by Eppstein et al. (2013), graphs produced under the preferential attachment model of Barabási & Albert (1999) have bounded degeneracy. Thus, these graphs also have bounded clique-core gap, and so Corollary 1 applies. In another example, consider the iterated

[7] In an alternative linear-time approach, first compute the d-core G' of G via Matula & Beck (1983) and suppose G' is d-regular. Then, $\omega = d + 1$ if and only if some connected component of G' contains exactly $d + 1$ vertices.

local transitivity (ITL) model of Bonato et al. (2009), which is motivated by properties of online social networks, specifically the idea that if $\{u, v\}$ are friends and $\{v, w\}$ are friends then $\{u, w\}$ are friends. The ITL model begins with an arbitrary initial graph G_0 and constructs a sequence of graphs $(G_t : t \geq 0)$ by "cloning" vertices. That is, G_{t+1} is constructed from G_t as follows. For every vertex v in G_t, a clone v' is added and connected to every vertex from $N_{G_t}(v) \cup \{v\}$. (Note that each iteration's clones form an independent set.) It can be observed that the clique number and degeneracy increase by one in each iteration, $\omega(G_{t+1}) = \omega(G_t) + 1$ and $d(G_{t+1}) = d(G_t) + 1$, so the clique-core gap does not change, $g(G_t) = g(G_0)$ for all $t \geq 1$. Thus, graphs constructed under the ILT model have increasing degeneracy but constant clique-core gap.

3.3 Algorithm Implementation

The interested reader can refer to Appendix A for more details about our implementation. This discussion covers the generation of the MD ordering, the construction of the subgraphs $\overline{G}[V_i]$, and the use of parallelization. While these implementation choices do not improve the worst-case bounds given in Section 3.2, they can have a discernible effect on the algorithm's practical performance.

4 Computational Experiments

The computational experiments reported in this section were conduced on a computing cluster running Linux x86_64, CentOS 7.2. Each node in the cluster was equipped with a 12-core Intel Xeon® E5-2620 v3 2.4GHz processor and 128 GB of RAM. Our algorithm was implemented in C++ and compiled with GCC 6.3.0. Each instance was solved by a single node of the cluster under a time limit of one hour.

We compare the results of our implementations with those obtained using the algorithms proposed by Buchanan et al. (2014), Rossi et al. (2015), and Verma et al. (2015), which we refer to as BWBP, RGG, and VBB, respectively.[8] These three algorithms were also coded in C++ and were compiled with GCC 6.3.0 according to the specifications provided by the authors. We note that the implementation of Rossi et al. (2015) was shown to dominate those of Csardi et al. (2006), Konc & Janežič (2007), Pattabiraman et al. (2013), so we exclude them from our experiments.

In addition to the aforementioned algorithms, there is also a recent series of papers (San Segundo et al. 2016, 2017,a) that report algorithms with fast computational times over some of the instances considered in this paper. In essence, these three papers describe minor variations of a "peeling" algorithm implemented in a bit-string encoding that, like in Rossi et al. (2015), combines an upper bound from vertex coloring with a branch-and-bound type of search. The computational experiments presented in these papers report solution times that are often significantly faster that those obtained with RGG and VBB. Our initial intention was to include their algorithms in our computational experiments as well. However, the website provided by the authors, which reportedly hosted their code, was down at the time this research was conducted.[9] We contacted the corresponding author who confirmed at the time of our inquiry that the code was indeed not available to share and provided instead the computer logs of their computational experiments. An examination of these logs reveals that the times reported in these papers do not account for the full execution of the algorithms. Instead, they only reflected the search time for a maximum clique over the subgraphs that resulted from the peeling process, omitting the time it takes to produce the k-core. Surprisingly, the times they report

[8] The algorithm in Buchanan et al. (2014) was implemented for this comparison using the clique solver by Östergård (2002) as a subroutine; the implementation of Rossi et al. (2015) was made available by the authors at `http://ryanrossi.com/pmc/` (last accessed: May 2019); and the implementation of Verma et al. (2015) was kindly provided by the authors.

[9] The website `https://www.biicode.com/pablodev/copt` that according to San Segundo et al. (2016) hosted the code was not available during the period from March 2017 to May 2019.

for the approach by Rossi et al. (2015)—the algorithm they compare their results against—include the preprocessing times.[10] It should be noted that the time to generate the k-core often represents a large portion of the overall running time for these algorithms and as such should not be ignored. For this reason *all* other approaches include these preprocessing times in their reports. Given the absence of an implementation to provide a fair comparison, we omit the algorithms from these papers from our computational experiments.

For the computational experiments, we use the fifty instances considered by Verma et al. (2015) and ten additional large instances. They were added to complement the test bed with extra instances from different domains with clique-core gaps ranging from small to large to better analyze the performance of our approach. All graphs were obtained from the Stanford Large Network Dataset Collection (SNAP. Last accessed: May 2018) and the 10th DIMACS Implementation Challenge (DIMACS_10. Last accessed: May 2018). These databases have a collection of large networks of sizes ranging from thousands to millions of vertices. They include social networks, web graphs, road networks, Internet networks, citation networks, collaboration networks, randomly generated graphs, and communication networks. The multitude of domains from which these networks originate, the very large sizes of them, and the fact that most real-life solvers commonly use them in their experiments make them suitable candidates for our computational experiments.

Different versions of graphs we used for the computational experiments are also available in other databases such as the Network Repository (Rossi & Ahmed 2015) (Last Accessed: May 2018). We notice, however, that the sizes of some of these graphs vary slightly from source to source. Therefore, some degeneracy values and maximum clique sizes reported in this paper may differ from those reported by Rossi et al. (2015), for they gathered the instances from the latter database. An example of these differences is the instance wiki-Talk, which is reported in this paper to have 2.394,385 vertices and 4.659,565 edges, whereas in Rossi et al. (2015) the graph is reported to have 92,117 vertices and 360,767 edges. Similar differences occur for the instances com-Orkut, soc-LiveJournal1, Soc-Epinions, and others.

For consistency, our experiments use the instances collected directly from both the SNAP and DIMACS_10 websites. Therefore, all algorithms were executed over the same computer architecture and with the same data sets as input. All directed graphs were converted to undirected graphs by replacing each directed edge with an undirected edge. Parallel edges and loops were removed.

Table 3 describes the graphs from the two datasets that were used for this study. The graphs were classified into nine categories based on their specific domain: (1) graphs from citation networks, (2) synthetic graphs created by different graph generators, (3) graphs derived from peer-to-peer Internet networks and graphs generated by crawling IP addresses, (4) portions of online social networks, (5) graphs generated by crawling websites, (6) graphs generated from sparse matrices collected from different domains, (7) road networks, (8) communication networks, and (9) graphs that associate products from the website of the e-commerce company Amazon based on the likelihood of being purchased in the same order. Table 3 also reports, for each graph, the number of vertices n, the number of edges m, the largest degree Δ, the degeneracy d, the size of a maximum clique ω, and the clique-core gap g.

The large graphs considered for the computational experiments typically consumed several gigabytes of hard drive space as text files. Since input/output operations with current hard drive technology are several orders of magnitude slower than most operations performed in RAM, reading the input files of real-life instances often takes much longer than solving for a maximum clique. For

[10] For instance, the computational log provided by the authors for solving the instance soc-pokec with two processors using the parallelized algorithm described in San Segundo et al. (2017a) reports a reading time of 10.68 seconds, a so-called setup time of 13.24 seconds that includes the calculation of the graph's k-core and the peeling process, and the search time over the reduced subgraph of 5.83 seconds. The authors report as the execution time the 5.83 seconds exclusively. This type of inconsistency was observed for all the instances they solved.

Table 3: Description of the graphs used in the computational experiments.

Type	Source	Name	n	m	Δ	d	ω	g
Citation	SNAP	Cit-HepTh	27,770	352,285	2,468	37	23	15
	SNAP	Cit-HepPh	34,546	420,877	846	30	19	12
	DIMACS_10	coAuthorsCiteseer	227,320	814,134	1,372	86	87	0
	DIMACS_10	citationCiteseer	268,495	1,156,647	1,318	15	13	3
	DIMACS_10	coAuthorsDBLP	299,067	977,676	336	114	115	0
	DIMACS_10	coPapersCiteseer	434,102	16,036,720	1,188	844	845	0
	DIMACS_10	coPapersDBLP	540,486	15,245,729	3,299	336	337	0
	DIMACS_10	er-fact1.5-scale20	1,048,576	10,904,496	45	14	3	12
	SNAP	cit-Patents	3,774,768	16,518,947	793	64	11	54
Synthetic	DIMACS_10	delaunay_n16	65,536	196,575	17	4	4	1
	DIMACS_10	kron_g500-simple-logn16	65,536	2,456,071	17,997	432	136	297
	DIMACS_10	preferentialAttachment	100,000	499,985	983	5	6	0
	DIMACS_10	G_n_pin_pout	100,000	501,198	25	7	4	4
	DIMACS_10	smallworld	100,000	499,998	17	7	6	2
	DIMACS_10	delaunay_n17	131,072	393,176	17	4	4	1
	DIMACS_10	delaunay_n18	262,144	786,396	21	4	4	1
	DIMACS_10	rgg_n_2_21_s0	2,097,152	14,487,995	37	18	19	0
	DIMACS_10	rgg_n_2_22_s0	4,194,304	30,359,198	36	19	20	0
	DIMACS_10	rgg_n_2_23_s0	8,388,608	63,501,393	40	20	21	0
	DIMACS_10	rgg_n_2_24_s0	16,777,216	89,345,197	40	20	21	0
Internet topology and peer-to-peer	SNAP	p2p-Gnutella04	10,876	39,994	103	7	4	4
	SNAP	p2p-Gnutella25	22,687	54,705	66	5	4	2
	DIMACS_10	as-22july06	22,963	48,436	2,390	25	17	9
	SNAP	p2p-Gnutella24	26,518	65,369	355	5	4	2
	SNAP	p2p-Gnutella30	36,682	88,328	55	7	4	4
	SNAP	p2p-Gnutella31	62,586	147,892	95	6	4	3
	DIMACS_10	caidaRouterLevel	192,244	609,066	1,071	32	17	16
	SNAP	as-skitter	1,696,415	11,095,298	35,455	111	67	45
Social	SNAP	wiki-Vote	7,115	100,762	1,065	53	17	37
	SNAP	soc-Epinions1	75,879	405,740	3,044	67	23	45
	SNAP	soc-Slashdot0811	77,360	469,180	2,539	54	26	29
	SNAP	soc-Slashdot0922	82,168	504,230	2,552	55	27	29
	SNAP	soc-pokec	1,632,803	22,301,964	14,854	47	29	19
	SNAP	soc-LiveJournal1	4,847,571	42,851,237	20,333	372	321	52
Web	SNAP	web-Stanford	281,903	1,992,636	38,625	71	61	11
	DIMACS_10	cnr-2000	325,557	2,738,969	18,236	83	84	0
	SNAP	web-NotreDame	325,729	1,090,108	10,721	155	155	1
	SNAP	web-BerkStan	685,230	6,649,470	84,230	201	201	1
	DIMACS_10	eu-2005	862,664	16,138,468	68,963	388	387	2
	SNAP	web-Google	875,713	4,322,051	6,332	44	44	1
	DIMACS_10	in-2004	1,382,908	13,591,473	21,869	488	489	0
	SNAP	wiki-topcats	1,791,489	25,444,207	238,342	99	39	61
	DIMACS_10	uk-2002	18,520,486	261,787,258	194,955	943	944	0
Sparse matrices	DIMACS_10	wave	156,317	1,059,331	44	8	6	3
	DIMACS_10	audikw1	943,695	38,354,076	344	47	36	12
	DIMACS_10	ldoor	952,203	22,785,136	76	34	21	14
	DIMACS_10	ecology1	1,000,000	1,998,000	4	2	2	1
	DIMACS_10	333SP	3,712,815	11,108,633	28	4	4	1
	DIMACS_10	cage15	5,154,859	47,022,346	46	25	6	20
Road	DIMACS_10	luxembourg.osm	114,599	119,666	6	2	3	0
	DIMACS_10	belgium.osm	1,441,295	1,549,970	10	3	3	1
Communication	SNAP	email-Enron	36,692	183,831	1,383	43	20	24
	DIMACS_10	cond-mat-2005	40,421	175,691	278	29	30	0
	SNAP	email-EuAll	265,214	364,481	7,636	37	16	22
	SNAP	wiki-Talk	2,394,385	4,659,565	100,029	131	26	106
	SNAP	com-Orkut	3,072,441	117,185,083	33,313	253	51	203
Product co-purchasing	SNAP	Amazon0302	262,111	899,792	420	6	7	0
	SNAP	Amazon0312	400,727	2,349,869	2,747	10	11	0
	SNAP	Amazon0601	403,394	2,443,408	2,752	10	11	0
	SNAP	Amazon0505	410,236	2,439,437	2,760	10	11	0

this reason, we do not report the reading times and start the clock after the algorithm loads the input graph into memory. This is a common practice (Verma et al. 2015, Rossi et al. 2015).

To reduce the variability of the running times caused by the hardware, we solve all instances five times and report the average. The variability between runs is generally small and has no significant impact on the overall running times. On average, it represents less than 3% of the total running time for all the runs and less than 1% for the instances whose overall execution lasts more than one second. However, using the average times provides for a better comparison between algorithms, particularly for instances that are solved rather quickly and when the running times of the algorithms differ only so slightly.

