`CliSAT`: a SAT-based exact algorithm for hard maximum clique problems

Pablo San Segundo[1]

Universidad Politécnica de Madrid (UPM), Centre for Automation and Robotics (CAR), Madrid, Spain, pablo.sansegundo@upm.es

Fabio Furini

Department of Computer, Control and Management Engineering Antonio Ruberti, Sapienza University of Rome, Rome, Italy, fabio.furini@uniroma1.it

David Alvarez

Universidad Politécnica de Madrid (UPM), Madrid, Spain, david.asanchez@upm.es

Panos Pardalos

Center for Applied Optimization, University of Florida (UF), Florida, USA, pardalos@ise.ufl.edu

Abstract

Given a graph, the maximum clique problem (MCP) asks for determining a complete subgraph with the largest possible number of vertices. We propose a new exact algorithm, called `CliSAT`, to solve the MCP to proven optimality. This problem is of fundamental importance in graph theory and combinatorial optimization due to its practical relevance for a wide range of applications. The newly developed exact approach is a combinatorial branch-and-bound algorithm that exploits the state-of-the-art branching scheme enhanced by two new bounding techniques with the goal of reducing the branching tree. The first one is based on graph colouring procedures and partial maximum satisfiability problems arising in the branching scheme. The second one is a filtering phase based on constraint programming and domain propagation techniques. `CliSAT` is designed for structured MCP instances which are computationally difficult to solve since they are dense and contain many interconnected large cliques. Extensive experiments on hard benchmark instances, as well as new hard instances arising from different applications, show that `CliSAT` outperforms the state-of-the-art MCP algorithms by several orders of magnitude.

Keywords: Combinatorial Optimization, Exact Algorithm, Branch-and-Bound Algorithm, Maximum Clique Problem.

[1]Corresponding author

1. Introduction

Let G be a simple undirected graph, we denote $V(G)$ its set of n vertices and $E(G)$ its set of m edges. Two vertices $u, v \in V(G)$ are called *adjacent* or *neighbors* if there is an edge $\{u, v\} \in E(G)$. A *clique* is a subset of pairwise adjacent vertices or, equivalently, a subset of vertices inducing a complete graph. The *maximum clique problem* (MCP) calls for determining a clique of G with the largest possible number of vertices, the size of which is known as the *clique number* $\omega(G)$. Figure 1 provides an example graph G with $n = 8$ vertices and $m = 22$ edges where $\omega(G) = 4$. A maximum clique is $K = \{v_1, v_2, v_3, v_4\}$ (the red vertices of the figure), the edges of the complete graph induced by these vertices are depicted with red lines. This graph contains multiple maximum cliques, another maximum clique is, e.g., the set $\{v_3, v_4, v_5, v_6\}$.

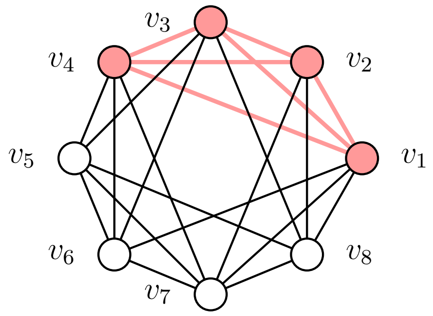


Figure 1: An example graph G with $\omega(G) = 4$. In red, a maximum clique $K = \{v_1, v_2, v_3, v_4\}$.

The MCP is one of the most studied combinatorial optimization problems in graph theory. It is known to be strongly $\mathcal{NP}$-hard and also hard to approximate within any polynomial factor unless $\mathcal{P} = \mathcal{ZPP}$ [10]. The MCP finds numerous applications which span disciplines such as computer vision [25, 22, 36], robotics [23], coding theory, network analysis and bioinformatics, see, e.g., [40].

In this work, we describe a new exact branch-and-bound (BnB) algorithm for the MCP that we call `CliSAT`. This algorithm is designed for hard MCP instances with up to several tenths of thousands of vertices. Hard MCP instances are those with many large interconnected cliques and they are typically dense. For these instances, the state-of-the-art techniques are combinatorial BnB algorithms (see, e.g., [44]) that employ bounding procedures based on graph coloring and partial maximum satisfiability (SAT) problems arising in the branching scheme. Our new exact algorithm is an enhancement of this class of algorithms that introduces new bounding procedures. These procedures, combined with the state-of-the-art branching scheme, are very effective in solving hard MCP instances as shown in the computational section. On the classical `DIMACS` set of instances, `CliSAT` compares favorably with previous state-of-the-art exact algorithms; moreover, on several new classes of hard instances `CliSAT` is the best performing algorithm, in some cases by several orders of magnitude.

It is important to mention that solving the MCP on very sparse massive graphs is, in practice, much easier than solving it for structured dense graphs. A different class of specialized algorithms has been specifically proposed in the literature for the former setting. This family of algorithms is based on tailored graph reduction techniques that are only effective for sparse instances (see e.g., [30] and [42]). The proposed algorithm `CliSAT`, even if it is not designed for this type of instances, is competitive with the state-of-the-art algorithms also for sparse graphs with up to 150.000 vertices.

1.1. Basic notation and definitions

Given a simple graph G and a subset of its vertices $W \subseteq V(G)$, we denote $G[W]$ the *induced graph* by W, i.e., the graph with vertex set $V(G[W])$ equal to W, and edge set $E(G[W])$ containing the subset of edges of $E(G)$ with both endpoints in W. The *complement graph*, denoted $\overline{G}$, has the same vertex set of G and edge set $\overline{E}(G) = \{u, v \in V(G) : \{u, v\} \notin E(G), u \neq v\}$. A subset $I \subseteq V(G)$ of pairwise non-adjacent vertices is called an *independent set* and it corresponds to a clique in $\overline{G}$. Moreover, let $N(u) = \big\{v \in V(G) : \{u, v\} \in E(G)\big\}$ denote the *neighbourhood* of a vertex $u \in V(G)$. A *vertex coloring* of a graph G is a partition of its vertex set into independent sets, also referred to as *color-classes*. The *vertex coloring problem* (VCP) calls for determining the minimum number of color-classes in any feasible vertex coloring, i.e., to determine the *chromatic number* $\chi(G)$ of the graph. We refer the interested reader to [18] for further details on the VCP. A (vertex) *k-coloring* of a graph G, which we denote $C_k(G)$, is a partition of $V(G)$ into k independent sets; precisely: $C_k(G) = \big\{I_1, I_2, \ldots, I_k\big\}$. Clearly, $\chi(G)$ provides an upper bound on the clique number $\omega(G)$, see, e.g., [1], and, consequently, so is the value k of any k-coloring of the graph, i.e., $\omega(G) \leq \chi(G) \leq |C_k(G)| = k$. Given a subset of vertices $W \subseteq V(G)$ and a partition of W into k independent sets, the k-coloring $C_k(G[W])$ is a called a *partial vertex coloring* of the graph G, i.e., a vertex coloring in which only the vertices of W are colored.

1.2. Reduction of the MCP to a partial maximum satisfiability problem

Given a graph G together with a k-coloring $C_k(G)$, we describe in this section a reduction, first proposed in [16], of the MCP to a *partial maximum satisfiability problem* (PMAX-SAT-P). A boolean variable $x \in \{0, 1\}$ is associated to two literals, a *positive literal*, denoted y and a *negative literal* denoted $\bar{y}$. The positive literal is true if $x = 1$ and the negative literal is true if $x = 0$. A *clause* is a finite collection of literals linked by logical operators (e.g., $\vee$ and $\wedge$). A *unit clause* refers to a clause with a single literal. Boolean formulas comprise clauses linked by logical operators. A Boolean formula in *conjunctive normal form* (CNF) is a conjunction of clauses, where a clause is a disjunction of literals.

The PMAX-SAT-P, associated to a MCP and a k-coloring, comprises two types of clauses denoted *hard clauses* and *soft clauses*. It calls for satisfying the maximum number of soft clauses, while satisfying all the hard ones. This PMAX-SAT-P features a vector of boolean variables $x \in \{0, 1\}^{|V(G)|}$, where each variable x_v represents a vertex $v \in V(G)$. Its $|\overline{E}(G)|$ hard clauses are associated to the non-edges of G. They contain only negative literals and encode the fact that at most one vertex from each pair of non-adjacent vertices of G can be part of a clique:

$$\big(\ \bar{y}_u\ \vee\ \bar{y}_v\ \big), \quad \forall \{u, v\} \in \overline{E}(G). \tag{1}$$

The hard clauses form a CNF boolean formula modelling the feasibility part of the MCP. The k soft clauses are associated to the independent sets of $C_k(G)$. They contain only positive literals and encode the fact that only one vertex from each independent set can be part of a clique:

$$\big(\ y_{v(I,1)}\ \vee\ y_{v(I,2)}\ \vee\ \ldots\ \vee\ y_{v(I,t)}\ \big), \quad \forall I \in C_k(G). \tag{2}$$

For each independent set $I \in C_k(G)$, the function $v(I, s)$ returns the vertex $v \in V(G)$ associated to the s-th vertex of I, and $t = |I|$. We denote $\mathscr{I}$ the collection of all the soft clauses (2), which form a CNF boolean formula modelling the objective function of the MCP, i.e., each satisfied clause corresponds to inserting the vertex of its true positive literal in a MCP solution. We denote $\mathrm{PSAT}(G, C_k(G))$ the PMAX-SAT-P associated to the graph G together with the k-coloring $C_k(G)$.

