Noname manuscript No.
(will be inserted by the editor)

Exact Combinatorial Algorithms and Experiments for Finding Maximum k-Plexes

Hannes Moser, Rolf Niedermeier, and
Manuel Sorge

the date of receipt and acceptance should be inserted later

Abstract We propose new practical algorithms to find maximum-cardinality k-plexes in graphs. A k-plex denotes a vertex subset in a graph inducing a subgraph where every vertex has edges to all but at most k vertices in the k-plex. Cliques are 1-plexes. In analogy to the special case of finding maximum-cardinality cliques, finding maximum-cardinality k-plexes is NP-hard. Complementing previous work, we develop exact combinatorial algorithms, which are strongly based on methods from parameterized algorithmics. The experiments with our freely available implementation indicate the competitiveness of our approach, for many real-world graphs outperforming the previously used methods.

Keywords Parameterized Algorithmics · Social Network Analysis · Biological Network Analysis · NP-Complete Graph Problems · Dense Subgraphs · s-Plexes · k-Dependent Sets

1 Introduction

To efficiently find maximum-cardinality cliques in graphs is a major challenge for algorithmic graph theory and accompanying algorithm engineering efforts (cf. DIMACS challenge (DIMACS, 1995)). The corresponding Maximum Clique problem is NP-hard and neither effective approximation nor parameterized approaches exist that allow for efficient algorithms with provable performance bounds. Hence, the use of heuristic

A preliminary version of this paper titled "Algorithms and experiments for clique relaxations—finding maximum s-plexes" appeared in the Proceedings of the 8th International Symposium on Experimental Algorithms (SEA 2009), June 3–6, 2009, Dortmund, Germany, volume 5526 of Lecture Notes in Computer Science, pages 233–244, Springer, 2009. Following a referee's advice, we replaced "s-plexes" by the more common term "k-plexes". The main work was done while all authors were with Friedrich-Schiller-Universität Jena.

Hannes Moser
E-mail: hannes.moser@uni-jena.de

Rolf Niedermeier, Manuel Sorge
Institut für Softwaretechnik und Theoretische Informatik, TU Berlin
E-mail: rolf.niedermeier@tu-berlin.de, manuel.sorge@campus.tu-berlin.de

approaches always has been an important tool for practical solutions of Maximum Clique. The concept of cliques, however, has been criticized for its overly restrictive nature asking for *complete* subgraphs. A more relaxed concept of a dense subgraph has been introduced by Seidman and Foster (1978) with the notion of k-plexes. A 1-plex is the same as a clique. For $k \geq 1$, a *k-plex* of a graph $G = (V, E)$ is a vertex set $S \subseteq V$ such that in the induced subgraph $G[S]$ every vertex has degree at least $|S| - k$. Unfortunately, finding maximum-cardinality k-plexes turns out to be computationally basically as hard as clique detection is (Balasundaram et al., 2009; Komusiewicz et al., 2009). Thus, recently the development of practical (heuristic) algorithms for k-plex detection has received some interest (Balasundaram et al., 2009; McClosky and Hicks, 2010; Wu and Pei, 2007). In this work, we contribute novel tools for the efficient detection of maximum-cardinality k-plexes. Other than the cited previous work (where Wu and Pei (2007) deal with k-plex *enumeration*), our algorithms draw on methods from parameterized algorithmics (Niedermeier, 2006).

The decision problem Maximum k-Plex for an integer $k \geq 1$ is defined as follows.

Maximum k-Plex
Input: An undirected graph $G = (V, E)$ and a nonnegative integer p.
Question: Is there a k-plex $S \subseteq V$ of size at least p?

In our experiments we actually choose to maximize the value of p. Recent work on clique finding has exploited the close connection (indeed, duality) between Maximum Clique and the Minimum Vertex Cover problem (Abu-Khzam et al., 2004, 2007; Chesler et al., 2005). We follow the same spirit here and make use of the duality between Maximum k-Plex and the Bounded-Degree-d Vertex Deletion problem (d-BDD for short):

Bounded-Degree-d Vertex Deletion
Input: An undirected graph $G = (V, E)$ and a nonnegative integer p.
Question: Is there a vertex set $S \subseteq V$ of size at most p making $G[V \setminus S]$ a graph of maximum degree d?

Clearly, we are interested in minimizing the value p. The point is that an n-vertex graph has a k-plex of size p if and only if its complement graph has a solution set for d-BDD of size $n - p$ with $d := k - 1$. We experimentally exploit this close connection by making use of fixed-parameter tractability results for d-BDD (Fellows et al., 2010; Komusiewicz et al., 2009) and adding some new results.

Known Results. The k-plex concept was introduced by Seidman and Foster (1978) in the context of social network analysis. Maximum k-Plex is NP-complete, which follows directly from the duality between Maximum k-Plex and d-BDD; d-BDD is NP-complete due to a general framework by Lewis and Yannakakis (1980). Balasundaram et al. (2009) also gave a direct NP-completeness proof by reduction from Maximum Clique. Concerning parameterized complexity, it is known that Maximum k-Plex is W[1]-hard with respect to the parameter p (Komusiewicz et al., 2009). There is also related work in the context of graph-based data clustering employing the k-plex concept (Guo et al., 2010; van Bevern et al., 2011).

Balasundaram et al. (2009) presented a 0/1 integer linear program for Maximum k-Plex, generalizing a known formulation for the special case Maximum Clique. In addition, they carried out a polyhedral study of the problem and discussed a branch-and-cut implementation as the basis of computational tests. McClosky and Hicks (2010) described combinatorial algorithms for Maximum k-Plex, both of heuristic (without

provable guarantees on the solution quality) and exact nature. Their heuristic algorithms are based on certain upper and lower bounds for vertex coloring and their exact algorithms are based on adapting known algorithms for Maximum Clique. Independently, Trukhanov (2008) performed similar studies as part of his PhD thesis. All these authors implemented their algorithms and did computational experiments with real-world and artificial graph instances. These three experimental studies serve as comparison points with our implementation. Wu and Pei (2007) gave an algorithm to *enumerate* all maximal k-plexes in a graph, also accompanied by experimental studies.

As mentioned before, an alternative route to solving Maximum k-Plex is to do a "detour" via d-BDD in the complement graph. This is our approach, which, thus, can also be interpreted as work on d-BDD. Concerning d-BDD, Nishimura et al. (2005) presented a depth-bounded search tree yielding a solving algorithm running in $O((d+p)^{p+3} \cdot p + n \cdot (d+p))$ time. Subsequently, an improved simple search tree algorithm running in $O((d+2)^p \cdot (d+p)^2 + n \cdot (d+p))$ time was described (Komusiewicz et al., 2009). Moreover, there is a generalization of a local optimization algorithm for Vertex Cover by Nemhauser and Trotter (1975), which yields an almost linear problem kernel for d-BDD, that is, a problem kernel of $O(p^{1+\epsilon})$ vertices for any constant $\epsilon > 0$ (Fellows et al., 2010).[1] Very recently, Chen et al. (2010) presented a $37p$-vertex problem kernel and a size-$O(3.24^p)$ search tree for the special case 2-BDD. The search tree and kernelization results all imply the fixed-parameter tractability of d-BDD with respect to the parameter p in case of constant d-values. Interestingly, this is the best one can hope for because in case of unbounded d-values the problem is W[2]-complete (Fellows et al., 2010).

Finally, we remark that the task to generate a maximum-degree-d graph by a minimum number of vertex deletions is equivalent to finding a so-called d-dependent set of maximum size (Djidev et al., 1992; Dessmark et al., 1993). In recent work (Balasundaram et al., 2010) this is also referred to as the problem of finding maximum-cardinality co-$(d+1)$-plexes.

Our Contributions. On the theoretical side, we provide an improved depth-bounded search tree for 1-BDD (the search tree has size $O(2.31^p)$ instead of previously $O(3^p)$ (Komusiewicz et al., 2009)) and an algorithm for 1-BDD based on iterative compression (exponential factor 2^p). By duality, these algorithms can be used for finding 2-plexes. A side result—perhaps of independent interest—is a linear-time algorithm for Maximum Matching for bipartite graphs in which the vertices of one partite set have degree at most two. We also present a quickly computable $O(p(p+d)d)$-vertex problem kernel for d-BDD—the running time is $O(n \cdot (p+d))$. Moreover, we provide several very effective heuristics (still yielding optimal solution sets) helping to significantly boost the performance of the underlying fixed-parameter algorithms in applications. We perform a number of computational studies, comparing with previous work (Balasundaram et al., 2009; McClosky and Hicks, 2010; Trukhanov, 2008) on exact solutions for k-plex finding which mainly rely on integer linear programming and branch-and-bound. For several real-world graphs, we mostly achieved speedups by orders of magnitude when compared to the previous work. Our corresponding software (implemented using Objective Caml) is open source and freely available (see Section 5).

[1] The original statement of the result in the conference version (Fellows et al., 2009) is flawed. A corrected version with the results as described here is given in (Moser, 2009) and in the journal version (Fellows et al., 2010).

Organization of the Paper. In Section 2, we provide some concepts and notions used throughout the work. Then, we describe the general algorithmic approach to find maximum k-plexes by solving d-BDD (Section 3). After that, we give improved algorithms for 1-BDD (Section 4). Then, we describe our implementation (Section 5) and the corresponding experiments (Section 6). We conclude the paper with a brief outlook (Section 7).

2 Preliminaries

In this paper, all graphs are simple and undirected. Throughout our work, n denotes the number of vertices in a graph. For a graph $G = (V, E)$ and a vertex set $S \subseteq V$, we write $G[S]$ to denote the graph induced by S in G, that is, $G[S] := (S, \{e \in E \mid e \subseteq S\})$. For a vertex $v \in V$, we write $G - v$ instead of $G[V \setminus \{v\}]$ and for a vertex set $S \subseteq V$ we write $G - S$ instead of $G[V \setminus S]$. We define $N(v) := \{u \in V \mid \{u, v\} \in E\}$, $N[v] := N(v) \cup \{v\}$; the *degree* of a vertex v is $|N(v)|$. If every vertex in G has degree at most d, then we say that G has *maximum degree* d. A vertex set $S \subseteq V$ is a *bdd-d-set* if $G - S$ has maximum degree d. For a graph $G = (V, E)$, an edge subset $M \subseteq E$ is called a *matching* if the edges in M are pairwise disjoint. A matching M is *maximal* if there exists no edge $e \in (E \setminus M)$ such that $M \cup \{e\}$ is a matching. A matching M is *maximum* if there exists no larger matching. The Maximum Matching problem is to compute a maximum matching in a given graph.