4.1 Instances Solved in Linear Time

We observe that fourteen of the sixty instances in our testbed are solved in linear time. We report the times in seconds to solve these instances in Table 4. We consider these fourteen instances to be too easy and exclude them in subsequent experiments as they would yield little insight.

Eight of these instances have d-regular d-cores and satisfy the property $d+1=\omega$. Thus, they are solved in linear time by **main** (see Remark 2). Interestingly, uk-2002, which is the largest graph in our testbed with more than 18 million vertices and 260 million edges, is one of them. Four of the nine citation networks share this property too.

Six other instances are solved "for free" by the MD ordering, because it provides matching lower and upper bounds on ω. We have already seen the upper bound $\omega \leq d+1$. Meanwhile, for the lower bound L, observe that if the right-degree of a vertex v_i is equal to the number of vertices to its right in the ordering $(v_1, v_2, \ldots, v_n)$, i.e., if $\text{rdeg}(v_i) = |\{v_{i+1}, \ldots, v_n\}|$, then $\{v_i, \ldots, v_n\}$ is a clique of size $n-i+1$, as v_i has the smallest degree in subgraph $G[\{v_i, \ldots, v_n\}]$. The largest clique found using this idea comes by picking the smallest i satisfying this property. If this yields a clique of size $d+1$, then $\omega = d+1$ and **main** can halt as soon as the MD ordering algorithm finishes.

Table 4: Instances solved in linear time by the proposed algorithm.

Property	Name	n	m	d	ω	g	Time
d-regular d-core	cond-mat-2005	40,421	175,691	29	30	0	0.00
	coAuthorsCiteseer	227,320	814,134	86	87	0	0.03
	coAuthorsDBLP	299,067	977,676	114	115	0	0.04
	coPapersCiteseer	434,102	16,036,720	844	845	0	0.17
	coPapersDBLP	540,486	15,245,729	336	337	0	0.20
	rgg_n_2_21_s0	2,097,152	14,487,995	18	19	0	0.55
	rgg_n_2_22_s0	4,194,304	30,359,198	19	20	0	1.33
	uk-2002	18,520,486	261,787,258	943	944	0	15.72
$L = d+1$	preferentialAttachment	100,000	499,985	5	6	0	0.03
	cnr-2000	325,557	2,738,969	488	489	0	0.08
	Amazon0312	400,727	2,349,869	10	11	0	0.17
	in-2004	1,382,908	13,591,473	5	6	0	0.47
	rgg_n_2_23_s0	8,388,608	63,501,393	20	21	0	6.13
	rgg_n_2_24_s0	16,777,216	89,345,197	20	21	0	9.30

4.2 Results for the New Algorithm

In this section, we discuss the performance of our approach on the remaining 46 instances. We begin by presenting the results of our algorithm when executed over a single thread. Table 5 presents the maximum clique size ω, the clique-core gap g, the lower bound L produced by the MD ordering, and the time in seconds taken by our algorithm. We also report the time spent computing the MD ordering (step 1 in **main**), generating the subgraphs $\overline{G}[v_i]$ (steps 3(a) and 4 in **main**), and solving the corresponding vertex cover subproblems (steps 3(b) and 5 in **main**).

Table 5: Computational results for the remaining 46 instances. We highlight in bold the instances for which the proposed approach takes more than five seconds to solve.

							Time			
Instance	n	m	L	d	ω	g	MD ordering	Subgraph generation	V.C.	Total
Cit-HepPh	34,546	420,877	18	30	19	12	0.01	0.15	0.01	0.18
Cit-HepTh	27,770	352,285	20	37	23	15	0.01	0.17	0.01	0.20
citationCiteseer	268,495	1,156,647	10	15	13	3	0.09	0.02	0.00	0.11
er-fact1.5-scale20	**1,048,576**	**10,904,496**	**2**	**14**	**3**	**12**	**1.64**	**5.90**	**1.26**	**9.12**
cit-Patents	3,774,768	16,518,947	10	64	11	54	2.25	0.86	1.32	4.52
delaunay_n16	65,536	196,575	3	4	4	1	0.01	0.01	0.02	0.05
kron_g500-simple-logn16	**65,536**	**2,456,071**	**135**	**432**	**136**	**297**	**0.04**	**7.17**	**2,651.29**	**2,663.15**
G_n_pin_pout	100,000	501,198	3	7	4	4	0.03	0.18	0.04	0.27
smallworld	100,000	499,998	5	7	6	2	0.03	0.06	0.00	0.10
delaunay_n17	131,072	393,176	3	4	4	1	0.02	0.02	0.04	0.11
delaunay_n18	262,144	786,396	3	4	4	1	0.05	0.04	0.09	0.23
p2p-Gnutella04	10,876	39,994	2	7	4	4	0.00	0.01	0.00	0.01
p2p-Gnutella25	22,687	54,705	2	5	4	2	0.00	0.01	0.00	0.01
as-22july06	22,963	48,436	14	25	17	9	0.00	0.00	0.00	0.01
p2p-Gnutella24	26,518	65,369	2	5	4	2	0.00	0.01	0.00	0.01
p2p-Gnutella30	36,682	88,328	2	7	4	4	0.00	0.01	0.00	0.02
p2p-Gnutella31	62,586	147,892	2	6	4	3	0.01	0.02	0.00	0.04
caidaRouterLevel	192,244	609,066	6	32	17	16	0.04	0.04	0.00	0.09
as-skitter	1,696,415	11,095,298	57	111	67	45	0.94	0.61	0.10	1.68
Wiki-Vote	7,115	100,762	16	53	17	37	0.00	0.10	0.15	0.27
soc-Epinions1	75,879	405,740	21	67	23	45	0.01	0.29	0.26	0.60
soc-Slashdot0811	77,360	469,180	25	54	26	29	0.02	0.21	0.05	0.29
soc-Slashdot0922	82,168	504,230	26	55	27	29	0.02	0.21	0.05	0.30
soc-pokec	**1,632,803**	**22,301,964**	**15**	**47**	**29**	**19**	**2.52**	**10.04**	**0.22**	**12.93**
soc-LiveJournal1	4,847,571	42,851,237	320	372	321	52	4.39	0.25	0.23	4.95
web-Stanford	281,903	1,992,636	13	71	61	11	0.08	0.07	0.00	0.15
web-NotreDame	325,729	1,090,108	154	155	155	1	0.07	0.00	0.00	0.07
web-BerkStan	685,230	6,649,470	201	201	201	1	0.25	0.00	0.00	0.25
eu-2005	862,664	16,138,468	387	388	387	2	0.47	0.03	0.00	0.50
web-Google	875,713	4,322,051	44	44	44	1	0.33	0.00	0.00	0.33
wiki-topcats	**1,791,489**	**25,444,207**	**5**	**99**	**39**	**61**	**2.54**	**9.58**	**0.28**	**12.58**
wave	156,317	1,059,331	5	8	6	3	0.04	0.23	0.04	0.33
audikw1	**943,695**	**38,354,076**	**30**	**47**	**36**	**12**	**0.72**	**31.84**	**1.14**	**34.30**
ldoor	**952,203**	**22,785,136**	**21**	**34**	**21**	**14**	**0.42**	**8.94**	**7.19**	**19.04**
ecology1	1,000,000	1,998,000	2	2	2	1	0.15	0.23	0.09	0.55
333SP	3,712,815	11,108,633	3	4	4	1	1.10	0.14	0.15	1.49
cage15	**5,154,859**	**47,022,346**	**5**	**25**	**6**	**20**	**2.58**	**24.34**	**3.71**	**32.77**
luxembourg.osm	114,599	119,666	2	2	3	0	0.02	0.00	0.00	0.02
belgium.osm	1,441,295	1,549,970	3	3	3	1	0.28	0.00	0.00	0.29
Email-Enron	36,692	183,831	17	43	20	24	0.01	0.07	0.03	0.12
Email-EuAll	265,214	364,481	14	37	16	22	0.02	0.04	0.02	0.09
Wiki-Talk	**2,394,385**	**4,659,565**	**25**	**131**	**26**	**106**	**0.29**	**1.34**	**34.82**	**36.92**
com-Orkut	**3,072,441**	**117,185,083**	**14**	**253**	**51**	**203**	**13.10**	**150.95**	**132.91**	**302.99**
Amazon0302	262,111	899,792	5	6	7	0	0.09	0.00	0.00	0.09
Amazon0505	410,236	2,439,437	6	10	11	0	0.19	0.00	0.00	0.19
Amazon0601	403,394	2,443,408	8	10	11	0	0.19	0.00	0.00	0.19

A glance at Table 5 shows that our approach is able to solve most instances quite rapidly, with 33 out of the 46 instances solved in less than one second and five of the remaining 13 instances taking between one and ten seconds. All but two graphs are solved in less than 40 seconds. The nine instances for which our approach took more than five seconds are highlighted in bold, as we will focus our discussion on them.

There are several observations that can be drawn from these results. First, as one might expect, the size of the instance is an important factor to consider when it comes to the difficulty of finding maximum cliques. With two notable exceptions (kron_g500-simple-logn16 and Wiki-Talk), graphs having fewer than ten million edges are solved by our approach in less than one second, whereas larger instances naturally take longer.

Second, as discussed in previous sections, it is clear that the clique-core gap plays a significant role in the efficacy of our algorithm. As evidenced by the times presented in Table 5, all instances but one that have a clique-core gap less than ten are solved within one second. Moreover, the synthetic instance kron_g500-simple-logn16, which has the largest clique-core gap $g = 297$ is by far the instance that takes the longest to solve despite having significantly fewer vertices than other instances. Also, the instances that take the second and third most time to solve are communication networks com-Orkut and Wiki-Talk, which have the second and third largest clique-core gaps of 203 and 106, respectively. Interestingly, as will be shown in Section 4.3, the instances for which the clique-core gap exceeds 200 (kron_g500-simple-logn16 and com-Orkut) are challenging, not only for our algorithm, but also for other approaches with which we compare. Another observation is that it is quite common for the graphs to have a small clique-core gaps compared to their degeneracy d. On average, this difference is roughly 26, which is about 39% the value of d. These averages fall to 19 and 30% if all 60 instances are considered.

Third, the results in Table 5 also highlight the fact that the worst-case analysis of our algorithm can be pessimistic in the sense that instances with moderate clique-core gaps are sometimes easy to solve. Indeed, there are several cases, including citation network cit-Patents, peer-to-peer network as-skitter, and social networks soc-Epinions1 and soc-LiveJournal1, that have moderate clique-core gaps (at least 45), but our algorithm is able to solve them quite rapidly (less than 5 seconds). In contrast, other instances with significantly smaller clique-core gaps (20 or less), like social network soc-pokec and graphs from sparse matrices audikw, ldoor and cage15, take significantly longer to be solved. We provide explanations for this behavior in Section 5. In particular, we will see that the times are also highly correlated with the number of subgraphs that **main** must process, as well as the difficulty of the resulting vertex cover subproblems.

Fourth, the times presented in Table 5 show that, for the majority of instances, the algorithm's bottleneck is generating the subgraphs. Indeed, for the instances which algorithm **main** takes more than one second to solve, generating the subgraphs takes on average 43% of the total time, whereas solving the resulting vertex cover subproblems takes on average only 27%. Four notable examples are the instances soc-pokec, wiki-topcats, audikw1, and cage15, for which the subgraph generation takes more than 74% of the total time. As one may expect, the results also show that for the three instances with a clique-core gap greater than 100, the algorithm spends a considerable amount of time solving the vertex cover subproblems. The two most extreme cases are kron_g500-simple-logn16 and wiki-Talk in which **main** spends more than 94% of its time in the vertex cover subroutine. The case of graph com-Orkut is also noteworthy as **main** spends similar amounts of time generating the subgraphs and solving vertex cover subproblems. Additionally, among the 46 instances, com-Orkut is also the one for which generating the MD ordering takes the longest, which makes sense given that it is one of the largest graphs in our test bed, having more than three million vertices and 100 million edges.

4.3 Performance comparison with other approaches

In this section, we compare our running times with those of BWBP, VBB, and RGG. Table 6 presents the times in seconds of the four approaches. We highlight in bold the fastest times to solve each instance and mark with an asterisk (*) the cases where an algorithm fails to identify ω within the time limit. This only occurs a few times for algorithms BWBP and VBB when solving three of the largest instances in our test bed. We observed that the exact solver by Östergård (2002), which is used by both BWBP and VBB as a subroutine, appears to have difficulties solving maximum clique in some of the subgraphs produced by these two algorithms.

The results suggest that all four algorithms are fairly competitive across all instances. Our approach seems to outperform the others when the clique-core gap is relatively small, but tends to take longer for instances with large clique-core gaps, as one might expect. Interestingly, with the exception of communications network wiki-Talk (which is solved by BWBP, VBB, and RGG rather quickly), each instance for which our algorithm takes more than five seconds to solve is also challenging for the other algorithms. For these nine instances, the two algorithms that use a peeling procedure as the key component (VBB and RGG) seem to perform slightly better than the other two. Furthermore, for the two instances that have clique-core gaps exceeding 200, i.e., kron_g500-simple-logn16 and com-Orkut—which we recall are the graphs our algorithm takes the longest to solve—RGG is clearly the fastest of the four algorithms.

The case of the synthetic graph kron_g500-simple-logn16 is worth highlighting, as it is an instance where all algorithms (except for RGG) struggle. Given that BWBP and **main** can take time exponential in d and g, respectively, and because this instance has both a large degeneracy and a large clique-core gap, it is understandable why they struggle here. The strong performance exhibited by RGG for this instance appears to be a result of its aggressive pruning strategies.