1.3. PMAX-SAT-P based upper bounds on the clique number

For a given graph G together with a coloring $C_k(G)$, upper bounds on the clique number $\omega(G)$ can be derived by reasoning and propagating the information of the hard clauses (1) and soft clauses (2) of the associated $\mathrm{PSAT}(G, C_k(G))$. It is straightforward to see that the existence of a clique of size k in G requires that all the k soft clauses (2) are satisfied. A subset $\mathscr{C} \subseteq \mathscr{I}$ of soft clauses (2) where at most $|\mathscr{C}| - 1$ of them can be satisfied, is called a *conflict*. A conflict-detection procedure determines a conflict by setting to false literals, i.e., removing them from the hard and soft clauses, while preserving logical entailment, until a clause becomes empty. A conflict logically entails an *empty clause*, i.e., a clause that contains no literals and, by definition, evaluates to false. If a conflict is found in $\mathrm{PSAT}(G, C_k(G))$, a clique of size k cannot exist in G; accordingly, $k - 1$ is in this case an upper bound on $\omega(G)$.

Unit Propagation (`UP`) is one of the main conflict-detection procedures, see [5]. It exploits the fact that a unit clause can only be satisfied by fixing its literal to true and, consequently, removing the negated literal from the remaining clauses. `UP` is applied iteratively after each removal until either *i*) there are no more unit clauses, or (*ii*) an empty clause is found. In the latter case, the soft clauses (2) in which a positive literal is set to true, together with the soft clause that becomes empty, determine a conflict.

Strong upper bounds on $\omega(G)$ can be obtained if more than one conflict is determined, see, e.g., [12, 13]. A collection of conflicts $\mathscr{P} = \{\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_{|\mathscr{P}|}\}$ is denoted a *proper set of conflicts* if for each pair of conflicts $(\mathscr{C}_a, \mathscr{C}_b)$ in $\mathscr{P}$, the set of soft clauses in $(\mathscr{C}_a \cup \mathscr{C}_b) \setminus (\mathscr{C}_a \cap \mathscr{C}_b)$ is also a conflict. In other words, the soft clauses that belong exactly to only one of the two conflicts also contain a conflict. If a proper set of conflicts is found, then $\omega(G) \leq k - |\mathscr{P}|$. Determining a proper set of conflicts can be done iteratively, one conflict at a time, see, e.g., [12, 13]. We recall, in what follows, the overarching idea of such procedures. For each conflict $\mathscr{C}$ found, the $\mathrm{PSAT}(G, C_k(G))$ is modified in such a way that the set of clauses in $\mathscr{C}$ are satisfiable. Precisely, the graph G is enlarged with $|\mathscr{C}|$ additional vertices, by inserting a new vertex per independent set associated to the clauses of $\mathscr{C}$. Each new vertex is connected to every vertex $V(G)$ in the graph, except to those vertices associated to the literals of its clause. In this way, we obtain a new graph called the *transformed graph of a conflict*, which we denote $G(\mathscr{C})$. The new $C_k(G(\mathscr{C}))$ is obtained from the original $C_k(G)$ by colouring each new vertex with the color class of its associated clause. In addition, a new $\mathrm{PSAT}\big(G(\mathscr{C}), C_k(G(\mathscr{C}))\big)$ can be defined in which the relaxed clauses of $\mathscr{C}$ are satisfiable. This problem is used to determine additional conflicts. A set of conflicts iteratively determined in this way is, by nature, a proper set of conflicts.

In addition to `UP`, and when no unit clauses are available, the *failed literal* conflict-detection procedure (`FL`), another well-established inference procedure used by SAT solvers, can be used in this context to determine conflicts. A positive literal of a soft clause is denoted *failed* if an empty clause is determined when it is set to true. If every literal in a clause is proven failed by successive calls to `FL`, a conflict has been found. The soft clauses of this conflict are those in which a positive clause is fixed to true by the different calls to `FL` together with the corresponding empty clauses.

1.4. Literature review on exact MCP algorithms

A large amount of effort has been devoted to solving the MCP to proven optimality. We refer the reader to [44] for a detailed survey on this topic. A complete overview of exact algorithms is out of the scope of this work. In what follows, we describe what we consider the most relevant ones together with their corresponding breakthroughs. One of the first successful BnB algorithms is described in [3], where a tailored n-ary branching scheme for the MCP is proposed. A bounding technique based on vertex colouring is described in [6], an idea almost universally employed by modern exact MCP algorithms, see, e.g., [39, 26, 27, 24]. One of the major breakthroughs of the last decade is the bounding technique proposed in [15, 16]. This family of upper bounds is based on partial maximum satisfiability problems arising in the branching scheme. Thanks to this new idea, the exact MCP algorithms have substantially improved their performance. Some of the state-of-art algorithms of this type are, e.g., [28, 31, 12, 13]. Finally, bitstring optimizations are known to be an additional source of efficiency, see, e.g., [26, 27, 24, 28, 31].

To the best of our knowledge, the more successful exact algorithm for hard dense MCP instances is `MoMC`, which is described in [12]. In the computational section, we compare the performance of our new algorithm `CliSAT` against `MoMC`, as well as several other efficient exact algorithms and integer linear programming (ILP) models solved by a state-of-the-art commercial solver.

Another recent stream of research aims at determining the clique number of real and very sparse massive graphs, such as those associated with social networks. Specialized algorithms exploit the scale-free nature of such graphs, i.e., graphs whose degree distribution follow a power law. These algorithms are able to solve the MCP to proven optimality in networks with millions of vertices, see, e.g., [30, 42, 9]. The techniques employed to determine a maximum clique for these instances are typically not effective for hard and dense MCP instances. For sparse massive instances, the most successful exact algorithms are `dOmega`, proposed in [42], and `BBMCSP`, proposed in [30]. These two algorithms are compared against `CliSAT` in the computational section.

It is also worth mentioning that exact algorithms have been developed in recent years for variants and generalization of the MCP. Efficient exact algorithms for the maximum vertex weighted clique problem are described in [33, 11], while exact algorithms for the edge-weighted case are described in [32, 35]. In addition, exact algorithms for vertex and edge interdiction variants of the MCP have been described in [8] and [7]. Finally, a recent exact algorithm for the knapsack problem with conflicts is described in [4]; this problem corresponds to the MCP with an additional knapsack constraint.

1.5. Methodological contributions and outline of the article

The main contribution of this paper is the development and the extensive testing of a new exact algorithm for the maximum clique problem. The algorithm, called `CliSAT`, is designed for hard MCP instances and is built upon the state-of-the-art procedures of the best-performing MCP algorithms in the literature. `CliSAT` integrates modern branching schemes with effective bounding techniques to reduce the size of the branching tree. The two state-of-art bounding mechanisms are based on graph colouring procedures and partial maximum satisfiability problems arising in the branching scheme. Starting from these cutting-edge techniques, `CliSAT` exploits new routines which are crucial for improving its performance to solve hard MCP instances.

Section 2 is entirely devoted to the presentation of the new algorithm. The first section §2.1 presents the state-of-the-art incremental branching scheme of `CliSAT`. This n-ary scheme employs the notions of branching and pruned sets of vertices and is described in §2.2. In §2.2.1 we present the most effective state-of-the-art techniques employed by MCP algorithms to enlarge the pruned set, which are based on PMAX-SAT-P-based upper bounds. In this context, the new `SATCOL` procedure presented in §2.2.2 is the first methodological improvement of `CliSAT`. Its goal is to further enlarge the pruned set by combining coloring-based and PMAX-SAT-P-based upper bounds. A second important methodological contribution is the the filtering phase of `CliSAT` described in §2.3. To the best of our knowledge, `CliSAT` is the first exact MCP algorithm to employ constraint programming and domain propagation techniques to filter vertices from the branching set, i.e., to completely remove them from branching subtrees. To this end, two *ad hoc* procedures are designed: the first one, denoted `FiltCOL`, is presented in §2.3.1; the second one, denoted `FiltSAT`, is presented in §2.3.2. After explaining the incremental upper bounds also employed by `CliSAT` in §2.4, the pseudocode for the algorithm is discussed in §2.5. Extensive experiments on hard benchmark MCP instances, as well as new hard instances arising from different applications, are presented in Section 3. Our computational campaign demonstrates the effectiveness of `CliSAT` on solving hard MCP instances and demonstrates that `CliSAT` outperforms the state-of-the-art MCP algorithms, for some classes of instances, by orders of magnitude. Section 4 concludes the paper summarizing the principal algorithmic improvements of `CliSAT` and outlines several promising lines of future research.

2. The new exact BnB algorithm: `CliSAT`

In this section, we describe the new BnB exact algorithm `CliSAT` for the MCP. `CliSAT` employs an n-ary branching scheme of a constructive type that iteratively builds a clique by adding one vertex at a time in a recursive fashion. We denote $\hat{K} \subseteq V(G)$ the *subproblem clique* associated to a branching node. Precisely, $\hat{K}$ contains the vertices fixed during branching and added to the subproblem clique in the nodes preceding the current one. Moreover, each branching node is associated to a *subproblem graph*, denoted $\hat{G}$. This graph contains the vertices which can be added singularly to $\hat{K}$, see §2.1. During its execution, `CliSAT` keeps track of the *incumbent solution*, denoted K_{inc}. The size $|K_{inc}|$ of the incumbent solution is denoted lb (a lower bound on $\omega(G)$). Moreover, if a larger clique is found during the branching, i.e., if the condition $|\hat{K}| > lb$ holds, both K_{inc} and lb are updated accordingly. After the execution of `CliSAT`, K_{inc} corresponds to a maximum clique of G and, accordingly, $lb = \omega(G)$.

The main idea of the branching scheme is to partition the set of vertices of the subproblem graph $\hat{G}$ into two subsets: *i*) the *branching set* B and *ii*) the *pruned set* P (see §2.2). This idea has been used in the state-of-the-art combinatorial BnB algorithms for the MCP and their variants, see, e.g., [12, 13, 14, 24, 26, 27, 28, 31, 33, 32]. By definition of P, at least one vertex from $B = V(\hat{G}) \setminus P$ is necessary to improve the incumbent solution K_{inc}. Accordingly, branching on any of the vertices in P is not necessary in a given branching node, and the algorithm backtracks when the set B is empty. After the pruned and branching sets are determined, `CliSAT` carries out a $|B|$-ary branching operation, creating a branching node for every vertex in B by adding it to the current subproblem clique $\hat{K}$ (see §2.1).