A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$. A parameterized problem L is *fixed-parameter tractable* if it can be decided in $f(p) \cdot |I|^{O(1)}$ time whether $(I, p) \in L$, where f is a computable function depending only on the parameter p (Downey and Fellows, 1999; Flum and Grohe, 2006; Niedermeier, 2006). Problem kernelization is a core tool to develop parameterized algorithms (Bodlaender, 2009; Guo and Niedermeier, 2007; Hüffner et al., 2008; Niedermeier, 2006). A kernelization is often described with a set of *data reduction rules* that are applied to the problem instance (I, p) and that change this instance into a smaller instance I' with parameter $p' \leq p$ in polynomial time such that (I, p) is a yes-instance if and only if (I', p') is a yes-instance. An instance to which none of a given set of data reduction rules applies is called *reduced* with respect to the rules. We call a reduced instance *problem kernel* if it has size $f(p)$ for a function f depending only on p. We also employ search trees for our fixed-parameter algorithms. Search tree algorithms work in a recursive manner. The number of recursion calls is the number of nodes in the according tree. This number is governed by linear recurrences with constant coefficients. These can be solved by standard mathematical methods (Niedermeier, 2006). If the algorithm solves a problem instance of size s and calls itself recursively for problem instances of sizes $s - d_1, \dots, s - d_i$, then $(d_1, \dots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence $T_s = T_{s-d_1} + \cdots + T_{s-d_i}$ for the asymptotic size T_s of the overall search tree.

3 Basic Algorithms

Our approach to compute maximum-cardinality k-plexes makes use of the duality between Maximum k-Plex and d-BDD (see Section 1). Therefore, the first step is to

compute the complement of the input graph.[2] Then, the second step is to solve d-BDD on this complement graph for $d := k - 1$. Finally, the minimum bdd-d-set is translated back into a maximum k-plex in the input graph.

In the following, we assume that graph G is the input to d-BDD. We suppose that the parameter p denoting the number of allowed vertex deletions is given (see Section 5 for details on how p is obtained). Our main algorithm to solve d-BDD uses a bounded search tree and polynomial-time data reduction rules interleaving with the search tree. In general, the branching strategy of the search tree algorithm chooses a vertex v of degree at least $d+1$, and then branches into the subcases of deleting v and every possibility of deleting all but d neighbors of v. We refer to this by saying that the strategy *branches on v and $N(v)$*. In practice, it is favorable to delete many vertices in each branching step, that is, v should be a vertex of high degree.

In the next subsections, we first describe the data reduction rules applied in each search tree node (Subsection 3.1) and then the search tree algorithm itself (Subsection 3.2).

3.1 Data Reduction Rules

The best known problem kernel for d-BDD in terms of (asymptotic) kernel size is an "almost linear" problem kernel of $O(p^{1+\epsilon})$ vertices for any constant $\epsilon > 0$ (Fellows et al., 2010). However, preliminary experiments showed that the corresponding kernelization algorithm often fails to significantly reduce the graphs we considered (basically due to the constant in the kernel size and because the parameter p is not very small). For this reason, we use a modified heuristic version of the corresponding data reduction rule. Before getting into more detail, we describe some further simple data reduction rules.

The following algorithm comprises several data reduction rules and maintains a set S which is used to store the vertices that are assumed to be in an optimal bdd-d-set.

Reduction Rule 1 (high-degree rule) *If a vertex v has degree more than $d+p$, then delete v from G, add v to S, and decrease p by one.*

For the correctness of this rule, observe that $v \notin S$ would imply that more than p neighbors of v have to be in S, a contradiction, since $|S| \leq p$.

Reduction Rule 2 (low-degree rule) *If there is a vertex v such that each vertex in $N[v]$ has degree at most d, then delete v from G.*

This rule is obviously correct as no minimum-cardinality solution would contain v, and the neighbors of v still have bounded degree d after the deletion of v.

Reduction Rules 1 and 2 can be used to show a simple quadratic-vertex problem kernel. Note that the running time is "almost linear", whereas the known $O(p^{1+\epsilon})$-vertex kernel for any constant $\epsilon > 0$ only runs in $O(n^4 \cdot m)$ time (Fellows et al., 2010).

Theorem 1 Bounded-Degree-d Vertex Deletion *admits a problem kernel containing $O(p \cdot (p+d) \cdot d)$ vertices, which can be computed in $O(n \cdot (p+d))$ time.*

[2] Note that we avoid a potentially quadratic blow-up of the edge number in the complement graph by only simulating rather than constructing it.

Proof Consider a graph G that is reduced with respect to Reduction Rule 1 and Reduction Rule 2. Assume that S is a size-p bdd-d-set of the reduced graph. Due to Reduction Rule 1, each vertex has degree at most $d+p$. Therefore, $|N(S)| \leq p(d+p)$. Since S is a bdd-d-set, each vertex in $D := N(S)$ has degree at most d in $G-S$. Thus, for $F := N(D) \setminus S$ we know that $|F| \leq p(d+p)d$. Due to Reduction Rule 2, there are no further vertices outside $S \cup D \cup F$, that is, $S \cup D \cup F$ contains all vertices of the reduced instance, yielding the claimed bound $p + p(d+p) + p(d+p)d = O(p(p+d)d)$ on the number of vertices in the reduced instance.

To show the running time, we use that a yes-instance contains at most $n(p+d)$ edges. Assume for the purpose of contradiction that (G, p) is a yes-instance and G contains more than $n(p+d)$ edges. Then, G contains an induced subgraph of minimum degree $d+p+1$: simply delete all vertices of degree at most $d+p$ from G. This removes at most $n(d+p)$ edges and, hence, after the deletion of all vertices of degree at most $d+p$, there remains an induced subgraph of G of minimum degree $d+p+1$. In this subgraph of minimum degree $d+p+1$, assuming that any vertex is not in a bdd-d-set S implies that more than p of its neighbors have to be in S. Thus, there is no bdd-d-set of size at most p in this graph, and the graph G is a no-instance.

Next, we show the running time. First, the algorithm tests whether $|E(G)| > n(p+d)$, and, if so, aborts and returns "no-instance". The test can be performed in $O(n(p+d))$ time. Otherwise, proceed by iterating over all vertices and deleting them if they have degree greater than $p+d$. Hence, it takes $O(|V(G)|+|E(G)|) = O(n(p+d))$ time in total in order to apply Reduction Rule 1.

Likewise, iterating over all vertices, checking the condition in Reduction Rule 2, and deleting the corresponding vertices, can be done in $O(n(p+d))$ time. □

We did not include Reduction Rule 2 in the implementation as preliminary experiments showed that it has almost no effect in practice or may even slow down the algorithm. Two more reduction rules we considered follow.

Reduction Rule 3 (degree-one rule) *If a vertex $v \in V$ has at least $d+1$ degree-one neighbors, then delete v from G, add v to S, and decrease p by one.*

Concerning the correctness of Reduction Rule 3, observe that at least v or one of its degree-one neighbors has to be in an optimal solution. If a degree-one neighbor w of v is in an optimal bdd-d-set S but $v \notin S$, then one can simply remove w from S and add v to it, obtaining a bdd-d-set of the same size. Therefore, it is safe to assume that an optimal solution contains v. Moreover, an exhaustive application of Reduction Rule 3 only takes linear time. We omit the straightforward details.

The final data reduction rule, called *BDD-NT rule*, is a heuristic version of an $O(p^{1+\epsilon})$-vertex kernelization (Fellows et al., 2010). The core of this kernelization is the algorithm FindExtremal. The input for FindExtremal is a graph $G = (V, E)$ and a bdd-d-set X. For $d \leq 1$, if $|V \setminus X| > (d+1)^2 \cdot |X|$, then FindExtremal returns in $O(n^3 \cdot m)$ time two vertex subsets $A' \subseteq X$ and $B' \subseteq V \setminus X$. The sets A' and B' have the property that for vertices in A' it can already be decided to put them into the solution set, and vertices in B' can be ignored for finding a solution and B' is not empty (thus, we have a guarantee that a data reduction rule based on FindExtremal will successfully reduce the graph). Thus, assuming that X is given, one can simply test whether the condition $|V \setminus X| > (d+1)^2 \cdot |X|$ is fulfilled, and if so, apply FindExtremal. The resulting set A' can be safely assumed to be in a minimum-cardinality bdd-d-set of G, and we can add A' to the solution set S and

Algorithm: BddReduction (G, X, p)
Input: A graph $G = (V, E)$, a bdd-d-set X for G, and an integer $p \geq 0$.
Output: A reduced instance (G, p), a modified bdd-d-set X for G, a set S of vertices that have already been deleted by some data reduction rule.

```
 1 S ← ∅
 2 repeat
 3     while ∃v ∈ V : deg(v) > d + p                          ▷ High-degree rule
 4         G ← G − v; X ← X \ {v}; S ← S ∪ {v}; p ← p − 1
 5     while ∃v ∈ V : v has at least d + 1 deg-1 neighbors     ▷ Degree-1 rule
 6         G ← G − v; X ← X \ {v}; S ← S ∪ {v}; p ← p − 1
 7     if |N(X)| > (d + 1) · |X| then
 8         call BDD-NT rule to obtain vertex sets A′ and B′
 9         G ← G − (A′ ∪ B′); X ← X \ A′; S ← S ∪ A′; p ← p − |A′|
10 until none of the rules applies.
11 return G, p, X, S
```

Fig. 1: Pseudo-code of the basic algorithm exhaustively applying the data reduction rules.

delete A' from G, decreasing the parameter by $|A'|$. The vertices in B' do not have to be considered and can be deleted from the graph. Basically, the same principle works for $d \geq 2$ using the condition $|V \setminus X| = \omega(|X|^{1+\epsilon})$.