To summarize the performance of the algorithms, we report the *shifted geometric mean* (SGM), both for the test set as a whole (all 60 instances), and also for insightful subgroups (e.g., real-life graphs). The use of the SGM to compare algorithms originates with Achterberg (2007) and has since been adopted by many others, e.g., Mittelmann (2018). Given solution times t_1, t_2, ..., t_k of a given algorithm and a time shift $s > 0$, the shifted geometric mean is defined as

$$\gamma_s(t_1, \ldots, t_k) = \left(\prod_{i=1}^{k} \max\{t_i + s, 1\} \right)^{\frac{1}{k}} - s.$$

Since most instances are solved by all algorithms in less than one second, we set the time shift s as one second. The times taken by BWBP, VBB, and RGG on the linear-time solvable instances were not reported in our previous tables, but they are needed for our SGM calculations, so we provide them in Appendix B. On some instances, algorithms BWBP and VBB are unable to finish in a time limit of one hour, in which case 3,600 seconds is used in the SGM calculations.

We further classify the 60 graphs into four different groups based on their clique-core gap and calculate the shifted geometric mean for each algorithm on each of these groups. We also compare the solution times for the 49 instances that arise from real-life applications (i.e., excluding the 11 synthetic graphs). Table 7 presents both the unscaled and scaled shifted geometric means of the solution times. The scaled values are calculated with respect to the best performance in each group, so that 1.00 indicates the fastest approach and larger values indicate slower performance.

The results from Table 7 indicate that algorithms **main** and RGG are undoubtedly the top performers on this test bed, obtaining similar average times on all the 60 instances and improving upon the averages of BWBP and VBB by about 50% and 250%, respectively. Algorithm VBB seems to be the one with the worst average; however, the large difference in performance can be partially explained by the fact that VBB failed to solve two instances that the top performers solve rather quickly.

Table 6: Comparison of algorithms **main**, BWBP, VBB, and RGG. For each instance, the best times are highlighted in bold. The asterisk (*) indicates an inability to find ω within the time limit of one hour.

						Time			
Instance	n	m	d	ω	g	**main**	BWBP	VBB	RGG
Cit-HepTh	34,546	420,877	30	19	12	0.18	0.34	**0.12**	0.19
Cit-HepPh	27,770	352,285	37	23	15	0.20	0.37	**0.12**	0.20
citationCiteseer	268,495	1,156,647	15	13	3	**0.11**	0.14	0.53	0.26
er-fact1.5-scale20	1,048,576	10,904,496	14	3	12	9.12	13.64	17.15	**6.87**
cit-Patents	3,774,768	16,518,947	64	11	54	**4.52**	5.29	12.79	5.89
delaunay_n16	65,536	196,575	4	4	1	**0.05**	0.08	0.22	0.09
kron_g500-simple-logn16	65,536	2,456,071	432	136	297	2,663.15	815.16	335.94	**13.31**
G_n_pin_pout	100,000	501,198	7	4	4	0.27	0.50	0.42	**0.23**
smallworld	100,000	499,998	7	6	2	**0.10**	0.27	0.41	0.22
delaunay_n17	131,072	393,176	4	4	1	**0.11**	0.16	0.22	0.18
delaunay_n18	262,144	786,396	4	4	1	**0.23**	0.33	0.45	0.36
p2p-Gnutella04	10,876	39,994	7	4	4	**0.01**	0.03	**0.01**	0.02
p2p-Gnutella25	22,687	54,705	5	4	2	**0.01**	0.03	0.44	0.02
as-22july06	22,963	48,436	25	17	9	**0.01**	**0.01**	**0.01**	**0.01**
p2p-Gnutella24	26,518	65,369	5	4	2	**0.01**	0.04	0.69	0.02
p2p-Gnutella30	36,682	88,328	7	4	4	**0.02**	0.06	0.12	0.03
p2p-Gnutella31	62,586	147,892	6	4	3	**0.04**	0.10	0.07	0.05
caidaRouterLevel	192,244	609,066	32	17	16	**0.09**	0.12	0.23	0.15
as-skitter	1,696,415	11,095,298	111	67	45	**1.68**	2.57	6.21	2.74
wiki-Vote	7,115	100,762	53	17	37	0.27	0.21	0.10	**0.09**
soc-Epinions1	75,879	405,740	67	23	45	0.60	0.56	**0.26**	0.31
soc-Slashdot0811	77,360	469,180	54	26	29	0.29	0.33	**0.15**	0.20
soc-Slashdot0922	82,168	504,230	55	27	29	0.30	0.34	**0.16**	0.22
soc-pokec	1,632,803	22,301,964	47	29	19	13.50	12.93	19.45	**11.20**
soc-LiveJournal1	4,847,571	42,851,237	372	321	52	**4.95**	*	*	9.07
web-Stanford	281,903	1,992,636	71	61	11	**0.15**	0.25	2.96	0.53
web-NotreDame	325,729	1,090,108	155	155	1	**0.07**	0.09	0.12	0.23
web-BerkStan	685,230	6,649,470	201	201	1	**0.25**	0.29	6.96	1.33
eu-2005	862,664	16,138,468	388	387	2	**0.50**	0.72	26.29	3.06
web-Google	875,713	4,322,051	44	44	1	**0.33**	0.35	1.15	0.82
wiki-topcats	1,791,489	25,444,207	99	39	61	**12.58**	14.94	*	20.04
wave	156,317	1,059,331	8	6	3	**0.33**	1.39	0.63	0.49
audikw1	943,695	38,354,076	47	36	12	34.30	92.87	**14.10**	32.12
ldoor	952,203	22,785,136	34	21	14	19.04	36.58	**10.79**	14.36
ecology1	1,000,000	1,998,000	2	2	1	**0.55**	2.76	1.05	1.04
333SP	3,712,815	11,108,633	4	4	1	**1.49**	1.86	50.24	5.79
cage15	5,154,859	47,022,346	25	6	20	32.77	56.83	**23.38**	27.49
luxembourg.osm	114,599	119,666	2	3	0	0.02	0.02	0.18	**0.01**
belgium.osm	1,441,295	1,549,970	3	3	1	0.29	0.29	0.89	**0.28**
email-Enron	36,692	183,831	43	20	24	0.12	0.16	0.12	**0.09**
email-EuAll	265,214	364,481	37	16	22	**0.09**	0.11	0.12	0.12
wiki-Talk	2,394,385	4,659,565	131	26	106	36.92	**3.89**	5.82	5.25
com-Orkut	3,072,441	117,185,083	253	51	203	302.99	275.67	*	**241.00**
Amazon0302	262,111	899,792	6	7	0	**0.09**	**0.09**	0.32	0.17
Amazon0601	410,236	2,439,437	10	11	0	0.18	0.19	1.95	**0.13**
Amazon0505	403,394	2,443,408	10	11	0	0.18	0.19	2.19	**0.13**

Table 7: Comparison of the shifted geometric means (unscaled and scaled) of the times for algorithms **main**, BWBP, VBB, and RGG for runs executed on a single thread. The time shift s was set to one second.

		main		BWBP		VBB		RGG	
Instance group	Graphs	Unscaled	Scaled	Unscaled	Scaled	Unscaled	Scaled	Unscaled	Scaled
$g \in [0, 1)$	18	**0.73**	**1.00**	1.55	2.11	3.06	4.17	0.96	1.31
$g \in [1, 10)$	20	**0.21**	**1.00**	0.36	1.75	1.12	5.37	0.47	2.25
$g \in [10, 100)$	19	**2.55**	**1.00**	5.09	1.99	3.36	2.49	2.65	1.04
$g \in [100, 1000)$	3	312.15	11.62	102.36	3.81	201.30	7.49	**26.87**	**1.00**
Real-life	49	**1.30**	**1.00**	2.06	1.58	4.29	3.30	1.43	1.10
All	60	1.50	1.02	2.28	1.55	3.80	2.58	**1.48**	**1.00**

The SGMs also corroborate the intuition that our approach should be fastest when the clique-core gap is small ($g < 10$). Interestingly, algorithm **main** also seems to fare quite well on instances with moderate clique-core gaps (i.e., within the $[10, 100)$ interval), performing slightly better than RGG and better than BWBP and VBB. In contrast, on the instances with large clique-core gaps ($g > 100$), **main** is significantly slower than the other three algorithms, especially RGG. This notable performance gap can be directly attributed to the large amount of time it takes **main** to solve the synthetic instance kron_g500-simple-log16.

Another observation is that **main** performs better than the other algorithms on graphs arising from real-life applications. This supports the idea that our approach, which was developed and analyzed using *worst-case* analysis, is actually competitive with the best approaches from the literature on *real-life* graphs.

5 Further Observations

As observed in Section 4.2, our approach performs relatively well even in several cases where the clique-core gap g is somewhat large. For example, graphs cit-Patents, as-skitter, soc-Epinions1, and soc-LiveJournal1 are solved in less than 5 seconds despite having moderate clique-core gaps (not less than 45). Why is this the case? Where in the worst-case runtime analysis of algorithm **main** are we being overly pessimistic?

We make two observations that help to explain this phenomenon. For some instances G,

1. very few of the graphs $\overline{G}[V_i]$ need to be generated;
2. very few branches are generated.

In cases where the first observation applies, the time to generate the subgraphs $O(|D|d^2)$ may be significantly less than the worst-case bound $O((n-d)d^2)$ that appears in Theorem 1. For example, consider the graphs soc-LiveJournal1 and cage15 in Table 8. They are similarly sized, each having roughly 5 million vertices and 45 million edges. Perhaps surprisingly, soc-LiveJournal1 is solved more quickly than is cage15 (5 seconds vs. 33 seconds), despite having a clique-core gap that is *larger* by 32. Our best explanation for this phenomenon is the number of subgraphs $\overline{G}[V_i]$ that have to be generated by **main**. For soc-LiveJournal1 this number is only 152, which is a tiny fraction of the $n - d + 1 = 4,847,200$ subgraphs possible. In contrast, for the graph cage15, roughly 90% of the more than 5 million subgraphs $\overline{G}[V_i]$ are generated by our approach. As can be seen in Table 8, it is common for **main** to generate only a small fraction of the possible subgraphs; in 31 out of the 60 instances, fewer than 2% of the subgraphs have to be generated.

In cases where the second observation applies, the time to explore the search trees is considerably less than the exponential term from Corollary 1 would suggest. A first explanation for the small

number of branches is that the preprocessing provided by the Buss kernel, Nemhauser-Trotter kernel, and degree-two reduction rules is very powerful, often solving the k-vertex cover problems at the root node of the search tree. For example, consider the graphs er-fact1.5-scale20, soc-pokec, audikw1, and ldoor. Even though **main** solves over a million k-vertex cover subproblems for each of these graphs, it does not branch once. This explains why the 1.2 million k-vertex cover subproblems for soc-pokec are solved in just 0.22 seconds (see Table 5). The ability to entirely avoid branching is quite common, occurring on 33 of the 46 instances, including all graphs that have clique-core gaps less than 20 (see Table 8). A second explanation for the small number of branches relates to the values of k in the k-vertex cover subproblems. While **main**'s running time is expressed with respect to the worst case where $k = g$, most of the subproblems are solved for smaller values of k. These small-k subproblems are easier in worst-case terms and may also be more amenable to preprocessing (recalling that all instances with $g < 20$ were solved without branching).

For further analysis, we introduce the right-degree distribution.

Definition 5 (right-degree distribution) *Given a graph $G = (V, E)$ and a vertex ordering $\sigma = (v_1, v_2, \ldots, v_n)$, we define the right-degree distribution of pair (G, σ) to be a function that indicates the fraction of vertices with right-degree equal to k, for each $k \geq 0$.*

Figure 3 presents the right-degree distributions for twelve graphs from our test bed and for four others from the 2nd DIMACS Implementation Challenge. In each case, we use our MD ordering as the vertex ordering σ. Here, the horizontal axis represents the right-degree, and the vertical axis represents the percentage of vertices having such a right-degree. The right-degree distributions for all 60 graphs from our test bed can be found in Figure 5 of Appendix B.

In these distributions, the position of ω (indicated by the dashed line) is important for the following reason. The mass of distribution's tail to the *right* of this dashed line (essentially) indicates the fraction of the possible subgraphs $\overline{G}[V_i]$ that will be generated by **main**. For most of the real-life graphs, we see something like as-skitter or web-Stanford, where this right-tail has small mass, meaning that very few subgraphs will be generated. This happens quite often on the real-life graphs, which can be observed by looking in Appendix B. In contrast, for the graphs er-fact1.5-scale20, cage15, and the four DIMACS_2 graphs, the right-tail is nearly the entire mass, meaning that a large fraction of the subgraphs will be generated.

The *width* of the right-tail also has a meaningful interpretation. It is (essentially) equivalent to the clique-core gap g, which is important given the (possibly) exponential dependence of **main** on g. However, not all width-g right-tails are created equal.

The *shape* of the right-tail also influences the performance of algorithm **main**. A good example of this is cit-Patents. This graph has a moderate clique-core gap of $g = 54$, as can been observed by the moderate width of the right-tail. However, this tail is usually very thin, especially at its right-most end, meaning that only for very few subgraphs $\overline{G}[V_i]$ will **main** need to solve for a k-vertex cover with k being moderately large. Indeed, so many of them will be solved for tiny values of k that the average value of k is only 4 (see Table 8). Similarly, cage15 has a clique-core gap of 20 and our approach solves more than 21 million instances of the k-vertex cover problem (!); however, so much of the right-tail is concentrated near to ω that the average value of k is only 3, and **main** is able to finish in only 33 seconds.