We consider the vertex set $V(G)$ of the input graph G sorted according to a given *initial ordering* $(v_1, v_2, \ldots, v_n)$, see §2.5 for further details on this topic. We denote $V_i(G) \subseteq V(G)$ the subset of vertices that comprises the first $i \leq n$ vertices of $V(G)$; precisely: $V_i(G) = \{v_1, \ldots, v_i\}$ with $i = 2, \ldots, n$, and $V_1(G) = \{v_1\}$. Moreover, we denote $V(v_i, G) \subseteq V(G)$ the subset of vertices that comprises the first i vertices of $V(G)$ intersected with the neighbourhood of the vertex v_i; precisely: $V(v_i, G) = V_i(G) \cap N(v_i)$, $i = 1, \dots, n$. We then define $|V(G)|$ graphs $G(v_i)$ as the ones induced by the (non-empty) sets of vertices $V(v_i, G)$; precisely:

$$G(v_i) = G[V(v_i, G)], \qquad i = 1, \dots, |V(G)|. \tag{3}$$

Figure 2 depicts the graphs $G(v_6)$ and $G(v_7)$ associated to the graph G of Figure 1. The vertices v_6 and v_7 appear in red, $V(G(v_6))$ and $V(G(v_7))$ in green. The edges of both graphs are drawn as thick black straight lines. The edges connecting vertex v_7 to the vertices preceding it according to the initial ordering are colored in blue. The edge that connects v_7 to v_8 appears as a dashed blue line, indicating that v_8 does not belong to $E(G(v_7))$. The same information is shown for $G(v_6)$. All the remaining edges in $E(G)$ are shown as dashed black lines.

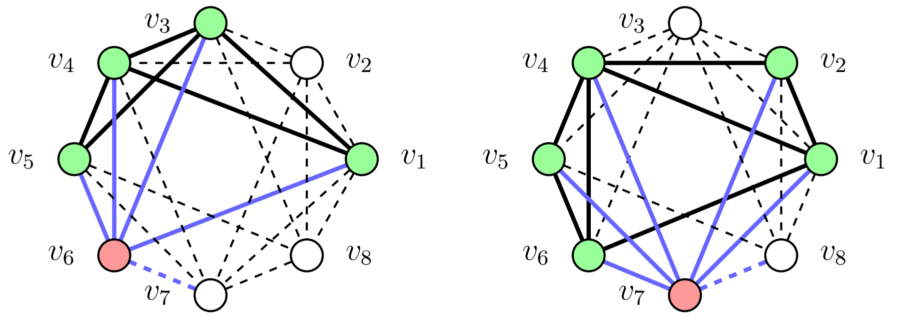


Figure 2: The graphs $G(v_6)$ (left part) and $G(v_7)$ (right part) associated to the graph G of Figure 1.

2.1. The incremental branching scheme of `CliSAT`

The input of the branching scheme of `CliSAT` corresponds to the family of graphs $G(v_i)$, $i = 2, \dots, n$. `CliSAT` executes a BnB procedure for each one of these graphs, examining them in order. We recall that $\hat{G}$ is the subproblem graph and $\hat{K}$ is the subproblem clique associated to a branching node. In the first level of the branching tree, $\hat{G}$ corresponds to one of the graphs $G(v_i)$ and $\hat{K}$ is the singleton $\{v_i\}$. In order to determine the subproblem graphs $\hat{G}$ for subsequent child nodes, `CliSAT` first partitions the vertex set $V(\hat{G})$ into the pruned and branching sets P and B, i.e., $V(\hat{G}) = P \cup B$ and $B \cap P = \emptyset$.

The pruned set P is a subset of vertices of $V(\hat{G})$ respecting the following condition:

$$|\hat{K}| + \overline{\omega}\big(\hat{G}[P]\big) \leq lb, \tag{4}$$

where $\overline{\omega}\big(\hat{G}[P]\big)$ is any upper bound on the clique number of $\hat{G}[P]$. The entire left hand side of (4) corresponds, *de facto*, to an upper bound on the clique number of the graph $G[\hat{K} \cup P]$. In other words, the condition states that the graph induced by the vertices in $\hat{K} \cup P$ does not contain a clique of size larger than $lb = |K_{inc}|$. Precisely, if a set P that respects the condition (4) is found, it means that, in order to improve the incumbent solution K_{inc}, it is necessary to add to $\hat{K}$ at least one of the vertices in $V(\hat{G})$ which is not in P. Consequently, we define the branching set B as $V(\hat{G}) \setminus P$. The specific way in which the P set is constructed by `CliSAT`, as well as the specific upper bounds on the clique number it employs, are presented in §2.2.

Once the sets P and B are created, the vertices of these sets are ordered according to the initial ordering $(v_1, v_2, \ldots, v_n)$ and relabelled as follows:

$$P = \{p_1, p_2, \ldots, p_{|P|}\} \quad \text{and} \quad B = \{b_1, b_2, \ldots, b_{|B|}\}. \tag{5}$$

`CliSAT` keeps track of the initial labels of the vertices $v \in V(G)$ by establishing a mapping between the vertices $p \in P$ and $b \in B$ and the corresponding vertices in $V(G)$. This is done efficiently with the help of its bitstring encoding of vertex sets in memory.

An example of the P and B sets is presented in the left part of Figure 3. Precisely, it shows the partition of the vertex set $\{v_1, v_2, v_4, v_5, v_6\}$ of the subproblem graph $G(v_7)$ of Figure 2 (the original graph G, we recall, is shown in Figure 1), into the sets $P = \{v_2, v_6\}$ (grey) and $B = \{v_1, v_4, v_5\}$ (black). The edges of $G(v_7)$ are depicted as thick black lines. In this example we assume G to be the input graph, so $\hat{K} = \{v_7\}$ (v_7 is shown in red). Since $\omega(\hat{G}[\{p_1, p_2\}]) = 1$, it follows that the size lb of the incumbent solution must be equal to 2 for the condition (4) to hold. For the sake of clarity, the vertices of the sets B and P are shown according to the relabelling established by Equation (5), i.e., $P = \{p_1, p_2\}$ and $B = \{b_1, b_2, b_3\}$. In blue, the edges incident to v_7 which are involved in the branching. The vertices v_3 and v_8 are shown in white (without a label) since they do not belong to $G(v_7)$, i.e., v_3 is not adjacent to v_7 and v_8 comes after v_7 in the initial ordering. Finally, the incident edges to v_3 and v_8 are drawn as dashed lines.

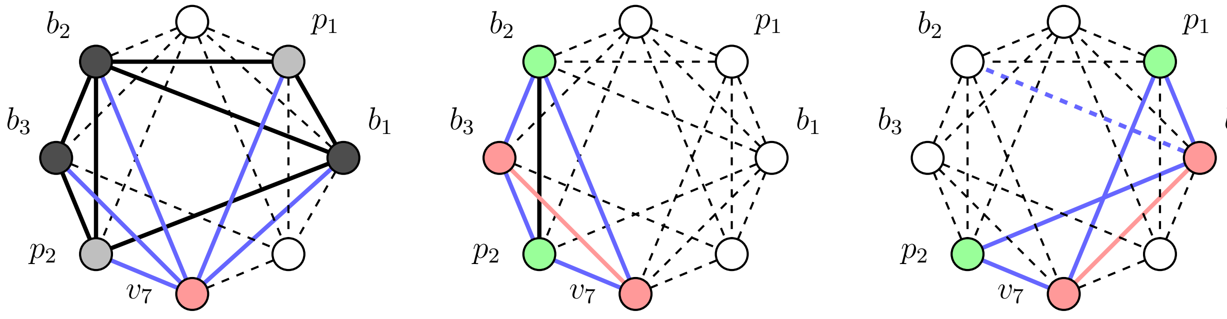


Figure 3: Left: the pruned and branching sets P and B for the subproblem graph $G(v_7)$ of Figure 2. Middle: the subproblem graph $\tilde{G}(b_3)$. Right: the subproblem graph $\tilde{G}(b_1)$.

To create the subproblem graphs $\hat{G}$ of the child nodes associated to branching on the vertices of the set B, we define a new family of graphs, called $\tilde{G}$. We denote $B_j(\hat{G}) \subseteq B$, the subset of vertices that comprises the first $j \leq |B|$ vertices; precisely: $B_j(\hat{G}) = \{b_1, \ldots, b_j\}$, with $j = 2, \ldots, |B|$, and $B_1(\hat{G}) = \{b_1\}$. In addition, we denote $\hat{V}(b_j, \hat{G}) \subseteq \{P \cup B\}$ the subset of vertices that comprises the intersection between the set P, together with the first j vertices of B, with the neighbourhood of the vertex b_j; precisely: $\hat{V}(b_j, \hat{G}) = \{P \cup B_j(\hat{G})\} \cap N(b_j)$, $j = 1, \ldots, |B|$. We then define $|B|$ graphs $\tilde{G}(b_j)$ as the graphs induced by the (non-empty) sets of vertices $\hat{V}(b_j, \hat{G})$; precisely:

$$\tilde{G}(b_j) = \hat{G}\big[\hat{V}(b_j, \hat{G})\big], \qquad j = 1, \ldots, |B|. \tag{6}$$

The graphs $\tilde{G}(b_j)$ become the subproblem graphs $\hat{G}$ in subsequent child nodes, and $\hat{K} \cup \{b_j\}$ the associated subproblem cliques. By construction, the vertices of $\tilde{G}(b_j)$ are connected to all the vertices of $\hat{K}$. `CliSAT` proceeds recursively until either all the vertices in B have been explored, or B becomes the empty set.