Unfortunately, for the graphs we experimented with (see Section 6), the constant hidden in the O-notation of the kernel size bound turns out to be too big. The larger constant compared to the case $d \leq 1$ is due to the fact that FindExtremal is based on stars[3] with $d+1+\lceil |X|^{\epsilon} \rceil$ leaves for $d \geq 2$ instead of $d+1$ leaves as in the case $d \leq 1$. However, referring to Fellows et al. (2010) for the details, FindExtremal can compute two sets A' and B' with the above-mentioned properties also if one only uses stars with $d+1$ leaves for $d \geq 2$. The drawback of such stars is that then the returned set B' might be empty (and thus we have no provably effective reduction rule anymore). By preliminary experiments, we found out that in practice it is very likely that FindExtremal terminates outputting two non-empty sets A' and B' with the above-mentioned properties if $|N(X)| > (d + 1) \cdot |X|$, using stars with $d + 1$ leaves for any constant d. We used this approach for our experiments, although one can construct instances where this adapted heuristic version would not reduce the graph.

Recall that the main algorithmic approach is a search tree interleaving with the data reduction rules. For the interleaving with the search tree, we initially compute a $(d + 2)$-approximate solution X, and then start branching, always keeping X up-to-date, that is, if a vertex is deleted from the graph, then also delete it from X. Then, in each search tree node simply test whether $|N(X)| > (d + 1) \cdot |X|$, and, if so, then apply the modified FindExtremal as described above to compute A' and B' and use these sets to reduce the graph.

Figure 1 presents the pseudo-code of the implemented exhaustive data reduction based on the three stated data reduction rules. We apply the data reduction rules exhaustively in each search tree node in the given order. The reason for this order is that the high-degree rule turns out to be the most effective in terms of number of

[3] A *star* is a tree where all of the vertices but one are leaves.

deleted vertices and, at the same time, it can be implemented to run very efficiently. The other two rules are less effective in general, and the BDD-NT rule is rather expensive in terms of running time, so it is the last rule that is considered.

3.2 Search Tree Algorithm

As outlined in the beginning of the section, the algorithmic strategy is to apply a search tree algorithm interleaving with the above data reduction rules. Additionally, we use heuristic tricks to speed up the algorithm. These tricks are described in Subsection 3.2.1, followed by a pseudo-code description of the entire search tree algorithm in Subsection 3.2.2.

3.2.1 Heuristic Improvements

While branching, the search tree algorithm maintains a bdd-d-set X which is based on a $(d+2)$-approximate solution.[4] This vertex set X gives an upper bound on the size of a minimum-cardinality bdd-d-set. When a vertex is deleted in the course of the branching, the bdd-d-set X is updated accordingly. First, we describe a heuristic, called "guided branching", which tries to select vertices to branch on such that X becomes small very quickly in the course of the branching process. Second, we describe two heuristics used to compute lower bounds on the size of a minimum bdd-d-set.

Guided Branching. The guided branching heuristic aims to "guide" the search tree algorithm to select vertices to branch on such that a bdd-d-set X becomes small very quickly in the branching process. There are two reasons why X should be small:

1. The set X is an upper bound on the size of a minimum-cardinality bdd-d-set and can be used to speed up the search, e.g., by setting $p := |X|$ if $|X| < p$ in some search tree node.
2. The BDD-NT rule is based on X; the BDD-NT rule can only be effective if X is small compared to $N(X)$ (we only apply the BDD-NT rule if $|N(X)| > (d+1) \cdot |X|$, see also Figure 1).

In order to faster decrease the size of X, it can be useful to branch on v and only a subset of $N(v)$. To this end, for a vertex v of maximum degree we branch on v and $N(v) \cap X$ if $|N(v) \cap X|/|N(v)| > 0.9$ (the value 0.9 has shown good results in preliminary experiments); otherwise, we simply branch on v and $N(v)$. This "90%-condition" prevents that the algorithm branches on $N(v) \cap X$ if it is too small; in this case, the benefit of reducing only X does not outweigh the less effective branching on v and $N(v) \cap X$ compared with branching on v and $N(v)$.

Edge-Count Test. The *edge-count test* checks whether the given d-BDD instance is a no-instance. The test counts how many edges can be deleted from the graph $G = (V, E)$ by at most p vertex deletions based on the vertex degree distribution of the graph. If there are too few such edges, then the graph cannot be turned into a graph with maximum degree d by at most p vertex deletions. The number of edges m' that can

[4] This $(d+2)$-approximate solution can be found by greedily computing a maximal collection of vertex-disjoint copies of stars with $(d+1)$ leaves.

Algorithm: BddSolve (G, X, p)
Input: A graph $G = (V, E)$, a bdd-d-set X for G, and an integer $p \geq 0$.
Output: A minimum-cardinality bdd-d-set S for G with $|S| \leq p$, or "no-instance".

```
1  G, p, X, S ← BddReduction(G, X, p)                    ▷ Also see Figure 1
2  if p < 0 then return "no-instance"
3  l ← greedily computed lower bound on the size of a minimum bdd-d-set.
4  if p < l or edge-count test tells "no-instance" then return "no-instance"
5  if maximum degree of G is d then return S
6  Among all vertices of maximum degree, choose a vertex v.
7  if |N(v) ∩ X| > d and |N(v) ∩ X|/|N(v)| > 0.9 then
8        for all size (|N(v) ∩ X| − d)-subsets C ⊆ N(v) ∩ X do
9              call BddSolve (G − C, X \ C, p − |C|)      ▷ Branch on N(v) ∩ X
10       call BddSolve (G − v, X \ {v}, p − 1)                 ▷ ... and v.
11 else branch analogously to lines 9–10 on N(v) and v.
12 if all recursive calls of BddSolve returned "no-instance" then
13       return "no-instance"
14 else return S ∪ S', where S' is a smallest set returned by the BddSolve calls.
```

Fig. 2: Pseudo-code of the basic search tree algorithm incorporating data reduction rules to compute a minimum bdd-d-set.

be removed by p vertex deletions is computed by sorting the vertices of G by non-decreasing degree and summing up the degrees of the first p vertices in this order. Then, one tests whether $m - m' > dn/2$. If so, then (G, p) is a no-instance, since the minimum number of edges remaining in the graph after at most p vertex deletions is greater than the maximum number of edges that are allowed in an n-vertex graph of maximum degree d.

Lower Bound Heuristic. In order to derive a lower bound on the size of a bdd-d-set, we greedily compute a packing of vertex-disjoint stars with $d + 1$ leaves. Since an optimal bdd-d-set has to contain at least one vertex of each star, the number of stars is a lower bound.

3.2.2 Algorithmic Details

Figure 2 gives the pseudo-code of the basic search tree algorithm to compute a minimum bdd-d-set of size at most p for a graph, including the data reduction rules and the heuristic improvements. The algorithm branches at most p times yielding a worst-case search tree size of $O((d + 2)^p)$. This upper bound corresponds to the case that in each branching step the set $N' := N(v) \cap X$ has size $d + 1$ and the algorithm branches into the case of deleting v and into $d + 1$ cases of deleting a vertex in N'.

In each search tree node, we kernelize the instance before branching, using the simple $O(p^2)$-vertex problem kernel (Theorem 1). In total, this yields a running time of $O((d + 2)^p \cdot p^2 + n(p + d))$. An improved interleaving of the depth-bounded search tree with a problem kernel (see Niedermeier and Rossmanith (2000)) yields a running time of $O((d + 2)^p + n(p + d))$.

Theorem 2 Bounded-Degree-d Vertex Deletion *can be solved in* $O((d + 2)^p + n \cdot (p + d))$ *time.*

This is slightly better than the (very similar) previous best-known algorithm for d-BDD with $O((d+2)^p \cdot (d+p)^2 + n \cdot (d+p))$ running time (Komusiewicz et al., 2009), but which, in contrast to our algorithm, is also capable of enumerating all minimal solution sets.

4 Improved Algorithms for Bounded-Degree-1 Vertex Deletion

In this section, we describe improved algorithms for 1-BDD, which corresponds to the practically relevant special case Maximum 2-Plex (see Section 1).

4.1 Search Tree Algorithm

Compared to Theorem 2, we give a more refined branching strategy with an improved search tree size of $O(2.31^p)$. We refrain from conceivable further asymptotic improvements in order to keep the algorithm efficient and easy to implement.

Theorem 3 Bounded-Degree-1 Vertex Deletion *can be solved in* $O(2.31^p + pn)$ *time.*

Proof We start with considering a vertex v of degree $t > 1$. As a necessary condition to transform the graph into one with maximum vertex degree one, either v needs to be deleted or all but one of its neighbors. Let $N(v) = \{u_1, \dots, u_t\}$. We branch into the following $t+2$ subcases:

1. Delete v from G and decrease p by one.
2. Delete $N(v)$ from G and decrease p by $|N(v)|$.
3. For each $u_i \in N(v)$, $1 \le i \le t$, delete $N' := (N(v) \setminus \{u_i\}) \cup (N(u_i) \setminus \{v\})$ from G and decrease p by $|N'|$.

The correctness of this branching can be seen as follows. First, clearly in each subcase v either gets deleted (case 1) or it gets maximum degree one (degree zero in case 2 and degree one in case 3). Second, the branching covers all possibilities how v can be made a maximum-degree-one vertex: one can keep at most one vertex from $N(v)$ (case 3), the rest has to be deleted. If u_i is the neighbor that shall not be deleted, then clearly all vertices from $N(v) \setminus \{u_i\}$ have to be deleted and all neighbors of u_i except for v (that is, $(N(u_i) \setminus \{v\})$ have to be deleted. The case of deleting all of $N(v)$ (case 2) also needs to be considered, otherwise one would miss the situation that all of v's neighbors have to be deleted for reasons lying outside the neighborhood of v. One obtains a branching into $t+2$ cases with the corresponding branching vector

$$(1, t, t-1+|N(u_1) \setminus N[v]|, \dots, t-1+|N(u_t) \setminus N[v]|).$$

It is not hard to check[5] that the worst-case branching vector occurs for $t = 2$ and $|N(u_1) \setminus N[v]| = |N(u_2) \setminus N[v]| = 1$, yielding $(1, 2, 2, 2)$ with the branching number 2.31. In analogy to the search tree algorithm for d-BDD for general d (Subsection 3.2.2) we can interleave the search tree with problem kernelization, which results in $O(2.31^p + pn)$ running time in total. □

[5] We omit some details here; basically, one can argue that for $t = 2$ cases where $|N(u_1) \setminus N[v]| = 0$ are actually easier (often avoiding branching at all) and $t > 2$ gives branching vectors with smaller branching numbers.