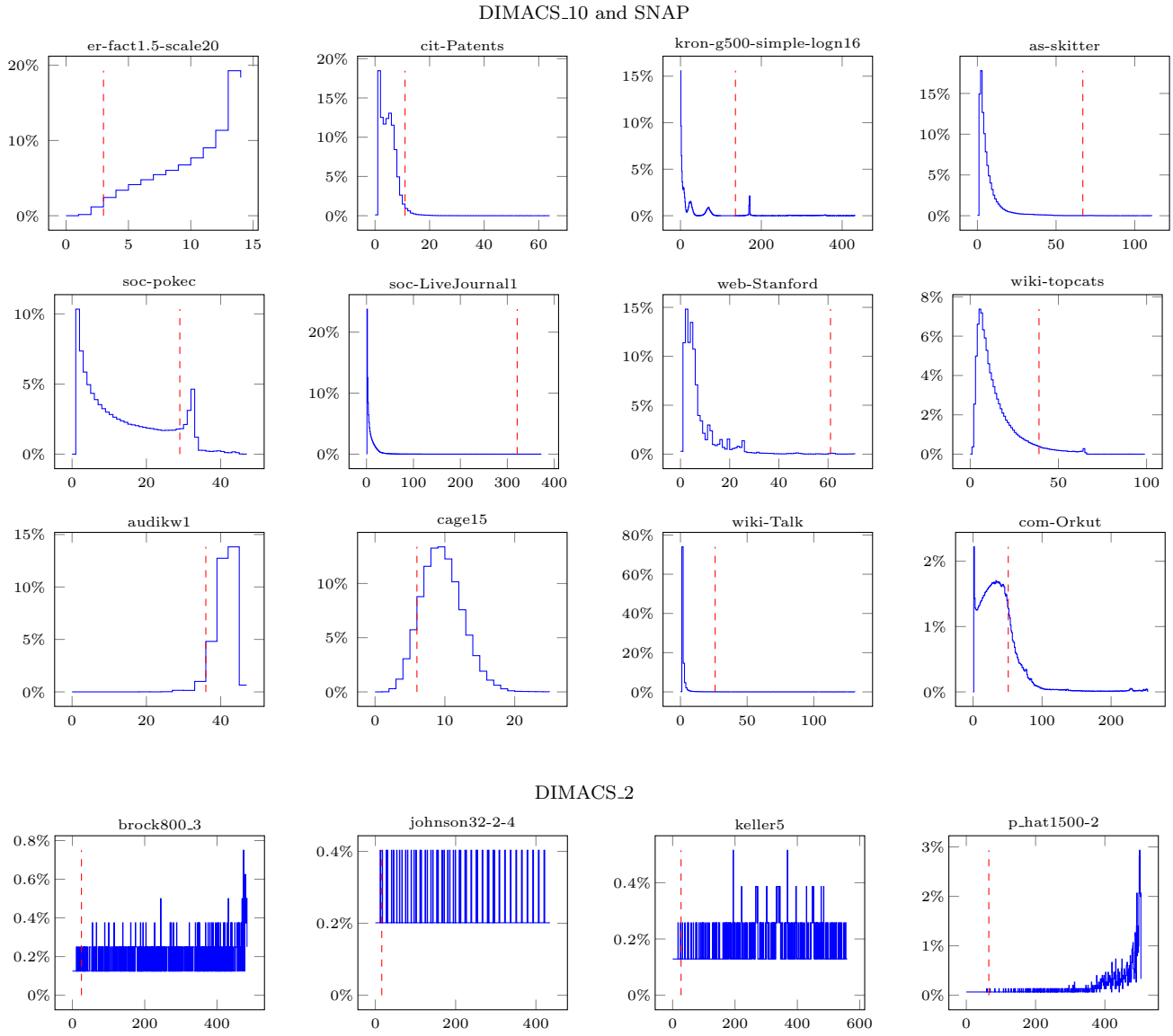
In a quite different example, consider the DIMACS_2 graph named p_hat1500-2. This instance is known to be challenging and, to our knowledge, its clique number is still unknown. (So, the dashed line marks the best existing lower bound.) Given this difficulty, it makes sense that it has an awful right-degree distribution. Its right-tail contains nearly all of the mass, meaning that **main** would generate nearly all of the subgraphs. Its right-tail is quite long, meaning that **main** would need to solve k-vertex cover for a large value of k (perhaps $k \approx 440$). Finally, the shape of the right-tail is no

Table 8: More details from the experiments. We report the number of subgraphs $\overline{G}[V_i]$ generated, the percentage of subgraphs generated out of $n - d + 1$, the average number of vertices $|V_i|$ in these subgraphs, the number of times the vertex cover subroutine is called, the average value of q_i of the q_i-vertex cover subproblems that are solved, and the total number of branches (#B) generated by search trees of all the q_i-vertex cover subproblems. Averages are rounded to integers.

Instance	n	d	ω	g	Time	Subgraphs generated	% out of $n-d+1$	Avg. $\lvert V_i\rvert$	Calls to V.C.	Avg. q_i	#B
Cit-HepPh	34,546	30	19	12	0.18	7,618	22	23	42,887	3	0
Cit-HepTh	27,770	37	23	15	0.20	6,043	22	29	41,336	4	0
citationCiteseer	268,495	15	13	3	0.11	2,435	1	13	3,767	0	0
er-fact1.5-scale20	**1,048,576**	**14**	**3**	**12**	**9.12**	**1,034,648**	**99**	**11**	**8,810,258**	**4**	**0**
cit-Patents	3,774,768	64	11	54	4.52	117,982	3	13	505,845	4	290,504
delaunay_n16	65,536	4	4	1	0.05	13,347	20	4	26,693	0	0
kron_g500	**65,536**	**432**	**136**	**297**	**2,663.15**	**6,747**	**10**	**220**	**575,752**	**75**	**46,650,543**
G_n_pin_pout	100,000	7	4	4	0.27	86,671	87	5	288,314	1	0
smallworld	100,000	7	6	2	0.10	30,896	31	6	36,668	0	0
delaunay_n17	131,072	4	4	1	0.11	26,831	20	4	53,656	0	0
delaunay_n18	262,144	4	4	1	0.23	53,648	20	4	107,163	0	0
p2p-Gnutella04	10,876	7	4	4	0.01	6,048	56	5	14,957	1	0
p2p-Gnutella25	22,687	5	4	2	0.01	6,927	31	4	10,079	0	0
as-22july06	22,963	25	17	9	0.01	110	0	21	600	3	0
p2p-Gnutella24	26,518	5	4	2	0.01	8,420	32	4	13,190	0	0
p2p-Gnutella30	36,682	7	4	4	0.02	11,668	32	4	18,418	0	0
p2p-Gnutella31	62,586	6	4	3	0.04	19,087	30	4	29,922	0	0
caidaRouterLevel	192,244	32	17	16	0.09	2,491	1	21	11,751	4	0
as-skitter	1,696,415	111	67	45	1.68	3,490	0	77	38,333	10	8
Wiki-Vote	7,115	53	17	37	0.27	2,243	32	37	47,869	13	9,035
soc-Epinions1	75,879	67	23	45	0.60	4,736	6	43	101,433	15	13,206
soc-Slashdot0811	77,360	54	26	29	0.29	4,706	6	35	45,319	6	1,093
soc-Slashdot0922	82,168	55	27	29	0.30	4,677	6	36	44,989	6	1,046
soc-pokec	**1,632,803**	**47**	**29**	**19**	**12.93**	**244,868**	**15**	**32**	**1,218,334**	**3**	**0**
soc-LiveJournal1	4,847,571	372	321	52	4.95	152	0	355	5,338	20	0
web-Stanford	281,903	71	61	11	0.15	918	0	64	4,359	4	0
web-NotreDame	325,729	155	155	1	0.07	8	0	154	11	0	0
web-BerkStan	685,230	201	201	1	0.25	1	0	201	1	0	0
eu-2005	862,664	388	387	2	0.50	15	0	388	28	0	0
web-Google	875,713	44	44	1	0.33	3	0	44	3	0	0
wiki-topcats	**1,791,489**	**99**	**39**	**61**	**12.58**	**98,095**	**5**	**50**	**1,248,588**	**9**	**304**
wave	156,317	8	6	3	0.33	152,919	98	7	280,717	1	0
audikw1	**943,695**	**47**	**36**	**12**	**34.30**	**906,531**	**96**	**41**	**5,413,233**	**3**	**0**
ldoor	**952,203**	**34**	**21**	**14**	**19.04**	**728,774**	**77**	**26**	**4,431,069**	**4**	**0**
ecology1	1,000,000	2	2	1	0.55	998,001	100	2	998,001	0	0
333SP	3,712,815	4	4	1	1.49	137,606	4	4	274,283	0	0
cage15	**5,154,859**	**25**	**6**	**20**	**32.77**	**4,625,580**	**90**	**10**	**21,574,632**	**3**	**39**
luxembourg.osm	114,599	2	3	0	0.02	248	0	2	248	0	0
belgium.osm	1,441,295	3	3	1	0.29	1	0	3	1	0	0
Email-Enron	36,692	43	20	24	0.12	2,238	6	30	25,803	7	110
Email-EuAll	265,214	37	16	22	0.09	1,534	1	26	17,006	7	94
Wiki-Talk	**2,394,385**	**131**	**26**	**106**	**36.92**	**14,787**	**1**	**52**	**401,229**	**29**	**2,142,798**
com-Orkut	**3,072,441**	**253**	**51**	**203**	**302.99**	**704,585**	**23**	**79**	**20,635,158**	**44**	**925,580**
Amazon0302	262,111	6	7	0	0.09	1	0	6	1	0	0
Amazon0505	410,236	10	11	0	0.19	148	0	10	148	0	0
Amazon0601	403,394	10	11	0	0.19	6	0	10	6	0	0

Note: The name of the instance kron_g500-simple-logn16 was shortened to reduce the width of the table.

Figure 3: Right-degree distribution of a selection of instances. The horizontal axis represents the right-degree and the vertical axis the percentage of vertices having such a right-degree. The dashed line marks ω.



Note: The dashed line for p_hat1500-2 marks the best known solution.

good either, being concentrated to its right-most end, meaning that **main** would need to solve *many* k-vertex cover instances with k being large. These three properties make p_hat1500-2 quite unlike the real-life instances from the DIMACS_10 and SNAP testbeds and quite a challenge for algorithm **main**.

6 Conclusion

In this paper, we aim to answer the questions: Why is the maximum clique problem often easy in practice? Is there a rigorous, worst-case explanation for this phenomenon? In this pursuit, we first observe that real-life instances often have a small clique-core gap $g := (d+1) - \omega$, which we define to be the difference between the clique number ω and its degeneracy-based upper bound $d+1$. Indeed, more than half of the large, real-life instances considered in this paper have clique-core gaps of 0, 1, or 2, and these instances are solved in mere seconds by the approaches of Verma et al. (2015) and Rossi et al. (2015). In contrast, synthetic benchmark instances often have clique-core gaps of 365 or more and remain unsolved after hours of computation (Prosser 2012).

We show that these observations are not just a coincidence. Hard instances must have large clique-core gaps, and instances with small clique-core gaps are easy. This is shown via an algorithm for the maximum clique problem that is fixed-parameter tractable with respect to g. In the particular case that g is a constant, our approach runs in time $O(dm) = O(m^{1.5})$. This provides the first rigorous, worst-case explanation for why instances like uk-2002 are easy to solve despite having more than 260 million edges and a degeneracy of nearly one thousand. Detailed computational experiments show that our algorithm is competitive with the best approaches from the literature, particularly on real-life graphs and other graphs that have clique-core gaps less than one hundred. Even when g is one hundred or more, our approach can still be practical, which we explain by the shape of the right-degree distribution and the clique number's position in it. Our implementation is publicly available for others to improve upon, extend, or compare with it.

6.1 Open Problems and Alternative Parameterizations

One might wonder if there are better parameterizations for the maximum clique problem. For example, if given a k-coloring of the graph, is it fpt with respect to the parameter p to check whether $\omega \geq k - p$? If so, this would generalize our approach, since one can find a coloring with $k = d + 1$ colors in linear time (Matula & Beck 1983). This turns out to be a bad parameterization with respect to p, since the case $p = 0$ is NP-hard, say, by the usual reduction from 3-SAT to Clique. Essentially the same result was stated by Akiba & Iwata (2016) in the context of parameterizations above the clique cover number for the minimum vertex cover problem.

In a related question, one might ask if the maximum clique problem is fpt with respect to the parameter k when given a k-coloring. If so, this would generalize the approaches of Eppstein et al. (2013), Buchanan et al. (2014), Manoussakis (2014). But, this is another bad parameterization, as this amounts to the multicolored clique problem, which is widely believed not to be fpt, see Theorem 13.7 of Cygan et al. (2015).

We leave with a few open problems. Our algorithm **main** checks if $\omega = d+1$ in linear time provided that the d-core of the graph is d-regular. However, it can take time $\Omega(m^{1.5})$ when d-regularity is not assumed. This motivates our first open question.

Open Problem 1 *Is there a linear-time algorithm to test if $\omega = d + 1$?*

If this first question is answered in the affirmative a natural next question is as follows. Recall that our approach runs in time $O(dm) = O(m^{1.5})$ when p is a constant.