Figure 3 shows the graphs $\tilde{G}(b_3)$ (middle part) and $\tilde{G}(b_1)$ (right part) associated to the branching set B. As in previous figures, the set of vertices of both graphs, i.e., $\{p_2, b_2\}$ and $\{p_1, p_2\}$ respectively, are colored in green. By branching on the vertex b_3 (resp. b_1), $\hat{K}$ becomes $\{v_7, b_3\}$ (resp. $\{v_7, b_1\}$) and its unique edge, i.e., $\{v_7, b_3\}$ (resp. $\{v_7, b_1\}$), is shown as a red line. In blue, the edges that connect the vertices of $\tilde{G}(b_3)$ and $\tilde{G}(b_1)$ to the associated $\hat{K}$. The edge set of $\tilde{G}(b_1)$ is empty, while the edge set of $\tilde{G}(b_3)$ is the singleton $\{p_2, b_2\}$ (drawn as a black line). The edge $\{b_1, b_2\}$ is drawn as a dashed blue line in the right part of the figure to indicate that b_2 does not belong to $V(\tilde{G}(b_1))$, since b_2 comes after b_1 in the new labelling (see equation (5)). All the remaining edges of $E(G)$ are shown as dashed black lines.

We denote this way of branching *incremental* hereafter, as opposed to the more traditional branching scheme that considers the child subproblems derived from the full neighbourhood of the vertices selected for branching, see, e.g., [23, 28, 31]. Incremental branching has been employed by the recent efficient algorithms `MoMC`[12] and `IncMC2`[13], and we have adopted this strategy for our algorithm `CliSAT`. Finally, it is worth mentioning the relation between the incremental branching of `CliSAT` and the Russian Doll Search (RDS) branching scheme described in the literature, see, e.g., [41]. In RDS, the original problem is divided into hierarchical subproblems (dolls) that are explored according to increasing order, i.e., the first doll contains a singleton, and the last doll corresponds to the original problem. RDS has been employed successfully for the MCP in the well known algorithm `CLIQUER` [20], also evaluated in this work. The subproblems determined by the incremental branching scheme are reminiscent of doll subproblems, but the (doll) decomposition occurs in every branching node, whereas in RDS it is restricted to the original problem.

2.2. Determining the pruned and branching sets

In this section we explain the techniques used by `CliSAT` to determine the branching and pruned sets. We recall that the branching operations of `CliSAT` require determining a pruned set $P \subseteq V(\hat{G})$ respecting the condition (4). One such type of pruned set, which we denote P_C, is determined by a partial κ-coloring:

$$C_\kappa(\hat{G}[P_C]) = \{I_1, I_2, \ldots, I_\kappa\}, \quad \text{where} \quad \kappa = lb - |\hat{K}|. \tag{7}$$

The value κ corresponds to an upper bound $\overline{\omega}(\hat{G}[P_C])$, and the κ-coloring is a collection of κ independent sets (§1.1). `CliSAT` employs the *greedy independent-set sequential colouring procedure* to compute a κ-coloring. This procedure is referred to as `ISEQ`, and was first proposed (in connection with the MCP) in [26, 27]. We outline, in what follows, the main operations of `ISEQ`, and refer the reader to the aforementioned papers for further details. Given a vertex ordering $(v_1, v_2, \ldots, v_{|V(\hat{G})|})$ of $V(\hat{G})$, each iteration of `ISEQ` builds an independent set processing the vertices in order. At each step within an iteration, and starting from the empty set, a vertex is added to the independent set under construction if it is not linked to any of its vertices. `ISEQ` continues iterating until κ independent sets are determined. It is worth mentioning that `CliSAT` implements `ISEQ` efficiently using a bitstring encoding of the vertex sets in memory, see [26].

We show the operations of `ISEQ` considering the subproblem graph $\hat{G}$ depicted in the left part of Figure 4. This graph is chosen since it features a gap of one unit between its clique number and its chromatic number, i.e., $\omega(\hat{G}) = 4$ and $\chi(\hat{G}) = 5$. In addition, $\hat{G}$ is considered associated to a branching node with $|\hat{K}| = 1$ and $lb = 5$, so $\kappa = 4$ according to equation (7). Given the ordering $(v_1, v_2, \ldots, v_7)$ of the vertex set $V(\hat{G})$, `ISEQ` determines the following 4 independent sets in order: $I_1 = \{v_1\}$, $I_2 = \{v_2\}$, $I_3 = \{v_3, v_4\}$ and $I_4 = \{v_5, v_6\}$. Each independent set is depicted with a different color. The grey vertices in the right part of the figure correspond to the pruned set $P_C = \{p_1, p_2, \ldots, p_6\}$, shown after the relabelling according to Equation (5). The remaining vertex b_1 (depicted in black) becomes the branching set B. The edges incident to b_1 are represented as dashed lines.

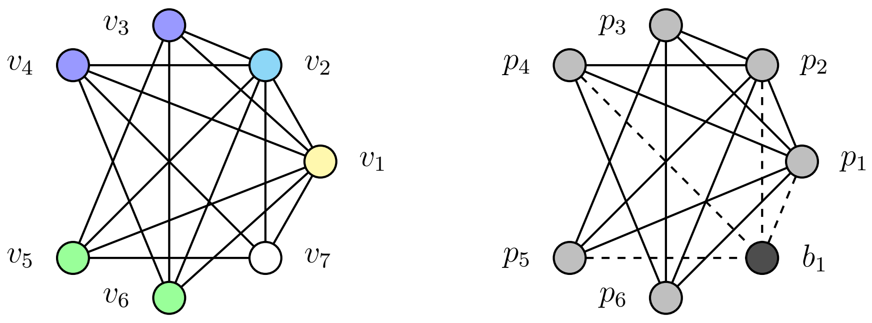


Figure 4: To the left, a subproblem graph $\hat{G}$ associated to a branching node of `CliSAT`. The coloured vertices correspond to the independent sets determined by `ISEQ`. To the right, the corresponding pruned set $P_C = \{p_1, p_2, \ldots, p_6\}$ and the branching set $B = \{b_1\}$.

In this example, the `ISEQ` procedure is not able to construct a pruned set $P_C = V(\hat{G})$, so branching is necessary. The example illustrates the limits of using a (heuristic) vertex colouring procedure to create the set P. Since $\chi(\hat{G}) = 5$, the branching node cannot be pruned even with an optimal colouring.

2.2.1. Enlarging the pruned set with PMAX-SAT-P-based upper bounds

Given a κ-colouring $C_\kappa(\hat{G}[P_C])$, defined in the previous §2.2, the set P_C can be enlarged by adding vertices from $B = V(\hat{G}) \setminus P_C$, one at a time, using the the PMAX-SAT-P upper bound presented in §1.3. We describe in what follows the state-of-the-art procedure of this type employed by, e.g., [12]. Hereafter, we denote for short $P_C \cup \{b\}$ as P_b. A vertex $b \in B$ can be added to P_C if an upper bound $\overline{\omega}(\hat{G}[P_b]) \leq lb - |\hat{K}|$ can be determined. To this end, a partition of $V(\hat{G}[P_b])$ into $\kappa + 1$ color classes is created by assigning the vertex b to a new color class. Precisely, $C_{\kappa+1}(\hat{G}[P_b]) = C_\kappa(\hat{G}[P_C]) \cup \{b\}$ and the associated $\text{PSAT}(\hat{G}[P_b], C_{\kappa+1}(\hat{G}[P_b]))$ can be used to prove that $\overline{\omega}(\hat{G}[P_b]) = \kappa$ if the `UP` procedure determines a conflict $\mathscr{C}$ (starting from the unit clause of $\{b\}$). If a conflict $\mathscr{C}$ is found, P_b becomes the new pruned set and b is removed from the branching set B. In order to add more than one vertex from B to P_C, it is necessary to find a proper set of conflicts by iteratively building the transformed-graphs as explained in §1.2. The effect on the branching tree is twofold: *i*) a node is fathomed if the branching set B becomes empty; *ii*) the number of child nodes is reduced if the set P_C is enlarged, see §2.1.

We illustrate this technique by referring again to the subproblem graph $\hat{G}$ in Figure 4 and $C_4(\hat{G}[P_C])$. We recall that the branching set is $B = \{b_1\}$ and the pruned set is $P_C = \{p_1, p_2, \ldots, p_6\}$, and show how to obtain $\overline{\omega}(\hat{G}[P_b]) = 4$ after determining $C_5(\hat{G}[P_b])$ as explained previously. Precisely, the associated $\text{PSAT}(\hat{G}[P_b]), C_5(\hat{G}[P_b])$ contains the 5 soft clauses: $(y_{p_1}), (y_{p_2}), (y_{p_3} \vee y_{p_4}), (y_{p_5} \vee y_{p_6}), (y_{b_1})$, and it is possible to determine a conflict by executing `UP` on the unit clause (y_{b_1}). The reasoning is as follows: setting to true the literal y_{b_1} removes the literals y_{p_3} and y_{p_6} (according to the hard clauses $(\bar{y}_{b_1} \vee, \bar{y}_{p_3})$ and $(\bar{y}_{b_1} \vee, \bar{y}_{p_6})$) so that both clauses $(y_{p_3} \vee y_{p_4})$ and $(y_{p_5} \vee y_{p_6})$ become unit. Finally, setting to true the literal y_{p_4} empties the other unit clause, resulting in the conflict $\{(y_{b_1}), (y_{p_3} \vee y_{p_4}), (y_{p_5} \vee y_{p_6})\}$. Consequently, $P = V(\hat{G})$, $B = \emptyset$ and the branching node is fathomed. As can be seen, the PMAX-SAT-P-based upper bounds can be stronger than the chromatic number.