4.2 Iterative Compression

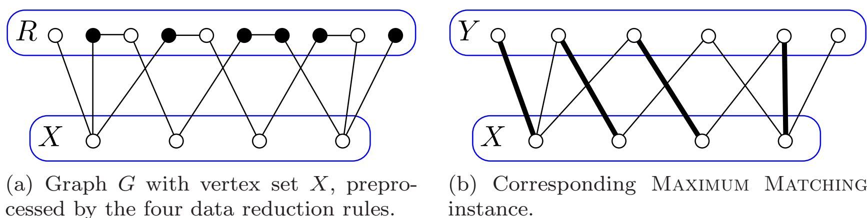
In this section, we present an asymptotically improved algorithm for 1-BDD using the iterative compression technique (see (Guo et al., 2009) for a survey on this technique) running in $O(2^p \cdot p^2 + pn)$ time.

The general idea behind our iterative compression is as follows. Start with $V' = \emptyset$ and $S = \emptyset$; clearly, S is a bdd-1-set for $G[V']$. Iterating over all graph vertices, step by step add one vertex $v \notin V'$ from V to both V' and S. Then, S is still a bdd-1-set for $G[V']$, although possibly not a minimum one. One can, however, obtain a minimum one by applying a compression routine. It takes a graph G and a bdd-1-set S for G, and returns a minimum-cardinality bdd-1-set for G. Since eventually $V' = V$, one obtains an optimal solution for G once the algorithm returns S. Hence, the main task is to design a compression routine. Consider a smaller bdd-1-set S' as a modification of a larger bdd-1-set S for the graph $G = (V, E)$. This modification retains some vertices $Y \subseteq S$ as part of the solution set, while the other vertices in $X := S \setminus Y$ are replaced by new vertices X' from $V \setminus S$, where $|X'| < |X|$. The idea is to try by brute force all $2^{|S|} - 1$ nontrivial partitions of S into these two sets Y and X. For each such partition, the vertices from Y are immediately deleted, since we already decided to put them into the smaller bdd-1-set. Now, the task is either to find a bdd-1-set X' of size less than $|X|$ for the remaining graph, where X' and X are disjoint, or to prove that no such X' exists. We call this remaining task Disjoint Compression Task. One requirement for the existence of a disjoint bdd-1-set X' is that $G[X]$ has maximum degree one. (If $G[X]$ has not, every bdd-1-set for G has to contain a vertex of X, thus there is no disjoint bdd-1-set.) This requirement can be verified in linear time. Therefore, in the following we assume that it is satisfied.

For the compression routine, let $G = (V, E)$ and $X \subseteq V$ with $G - X$ being a graph of maximum vertex degree one. Let $R := V \setminus X$. The general outline of the compression routine reads as follows. First, compute a set $X_1 \subseteq R$ of vertices that necessarily belong to the disjoint solution by applying a series of polynomial-time executable data reduction rules. Then, compute a set $X_2 \subseteq (R \setminus X_1)$ such that $X' := X_1 \cup X_2$ forms a minimum-cardinality solution. If in this process one encounters a situation showing that the given instance has no solution, then return "no-instance". Next, we describe the four data reduction rules. Initially, set $X_1 := \emptyset$.

1. For each edge $\{u, v\}$ in $G[X]$, set $X_1 := X_1 \cup (R \cap (N(u) \cup N(v)))$, delete u and v from G and X, and delete $R \cap (N(u) \cup N(v))$ from G and R.
2. Delete each vertex $u \in R$ that is adjacent to more than one vertex in X from G and R and set $X_1 := X_1 \cup \{u\}$.
3. For each edge $\{u, v\}$ in $G[R]$ such that $N(\{u, v\}) = \{w\}$, choose a vertex $x \in \{u, v\}$ that is adjacent to w, set $X_1 := X_1 \cup \{x\}$, and delete x from G and R.
4. Delete isolated vertices in R and isolated edges in $G[R]$ from G and R.

The correctness of the first two data reduction rules is due to the fact that each P_3 (that is, a three-vertex path) intersecting with X in two vertices can only be destroyed if the third vertex (from R) is in X_1. The third reduction rule is correct because the only neighbor of $\{u, v\}$ is the vertex in $N(\{u, v\}) = \{w\} \subseteq X$, thus $\{u, v, w\}$ induces either a triangle (K_3) or a P_3. As a consequence, it is optimal to add to X_1 a vertex from $\{u, v\}$ that is adjacent to w. The fourth reduction rule is obviously correct, as an optimal solution would never contain isolated vertices or a vertex from an isolated edge.



(a) Graph G with vertex set X, preprocessed by the four data reduction rules.

(b) Corresponding Maximum Matching instance.

Fig. 3: Reduced Disjoint Compression Task instance (a) together with a corresponding Maximum Matching instance (b). The bold edges in the Maximum Matching instance form a maximum matching M. The black vertices in G are the vertices of a minimum-cardinality bdd-1-set X_2, $X_2 \cap X = \emptyset$, corresponding to M.

These reduction rules can be applied exhaustively in $O(n + m)$ time if they are applied in the given order: To apply the first reduction rule, iterate through each edge in $G[X]$ and delete its neighbors in R, taking $O(n + m)$ time in total. Then, for the second rule, simply check for each vertex u in R whether there are at least two neighbors in X, and if so, delete u, which takes $O(n+m)$ for all vertices in R. To apply the third rule, iterate over each edge $\{u, v\}$ in $G[R]$ and test whether the neighborhood of $\{u, v\}$ contains only one vertex; if so, either u or v is deleted. In total, this takes $O(n + m)$ time. The fourth rule clearly takes $O(n+m)$ time. It is important to observe that after one particular reduction rule has been exhaustively applied, there can never again occur a situation where the same rule could be applied again. This guarantees that after applying the rules in their given order (the order is important, e.g., applying the fourth rule before the others could lead to an instance that contains isolated vertices or edges), the instance is reduced with respect to the four data reduction rules.

After these rules have been exhaustively applied, if $|X_1| > |X|$, then stop and return "no-instance". Otherwise, compute a minimum-cardinality set X_2 such that $X_1 \cup X_2$ is a minimum-cardinality bdd-1-set as follows.

In the following, assume that G, X, and R are reduced with respect to the above data reduction rules, that is, none of the rules can be applied anymore. The following properties of G, X, and R are important for the subsequent arguments.

- P1 The graph $G[R]$ consists of isolated vertices and edges (because X is a bdd-1-set),
- P2 X forms an independent set (first reduction rule),
- P3 each vertex in R is adjacent to exactly one vertex in X (reduction rules two, three, and four), and
- P4 each vertex in X is adjacent to at most one endpoint of each edge in $G[R]$ (P3 and third reduction rule).

An example of an instance with these properties is given in Figure 3a. Observe that for each edge in $G[R]$ at least one of its endpoints must belong to the new solution X' in order to obstruct all P_3's. Moreover, for each vertex $v \in X$, all but at most one neighbor must belong to X'. An optimal solution fulfilling these constraints can be easily found by reduction to Maximum Matching in bipartite graphs; the corresponding Maximum Matching instance is constructed by contracting every edge in R, and a maximum matching M in that instance directly corresponds to a solution X_2 in G (see Figure 3b for an example). Obviously, the input instance (G, p) is a yes-instance for 1-BDD if and only if $|X_1| + |X_2| \leq |X|$.

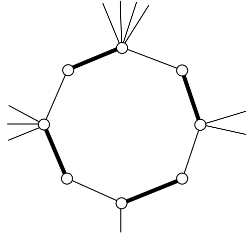


Fig. 4: A cycle C with vertices of degree two ("inner vertices") and vertices of degree at least two ("exits"). In this example, any maximum matching M can contain at most four exits, and any matching edge that contains an inner vertex also contains an exit. Hence, taking every second edge on C into the matching is always optimal. The bold edges show such a matching on C.

It remains to show the running time to solve the matching instance. Observe that in the Maximum Matching instance the vertices in one partite set have maximum degree two. Such an instance can be solved in linear time:

Lemma 1 *In a bipartite graph B with partite vertex sets X and Y such that the vertices in Y have maximum degree two,* Maximum Matching *can be solved in linear time.*

Proof The following is a description of an algorithm that finds a maximum matching M in B. It starts with an empty set M and then adds edges to it as follows.

First of all, one can safely assume that an edge incident to a degree-one vertex is always contained in a maximum matching. Therefore, in a first step, find in linear time all degree-one vertices. Then, repeat the following until there are no more degree-one vertices. Choose a degree-one vertex u, add its incident edge $\{u, v\}$ to M, and remove u and v from B. Removing v may imply that some of its neighbors obtain degree one. Hence, the set of degree-one neighbors is updated accordingly. The whole process of removing all degree-one vertices takes linear time since it takes $O(\deg(v))$ time to update the set of degree-one neighbors if v is removed.

The remaining graph has minimum degree two. Hence, each connected component contains a cycle. The connected components and a cycle in each of it can be found in linear time with depth-first search: Traverse a connected component in depth-first order, constructing a depth-first tree. If a vertex is discovered twice in the traversal, a cycle has been found and can be explicitly constructed by backtracking on the branches of the depth-first tree. A vertex is discovered twice in a depth-first search if and only if there is a cycle in the graph and the process of backtracking takes linear time.