Open Problem 2 *Is there an algorithm that, for any constant p, tests if $\omega \geq d + 1 - p$ in linear time?*

Finally, we discuss possible parameterizations for the minimum vertex coloring problem. It is natural to parameterize below degeneracy, i.e., to see if the chromatic number χ satisfies $\chi \leq d+1-p$. However, this is a bad parameterization as coloring is NP-hard on 4-regular planar graphs (Garey et al. 1976, Dailey 1980). Namely, the case $d = 4$ and $p = 2$ is NP-hard. For the same reason, parameterizing above the clique number ω is fruitless.

Open Problem 3 *Are there good parameterizations for coloring real-life graphs?*

A possibly exploitable fact is that checking whether an n-vertex graph has $\chi \leq n - k$ is fpt with respect to k. Indeed, the running time is bounded by $O^*(8^k)$ by the size $3k - 3$ kernel of Chor et al. (2004) and the $O^*(2^n)$ algorithm for coloring due to Björklund et al. (2009). However, χ is typically orders of magnitude smaller than n on real-life graphs, meaning that k would end up being rather large and an alternative parameterization is needed. Verma et al. (2015) found coloring to be more difficult than maximum clique in practice, so finding a good parameterization for vertex coloring is perhaps a more onerous task than ours.

Acknowledgements

This work was supported in part by the National Science Foundation through the awards CMMI-1635611, "Operational Decision-Making for Reach Maximization of Incentive Programs that Influence Consumer Energy-Saving Behavior" and CMMI-1662757, "Imposing Connectivity Constraints in Large-Scale Network Problems". The first author dedicates this work to Myriam Silva J. and Jose Luis Walteros M. for their unconditional support and love. The authors also thank the Center for computational Research (CCR) at the University at Buffalo for their support towards the execution of the computational experiments.

References

Abello, J., Pardalos, P. M. & Resende, M. G. C. (1999), On maximum clique problems in very large graphs, *in* 'External memory algorithms', Vol. 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, pp. 119–130.

Abu-Khzam, F. N., Collins, R. L., Fellows, M. R., Langston, M. A., Suters, W. H. & Symons, C. T. (2004), 'Kernelization algorithms for the vertex cover problem: Theory and experiments.', *ALENEX/ANALC* **69**.

Achterberg, T. (2007), Constraint integer programming, PhD thesis, Technische Universität Berlin.

Akiba, T. & Iwata, Y. (2016), 'Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover', *Theoretical Computer Science* **609**(1), 211–225.

Bader, G. D. & Hogue, C. W. (2003), 'An automated method for finding molecular complexes in large protein interaction networks', *BMC Bioinformatics* **4**(1), 2.

Balasubramanian, R., Fellows, M. & Raman, V. (1998), 'An improved fixed-parameter algorithm for vertex cover', *Information Processing Letters* **65**(3), 163–168.

Bar-Yehuda, R. & Even, S. (1985), A local-ratio theorem for approximating the weighted vertex cover problem, *in* G. Ausiello & M. Lucertini, eds, 'Analysis and Design of Algorithms for Combinatorial Problems', Vol. 109 of *North-Holland Mathematics Studies*, North-Holland, pp. 27–45.

Barabási, A.-L. & Albert, R. (1999), 'Emergence of scaling in random networks', *Science* **286**(5439), 509–512.

Batagelj, V. & Zaversnik, M. (2003), 'An $O(m)$ algorithm for cores decomposition of networks', *arXiv preprint arXiv:cs/0310049v1* .

Björklund, A., Husfeldt, T. & Koivisto, M. (2009), 'Set partitioning via inclusion-exclusion', *SIAM Journal on Computing* **39**(2), 546–563.

Bomze, I. M., Budinich, M., Pardalos, P. M. & Pelillo, M. (1999), The maximum clique problem, *in* 'Handbook of Combinatorial Optimization', Springer, pp. 1–74.

Bonato, A., Hadi, N., Horn, P., Prałat, P. & Wang, C. (2009), 'Models of online social networks', *Internet Mathematics* **6**(3), 285–313.

Borchers, B. (1999), 'CSDP, a C library for semidefinite programming', *Optimization methods and Software* **11**(1-4), 613–623.

Borchers, B. (2017), 'CSDP 6.2 user's guide'. https://github.com/coin-or/Csdp/blob/master/doc/csdpuser.pdf,last accessed: Aug, 2019.

Buchanan, A., Walteros, J. L., Butenko, S. & Pardalos, P. M. (2014), 'Solving maximum clique in sparse graphs: an $O(nm + n2^{d/4})$ algorithm for d-degenerate graphs', *Optimization Letters* **8**(5), 1611–1617.

Buss, J. F. & Goldsmith, J. (1993), 'Nondeterminism within P', *SIAM Journal on Computing* **22**(3), 560–572.

Carraghan, R. & Pardalos, P. M. (1990), 'An exact algorithm for the maximum clique problem', *Operations Research Letters* **9**(6), 375–382.

Chen, J., Fernau, H., Kanj, I. A. & Xia, G. (2007), 'Parametric duality and kernelization: Lower bounds and upper bounds on kernel size', *SIAM Journal on Computing* **37**(4), 1077–1106.

Chen, J., Kanj, I. A. & Jia, W. (2001), 'Vertex cover: Further observations and further improvements', *Journal of Algorithms* **41**(2), 280–301.

Chen, J., Kanj, I. A. & Xia, G. (2010), 'Improved upper bounds for vertex cover', *Theoretical Computer Science* **411**(40), 3736–3756.

Chor, B., Fellows, M. & Juedes, D. (2004), Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps, *in* 'Proceedings of the 30th international conference on Graph-Theoretic Concepts in Computer Science', Springer-Verlag, pp. 257–269.

Cohen, J. (2008), 'Trusses: Cohesive subgraphs for social network analysis', *National Security Agency Technical Report* p. 16.

Csardi, G., Nepusz, T. et al. (2006), 'The igraph software package for complex network research', *InterJournal, Complex Systems* **1695**(5), 1–9.

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M. & Saurabh, S. (2015), *Parameterized Algorithms*, Springer.

Dailey, D. P. (1980), 'Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete', *Discrete Mathematics* **30**(3), 289–293.

Dell, H. & Van Melkebeek, D. (2014), 'Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses', *Journal of the ACM* **61**(4), 23.

Downey, R. G. & Fellows, M. R. (1999), *Parameterized complexity*, Springer Science & Business Media.

Eppstein, D., Löffler, M. & Strash, D. (2013), 'Listing all maximal cliques in large sparse real-world graphs', *ACM Journal of Experimental Algorithmics* **18**(3), 3.1.

Erdös, P. & Hajnal, A. (1966), 'On chromatic number of graphs and set-systems', *Acta Mathematica Hungarica* **17**(1-2), 61–99.

Freuder, E. C. (1982), 'A sufficient condition for backtrack-free search', *Journal of the ACM* **29**(1), 24–32.

Garey, M. R., Johnson, D. S. & Stockmeyer, L. (1976), 'Some simplified NP-complete graph problems', *Theoretical Computer Science* **1**(3), 237–267.

Håstad, J. (1999), 'Clique is hard to approximate within $O(n^{1-\varepsilon})$', *Acta Mathematica* **182**(1), 105–142.

Iwata, Y., Oka, K. & Yoshida, Y. (2014), Linear-time FPT algorithms via network flow, *in* 'Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms', Society for Industrial and Applied Mathematics, pp. 1749–1761.

Kirousis, L. M. & Thilikos, D. M. (1996), 'The linkage of a graph', *SIAM Journal on Computing* **25**(3), 626–647.

Knuth, D. E. (1994), 'The sandwich theorem', *The Electronic Journal of Combinatorics* **1**(1), 1.

Konc, J. & Janežič, D. (2007), 'An improved branch and bound algorithm for the maximum clique problem', *MATCH–Communications in Mathematical and in Computer Chemistry* **58**, 569–590.

Li, C.-M. & Quan, Z. (2010), An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem, *in* 'Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence', pp. 128–133.

Li, W. & Zhu, B. (2018), 'A $2k$-kernelization algorithm for vertex cover based on crown decomposition', *Theoretical Computer Science* **739**, 80–85.

Lick, D. R. & White, A. T. (1970), 'k-degenerate graphs', *Canadian Journal of Mathematics* **22**, 1082–1096.

Lieder, F., Rad, F. B. A. & Jarre, F. (2015), 'Unifying semidefinite and set-copositive relaxations of binary problems and randomization techniques', *Computational Optimization and Applications* **61**(3), 669–688.

Lovász, L. (1979), 'On the shannon capacity of a graph', *IEEE Transactions on Information theory* **25**(1), 1–7.

Manoussakis, G. (2014), 'The clique problem on inductive k-independent graphs', *arXiv preprint arXiv:1410.3302* .

Manoussakis, G. (2016), 'New algorithms for cliques and related structures in k-degenerate graphs', *arXiv preprint arXiv:1501.01819v4* .

Matula, D. W. & Beck, L. L. (1983), 'Smallest-last ordering and clustering and graph coloring algorithms', *Journal of the ACM* **30**(3), 417–427.

Mittelmann, H. (2018), 'Benchmarks for optimization software'. http://plato.asu.edu/bench.html, last accessed: Aug, 2019.

Nagamochi, H. (2010), 'Minimum degree orderings', *Algorithmica* **56**(1), 17.

Nemhauser, G. L. & Trotter Jr, L. E. (1975), 'Vertex packings: structural properties and algorithms', *Mathematical Programming* **8**(1), 232–248.

Niedermeier, R. & Rossmanith, P. (2000), 'A general method to speed up fixed-parameter-tractable algorithms', *Information Processing Letters* **73**(3), 125–129.

Östergård, P. R. J. (2002), 'A fast algorithm for the maximum clique problem', *Discrete Applied Mathematics* **120**(1), 197–207.

Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., Liao, W.-k. & Choudhary, A. (2013), Fast algorithms for the maximum clique problem on massive sparse graphs, *in* 'International Workshop on Algorithms and Models for the Web-Graph', Springer, pp. 156–169.

Prosser, P. (2012), 'Exact algorithms for maximum clique: A computational study', *Algorithms* **5**(4), 545–587.

Pulleyblank, W. R. (1979), 'Minimum node covers and 2-bicritical graphs', *Mathematical Programming* **17**(1), 91–103.

Robson, J. M. (2001), Finding a maximum independent set in time $O(2^{n/4})$, Technical report, LaBRI, Université de Bordeaux I. https://www.labri.fr/perso/robson/mis/techrep.html, last accessed: Aug, 2019.

Rossi, R. A. (2014), Fast triangle core decomposition for mining large graphs, *in* 'Pacific-Asia Conference on Knowledge Discovery and Data Mining', Springer, pp. 310–322.

Rossi, R. A. & Ahmed, N. K. (2015), The network data repository with interactive graph analytics and visualization, *in* 'Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence'. Last accessed: Aug, 2019.
URL: *http://networkrepository.com*

Rossi, R. A., Gleich, D. F. & Gebremedhin, A. H. (2015), 'Parallel maximum clique algorithms with applications to network analysis', *SIAM Journal on Scientific Computing* **37**(5), C589–C616.

San Segundo, P., Artieda, J., Batsyn, M. & Pardalos, P. M. (2017), 'An enhanced bitstring encoding for exact maximum clique search in sparse graphs', *Optimization Methods and Software* **32**(2), 312–335.

San Segundo, P., Lopez, A., Artieda, J. & Pardalos, P. M. (2017a), 'A parallel maximum clique algorithm for large and massive sparse graphs', *Optimization Letters* **11**(2), 343–358.

San Segundo, P., Lopez, A. & Pardalos, P. M. (2016), 'A new exact maximum clique algorithm for large and massive sparse graphs', *Computers & Operations Research* **66**, 81–94.

San Segundo, P., Nikolaev, A. & Batsyn, M. (2015), 'Infra-chromatic bound for exact maximum clique search', *Computers & Operations Research* **64**, 293–303.

Savelsbergh, M. W. P. (1994), 'Preprocessing and probing techniques for mixed integer programming problems', *ORSA Journal on Computing* **6**(4), 445–454.

Seidman, S. B. (1983), 'Network structure and minimum degree', *Social Networks* **5**(3), 269–287.

Sloane, N. J. A. (2017), 'Challenge problems: Independent sets in graphs', http://www2.research.att.com/~njas/doc/graphs.html. Last accessed: Aug, 2019.

Szekeres, G. & Wilf, H. S. (1968), 'An inequality for the chromatic number of a graph', *Journal of Combinatorial Theory* **4**(1), 1–3.

Tomita, E. & Kameda, T. (2007), 'An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments', *Journal of Global Optimization* **37**(1), 95–111.

Tomita, E., Sutani, Y., Higashi, T., Takahashi, S. & Wakatsuki, M. (2010), A simple and faster branch-and-bound algorithm for finding a maximum clique, *in* M. S. Rahman & S. Fujita, eds, 'WALCOM: Algorithms and Computation', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 191–203.

Verma, A., Buchanan, A. & Butenko, S. (2015), 'Solving the maximum clique and vertex coloring problems on very large sparse networks', *INFORMS Journal on Computing* **27**(1), 164–177.

Wang, J. & Cheng, J. (2012), 'Truss decomposition in massive networks', *Proceedings of the VLDB Endowment* **5**(9), 812–823.

Watts, D. J. (1999), 'Networks, dynamics, and the small-world phenomenon', *American Journal of Sociology* **105**(2), 493–527.