2.2.2. Enlarging the pruned set with the `SATCOL` *procedure*

In what follows, we describe a new procedure, denoted `SATCOL`, that is employed by `CliSAT` to (potentially) enlarge the pruned set P_C by adding one independent set $I \subseteq B$ at a time. Each independent set is computed by one iteration of `ISEQ` on the vertices in B. We denote for short $P_C \cup I$ as P_I. A larger pruned set P_I is determined if a conflict $\mathscr{C}$ is found in PSAT$(\hat{G}[P_I], C_{\kappa+1}(\hat{G}[P_I]))$, where $C_{\kappa+1}(\hat{G}[P_I])$ corresponds to the κ-colouring $C_\kappa(\hat{G}[P_C])$ together with the independent set I. In such a case, $\bar{\omega}(\hat{G}[P_I]) = \kappa$, the new pruned set becomes P_I, I is removed from B, and the transformed-graph $\hat{G}[P_I](\mathscr{C})$ is computed. To find a conflict $\mathscr{C}$, `SATCOL` executes the procedure `FL` on each of the literals associated to the vertices of I attempting to prove them failed, see §1.3. It follows that, if $\mathscr{C}$ is found, the soft clause associated to I must be part of $\mathscr{C}$. `SATCOL` continues examining independent sets in B until it either fails to find a conflict, or the set $B = \varnothing$ and the branching node is fathomed. When the procedure stops, the pruned set determined in this way is denoted P_S. The transformed-graphs are necessary to ensure that the set of conflicts determined iteratively by `SATCOL` is a proper set of conflicts, see §1.2.

`SATCOL` presents a number of advantages with respect to prior state-of-the-art procedures that examine vertices in B individually. In the first place, `SATCOL`, creates a single soft clause per independent set I (if it is part of a conflict). An equivalent procedure that executes `UP` to find a conflict for each of the vertices in I, generates $|I|$ soft clauses and $|I|$ transformed-graphs during the reasoning. In addition, each transformation relaxes the clauses of the corresponding conflict with an additional literal, see §1.3, so emptying these clauses becomes more difficult in subsequent iterations. Keeping the number of soft clauses low (and of small size) is crucial for the overall efficiency of `SATCOL`. In the second place, `SATCOL` can also determine larger pruned sets, since it typically examines the vertices in B in a "better" order (according to independent sets) than the initial order. We illustrate this behaviour by means of the following example.

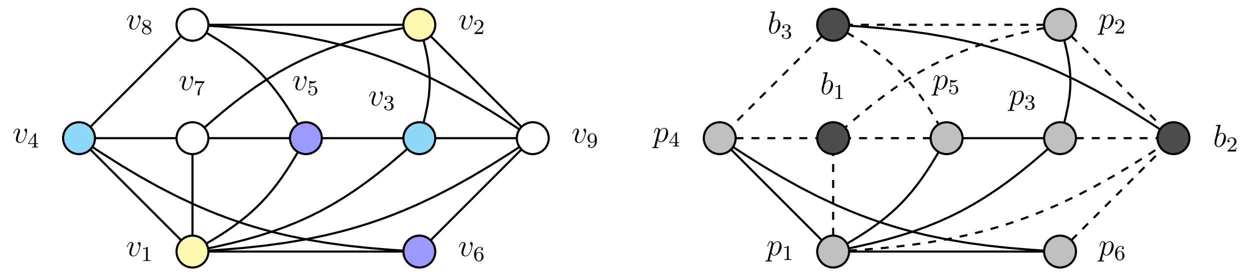


Figure 5: To the left, a subproblem graph $\hat{G}$ associated to a branching node of `CliSAT`. The coloured vertices correspond to the independent sets determined by `ISEQ`. To the right, the corresponding pruned set $P_C = \{p_1, p_2, \ldots, p_6\}$ and branching set $B = \{b_1, b_2, b_3\}$.

The left part of Figure 5 shows a new subproblem graph $\hat{G}$ associated to a branching node, with $|\hat{K}| = 1$ and $lb = 4$, so $\kappa = 3$ according to equation (7). The figure also shows the 3-colouring $C_3(G[P_C])$ determined by `ISEQ`. The right part of the figure depicts the relabelled vertices of the pruned set $P_C = \{p_1, p_2, \ldots, p_6\}$ (grey) and the branching set $B = \{b_1, b_2, b_3\}$ (black). The edges

with an endpoint in B appear dashed. `SATCOL` first examines the independent set $I = \{b_1, b_2\}$ from B, and the procedure `FL` determines a first conflict $\mathscr{C}_1 = \{(y_{p_3} \vee y_{p_4}), (y_{p_5} \vee y_{p_6}), (y_{b_1} \vee y_{b_2})\}$ in the associated $\text{PSAT}\big(\hat{G}[P_I], C_4(\hat{G}[P_I])\big)$, where, we recall, $P_I = P_C \cup I$. Consequently, P_I becomes the enlarged pruned set, and I is removed from B. In the next and final iteration, `SATCOL` considers the remaining vertex b_3 in B and finds a second conflict $\mathscr{C}_2 = \{(y_{p_1} \vee y_{p_2}), (y_{p_3} \vee y_{p_4}, z_1), (y_{p_5} \vee y_{p_6} \vee z_2), (y_{b_3})\}$ in the associated $\text{PSAT}\big(\hat{G}(\mathscr{C}_1), C_5(\hat{G}(\mathscr{C}_1))\big)$. For completeness we provide its 5 (unsatisfiable) soft clauses: $(y_{p_1} \vee y_{p_2})$, $(y_{p_3} \vee y_{p_4} \vee z_1)$, $(y_{p_5} \vee y_{p_6} \vee z_2)$, $(y_{b_1} \vee y_{b_2} \vee z_3)$ and (y_{b_3}). The added z literals correspond to the transformed-graph $\hat{G}(\mathscr{C}_1)$. As can be seen from the conflict $\mathscr{C}_2$, the reasoning involves (besides the unit clause of b_3) the soft clauses associated to the yellow, blue and cyan color classes in the figure.
Alternatively, we now consider the operations of the `UP` procedure on the vertices in B following the initial order, i.e., v_7, v_8 and v_9 (also b_1, b_3 and b_2). The first conflict determined by `UP` when setting y_{b_1} to true is $\mathscr{C}_1 = \{(y_{p_3} \vee y_{p_4}), (y_{p_5} \vee y_{p_6}), (y_{b_1})\}$, and a second conflict, when setting y_{b_3} to true, is $\mathscr{C}_2 = \{(y_{p_1} \vee y_{p_2})(y_{p_3} \vee y_{p_4} \vee z_1), (y_{b_1} \vee z_3), (y_{b_3})\}$. At this point, `UP` is unable to find a third conflict in the associated $\text{PSAT}\big((\hat{G}(\mathscr{C}_1))(\mathscr{C}_2), C_6((\hat{G}(\mathscr{C}_1))(\mathscr{C}_2))\big)$. Its 6 soft clauses are: $(y_{p_1} \vee y_{p_2} \vee z'_1)$, $(y_{p_3} \vee y_{p_4} \vee z_1 \vee z'_2)$, $(y_{p_5} \vee y_{p_6} \vee z_2)$, $(y_{b_1} \vee z_3 \vee z'_3)$, $(y_{b_3} \vee z'_4)$ and (y_{b_2}), where z and z' are the added literals corresponding to the conflicts $\mathscr{C}_1$ and $\mathscr{C}_2$ respectively. It is not difficult to see that setting y_{b_2} to true is unable to turn into unit any of the remaining (relaxed) clauses.

2.3. The filtering phase of `CliSAT`

We now describe one of the main algorithmic contributions of `CliSAT`. After the `ISEQ` procedure terminates and computes a partial κ-coloring of a subproblem graph $\hat{G}$, i.e., determines the pruned set P_C, `CliSAT` attempts to find a $(\kappa + 1)$-colouring of $\hat{G}$, $C_{\kappa+1}(\hat{G})$, by checking if the branching set B is an independent set. If this is the case, clearly B is the last color class of $C_{\kappa+1}(\hat{G})$. `CliSAT` exploits such a colouring to further reduce the branching tree.

A branching node where `CliSAT` is able to determine a $(\kappa+1)$-coloring of $\hat{G}$, i.e., $\hat{G}$ is $(\kappa+1)$-partite, is called a $(\kappa + 1)$*-partite branching node*. In these "special" branching nodes it is necessary to add to $\hat{K}$ exactly one vertex from each of the $\kappa + 1$ color classes in order to improve the lower bound *lb*. This is true since, in $(\kappa + 1)$-partite graphs, only one vertex from each of the $\kappa + 1$ color classes can make part of a clique.

The left part of Figure 6 shows a $(\kappa+1)$-partite subproblem graph $\hat{G}$ (associated to a $(\kappa+1)$-partite branching node) with $\kappa = 3$ (assuming $|\hat{K}| = 1$ and $lb = 4$). The `ISEQ` procedure determines the 3 independent sets: $I_1 = \{v_1, v_4, v_5\}$, $I_2 = \{v_2, v_6\}$ and $I_3 = \{v_3\}$ (shown with different colors in the figure). Moreover, `CliSAT` is able to determine a 4-coloring of $\hat{G}$, since the branching set $B = \{v_7, v_8\}$ forms an independent set. In the example, the 4-clique $\{v_1, v_2, v_3, v_7\}$ improves the *lb* value and, as can be seen, each of its vertices belongs to one of the color classes of the 4-colouring.