From now on, let B' be a connected component with partite vertex sets $X' \subseteq X$ and $Y' \subseteq Y$ and let C be a cycle in B'. Obviously, C has even length because B' is bipartite. Since the graph has minimum degree two, the vertices in Y' have degree exactly two, and, therefore, every second vertex on C has degree exactly two. Moreover, every vertex in $D := N(V(C))$ is from Y' and has degree two. There exists a maximum matching M that contains every second edge on C; any maximum matching M' without this property can be easily converted into a maximum matching of the same size with this property (see Figure 4). Thus, we add every second edge on C to M and remove $V(C)$ from B'. After that, the neighbors D of the removed cycle have

degree one, and, therefore, we can apply the process above of removing all degree-one vertices. This process will remove the whole connected component B' for the following reason. Let $v \in D$ and let w be its neighbor (we assume that C has been removed). Since $v \in Y'$, it follows that $w \in X'$, and since the process of removing degree-one vertices will remove w, the degree-two neighbors of w will obtain degree one. By applying this argument inductively, it is clear that the process of removing degree-one vertices will remove every vertex w' for which there exists a path between v and w' in B'. Thus, eventually, all vertices in the connected component B' are removed. Clearly, deleting $V(C)$ takes linear time, and, as explained above, the subsequent removal of degree-one vertices also takes linear time. □

Using this linear-time algorithm to solve the Maximum Matching instance, the Disjoint Compression Task for 1-BDD can be solved in linear time.

Theorem 4 Bounded-Degree-1 Vertex Deletion *can be solved in* $O(2^p \cdot p^2 + pn)$ *time.*

Proof By Theorem 1, we obtain an $O(p^2)$-vertex problem kernel for 1-BDD (with $O(p^4)$ edges) in $O(pn)$ time. To this kernel, we apply the BDD-NT-Rule and obtain an $O(p)$-vertex problem kernel for 1-BDD (with $O(p^2)$ edges) in $O(p^{12})$ time. We apply the above iterative compression algorithm to this problem kernel. This means that we have $O(p)$ iterations, each taking $O(2^p \cdot p^2)$ time. Herein, the factor $O(2^p)$ derives from trying all partitions of X into two subsets. The resulting total running time is $O(pn + p^{12} + 2^p \cdot p^2) = O(2^p \cdot p^2 + pn)$. □

To the best of our knowledge, 1-BDD provides the first nontrivial application of iterative compression where the corresponding Disjoint Compression Task is *linear-time* solvable. For other related problems (like Cluster Vertex Deletion) whose corresponding Disjoint Compression Tasks are polynomial-time solvable, it seems to be much more difficult to reach linear-time solvability (Hüffner et al., 2010).

5 Implementation with some Speedup Tricks

Our implementation is written in the functional programming language Objective Caml.[6] A reason for this choice was the availability of a purely functional graph data structure. This data structure makes the implementation of a search-tree-based algorithm much easier, since we do not have to care about undoing changes to the data structure that were applied in other search tree branches. Moreover, it is a stated (and usually achieved) goal of the Objective Caml developers that Objective Caml code runs at most twice as slow as code generated by a decent C compiler. Since we are dealing with exponential-time algorithms, algorithmic improvements usually lead to time savings that cannot be bounded by any constant factor, so such a constant factor seems of lesser importance concerning a qualitative assessment of performance. Our implementation is open source and it is freely available.[7]

Concerning the initial $(d+2)$-approximate solution X needed for the guided branching, it turned out that a greedy solution, computed by simply taking a vertex of highest degree into the solution until the remaining graph has bounded degree d, very often

[6] See `http://caml.inria.fr/`

[7] `http://theinf1.informatik.uni-jena.de/splex/`

was smaller than a $(d+2)$-approximate solution, although this method does not provably guarantee an approximation factor of $d+2$. Such a greedy solution is constructed at the beginning of the computation (before invoking the search tree algorithm), and its size is taken as the initial value of p (the maximum number of allowed vertex deletions). Our implementation contains many algorithmic tweaks that are not covered by the basic description from Figure 2 (Section 3). For instance, the effect of the guided branching can be improved by recomputing X from time to time in the course of the branching process. Moreover, it improves performance significantly if one updates the value of p when a branch has found a solution smaller than the initial p. For 1-BDD (that is, MAXIMUM 2-PLEX), we implemented the improved branching described in Subsection 4.1 instead of the branching shown in Figure 2.

In the following, we comment about some particularities of our search tree implementation. One of the most important issues was the computation of the complement graph (to transform between d-BDD and MAXIMUM k-PLEX, see Section 1), which has to be performed before executing the BDDSOLVE algorithm (Figure 2). For sparse graphs, the complement graph is dense and in practice the amount of time and memory to compute it exceeds often the time and memory needed for finding a maximum k-plex. Therefore, we implemented a wrapper that simulates a complement graph rather than actually computing it, which turned out to be much more efficient.

For the graphs we considered, it turned out that applying the data reduction rules (see Figure 1) in every search tree node yields the best results. In particular, the degree-one rule and the high-degree rule are mostly very effective.

We did not implement the iterative compression approach for $d = 1$ (see Theorem 4 in Subsection 4.2) because preliminary experiments showed that trying all 2^p subsets of a given solution or even just enumerating subsets that can yield a solution (excluding subsets that induce a graph of maximum degree $> d$) is not fast enough on the instances we tested. Indeed, it is a common observation that algorithms based on iterative compression actually are close to their analyzed worst-case behavior whereas algorithms based on search trees often perform much better in practice than their predicted worst-case behavior lets assume.

6 Experimental Results

All experiments were run on an AMD Athlon 64 3700+ machine with 2.2 GHz, 1 M L2 cache, and 3 GB main memory with Debian GNU/Linux 4.0 operating system and the Objective Caml 3.09.2 compiler. The experiments of Balasundaram et al. (2009) were performed on Dell Precision PWS690 machines with a 2.66 GHz Xeon Processor, 3 GB main memory, implemented using ILOG CPLEX 10.0. The experiments of McClosky and Hicks (2010) were run on a 2.2 GHz Dual-Core AMD Opteron processor with 3 GB main memory. Trukhanov (2008) used a Dell Optiplex GX260 computer with 3.2 GHz Pentium D processor and 2 GB of RAM. The processor speeds are more or less comparable, so we compare the running times directly without applying a correction factor. Note that for all three works (Balasundaram et al., 2009; McClosky and Hicks, 2010; Trukhanov, 2008) the corresponding source code is not publicly available.

Balasundaram et al. (2009) experimented with two main groups of graphs. One group can be characterized as social networks, which are derived from real-world data. The second group of graphs contains various graphs using the *Sanchis generator* (Sanchis and Jagota, 1996) and clique instances from the second DIMACS challenge (DI-

MACS, 1995). They also performed experiments on two biological networks. Balasundaram et al. (2009) used an integer linear programming formulation combined with branch & cut methods. One of their exact algorithms, called BC(MIS), generates cuts based on a greedily computed independent set. They combined this approach with an algorithm that iterates over all vertices and searches a k-plex only in the vicinity of each iterated vertex, combined with a low-degree reduction rule (which corresponds to the high-degree rule in the complement d-BDD instance). This variant is called *Iterative Peel-Branch-and-Cut (IPBC)* algorithm. In the following, we compare our approach with the BC(MIS) and IPBC algorithms and also with the exact algorithm "OsterPlex" by McClosky and Hicks (2010), which is an adapted version of an algorithm for finding maximum-cardinality cliques by Östergård (2002). The experiments of McClosky and Hicks (2010) cover almost all social networks that were analyzed by Balasundaram et al. (2009) and the instances from the DIMACS challenge. The algorithms studied by Trukhanov (2008) are also based on the clique algorithm by Östergård (2002). He compared them to the approaches by McClosky et al. and Balasundaram et al. using most of the aforementioned instances.

6.1 Social Networks

This graph group contains a set of Erdős collaboration networks (Grossman et al., 2007) (ERDŐS graphs), collaboration networks in computational geometry (Batagelj and Mrvar, 2006) (GEOM graphs), and text-mining networks based on Reuters news (Batagelj and Mrvar, 2006) (DAYS graphs). In social network analysis, high-density subgraphs play an important role since they represent, e.g., a group of persons that work closely together. Cliques are too sensitive with respect to missing edges: even if a group is very closely connected, there might be two persons who did not collaborate, but still can be considered very active members of the group.

ERDŐS graphs. Each vertex in an Erdős graph represents a scientist, and two vertices are adjacent if the corresponding scientists have published together. The graphs, obtained from Grossman et al. (2007), are named "ERDOS-x-y", where x represents the last two digits of the year for which the network was constructed, and y is the maximum distance from each vertex to Paul Erdős in the graph. As Balasundaram et al. (2009), McClosky and Hicks (2010) and Trukhanov (2008), we considered $x \in \{97, 98, 99\}$ and $y \in \{1, 2\}$.

GEOM graphs. Each vertex represents an author in computational geometry. For a threshold t, two authors are adjacent if they have more than t joint publications. The graphs are based on data from Beebe's bibliography page (Beebe, 2002) obtained from a computational geometry database (Jones, 2002). The graphs were obtained from Batagelj and Mrvar (2006) and named "GEOM-t", where $t \in \{0, 1, 2\}$ is the threshold.

DAYS graphs. The graphs are based on news released by Reuter during 66 days beginning with the terrorist attacks in New York on September 11, 2001. Each vertex is a selected word that appeared in the news. Given a threshold t, two words are connected by an edge if there exist more than t sentences in which both appear. The graphs, obtained from Batagelj and Mrvar (2006), are named "DAYS-t", where $t \in \{3, 4, 5\}$.