Wilson, A. T. (2009), Applying the boundary point method to an SDP relaxation of the maximum independent set problem for a branch and bound algorithm, Master's thesis, New Mexico Institute of Mining and Technology.

Zuckerman, D. (2006), Linear degree extractors and the inapproximability of max clique and chromatic number, *in* 'Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing', STOC '06, pp. 681–690.

Appendix

A Algorithm Implementation

In this appendix we discuss the implementation of several key components of algorithm **main**. We also present further enhancements that can be incorporated to speed up its performance in practice and analyze the results obtained by our algorithm when equipped with these enhancements.

A.1 Computing MD Orderings

To compute an MD ordering in linear time, the algorithm by Matula & Beck (1983) stores the vertices in a bucket-sort type structure using $\Delta + 1$ buckets labeled from 0 to Δ. The algorithm initializes each bucket i so that it contains the vertices whose degree is equal to i. At each iteration, to identify the vertex that will be appended to the ordering (i.e., the one with the smallest degree), the algorithm

selects a vertex from the first non-empty bucket. Then, since removing such a vertex reduces the degree of its neighbors exactly by one, the algorithm proceeds to relocate them into their corresponding new buckets based on their updated degree, i.e., decreases their bucket by one.

In their original paper, Matula & Beck (1983) proposed an implementation that utilizes doubly-linked lists to represent the degree buckets. Since both the insertion and removal operations take constant time on doubly-linked lists, relocating the vertices between buckets to maintain the sorted structure is done quite efficiently. This type of implementation was also adopted by Eppstein et al. (2013) and Verma et al. (2015) in their respective codes.

A related linear-time algorithm was later given by Batagelj & Zaversnik (2003). This algorithm does not produce an MD ordering, but does solve a similar problem of generating k-cores. We employ one if its key ideas. It encodes the bucket structure using *arrays* instead of doubly-linked lists. For practical reasons, arrays tends to outperform doubly-linked lists mainly because most modern programming languages store arrays contiguously in memory unlike doubly-linked lists which, given their pointer-based nature, are generally stored scattered across memory. Arrays tend to reduce the number of cache misses during the execution of the algorithm, resulting in an overall better performance despite the array and doubly-linked list implementations having the same worst-case complexity.

Here, we provide a minor modification of the algorithm proposed by Batagelj & Zaversnik (2003), which we adapt to compute MD orderings. Algorithm 1 preserves the linear-time complexity of the one by Matula & Beck (1983), but encodes the bucket structure using the following arrays.

- `buckets`: An array of size n that stores the $\Delta + 1$ buckets in consecutive order starting from 0 and ending in Δ. Each bucket i is represented by a subarray that contains the vertices with degree i. The number of contiguous positions allocated to each bucket in this array changes as the algorithm progresses given that the vertices get swapped from bucket to bucket when their degree decreases. Therefore, the algorithm requires us to keep track of the position of the first element of each bucket in this array. These positions are maintained in the following array.

- `ind`: An array of size $\Delta + 1$ that contains the position where each bucket begins in the `buckets` array. During execution it is possible for a bucket to become empty. In this case, the position stored by array `ind` for that bucket will be equal to the value for the subsequent bucket.

- `ord`: An array of size n containing the MD ordering.

- `rdeg`: An array of size n containing the right-degree of each vertex.

- `pos`: An array of size n indicating the position of each vertex in the `buckets` array. Upon termination, this array yields the position of each vertex in the MD ordering.

The main difference with the original version proposed by Batagelj & Zaversnik (2003) resides in the way in which the algorithm updates the buckets of the neighbors of a vertex v that is being appended to the MD ordering (i.e., steps (26)-(31)). To compute an MD ordering, it is required to consider the specific case in which both v and at least one of its neighbors u are in the same bucket. This extra step can be omitted when generating k-cores. Algorithm 1 also uses steps (16)-(18) to identify a lower bound L on ω, which is explained in Section 4.1.

A.2 Generating and Processing the Subgraphs

We now discuss the procedure for constructing the subgraphs $\overline{G}[V_i]$ in **main**, i.e., steps 3(a) and 5. First, since it is reasonable to expect larger graphs to contain larger cliques, we do not construct and process the subgraphs following the sequence given by the MD ordering. Instead, we begin by sorting the first $n - d$ vertices of the MD ordering in decreasing order with respect to their right-degree and

Algorithm 1 Algorithm for computing an MD ordering of a graph.

Input: A graph $G = (V, E)$
Output: The degeneracy d of G, a lower bound L on ω, an MD ordering $(v_1, \ldots, v_n)$ of G, and the right-degree of the vertices in V.

procedure MDOrdering(G)
  Initialize the `buckets` and `ind` arrays according to the degree of the vertices. Ties are broken lexicographically.
  Initialize the `pos` array based on the position of the vertices in the `buckets` array.
  $d \leftarrow 0$
  $\texttt{L} \leftarrow 0$
  for all $i \in V$ **do**
    $\texttt{rdeg}[i] \leftarrow |N(i)|$
  end for
  for all $i = 1, \ldots, n$ **do**
    $v \leftarrow \texttt{buckets}[i]$
    $\texttt{ord}[i] \leftarrow v$ ▷ v becomes part of the MD ordering
    $\texttt{ind}[\texttt{rdeg}[v]] \leftarrow \texttt{ind}[\texttt{rdeg}[v]] + 1$ ▷ the index for the bucket where v was is updated to the position right after v
    if $d < \texttt{rdeg}[v]$ **then**
      $d \leftarrow \texttt{rdeg}[v]$ ▷ the degeneracy is updated
    end if
    if $L = 0$ and $\texttt{rdeg}[v] = n - i$ **then** ▷ a lower bound is found
      $L \leftarrow n - i + 1$
    end if
    for all $u \in N(v)$ **do** ▷ This loop updates buckets of the neighbors of v
      if $\texttt{pos}[u] > \texttt{pos}[v]$ **then**
        let w be the first vertex in u's current bucket
        if $u \neq w$ **then**
          Swap u and w in the `buckets` array and update `pos` accordingly
        end if
        $\texttt{rdeg}[u] \leftarrow \texttt{rdeg}[u] - 1$
        if $\texttt{rdeg}[u] = \texttt{rdeg}[v]$ **then** ▷ u and v are in the same bucket
          $\texttt{ind}[\texttt{rdeg}[u]] \leftarrow \texttt{pos}[v] + 1$ ▷ the index of the new bucket of u is set to begin right after $\texttt{pos}[v]$
          $\texttt{ind}[\texttt{rdeg}[u]] \leftarrow \texttt{ind}[\texttt{rdeg}[v]] + 1$ ▷ the index of the bucket that contained v is set to begin right after $\texttt{pos}[u]$
        else
          $\texttt{ind}[\texttt{rdeg}[u]] \leftarrow \texttt{pos}[u] + 1$ ▷ the index of the new bucket of u is set to begin right after $\texttt{pos}[u]$
        end if
      end if
    end for
  end for
  return d, `L`, `ord`, and `rdeg`
end procedure

then proceed to run the fpt vertex cover algorithms starting with subgraph $\overline{G}[V_f]$, followed by the subgraphs given by this new ordered sequence (i.e., step 3(b) in **main**). Since the $\text{rdeg}(i) \leq d$ for all $i \in V$, sorting the vertices can be done in time $O(n)$ via bucket sort, which preserves the worst case complexity of the algorithm as stated in Theorem 1.

Sorting the vertices by their right-degree has a crucial effect on the running time of the algorithm in practice because it directly yields set D of step 2 in **main** for any value of p (i.e., the algorithm halts returning "no" as soon as the first vertex with right-degree less than $d - p$ appears in the sequence). Furthermore, in most of the instances we tested (see Section 5), maximum cliques are often found in the largest subgraphs and, given that a significantly large proportion of the subgraphs tend to be quite small (often much smaller than ω), several subgraphs do not even need to be generated in **main**. In fact, among the instances we tested, in average only 23% of the subgraphs need to be generated in our implementation (see Table 8).

To construct a graph $\overline{G}[V_i]$ in time $O(d^2)$ (i.e., step 3(a) in **main**), we first generate the right-neighborhoods of the first $n - d$ vertices (i.e., V_1, V_2, ..., V_{n-d}). Manoussakis (2016) provides an $O(m)$ time algorithm to construct these vertex subsets using as input the MD ordering. We notice, however, that these sets can be immediately generated while computing the MD ordering as follows. Whenever a vertex v is appended to the MD ordering, the algorithm proceeds to decrease the degree of the neighbors of v that have not yet been added to the MD ordering. These neighbors are in fact the members of right-neighborhood V_i and as such can be stored directly. Once the MD ordering is computed and all sets $V_1, V_2, \ldots, V_f$ are generated, the algorithm proceeds to construct each $\overline{G}[V_i]$ on the fly, following the subsequent steps of the algorithm by Manoussakis (2016).

Furthermore, given that algorithm **main** is executed iteratively for several values of p starting from $p = 0$ until $p = d + 1 - \omega$ is reached, during each call of **main** the algorithm is only required to process the subgraphs $G[v_i]$ for which $|V_i| \geq d - p$. Once graph $\overline{G}[V_i]$ is generated during some call of **main**, we keep it in memory and reuse it for subsequent values of p.

Generating the vertex sets V_1, V_2, ..., V_f while computing the MD ordering avoids traversing the adjacency lists an extra time, thus reducing the running time of the algorithm in practice. However, we notice that this minor variation does not improve upon the worst-case time complexity given by Manoussakis (2016) because the largest contributor to the complexity of constructing the graphs $\overline{G}[V_i]$ is the time spent generating the edge sets and not the vertex sets.

A.3 Algorithm Parallelization

The proposed algorithm can directly benefit from a parallelized implementation over multiple threads without requiring major modifications to its design. The key observation for the parallelized version of **main** comes from the fact that the fpt procedure for vertex cover that is executed over each of the generated subgraphs (steps 3(b) and 5) can be run independently in parallel by different threads without the need to exchange large amounts of information between such processes. The task of generating subgraphs $\overline{G}[V_i]$ and running the vertex cover algorithm on them can then be evenly distributed among independent threads, while maintaining a global atomic flag that stops the execution of all the threads in the event that any of those threads finds a vertex cover of the desired size.

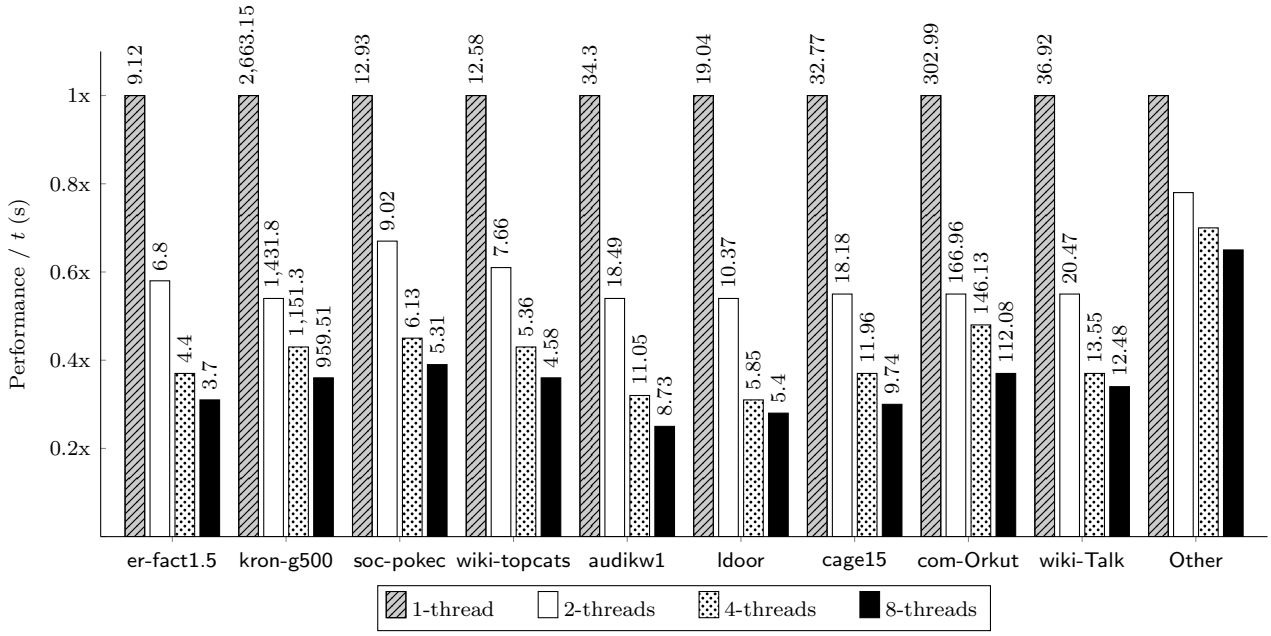
We now proceed to discuss the results of the parallelized version of our algorithm using two, four, and eight threads and analyze the performance improvements obtained. We compare the multi-threaded implementation with RGG exclusively, as the other algorithms were not coded to run in parallel. Table 9 presents the times in seconds of both approaches. We left the times obtained with a single processor to allow for a full comparison and highlight in bold the fastest times to solve each instance. As with previous tables, mark with an asterisk (*) the cases where an algorithm fails to identify ω within the time limit. Unsurprisingly, parallelization results in notable speedups on instances for which the clique-core gap is large. We notice however that for some easy instances with

small clique-core gaps, the overhead of creating and managing the additional threads may undermine the benefit of the parallelization, producing no substantial improvements. Additionally, the algorithm for computing an MD ordering is sequential, and so parallelization is not particularly helpful when the instance is solved soon after the MD ordering is found.