`CliSAT` exploits a $(\kappa + 1)$-partite subproblem graph $\hat{G}$ by discarding some vertices from $V(\hat{G})$ that cannot improve the incumbent solution. We call these operations of `CliSAT` the *filtering* phase of the algorithm. To the best of our knowledge, no state-of-the-art MCP algorithm employs filtering techniques, which are however extensively used for solving (Binary) Constraint Satisfaction Problems, see, e.g., [34, 45]. Moreover, filtering is a core technique in state-of-the-art Constraint Programming solvers, see e.g., [21]. Filtering vertices from $V(\hat{G})$ can have a substantial impact on the size of the branching tree, since, once a vertex is filtered, it is discarded from the entire branching subtree rooted in a $(\kappa + 1)$-partite branching node. In contrast, the vertices in the pruned set can still make part of a solution in subsequent child nodes and thus cannot be discarded.

The general condition to filter a vertex of a $(\kappa+1)$-subproblem graph $\hat{G}$ is to prove that it cannot make part of any clique of size $(\kappa + 1)$ contained in $\hat{G}$. In practice, a necessary condition which is easier to check is that the vertex is not linked to any of the vertices from another color class, given a $(\kappa+1)$-coloring of $\hat{G}$. To evaluate this condition efficiently, `CliSAT` employs the procedure `FiltCOL`, which is described in §2.3.1. An alternative necessary condition is that the corresponding literal of the vertex in the associated PSAT$(\hat{G}, C_{\kappa+1}(\hat{G}))$ is a failed literal. This is evaluated by a second procedure `FiltSAT` presented in §2.3.2.

2.3.1. The `FiltCOL` filtering procedure

`FiltCOL` is the efficient color-based procedure employed by `CliSAT` to filter vertices. To better explain the operations of `FiltCOL`, we first introduce some definitions and notation. We call *reference node* the $(\kappa + 1)$-partite root node of a subtree, and denote $\hat{G}_R$ its associated subroblem graph. We call *reference (vertex) colouring*, $C_{\kappa+1}(\hat{G}_R)$, the κ-colouring computed by `ISEQ`, see §2.2, together with the color class determined by the branching set B. The reference colouring $C_{\kappa+1}(\hat{G}_R)$ *induces* a colouring $C^r_\alpha(\hat{G})$, $\alpha < (\kappa + 1)$, in any α-partite subproblem graph $\hat{G}$ of the subtree rooted in the reference node. Precisely, $C^r_\alpha(\hat{G})$ is obtained when the vertices of $\hat{G}$ preserve the color class of $C_{\kappa+1}(\hat{G}_R)$. `FiltCOL` exploits the fact that $C^r_\alpha(\hat{G})$ differs from $C_\alpha(\hat{G})$ to filter vertices of $\hat{G}$.

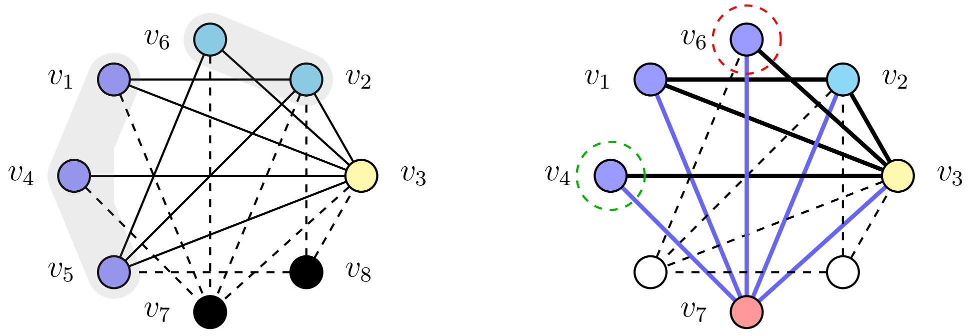


Figure 6: On the left, a $(\kappa+1)$-partite subproblem graph $\hat{G}$ of a $(\kappa+1)$-partite branching node with $\kappa = 3$, together with a 4-coloring. The independend sets $I_1 = \{v_1, v_4, v_5\}$, $I_2 = \{v_2, v_6\}$ and $I_3 = \{v_3\}$ are the first 3 colours; the branching set $B = \{v_7, v_8\}$, in black, is the 4-th colour. On the right part, the $(\kappa + 1)$-partite subproblem graph $\hat{G}$, with $\kappa = 2$, resulting from branching on the vertex v_7 in the reference node shown in the left part. Encircled vertices are filtered: v_6 (red) by `FiltCOL` and v_4 (green) by `FiltSAT`.

We illustrate the above notions by again referring to the example of Figure 6. Precisely, we consider its $(\kappa + 1)$-partite subproblem graph, with $\kappa = 3$, to be the reference branching node $\hat{G}_R$. The coloured vertices show the reference colouring $C_4(\hat{G}_R)$: $I_1(\hat{G}_R) = \{v_1, v_4, v_5\}$, $I_2(\hat{G}_R) = \{v_2, v_6\}$, $I_3(\hat{G}_R) = \{v_3\}$ and $I_4(\hat{G}_R) = \{v_7, v_8\}$ (left part). The right part of the figure depicts the $(\kappa + 1)$-partite child subproblem graph $\hat{G} = \hat{G}_R(v_7)$, with $\kappa = 2$, which results from branching on the vertex $v_7 \in G_R$ (pink). The edges of $\hat{G}$ appear in black; in blue the edges with an endpoint in v_7. The induced colouring $C^r_\alpha(\hat{G})$, $\alpha = 3$, is $I^r_1 = \{v_1, v_4\}$, $I^r_2 = \{v_2, v_6\}$ and $I^r_3 = \{v_3\}$. The coloured vertices correspond to $C_3(\hat{G})$, i.e. $I_1 = \{v_1, v_4, v_6\}$ and $I_2 = \{v_2\}$, together with the branching set $B = \{v_3\} = I_3$. As can be seen, $C^r_3(\hat{G}) \neq C_3(\hat{G})$, since the vertex v_6 (encircled in red) does not belong to the same color class.

In a nutshell, given a reference colouring $C_{\kappa+1}(\hat{G}_R)$ and an α-partite subproblem graph $\hat{G}$, $\alpha < \kappa+1$, `FiltCOL` computes the induced coloring $C^r_\alpha(\hat{G})$ while, at the same time, attempts to filter vertices of $\hat{G}$ that do not belong to its associated color class in $C_\alpha(\hat{G})$. In detail, the operations of `FiltCOL` are as follows. `FiltCOL` processes the vertices of $\hat{G}$ according to the initial order. At the beginning of each iteration, `FiltCOL` starts with an empty independent set I. The first time a vertex $v \in V(\hat{G})$ is added to I, the procedure determines a correspondence between I and the independent set $I(\hat{G}_R) \in C_{\kappa+1}(\hat{G}_R)$ to which v belonged in the reference coloring, i.e., $v \in I(\hat{G}_R)$. Then, for each additional vertex $w \in V(\hat{G})$ that can enlarge I, i.e., $I \cup \{w\}$ is an independent set, `FiltCOL` checks if the correspondence with $I(\hat{G}_R)$ is preserved, i.e., if $w \in I(\hat{G}_R)$. If this is the case, w is added to I. Alternatively, there are two possibilities: (a) the vertex w comes after the last vertex of $I(\hat{G}_R)$ according to the initial order, in which case it is filtered from $\hat{G}$. This is possible because w is not a member of $I(\hat{G}_R)$ and is non-adjacent to all its vertices. (b) the vertex w precedes the last vertex of $I(\hat{G}_R)$, in which case w is skipped for future iterations. In this case w cannot be filtered, since it could still be linked to other vertices of $I(\hat{G}_R)$ that are also in $\hat{G}$ and which have not yet been examined. The iteration ends when all the vertices in $V(\hat{G})$ have been considered. `FiltCOL` continues building independent sets until the induced colouring $C^r_\alpha(\tilde{G})$ is determined for the resulting reduced graph $\tilde{G}$.

Considering the suproblem graph $\hat{G}$ of Figure 6, `FiltCOL` is able to filter the vertex v_6 (encircled in red) in its first iteration with the following operations. Initially, I_1 is the empty set and vertex v_1 is added to I_1, establishing a correspondence with the independent set $I_1(\hat{G}_R) = \{v_1, v_4, v_5\}$ of the reference coloring $C_4(\hat{G}_R)$. Next, `FiltCOL` adds vertex v_4 to I_1 successfully, since $v_4 \in I_1(\hat{G}_R)$. Finally, v_6 is selected to enlarge I_1; however, since $v_6 \notin I_1(\hat{G}_R)$ and it has a higher index than the last vertex of $I_1(\hat{G}_R)$, i.e., v_5, it is filtered (removed) from the graph. In the remaining 2 iterations, the independent sets $I_2 = \{v_2\}$ and $I_3 = \{v_3\}$ are determined. The vertices of the reduced graph are $V(\tilde{G}) = \{v_1, v_2, v_3, v_4\}$.

Finally, we mention an important optimization related to `FiltCOL`. Once `CliSAT` executes both filtering procedures (`FiltCOL` and `FiltSAT`), and before branching, it keeps track of the vertices with the highest index from each of the α color classes of $C^r_\alpha(\hat{G})$. These α vertices, and not the ones from the reference colouring, are used to determine if a vertex is skipped or filtered during the execution of `FiltCOL` in the child nodes of $\tilde{G}$. In the example, and considering only the execution of `FiltCOL`, the vertices stored would be v_4, v_2 and v_3, for the independent sets I_1, I_2 and I_3 respectively.

2.3.2. The `FiltSAT` *filtering procedure*

Upon termination of `FiltCOL`, `CliSAT` executes the second filtering procedure `FiltSAT` on the reduced subproblem $(\kappa + 1)$-partite graph $\tilde{G}$, with $\kappa + 1 = \alpha$, attempting to filter additional vertices and, ultimately, fathom the node.