Table 1: Number of vertices, edges, graph density, and maximum k-plex sizes for $1 \leq k \leq 5$ for social networks.

graph	$\|V\|$	$\|E\|$	density	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
ERDOS-97-1	472	1314	0.01182	7	8	9	11	12
ERDOS-98-1	485	1381	0.01177	7	8	9	11	12
ERDOS-99-1	492	1417	0.01173	7	8	9	11	12
ERDOS-97-2	5488	8972	0.00060	7	8	9	11	12
ERDOS-98-2	5822	9505	0.00056	7	8	9	11	12
ERDOS-99-2	6100	9939	0.00053	8	8	9	11	12
GEOM-0	7343	11898	0.00044	22	22	22	22	22
GEOM-1	7343	3939	0.00015	10	10	11	12	13
GEOM-2	7343	1976	0.00007	8	8	10	11	11
DAYS-3	13332	5616	0.00006	8	10	11	13	13
DAYS-4	13332	3251	0.00004	7	8	9	11	11
DAYS-5	13332	2179	0.00003	7	7	8	10	11
H. Pylori	1570	1399	0.00114	3	5	6	7	8
S. Cerevisiae	2112	2203	0.00099	6	6	7	7	8
S. Pombe	1053	2884	0.00521	8	9	10	11	13

These three types of graphs have in common that they are sparse and show a power-law degree distribution. See Table 1 for an overview on the number of vertices, edges, graph density, and maximum-cardinality k-plex sizes (for $1 \leq k \leq 5$) for ERDŐS, GEOM, and DAYS graphs.

We compared the IPBC algorithm (Balasundaram et al., 2009), the OsterPlex algorithm (McClosky and Hicks, 2010) and Trukhanov's algorithm (Trukhanov, 2008) with our methods. We discovered experimentally that guided branching has a strong effect on the running time for the social network instances, while the BDD-NT rule and the edge-count rule had only minuscule effects. Therefore, we performed experiments with and without guided branching. The resulting running times for the ERDŐS, GEOM, and DAYS graphs are given in Tables 2–4, respectively. For the ERDŐS graphs, our method with guided branching outperformed the approaches by Balasundaram et al. (2009), McClosky and Hicks (2010) and—for $k \in \{4, 5\}$—also Trukhanov's algorithm by one or two orders of magnitude. Guided branching was especially very effective for higher values of k. An explanation for this is that since the graphs are very sparse, their complements, on which we solve d-BDD, are very dense. For this reason, the high-degree rule (Figure 1) applied extremely well, and the vertices that were chosen to branch on had mostly a high degree, which makes branching very effective. The guided branching accelerates the search process: the bdd-d-set X is relatively big (on a dense graph, many vertices have to be put into a bdd-d-set), thus just branching on v and $N(v) \cap X$ instead on v and $N(v)$ (Figure 2) still results in a good branching, because $N(v) \cap X$ is not significantly smaller than $N(v)$. Thus, we still have a good branching while having the benefit from the guided branching that our greedy solution X becomes small very quickly. To our surprise, the BDD-NT rule (almost) did not apply at all. The reason was that X (see "guided branching" in Subsection 3.2.1) was rather big, and we applied the high-degree rule first (see Figure 1), which reduces the graph so effectively that the condition for applying the BDD-NT rule was (almost) never met. When switching off the high-degree rule, almost all reduction was then performed by the BDD-NT rule.

For the GEOM graphs, we observed similar speedups of up to two orders of magnitude (see Table 3). Interestingly, for some instances our approach did not branch at

Table 2: Running times and numbers of search tree nodes for ERDŐS graphs compared with the running times of the IPBC (Balasundaram et al., 2009), OsterPlex (McClosky and Hicks, 2010) and Trukhanov's algorithm. Note that our and the OsterPlex experiments were aborted after one hour. Also note that OsterPlex was not tested for $k = 5$.

k	graph	IPBC	OsterPlex	Trukhanov	search tree algorithm			
					no guided branching		guided branching	
		seconds	seconds	seconds	seconds	nodes	seconds	nodes
2	E-97-1	1.5	0	0.16	0.46	91	0.26	674
	E-97-2	392.9	1253	2.03	8.35	141	4.76	2356
	E-98-1	1.7	0	0.17	0.24	78	0.14	838
	E-98-2	464.3	1514	2.19	7.3	116	5.88	2855
	E-99-1	1.8	0	0.17	0.31	100	0.17	931
	E-99-2	526.5	1757	2.31	8.99	127	7.05	3391
3	E-97-1	1.8	19	0.16	0.95	4422	0.57	5935
	E-97-2	394.1	≥3600	3.78	10.56	30165	12.53	63137
	E-98-1	1.8	20	0.19	0.63	5998	0.98	12947
	E-98-2	457.1	≥3600	4.44	20.1	55389	23.58	143441
	E-99-1	1.8	21	0.19	0.99	8116	1.8	20188
	E-99-2	520.0	≥3600	4.73	26.76	69522	33.82	172777
4	E-97-1	2.2	1897	0.22	0.7	8949	1.12	14528
	E-97-2	424.0	≥3600	60.64	6.61	25032	8.86	43819
	E-98-1	2.8	1675	0.28	0.67	8840	1.14	14528
	E-98-2	614.7	≥3600	83.47	8.52	33360	10.31	49908
	E-99-1	1.8	1783	0.19	1.08	12793	1.47	17058
	E-99-2	526.3	≥3600	90.41	16.68	72695	17.23	75940
5	E-97-1	5.7	–	0.30	18.29	177845	6.12	75830
	E-97-2	1042.8	–	1932.11	1434.04	4948746	45.07	313441
	E-98-1	7.9	–	1.36	40.7	347168	6.11	74812
	E-98-2	1664.6	–	2920.34	≥3600	8815201	52.81	357568
	E-99-1	9.9	–	0.28	98.4	697284	8.04	93790
	E-99-2	653.5	–	3172.94	≥3600	7606070	122.6	665268

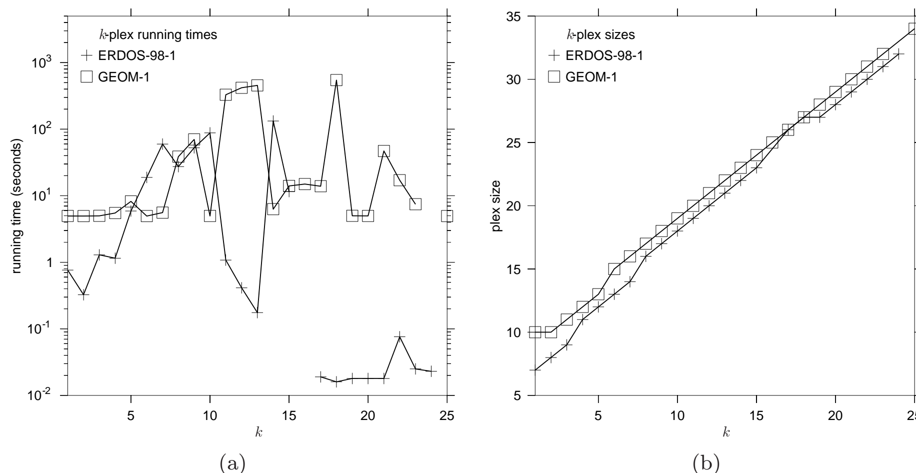
Table 3: Running times and numbers of search tree nodes for GEOM graphs.

k	graph	IPBC	OsterPlex	search tree algorithm			
				no guided branching		guided branching	
		seconds	seconds	seconds	nodes	seconds	nodes
2	GEOM-0	2384.4	397	9.61	0	9.73	0
	GEOM-1	753.2	1118	4.92	14	5.23	129
	GEOM-2	530.6	1145	3.37	13	3.36	192
3	GEOM-0	2387.1	≥3600	9.63	0	9.67	0
	GEOM-1	747.7	≥3600	6.04	7472	5.15	1128
	GEOM-2	524.3	≥3600	3.3	1	3.42	1
4	GEOM-0	2383.7	≥3600	9.74	0	9.6	0
	GEOM-1	743.7	≥3600	5.42	5021	5.45	6731
	GEOM-2	522.2	≥3600	3.44	1	3.46	1
5	GEOM-0	2298.1	–	9.65	0	9.64	0
	GEOM-1	691.6	–	6.91	21952	8.11	34089
	GEOM-2	472.6	–	8.02	58445	13.68	137262

all; it immediately found a solution using the data reduction rules. Since the data reduction rules were very effective and few branchings take place, the effect of the guided branching was not as pronounced as for the ERDŐS graphs. Note that Trukhanov (2008) did not include the GEOM graphs in his experiments.

Table 4: Running time and number of search tree nodes for DAYS graphs.

k	graph	IPBC	search tree algorithm: no guided branching		search tree algorithm: guided branching	
		seconds	seconds	nodes	seconds	nodes
2	DAYS-3	3367.8	20.34	10	20.44	330
	DAYS-4	2635.7	17.51	12	17.69	310
	DAYS-5	2462.9	0.1	16	0.11	262
3	DAYS-3	3395.4	60.52	119669	21.49	5160
	DAYS-4	3395.4	17.69	2282	17.85	3378
	DAYS-5	2445.5	0.27	2302	0.37	3596
4	DAYS-3	3489.8	20.49	1	20.45	1
	DAYS-4	3642.3	17.59	1	17.68	1
	DAYS-5	2426.3	89.73	550125	0.31	2983
5	DAYS-3	15336.9	81.28	423511	78.64	420962
	DAYS-4	6201.4	31.19	149970	37.74	222057
	DAYS-5	2820.8	$\geq$3600	6616431	2.09	18311



(a) (b)

Fig. 5: (a) Running times of our approach and (b) sizes of the resulting maximal k-plexes for $1 \leq d \leq 25$ on ERDOS-98-1 and GEOM-1 graphs. Missing data points are due to the exceeded running time limit of 60 minutes.

For the DAYS graph, we observed a speedup of up to three orders of magnitude (see Table 4) compared to the IPBC algorithm (Balasundaram et al., 2009). Note that McClosky and Hicks (2010) and Trukhanov (2008) did not include the DAYS graphs in their experiments.