Results indicate algorithm **main** solves nearly all instances more quickly than does RGG. It seems that **main** is able to exploit the parallelized execution, reducing the solution times in most cases, whereas RGG is only able to do so for the largest instances. In fact, the running times of RGG sometimes slightly deteriorate on some small instances when using multiple threads. A possible explanation of this slight deterioration is that, for instances that are solved rather fast, RGG spends most of the execution in parts of the algorithms that are not parallelized, like generating the k-cores, and so the overhead paid for parallelization is not worth it.

To further understand the benefits of parallelization, we visualize in Figure 4 the times for the nine most challenging instances for **main**, i.e., those instances taking longer than five seconds with the single-thread implementation. The bars are scaled by the single-thread time (1x) and a lower value indicates a performance improvement. As can be observed, when ran on two, four, and eight threads, **main** finishes in average 43%, 61% and, 67% faster than on a single thread, respectively. Since algorithm **main** spends most of its time generating subgraphs and solving vertex cover subproblems (see Table 4.2), which are straightforward tasks to parallelize, the multi-threaded implementation results in noticeable improvements. For comparison purposes, we also report the average performance on the other 37 instances. As might be expected, the benefits of parallelization are less pronounced on them.

Figure 4: The benefits of parallelization for the nine instances that took **main** more than five seconds to solve on a single thread. The total times are reported above the bars.



Note: The names er-fact1.5-scale20 and kron_g500-simple-logn16 were shortened to reduce the width of the chart.

Table 9: Comparison of algorithms **main**, BWBP, VBB, and RGG. For each instance, the best times are highlighted in bold. The asterisk (*) indicates an inability to find ω within the time limit of one hour.

				Time									
				1 thread				2 threads		4 threads		8 threads	
Instance	d	ω	g	**main**	BWBP	VBB	RGG	**main**	RGG	**main**	RGG	**main**	RGG
Cit-HepTh	30	19	12	0.18	0.34	**0.12**	0.19	**0.11**	0.21	**0.07**	0.19	**0.06**	0.24
Cit-HepPh	37	23	15	0.20	0.37	**0.12**	0.20	**0.11**	0.21	**0.07**	0.19	**0.07**	0.24
citationCiteseer	15	13	3	**0.11**	0.14	0.53	0.26	**0.11**	0.37	**0.12**	0.38	**0.10**	0.42
er-fact1.5-scale20	14	3	12	9.12	13.64	17.15	**6.87**	6.80	**6.30**	**4.40**	5.34	**3.70**	5.69
cit-Patents	64	11	54	**4.52**	5.29	12.79	5.89	**4.29**	7.12	**4.09**	6.96	**3.80**	7.11
delaunay_n16	4	4	1	**0.05**	0.08	0.22	0.09	**0.04**	0.10	**0.03**	0.09	**0.03**	0.11
kron_g500	432	136	297	2,663.15	815.16	335.94	**13.31**	1,431.80	**7.85**	1,151.30	**6.08**	959.51	**4.42**
G_n_pin_pout	7	4	4	0.27	0.50	0.42	**0.23**	**0.16**	0.26	**0.11**	0.24	**0.09**	0.27
smallworld	7	6	2	**0.10**	0.27	0.41	0.22	**0.07**	0.25	**0.05**	0.22	**0.05**	0.25
delaunay_n17	4	4	1	**0.11**	0.16	0.22	0.18	**0.08**	0.21	**0.06**	0.19	**0.06**	0.21
delaunay_n18	4	4	1	**0.23**	0.33	0.45	0.36	**0.16**	0.41	**0.11**	0.37	**0.11**	0.42
p2p-Gnutella04	7	4	4	**0.01**	0.03	**0.01**	0.02	**0.01**	0.03	**0.01**	0.02	**0.01**	0.04
p2p-Gnutella25	5	4	2	**0.01**	0.03	0.44	0.02	**0.01**	0.03	**0.01**	0.02	**0.01**	0.06
as-22july06	25	17	9	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.02	**0.01**	0.02	**0.01**	0.03
p2p-Gnutella24	5	4	2	**0.01**	0.04	0.69	0.02	**0.01**	0.04	**0.01**	0.03	**0.01**	0.06
p2p-Gnutella30	7	4	4	**0.02**	0.06	0.12	0.03	**0.02**	0.04	**0.01**	0.04	**0.01**	0.06
p2p-Gnutella31	6	4	3	**0.04**	0.10	0.07	0.05	**0.03**	0.07	**0.02**	0.06	**0.02**	0.08
caidaRouterLevel	32	17	16	**0.09**	0.12	0.23	0.15	**0.07**	0.21	**0.07**	0.21	**0.07**	0.29
as-skitter	111	67	45	**1.68**	2.57	6.21	2.74	**1.36**	3.71	**1.31**	3.72	**1.15**	4.64
wiki-Vote	53	17	37	0.27	0.21	0.10	**0.09**	0.15	**0.08**	0.09	**0.07**	**0.07**	0.11
soc-Epinions1	67	23	45	0.60	0.56	**0.26**	0.31	0.35	**0.28**	**0.22**	0.23	**0.18**	0.25
soc-Slashdot0811	54	26	29	0.29	0.33	**0.15**	0.20	**0.17**	0.22	**0.10**	0.20	**0.10**	0.24
soc-Slashdot0922	55	27	29	0.30	0.34	**0.16**	0.22	**0.19**	0.24	**0.11**	0.21	0.12	0.24
soc-pokec	47	29	19	13.50	12.93	19.45	**11.20**	**9.02**	10.31	**6.13**	8.98	**5.31**	10.22
soc-LiveJournal1	372	321	52	**4.95**	*	*	9.07	**4.73**	13.24	**4.46**	13.16	**4.15**	13.84
web-Stanford	71	61	11	**0.15**	0.25	2.96	0.53	**0.12**	0.71	**0.12**	0.71	**0.10**	0.74
web-NotreDame	155	155	1	**0.07**	0.09	0.12	0.23	**0.07**	0.37	**0.07**	0.36	**0.07**	0.49
web-BerkStan	201	201	1	**0.25**	0.29	6.96	1.33	**0.26**	2.01	**0.27**	2.04	**0.26**	2.17
eu-2005	388	387	2	**0.50**	0.72	26.29	3.06	**0.48**	4.69	**0.51**	4.69	**0.50**	6.25
web-Google	44	44	1	**0.33**	0.35	1.15	0.82	**0.34**	1.29	**0.35**	1.31	**0.34**	1.44
wiki-topcats	99	39	61	**12.58**	14.94	*	20.04	**7.66**	11.41	**5.36**	8.44	**4.58**	6.70
wave	8	6	3	**0.33**	1.39	0.63	0.49	**0.20**	0.53	**0.13**	0.48	**0.12**	0.55
audikw1	47	36	12	34.30	92.87	**14.10**	32.12	**18.49**	24.28	**11.05**	19.97	**8.73**	19.58
ldoor	34	21	14	19.04	36.58	**10.79**	14.36	**10.37**	12.29	**5.85**	9.90	**5.40**	10.91
ecology1	2	2	1	**0.55**	2.76	1.05	1.04	**0.41**	1.16	**0.33**	1.01	**0.31**	1.05
333SP	4	4	1	**1.49**	1.86	50.24	5.79	**1.25**	5.94	**1.31**	5.02	**1.43**	5.06
cage15	25	6	20	32.77	56.83	**23.38**	27.49	**18.18**	24.56	**11.96**	20.00	**9.74**	18.44
luxembourg.osm	2	3	0	0.02	0.02	0.18	**0.01**	0.02	**0.01**	0.02	**0.01**	**0.02**	0.03
belgium.osm	3	3	1	0.29	0.29	0.89	**0.28**	**0.29**	0.45	**0.33**	0.45	**0.29**	0.50
email-Enron	43	20	24	0.12	0.16	0.12	**0.09**	**0.07**	0.10	**0.04**	0.09	**0.04**	0.12
email-EuAll	37	16	22	**0.09**	0.11	0.12	0.12	**0.07**	0.15	**0.06**	0.16	**0.05**	0.22
wiki-Talk	131	26	106	36.92	**3.89**	5.82	5.25	20.47	**3.76**	13.55	**2.97**	12.48	**2.78**
com-Orkut	253	51	203	302.99	275.67	*	**241.00**	166.96	**146.02**	146.13	**105.30**	112.08	**78.54**
Amazon0302	6	7	0	**0.09**	**0.09**	0.32	0.17	**0.09**	0.27	**0.10**	0.28	**0.09**	0.38
Amazon0601	10	11	0	0.18	0.19	1.95	**0.13**	**0.18**	0.22	**0.21**	0.22	**0.18**	0.29
Amazon0505	10	11	0	0.18	0.19	2.19	**0.13**	**0.18**	0.22	**0.20**	0.22	**0.18**	0.29

Note: The name kron_g500-simple-logn16 was shortened to reduce the width of the table.

A.4 Further Implementation Enhancements

There are several enhancements that can be easily incorporated into our approach to potentially reduce its running time. We briefly discuss a few of them in this section and provide details of their performance.

First, as described in Section 3, algorithm **main** is executed sequentially for several values of p starting from $p = 0$ until $p = d + 1 - \omega$ is reached. This linear-search type of execution may become expensive whenever the clique-core gap is expected to be large. In such case, the proposed algorithm may benefit from a different type of search to reduce the number of times **main** is called. In the presence of a good lower bound L, the algorithm could run in a binary search fashion, instead of the linear search. The algorithm can use as the initial L the lower bound produced by the MD ordering procedure described in Section 4.1 or the result of any other heuristic.

A drawback of this variation is that it depends heavily on the quality of the lower bound L and since clique is hard to approximate (Håstad 1999, Zuckerman 2006), $d + 1 - L$ can be as bad as $\Omega(d)$. Thus, the worst-case bound of $1.28^g \operatorname{poly}(n)$ given in Corollary 1 cannot be maintained and is drastically affected becoming $1.28^d \operatorname{poly}(n)$. We have observed in practice that the binary search variation often reduces the computational times to solve some of the instances we consider in our experiments. However, given its negative effect on the worst-case complexity of our approach, we do not expand on this variation other than a brief discussion about some computational experiments.

A second possible enhancement takes advantage of the way in which our approach divides the search across many smaller subgraphs. This allows for one to execute several heuristics and other algorithms over each of these subgraphs to identify upper and lower bounds on the maximum cliques of subgraphs $G[V_1]$, $G[V_2]$, ..., $G[V_f]$, thus potentially refining the bounds on ω. This could also be used to identify subgraphs that are not worth exploring (i.e., subgraphs whose clique upper bound is smaller than the best lower bound of G). However, in an effort to stay true to the proposed algorithm, we do not employ these ideas in our implementation.

We focus our attention on the variation that runs **main** in a binary search fashion for values of p within interval $[0, d + 1 - L)$, until $p = d + 1 - \omega$ is reached. Here, we use as L the lower bound produced by the MD ordering. We compare the performance of this variant with the linear search version described in Section 3. Table 10 presents the times obtained by both variants on the nine instances for which the linear search takes more than five seconds to solve (see Section 4).

Table 10: Comparison of the times for algorithm **main** using linear search (LS) and binary search (BS). For each instance, the best times are highlighted in bold.

					Time							
					1 thread		2 threads		4 threads		8 threads	
Instance	L	d	ω	g	LS	BS	LS	BS	LS	BS	LS	BS
er-fact1.5-scale20	2	14	3	12	9.12	**8.01**	6.80	**5.98**	4.40	**3.89**	3.70	**3.52**
audikw1	30	47	36	12	34.30	**32.63**	18.49	**17.37**	11.05	**10.06**	8.73	**8.55**
ldoor	21	34	21	14	19.04	**13.44**	10.37	**7.27**	5.85	**4.05**	5.40	**3.85**
soc-pokec	15	47	29	19	**12.93**	13.37	**9.02**	9.24	**6.13**	7.10	**5.31**	6.63
cage15	5	25	6	20	32.77	**29.00**	18.18	**16.59**	11.96	**10.07**	9.74	**9.22**
wiki-topcats	5	99	39	61	12.58	**12.46**	7.66	**7.59**	5.36	**5.32**	4.58	**4.57**
wiki-Talk	25	131	26	106	36.92	**16.42**	20.47	**9.01**	13.55	**5.47**	12.48	**5.49**
com-Orkut	14	253	51	203	302.99	**256.53**	166.96	**146.26**	146.13	**105.53**	112.08	**90.78**
kron-g500	135	432	136	297	2,663.15	**469.01**	1,431.80	**247.60**	1,151.30	**157.85**	959.51	**163.46**

Note: The name of the instance kron_g500-simple-logn16 was shortened to reduce the width of the table.

It can be observed that binary search outperforms linear search for all instances but one (soc-pokec). The results also suggest that the speedups obtained by the binary search depend significantly on the clique-core gaps. As one might expect, for instances with clique-core gap less than 100, the

times of both variants are quite similar with the binary search only achieving speedups of a few seconds. In contrast, the cases in which the clique-core gaps exceed 100 (wiki-Talk, com-Orkut, and kron_g500-simple-logn16), the binary search is able to find ω significantly faster than the linear search. The most notable example is the case of kron_g500-simple-logn16 for which the single-thread binary search implementation finds ω more than 2,000 seconds faster.