`FiltSAT` exploits the following observations concerning the associated PSAT($\tilde{G}$, $C^r_\alpha(\tilde{G})$): *i)* if a failed literal is found, its associated vertex cannot be part of an α-clique in $\tilde{G}$ and the corresponding vertex can be filtered, i.e., removed from $\tilde{G}$; *ii)* if a conflict is found, an α-clique cannot exist in $\tilde{G}$ and, therefore, the node can be fathomed. The latter is true since, as explained in §1.3, a conflict found in PSAT($\tilde{G}$, $C^r_\alpha(\tilde{G})$) reduces the color-based upper bound $\overline{w}(\tilde{G}) = \alpha$ by one unit. The vertex associated to a failed literal can be filtered for similar reasons. `FiltSAT` attempts to filter every vertex in $V(\tilde{G})$ by executing the procedure `FL` on the associated literals in PSAT($\tilde{G}$, $C^r_\alpha(\tilde{G})$), starting from the vertices of the branching set B. Any literal proven failed by FL is filtered from $V(\tilde{G})$. The procedure ends when all the vertices in $V(\tilde{G})$ have been examined or any one of the PSAT($\tilde{G}$, $C^r_\alpha(\tilde{G})$) α clauses becomes empty, in which case the node is fathomed.

We illustrate the operations of FiltSAT by referring again to the example from Figure 6. Precisely, we consider the reduced subproblem graph $\tilde{G}$ that results from the execution of FiltCOL, where, we recall $V(\tilde{G}) = \{v_1, v_2, v_3, v_4\}$. FiltSAT executes FL on the literals of PSAT($\tilde{G}$, $R_3(\tilde{G})$), starting with the literal associated to the branching set y_{v_3}. In this case y_{v_3} cannot be filtered, since it is part of the solution $\{v_1, v_2, v_3\}$, but y_{v_4} is found to be a failed literal (v_4 is non-adjacent to the singleton vertex v_2 of I_2). Consequently, v_4 (encircled in green in the figure) is removed from $\tilde{G}$. The resulting graph $\tilde{G}[\{v_1, v_2, v_3\}]$ is a 3-clique, so the filtering is optimal.

Finally, it is worth mentioning that for the subproblem graph $\hat{G}_R$ of the reference node, FiltCOL is not executed since there is no reference coloring available. In this case only FiltSAT is run on PSAT($\hat{G}_R$, $C_{\kappa+1}(\hat{G}_R)$).

2.4. Incremental upper bounds

One of the advantages of the incremental branching scheme is that upper bounds on the large subproblems can be efficiently computed based on upper bounds of previously examined smaller subproblems. Such upper bounds, denoted *incremental* in [17], have been employed in recent SAT-based algorithms for the MCP, see, e.g., [12, 13], and are also employed by the new algorithm CliSAT. We briefly describe the incremental bound employed by CliSAT in what follows.

Let $\hat{G} = (\hat{V}, \hat{E})$ be a subproblem graph whose vertices are sorted according to the ordering $(\hat{v}_1, \hat{v}_2, \dots \hat{v}_{|\hat{V}|})$. We define $\mu(\hat{G}) = (\mu[\hat{v}_1], \mu[\hat{v}_2], \dots, \mu[\hat{v}_{|\hat{V}|}])$ as an ordered collection of $|\hat{V}|$ values associated to $\hat{V}$, such that each value $\mu[\hat{v}_i]$ is a valid upper bound on the clique number of the graph induced by $\hat{v}_i$ together with the set of adjacent vertices to $\hat{v}_i$ that precede it in the ordering. Precisely, this induced graph corresponds with a branching subproblem of CliSAT's incremental branching scheme. Furthermore, and owing to the hereditary nature of cliques, a valid value (upper bound) $\mu[\hat{v}_i]$, $2 < i \leq |\hat{V}|$, can always be computed in $O(|\hat{V}|)$, given the values of μ associated to the vertices in $\hat{V}$ preceding $\hat{v}_i$ ($\mu[\hat{v}_1] = 1$), as follows:

$$\mu[\hat{v}_i] = 1 + \max\left\{\mu[u] : u \in \hat{V}_{i-1}(\hat{G}), (u, \hat{v}_i) \in \hat{E}\right\}, \qquad i = 2, \dots |\hat{V}|, \tag{8}$$

where $\hat{V}_{i-1}(\hat{G})$ is the set of vertices that precede $\hat{v}_i$ in $\hat{V}$.

The values of $\mu(\hat{G})$ are dynamically updated during the execution of CliSAT according to Equation 8, taking into account as well the size of the incumbent solution obtained after examining the corresponding subproblem. They provide a computationally cheap upper bounding condition for reducing the number of branching child nodes for a given a branching set B. This condition is evaluated just after the child subproblem is determined, and before the bounding techniques described in the previous sections are executed. In practice, CliSAT considers the vertex ordering (5), i.e., vertices in the pruned set P first, followed by the vertices in the branching set B, to determine the values of μ in every node, as in [12]. The specific details concerning how the μ values are employed by CliSAT to prune the branching tree are described in Algorithm 1.

2.5. The algorithm `CliSAT`

The algorithm `CliSAT` produces a branching tree that interleaves the bounding procedures `SATCOL`, `FiltCOL` and `FiltSAT` presented in the previous sections with the general branching scheme described in §2.1. Pseudocode for `CliSAT` is presented in Algorithm 1. In the pseudocode, the steps (1-3) correspond to the initial preprocessing phase of `CliSAT`, which is covered at the end of this section. Branching takes place in the recursive call to FindMaxClique (step 7), and is described in what follows.

Algorithm 1: `CliSAT` algorithm for the maximum clique problem

```
Input: A simple graph $G = (V, E)$
Output: A maximum clique $K$ in $G$ ($lb = |K| = \omega(G)$)
1  $(v_1, v_2, \ldots, v_n) \leftarrow$ Sort$(V)$
2  $K \leftarrow$ FindClique$(V)$, $lb \leftarrow |K|$
3  Initialize $\mu(G)$
4  for $i \leftarrow |K| + 1$ to $n$ do
5      $\hat{V} \leftarrow \{v \in V_{i-1}(G) : \{v, v_i\} \in E\}$                ▷ child subproblem
6      $P \leftarrow \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_{|K|}\}$
7      FindMaxClique$(G[\hat{V}], \{v_i\}, P, \mu(G))$
8      $\mu[v_i] \leftarrow lb$

9  FindMaxClique$(\hat{G}, \hat{K}, P, \mu)$
10 $\hat{\mu} \leftarrow \{\mu[v] : v \in P\}$
11 $B = \{b_1, \ldots, b_{|B|}\} \leftarrow \hat{V} \setminus P$
12 for $l \leftarrow 1$ to $|B|$ do
13     Compute $\hat{\mu}[b_l]$                                              ▷ see Equation (8)
14     if $\hat{\mu}[b_l] + |\hat{K}| \leq lb$                                 ▷ skip the $l$-th subproblem
15      then
16         $P \leftarrow P \cup \{b_l\}$ and $B \leftarrow B \setminus \{b_l\}$
17     else
18         $\tilde{V} \leftarrow \{P \cap N(b_l)\} \cup \{b_j \in B : j < l, \{b_j, b_l\} \in \hat{E}\}$   ▷ child subproblem
19         if $\tilde{V} = \emptyset$ then
20             if $|\hat{K}| > lb$ then $lb \leftarrow |\hat{K}|$ and $K \leftarrow \hat{K}$
21             return
22         if the current branching node is $(\kappa + 1)$-partite              ▷ Section 2.3
23          then
24             $(\tilde{P}, \tilde{B}) \leftarrow$ FiltCOL $(\tilde{V})$                           ▷ Section 2.3.1
25             $(\tilde{P}, \tilde{B}) \leftarrow$ FiltSAT $(\tilde{P}, \tilde{B})$                        ▷ Section 2.3.2
26         else
27             $(\tilde{P}, \tilde{B}) \leftarrow$ SATCOL $(\tilde{V})$;                          ▷ Section 2.2.2
28         if $\tilde{B} \neq \emptyset$ then
29             FindMaxClique$(\hat{G}[\tilde{V}], \hat{K} \cup \{b_l\}, \tilde{P}, \hat{\mu})$
30     $\hat{\mu}[b_l] \leftarrow \min\{\hat{\mu}[b_l], lb - |\hat{K}|\}$
```

At the end of its preprocessing phase, `CliSAT` branches on the vertices in V (according to the initial order established in step 1) starting from the $|K|$-th + 1 vertex (the first $lb = |K|$ vertices are skipped, since they cannot improve the initial clique by themselves). Then, for each vertex $v_i \in V$, $i = lb + 1, \ldots, n$, selected for branching, `CliSAT` determines the set of vertices $\hat{V}$ of the child subproblem, i.e., the adjacent vertices to v_i that precede it in V (step 5), computes a trivial Pruned Set P that comprises the first lb vertices in $\hat{V}$ (step 6) and calls the recursive procedure FindMaxClique to explore $G(\hat{V})$ (step 7). On backtracking, the value of μ corresponding to the branched vertex v_i is updated with lb (step 8).

Inside a branching node, the sets P and B always store the vertices according to their index number, the predetermined initial order of the vertices in G. This operation is done efficiently with the help of bitsets. The first task executed by FindMaxClique is to compute the values of $\hat{\mu}$ for the vertices in P. Since these vertices will not be branched on, preliminary tests established that the best compromise between efficiency and pruning ability was to give them the corresponding values in the father node (step 10). This efficient *inheritance* (originally described in [12], to the best of our knowledge, in combination with incremental branching) is possible because, as stated previously, the order of the vertices in P is preserved in every node. Since child subproblems are always subsets of father subproblems, the upper bound values concerning the latter are also valid for the former. In contrast, the values of $\hat{\mu}$ for each branching vertex in B are computed in step 13 according to Equation (8).