Since the preceding experiments indicate that the running time of our approach does not increase too much with increasing k (recall that $k = d + 1$), we performed experiments on two of the real-world graphs (of medium difficulty) for $1 \leq k \leq 25$. The results are shown in Figure 5a. For most values of k, the instances could be solved within some seconds, only very few took several minutes, and exactly three could not be solved within the time limit of one hour. We also observed in Figure 5b that the size of a maximum k-plex increases almost linearly with the value of k. We conclude

Table 5: Running time and number of search tree nodes for the biological networks.

k	graph	IPBC	Trukhanov	search tree algorithm			
				no guided branching		guided branching	
		seconds	seconds	seconds	nodes	seconds	nodes
1	H. Pylori	11.5	-	5.54	357	4.50	405
	S. Cerevisiae	44.1	-	0.29	0	0.28	0
	S. Pombe	-	-	1.23	45	1.48	251
2	H. Pylori	12.6	0.53	11.40	289	4.42	5223
	S. Cerevisiae	46.4	0.69	0.29	1	0.29	1
	S. Pombe	-	-	0.96	28	1.12	1754
3	H. Pylori	37.8	0.55	46.77	233060	44.73	216341
	S. Cerevisiae	26.6	0.72	0.29	1	0.28	1
	S. Pombe	-	-	13.99	50127	2.26	14746
4	H. Pylori	29.3	0.78	$\geq$3600	$\geq$7212876	1655.40	7264346
	S. Cerevisiae	45.0	0.80	0.31	23	0.32	23
	S. Pombe	-	-	19.70	126160	27.65	194344
5	H. Pylori	133.1	2.36	$\geq$3600	$\geq$13401599	$\geq$3600	$\geq$14046277
	S. Cerevisiae	41.8	1.69	277.45	1224879	1.86	6532
	S. Pombe	-	-	0.83	1	0.84	1

that our approach seems to be able to find maximum k-plexes for a wide range of the parameter k for these types of graphs.

6.2 Biological Networks

Balasundaram et al. (2009) and Trukhanov (2008) performed experiments on two biological networks, namely protein-protein interaction networks of *H. Pylori* and *S. Cerevisiae*. In these graphs, vertices represent proteins and edges indicate that the pair of proteins forming the endpoints are known to interact. In such protein-protein interaction networks, k-plexes correspond to functional modules (see, e.g, (Balasundaram et al., 2005)). We add one additional network to the set of instances, namely the protein-protein interaction network of *S. Pombe* (fission yeast). The graph of S. Pombe was generated using data from the BioGRID database (`http://www.thebiogrid.org`). See Table 1 for the number of vertices and edges, density, and maximum k-plex sizes for $1 \leq k \leq 5$ for these biological networks. For the biological instances, we observe a very different behavior of our approach depending on the network (Table 5). For S. Cerevisiae and S. Pombe our algorithm performs very good for all considered values of k. For S. Cerevisiae, it is about two orders of magnitude faster than the IPBC algorithm by Balasundaram et al. (2009) and comparable to the approach by Trukhanov (2008). However, for H. Pylori the running time of our approach increases very quickly with increasing k and is much slower than Trukhanov's algorithm and slower than the IPBC algorithm for $k \geq 4$. A plausible explanation for this behavior is that neither the high-degree rule nor the BDD-NT rule can reduce the graph sufficiently; therefore, the parameter p is still rather large when the algorithm starts branching and therefore the search space combinatorially explodes. It would be interesting to learn why the IPBC algorithm and Trukhanov's algorithm perform so extremely well on this instance compared to our approach; this could help to combine the "best of all worlds" into one algorithm.

6.3 Sanchis and DIMACS Graphs

The second group of graphs considered by Balasundaram et al. (2009) contains various graphs using the *Sanchis generator* (Sanchis and Jagota, 1996) and clique instances from the second DIMACS challenge (DIMACS, 1995). The Sanchis generator (Sanchis and Jagota, 1996) produces graphs with known maximum clique size with a specified number of vertices n and edges m, and a construction parameter r. As Balasundaram et al. (2009), we fixed the maximum clique size at $\lceil n/5 \rceil$, and the construction parameter to $\lfloor 0.75(n/c-1) \rfloor$. The number of edges is determined by the density ϱ, that is, we computed the number of edges as $m := \lfloor \varrho n(n-1)/2 \rfloor$. We performed experiments for $n \in \{100, 200\}$ and $\varrho \in \{0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$.

We observed the same general behavior as for the BC(MIS) algorithm, that is, dense Sanchis graphs are harder to solve than sparse ones, and graphs with many vertices are harder to solve than graphs with few vertices. However, since the algorithm by Trukhanov (2008) appears to be clearly superior to our and the BC(MIS) algorithms in the Sanchis instances, we discarded a more detailed performance comparison. The general observation is that Trukhanov's algorithm outperforms the running times of the other algorithms approximately in a range between two and three orders magnitude.

Finally, we briefly report about our findings concerning instances from the DIMACS challenge. We compare with the BC(MIS) algorithm (Balasundaram et al., 2009), the OsterPlex algorithm (McClosky and Hicks, 2010) and Trukhanov's algorithm (Trukhanov, 2008). The results, which cover all instances that are used by Balasundaram et al. (2009), are shown in Table 6. Summarizing, out of the 32 considered instances we could solve 25 instances for $k = 1$ and 17 instances for $k = 2$, while BC(MIS) could solve 20 instances for $k = 1$ and 16 instances for $k = 2$ within a running time limit of three hours. Compared to the OsterPlex algorithm and Trukhanov's algorithm, we could solve within one hour all but five instances for $k = 2$, which they could solve within this time. Also, we could solve one instance within the time limit, which they could not. Summarizing, BC(MIS) is comparable with our approach, and OsterPlex and Trukhanov's algorithm are at least as good as our approach for these instances. In general, "hard" instances cannot be solved efficiently by either of the four compared algorithms and "easy" instances are solved quickly by all the four algorithms, but there are a few exceptions where one method seems to outperform the others. In this respect, it would be interesting to study whether the OsterPlex, BC(MIS) and Truhkanov's algorithms could be efficiently combined with ours.

6.4 Concluding Remarks

The three data reduction rules (Figure 1) behave very differently in our experiments. The simple high-degree rule is the most effective and time-efficient data reduction rule, although in theory it does not yield the best-possible problem kernel size-bound. We recommend to apply it in every search tree node, especially in dense d-BDD instances. The degree-one rule is less often applied, but still it is quite effective on some instances. Concerning the BDD-NT rule, it is also applied less often than the high-degree rule. We repeated some of the above experiments with the high-degree rule and the degree-one rule disabled. Then, the applications of the BDD-NT rule increase dramatically. However, we observed that the running time with and without BDD-NT rule is approximately the same for most of the instances. We observed two reasons for this behavior:

Table 6: Table showing the running times of our algorithm and the corresponding k-plex sizes of DIMACS instances. A "B" superscript means that Balasundaram et al. (2009) solved the corresponding instance to optimality within three hours, but our algorithm did not terminate within this time. A "b" superscript means that we solved the corresponding instance to optimality within three hours, but the algorithm of Balasundaram et al. (2009) did not terminate within this time. Likewise, "M", "m", "T" and "t" superscripts compare our algorithm with those of McClosky and Hicks (2010) and Trukhanov (2008), respectively, but with the time limit one hour. A "*" superscript indicates that the corresponding instance was not part of the testbed for the algorithm considered by Trukhanov (2008). If our algorithm did not terminate within the running time limit of three hours, then we state the lower bound x and upper bound y of the maximum k-plex size that could be computed in the given time as an interval $[x, y]$.

graph	$\|V\|$	density	1-plex size	seconds	2-plex size	seconds
c-fat200-1	200	0.077	12	0.21	12	1.10
c-fat200-2	200	0.163	24	0.42	24	3.53
c-fat200-5	200	0.426	58	1.17	58	22.44
c-fat500-1	500	0.036	14	3.95	14	11.01
c-fat500-2	500	0.073	26	7.25	26	50.21
c-fat500-5	500	0.186	64	17.61	64	350.56
c-fat500-10	500	0.374	126	36.28	126	1547.25
hamming6-2	64	0.905	32	0.00	32	1.77
hamming6-4	64	0.349	4	0.05	6	0.24
hamming8-2	256	0.969	128	0.21	$[127,192]^{BMT}$	> 10800
hamming8-4	256	0.639	16	243.11	$[16,171]^{BMT}$	> 10800
hamming10-2	1024	0.990	512	12.44	$[511,768]^{M*}$	> 10800
hamming10-4	1024	0.829	[30,512]	> 10800	$[34,683]^{*}$	> 10800
johnson8-2-4	28	0.556	4	0.00	5	0.02
johnson8-4-4	70	0.768	14	0.44	14	40.70
MANN_a9	45	0.927	16	0.00	26	0.09
MANN_a27	45	0.927	126	2.05	236^{mt}	3417.35
MANN_a45	1035	0.996	345^{b}	469.16	$[662,697]^{*}$	> 10800
keller4	171	0.649	11	21.38	15^{MT}	4583.84
brock200_1	200	0.745	21^{b}	794.73	[24,134]	> 10800
brock200_2	200	0.496	12	23.13	13^{b}	606.16
brock200_4	200	0.658	17	204.58	20^{bT}	9691.01
brock400_2	400	0.749	[23,200]	> 10800	[26,267]	> 10800
brock400_4	400	0.749	[24,200]	> 10800	[26,267]	> 10800
brock800_2	800	0.651	[19,400]	> 10800	$[21,534]^{*}$	> 10800
brock800_4	800	0.650	[19,400]	> 10800	$[22,534]^{*}$	> 10800
p_hat300-1	300	0.244	8	29.72	10^{b}	502.48
p_hat300-2	300	0.489	25^{b}	242.77	$[28,200]^{T}$	> 10800
p_hat300-3	300	0.744	36^{b}	8154.08	[39,200]	> 10800
p_hat700-1	700	0.249	11^{b}	1464.41	$[11,467]^{M*}$	> 10800
p_hat700-2	700	0.498	[40,350]	> 10800	$[47,467]^{*}$	> 10800
p_hat700-3	700	0.748	[58,350]	> 10800	$[69,467]^{*}$	> 10800

one is the running time of the BDD-NT rule; in the present version, it seems to be still too slow to be used to effectively speed up the search tree algorithm. A bottleneck in the BDD-NT rule is the computation of a maximum flow, which we implemented using a simple augmenting path computation; however, this method turned out to be still faster than using an existing (experimental) maximum flow library for Objective Caml. A more sophisticated routine to compute maximum flows could significantly speed up the search process. The second reason is that the greedy solution X is rather

big in almost all the tested instances. In many cases, the condition $|N(X)| > (d+1)|X|$ (cf. Figure 1) is satisfied only in search tree nodes that are very close to the leaves of the search tree. The input instance for such search tree nodes often contains only a few vertices, and the instance is solved very efficiently within a few branching steps. The benefit of reducing such small instances before branching has no big influence, compared to the application of, e.g., the high-degree rule on the input graph even *before* starting to branch. For the special case of 2-BDD respectively MAXIMUM 3-PLEX one might test the benefits of the recent results of Chen et al. (2010).