B Additional Tables and Figures

Table 11: Comparison of the times for algorithms **main**, BWBP, VBB, and RGG for the instances solved in linear time by our approach. For each instance, the best times are highlighted in bold.

Property	Name	n	m	d	ω	g	**main**	BWBP	VBB	RGG
	cond-mat-2005	40,421	175,691	29	30	0	**0.00**	0.02	0.01	0.01
	coAuthorsCiteseer	227,320	814,134	86	87	0	**0.03**	0.13	0.06	0.04
	coAuthorsDBLP	299,067	977,676	114	115	0	**0.04**	0.19	0.06	0.05
d-regular d-core	coPapersCiteseer	434,102	16,036,720	844	845	0	**0.17**	1.86	2.23	0.50
	coPapersDBLP	540,486	15,245,729	336	337	0	**0.20**	1.94	1.31	0.52
	rgg_n_2_21_s0	2,097,152	14,487,995	18	19	0	**0.55**	2.35	0.71	1.24
	rgg_n_2_22_s0	4,194,304	30,359,198	19	20	0	**1.33**	5.55	1.48	3.31
	uk-2002	18,520,486	261,787,258	943	944	0	**15.72**	39.73	70.11	16.34
	preferentialAttachment	100,000	499,985	5	6	0	**0.03**	0.10	0.21	0.31
	cnr-2000	325,557	2,738,969	488	489	0	**0.08**	0.32	2.75	0.09
	Amazon0312	400,727	2,349,869	10	11	0	0.17	0.50	1.84	**0.14**
$L = d + 1$	in-2004	1,382,908	13,591,473	5	6	0	**0.47**	1.46	7.67	**0.47**
	rgg_n_2_23_s0	8,388,608	63,501,393	20	21	0	6.13	12.14	**2.74**	7.98
	rgg_n_2_24_s0	16,777,216	89,345,197	20	21	0	9.30	17.44	**8.21**	12.18

Figure 5: Right-degree distribution of the different instances. The horizontal axis represents the right-degree and the vertical axis the percentage of vertices having such a right-degree. The red dashed line marks ω.

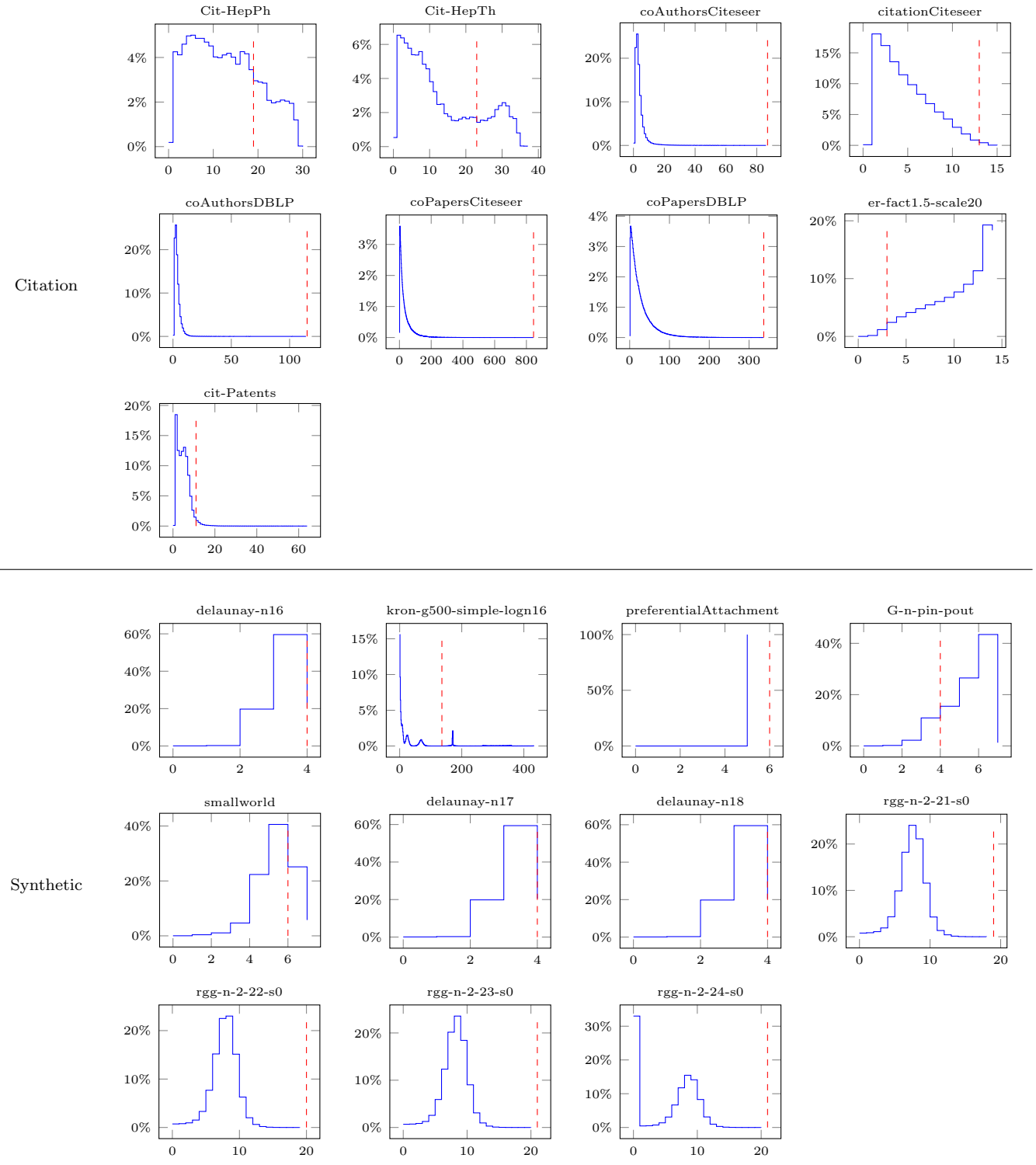


Figure 5: (Continued) Right-degree distribution of the different instances. The horizontal axis represents the right-degree and the vertical axis the percentage of vertices having such a right-degree. The red dashed line marks ω.

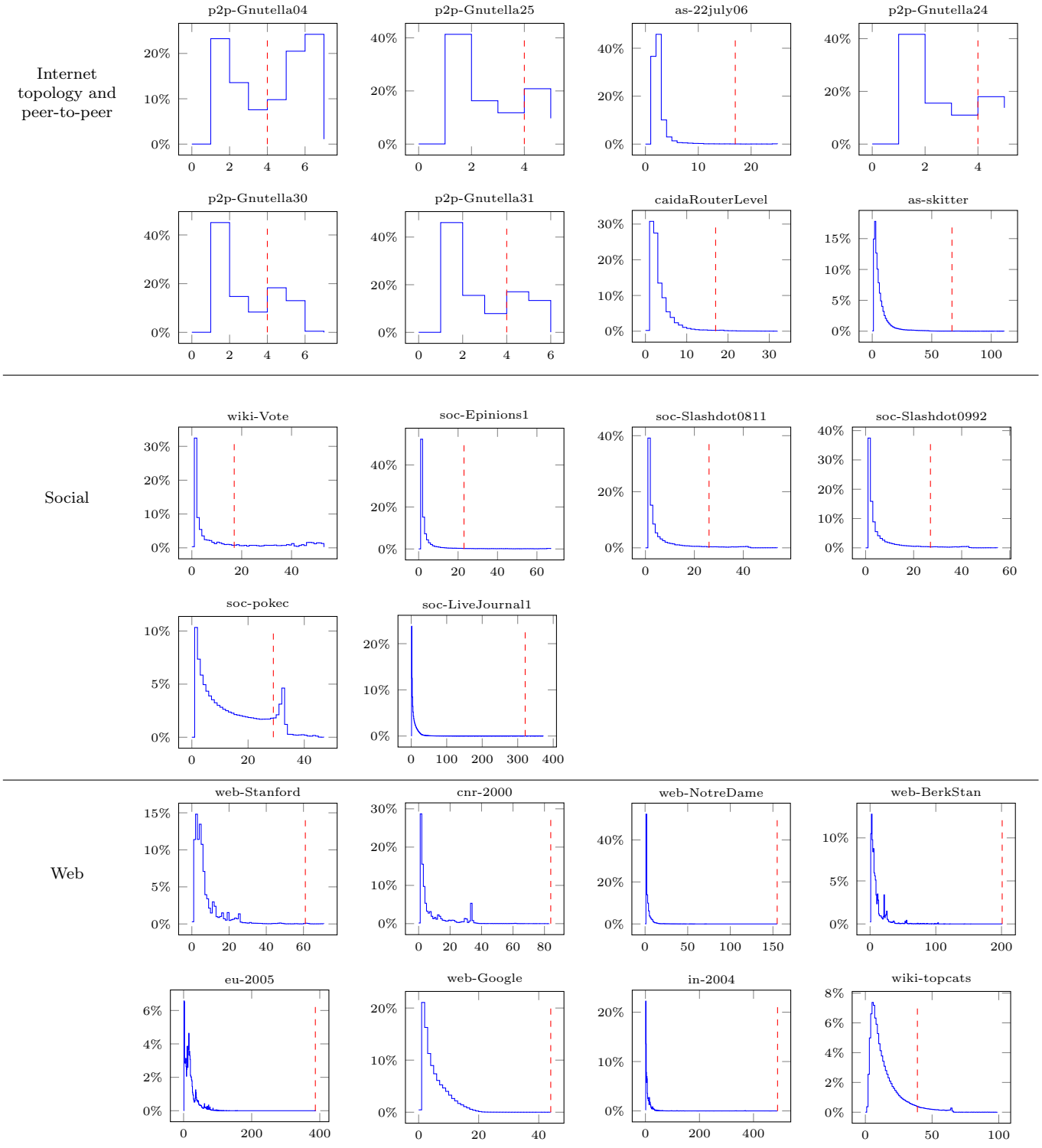
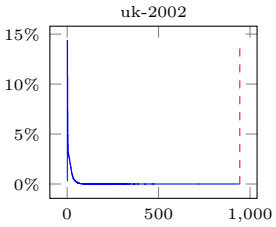
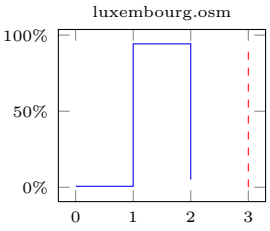
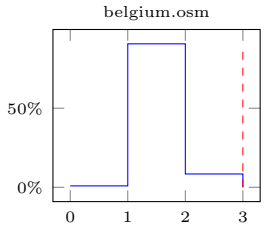


Figure 5: (Continued) Right-degree distribution of the different instances. The horizontal axis represents the right-degree and the vertical axis the percentage of vertices having such a right-degree. The red dashed line marks ω.

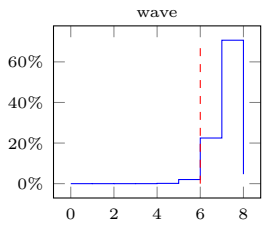


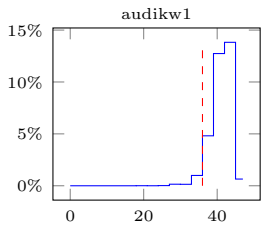
Road

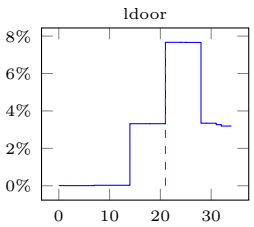


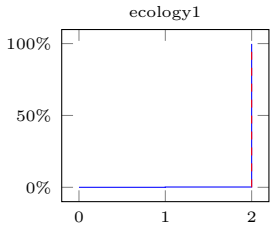


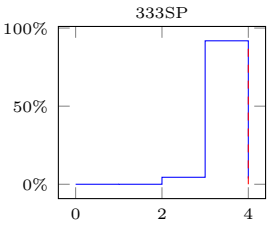
Sparse matrices

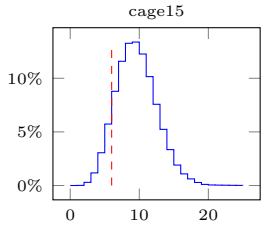




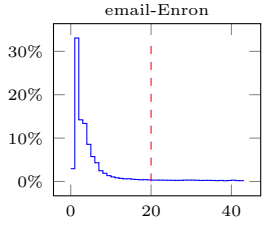


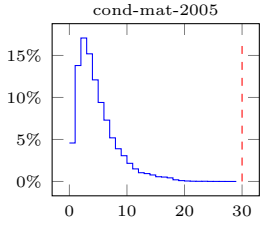


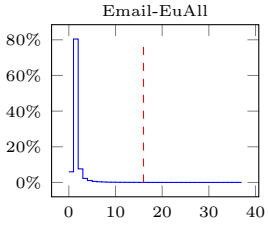


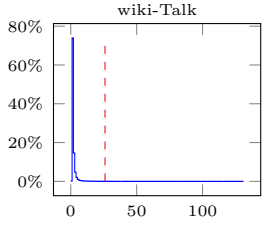


Communication









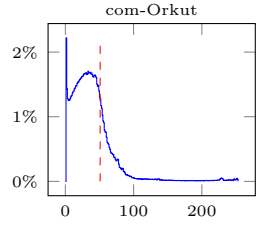


Figure 5: (Continued) Right-degree distribution of the different instances. The horizontal axis represents the right-degree and the vertical axis the percentage of vertices having such a right-degree. The red dashed line marks ω.