Since data reduction rules are in some sense universal (that is, they can be always applied before solving an instance with virtually any method), it makes sense to combine the data reduction rules presented in this work with the BC(MIS), the OsterPlex and Trukhanov's algorithm. This is an interesting topic for future research.

7 Outlook

In some analogy to previous work on maximum-cardinality clique finding (Abu-Khzam et al., 2004, 2007; Chesler et al., 2005), we demonstrated that a fixed-parameter approach provides competitive algorithms for finding maximum-cardinality k-plexes. Clearly, due to the NP-hardness of the problem, there are limitations concerning the range of practical feasibility. On the one hand, we believe that there is still some room for further tuning our algorithms and implementations (which in future work also should be compared with other approaches in an experimental study being based on the *same* platform); on the other hand, we think that at some point more restrictions such as the one of "isolation" (see (Ito and Iwama, 2009; Komusiewicz et al., 2009; Hüffner et al., 2009)) have to be imposed in order to gain practical algorithms. Our focus was on finding k-plexes of maximum size; studies concerning efficient approximation algorithms are left open.

It is conceivable that the presented algorithms can be converted into an enumerative algorithm if one does not use the BDD-NT-Rule and the Degree-One Rule. Since the whole search space has to be traversed, it seems also clear that the guided branching heuristic would not show effect. Hence, isolation concepts might be necessary in order to derive efficient enumeration algorithms, also because for $k \geq 2$ the search space for MAXIMUM k-PLEX is larger than the search space for MAXIMUM CLIQUE (for which the isolation concepts were originally designed).

A general message to be taken from our work is that even when considering exponential-time algorithms (particularly fixed-parameter algorithms), then still polynomial running time factors with high-degree polynomials may turn such algorithms impractical. Hence, struggling for (almost) linear-time factors in fixed-parameter algorithms is an important, so far often neglected matter of great practical significance.

Acknowledgements Our work has been supported by the Deutsche Forschungsgemeinschaft, project AREG (algorithms for generating quasi-regular structures in graphs, NI 369/9). We are grateful to Balabhaskar Balasundaram (Oklahoma State University, Stillwater, U.S.A.) for providing test instances. We thank the anonymous referees of *Journal of Combinatorial Optimization* for their constructive feedback.

References

Abu-Khzam FN, Collins RL, Fellows MR, Langston MA, Suters WH, Symons CT (2004) Kernelization algorithms for the Vertex Cover problem: Theory and experiments. In: Proc. 6th ALENEX, ACM/SIAM, pp 62–69 2, 23

Abu-Khzam FN, Fellows MR, Langston MA, Suters WH (2007) Crown structures for vertex cover kernelization. Theory Comput Syst 41(3):411–430 2, 23

Balasundaram B, Butenko S, Trukhanov S (2005) Novel approaches for analyzing biological networks. J Comb Optim 10(1):23–39 20

Balasundaram B, Butenko S, Hicks IV (2009) Clique relaxations in social network analysis: The maximum k-plex problem. Oper Res DOI 10.1287/opre.1100.0851, available electronically 2, 3, 15, 16, 17, 18, 19, 20, 21, 22

Balasundaram B, Chandramouli S, Trukhanov S (2010) Approximation algorithms for finding and partitioning unit-disk graphs into co-k-plexes. Optim Lett 4:311–320 3

Batagelj V, Mrvar A (2006) Pajek datasets. URL `http://vlado.fmf.uni-lj.si/pub/networks/data/`, accessed January 2009 16

Beebe NH (2002) Nelson H.F. Beebe's bibliographies page. URL `http://www.math.utah.edu/~beebe/bibliographies.html` 16

van Bevern R, Moser H, Niedermeier R (2011) Approximation and tidying—a problem kernel for s-plex cluster vertex deletion. Algorithmica DOI 10.1007/s00453-011-9492-7, available electronically 2

Bodlaender HL (2009) Kernelization: New upper and lower bound techniques. In: Proc. 4th IWPEC, Springer, LNCS, vol 5917, pp 17–37 4

Chen ZZ, Fellows M, Fu B, Jiang H, Liu Y, Wand L, Zhu B (2010) A linear kernel for co-path/cycle packing. In: Proc. 6th AAIM, Springer, LNCS, vol 6124, pp 90–102 3, 23

Chesler EJ, Lu L, Shou S, Qu Y, Gu J, Wang J, Hsu HC, Mountz JD, Baldwin NE, Langston MA, Threadgill DW, Manly KF, Williams RW (2005) Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function. Nat Genet 37(3):233–242 2, 23

Dessmark A, Jansen K, Lingas A (1993) The maximum k-dependent and f-dependent set problem. In: Proc. 4th ISAAC, Springer, LNCS, vol 762, pp 88–97 3

DIMACS (1995) Maximum clique, graph coloring, and satisfiability. Second DIMACS implementation challenge. URL `http://dimacs.rutgers.edu/Challenges/`, accessed November 2008 1, 15, 21

Djidev H, Garrido O, Levcopoulos C, Lingas A (1992) On the maximum k-dependent set problem. Tech. Rep. LU-CS-TR:92-91, Department of Computer Science, Lund University, Sweden 3

Downey RG, Fellows MR (1999) Parameterized Complexity. Springer 4

Fellows MR, Guo J, Moser H, Niedermeier R (2009) A generalization of Nemhauser and Trotter's local optimization theorem. In: Proc. 26th STACS, IBFI Dagstuhl, Germany, pp 409–420 3

Fellows MR, Guo J, Moser H, Niedermeier R (2010) A generalization of Nemhauser and Trotter's local optimization theorem. J Comput System Sci DOI 10.1016/j.jcss.2010.12.001, available electronically 2, 3, 5, 6, 7

Flum J, Grohe M (2006) Parameterized Complexity Theory. Springer 4

Grossman J, Ion P, Castro RD (2007) The Erdős number project. URL `http://www.oakland.edu/enp/`, accessed January 2009 16

Guo J, Niedermeier R (2007) Invitation to data reduction and problem kernelization. SIGACT News 38(1):31–45 4

Guo J, Moser H, Niedermeier R (2009) Iterative compression for exactly solving NP-hard minimization problems. In: Algorithmics of Large and Complex Networks, LNCS, vol 5515, Springer, pp 65–80 11

Guo J, Komusiewicz C, Niedermeier R, Uhlmann J (2010) A more relaxed model for graph-based data clustering: s-plex cluster editing. SIAM J Discrete Math 24(4):1662–1683 2

Hüffner F, Niedermeier R, Wernicke S (2008) Techniques for practical fixed-parameter algorithms. The Computer Journal 51(1):7–25 4

Hüffner F, Komusiewicz C, Moser H, Niedermeier R (2009) Isolation concepts for clique enumeration: Comparison and computational experiments. Theor Comput Sci 410(52):5384–5397 23

Hüffner F, Komusiewicz C, Moser H, Niedermeier R (2010) Fixed-parameter algorithms for cluster vertex deletion. Theory Comput Syst 47:196–217 14

Ito H, Iwama K (2009) Enumeration of isolated cliques and pseudo-cliques. ACM Trans Algorithms 5(4):40:1–21 23

Jones B (2002) Computational geometry database. URL `http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html` 16

Komusiewicz C, Hüffner F, Moser H, Niedermeier R (2009) Isolation concepts for efficiently enumerating dense subgraphs. Theor Comput Sci 410(38-40):3640–3654 2, 3, 10, 23

Lewis JM, Yannakakis M (1980) The node-deletion problem for hereditary properties is NP-complete. J Comput System Sci 20(2):219–230 2

McClosky B, Hicks IV (2010) Combinatorial algorithms for the maximum k-plex problem. J Comb Optim DOI 10.1007/s10878-010-9338-2, available electronically 2, 3, 15, 16, 17, 18, 19, 21, 22

Moser H (2009) Finding optimal solutions for covering and matching problems. PhD thesis, Institut für Informatik, Friedrich-Schiller Universität Jena 3

Nemhauser GL, Trotter LE (1975) Vertex packings: Structural properties and algorithms. Math Program 8:232–248 3

Niedermeier R (2006) Invitation to Fixed-Parameter Algorithms. Oxford University Press 2, 4

Niedermeier R, Rossmanith P (2000) A general method to speed up fixed-parameter-tractable algorithms. Inf Process Lett 73(3–4):125–129 9

Nishimura N, Ragde P, Thilikos DM (2005) Fast fixed-parameter tractable algorithms for nontrivial generalizations of Vertex Cover. Discrete Appl Math 152(1–3):229–245 3

Östergård PRJ (2002) A fast algorithm for the maximum clique problem. Discrete Appl Math 120(1-3):197–207 16

Sanchis LA, Jagota A (1996) Some experimental and theoretical results on test case generators for the maximum clique problem. INFORMS J Comput 8(2):103–117 15, 21

Seidman SB, Foster BL (1978) A graph-theoretic generalization of the clique concept. J Math Sociol 6:139–154 2

Trukhanov S (2008) Novel approaches for solving large-scale optimization problems on graphs. PhD thesis, A&M Universtity, Texas 3, 15, 16, 17, 18, 19, 20, 21, 22

Wu B, Pei X (2007) A parallel algorithm for enumerating all the maximal k-plexes. In: Emerging Technologies in Knowledge Discovery and Data Mining, Springer, Lecture

Notes in Artificial Intelligence, vol 4819, pp 476–483 2, 3