Pruning with the (upper bound) values of $\hat{\mu}$ occurs prior to the computation of each new child subproblem in step 14. If the pruning is successful, the vertex $b_l \in B$, $l = 1 \ldots\ |B|$ is added to P and removed from B (and the corresponding subproblem is not explored); otherwise, the child subproblem is determined in step 18. If the latter corresponds to a leaf node that improves the current solution, the incumbent clique is updated in step 20; else the child node is either processed according to the procedure `SATCOL`, or, in case the node is $(\kappa + 1)$-partite, according to the filtering procedures `FiltCOL` and `FiltSAT` (steps 22-27), see the Sections 2.2.2, 2.3.1 and 2.3.2 respectively. Finally, if at this point the child node has not been fathomed, `CliSAT` branches to the child subproblem in a recursive fashion (step 29).

We conclude this section by presenting the initial preprocessing phase of `CliSAT`. This phase comprises the following 3 operations executed in the first 3 steps of the algorithm: (i) an initial ordering of the vertices (step 1); (ii) a clique is computed heuristically (step 2); (iii) the collection of upper bound values μ is initialized (step 3). We describe the three operations in the following.

It is well established in the literature that the initial ordering of vertices plays an important role in BnB algorithms for the MCP, see, e.g. [19]. More precisely, state-of-the-art exact MCP algorithms employ two different orderings: (i) *degenerate* degree-based (`DEG-SORT`) and (ii) color-based (`COLOR-SORT`). The term *degenerate* in (i) refers to the fact that the sorting criterium (vertex degree) is dynamic, i.e., it is recomputed on the remaining unsorted vertices each time a vertex is selected. These two orderings are briefly presented in what follows; for a more in-depth analysis we refer the interested reader to [29].

The most frequently employed ordering is `DEG-SORT`, which can be traced back to [3]. In its basic form, the degree-based ordering $(v_1, v_2, \ldots, v_n)$ is such that v_n is a vertex with smallest degree in G, v_{n-1} is a vertex with smallest degree in the induced graph $G[V_{n-1}(G)]$, and so on. A successful color-based ordering for the MCP was first described in [17] to the best of our knowledge.
`COLOR-SORT` partitions V into k independent sets $\{I_1, I_2, \ldots, I_k\}$, such that I_1 is a maximum

independent set in G, I_2 is a maximum independent set in the induced subgraph $G[V \setminus I_1]$, and so on. Moreover, `CliSAT` considers the following order within each independent set I: for any pair of vertices $(v_i, v_j) \in I$ such that $1 \leq i < j \leq n$ (v_i precedes v_j), the degree of v_i is greater or equal to the degree of v_j. It is worth noting that finding partitions of maximum independent sets is as computationally hard as the original problem. However, hard MCP instances are normally dense or very dense, and, therefore, determining maximum independents sets is expected to be easy. In practice, to determine `COLOR-SORT` we execute `CliSAT` on the complement graph and search for maximum cliques with a fixed time limit.

Depending on the actual instance, `CliSAT` employs either `DEG-SORT` or `COLOR-SORT`. Extensive preliminary tests carried out showed that, in the general case, `COLOR-SORT` improves the efficiency of `CliSAT` when the size k of the independent set partition provides a tight upper bound on $\omega(G)$. If this is not the case, `DEG-SORT` is to be preferred. This is consistent with the results found in the literature, see, e.g., [29]. The procedure referred to as Sort(V) in step 1 of the pseudocode, selects the concrete ordering and is adapted from [29]; we refer the reader to the latter for further details. When Sort(V) terminates, the adjacency matrix of G is processed so that the vertex order becomes the index order of the vertices in G, i.e., we compute an isomorphic graph to G which becomes the new input graph to `CliSAT`. This optimization was originally described in the bitstring algorithm [26] to the best of our knowledge.

To compute an initial clique K_{inc} (step 2 of the initial preprocessing phase), `CliSAT` executes the multi-start tabu search heuristic AMTS [43] with a reduced time limit (see the computational section §3), and sets lb accordingly, i.e., $lb = |K_{inc}|$. Finally, $\mu(G)$ is initialized in step 3 according to Equation (8) ($\mu[v_1] = 1$). In addition, the first $|K_{inc}|$ values of μ are bounded by lb, and its remaining values are bounded by the size k of the independent set partition determined by `COLOR-SORT`.

3. Computationals

In this section we assess the computational performance of the new BnB algorithm `CliSAT` presented in this work. The goal of this computational study is twofold: *i)* to evaluate the performance of `CliSAT` with respect to its main components, covered in §3.2; *ii)* to compare `CliSAT` against the state-of-art algorithms in the literature, covered in §3.3 and 3.4.

3.1. Experimental setting and testbed of instances

All the experiments have been carried out on a 20-core Intel(R) Xeon(R) CPU E5-2690 v2@3.00GHz, disposing of 128 GB of main memory and running a 64 bit Linux operating system. The source code was compiled with gcc 5.4.0 and the -o3 optimization flag. The configuration parameters of `CliSAT` are as follows. In all the runs, during `CliSAT`'s initial preprocessing phase the heuristic AMTS is executed with a time limit of 0.05 seconds to determine an initial large clique. The time limit to determine each maximum independent set required by `COLOR-SORT` is also fixed to 0.05 seconds (see the description of the initial preprocessing phase of `CliSAT` at the end of the previous §2.5 for an explanation of this threshold).

For the tests, we have considered a testbed of 771 instances which comprises 501 structured instances (see Table 1) and 270 uniform random instances (see Table 4). The choice of orders and densities of the 270 random instances is in accordance with similar tests that can be found in the literature for exact MCP algorithms, see, e.g., Table 2 of [31]. The 501 structured instances can be divided into the following 4 categories (datasets): (*i*) the 86 instances from the 2nd DIMACS Challenge (`http://dimacs.rutgers.edu/programs/challenge/`); (*ii*) the 41 instances from the `BHOSHLIB` dataset; (*iii*) 223 representative instances derived from binary constraint satisfaction problems (BCSPs), which we denote the `CSPLIB` dataset and (*iv*) 151 hard MCP instances taken from different sources, hereafter the miscellaneous dataset `MISCLIB`. The 501 structured instances are publicly available in the github repository `https://github.com/psanse/CliSAT`. We consider this extended dataset wrt typical clique benchmarks employed elsewhere an additional contribution of this work, and hope it will stimulate further research in this field. Moreover, the repository also contains additional comparison performance results of `CliSAT` that complement those reported in this section.

The `DIMACS` and `BHOSHLIB` datasets are consistently employed in the literature to test exact MCP algorithms. The instances of our `CSPLIB` dataset are obtained as follows: vertices represent specific values of variable domains, and there is an edge between two vertices if the corresponding 2 values are compatible according to the constraints imposed on the original BCSP. It is worth mentioning that all the instances from the `BHOSHLIB` dataset also derive from BCSPs, which has motivated the choice of the `CSPLIB`. Last of all, the miscellaneous dataset `MISCLIB` comprises 4 families: (*i*) the 20 instances of the recent `evil` dataset [38], claimed to be harder than the `BHOSHLIB` dataset ; (*ii*) 3 instances derived from monotone matrices (`mon`) [37]; (*iii*) 78 instances (denoted `vc`) derived from the 200 vertex cover problems from the PACE Challenge (Track 1a) (`https://pacechallenge.org/2019/vc/`). Precisely, we have included those instances from the PACE Challenge with less than $8,000$ vertices; (*iv*) the first 50 (out of more than $49,150$) instances of the Gordon Royle's 17-clue Sudoku collection (`https://github.com/t-dillon/tdoku/blob/master/data.zip`), and referred to as `sud` in the following. In the `sud` instances, vertices represent a specific number and square of the 9×9 Sudoku grid, and there is an edge between two vertices if the corresponding (number, square) pairs are compatible according to the rules of the game. All the instances of `sud` have 729 vertices and a unique maximum clique of order 81.

Table 1 shows information related to the number of instances (#inst.), order ($|V|$) and density ($d(G)$) of the 501 dataset of structured instances classified by categories (datasets) and aggregated by families. As can be seen from the table, the average density of the different families of instances is high, i.e. with the exception of the family `c-fat`, the smallest average density is 0.49. Moreover, in 15 out of the 24 families, the average density is greater than 0.75. It is also worth noting that some of the instances were not solved by any of the algorithms tested and remain open.

3.2. Empirical analysis of the main components of `CliSAT`

In this section we evaluate the impact on the performance of `CliSAT` of its main components. For this purpose we have selected 67 representative instances from the 501 structured instance dataset. The collection covers most of the original families and has been chosen to be relatively easy for `CliSAT` i.e., none of the instances take the algorithm more than 500 seconds to prove optimality; the time limit for these experiments was fixed at $1,800$ seconds.

Table 2 summarizes the results obtained. The table shows the number of instances solved to proven optimality (#opt), and the time (in seconds) spent by the different algorithmic variants to prove optimality, i.e., those instances in which a time limit was reached are not included. Specifically, we report performance results for the following procedures: (*i*) the algorithm `CliSAT`; (*ii*) `CliSAT` without the `SATCOL` procedure (described in §2.2.2); (*iii*) `CliSAT` without the filtering

Table 1: Information on the dataset of 501 structured MCP instances considered in this work.

			number of vertices $\|V\|$			edge density $d(G)$		
category	family	#instances	min	avg	max	min	avg	max
`DIMACS`	`brock`	12	200	466.7	800	0.50	0.67	0.75
	`C`	7	125	1,410.7	4,000	0.50	0.79	0.90
	`c-fat`	7	200	371.4	500	0.04	0.19	0.43
	`dsjc`	7	250	678.6	1,000	0.10	0.50	0.90
	`gen`	5	200	320.0	400	0.90	0.90	0.90
	`ham`	6	64	448.0	1,024	0.35	0.78	0.99
	`john`	5	28	208.8	496	0.56	0.78	0.91
	`keller`	3	171	1,436.0	3,361	0.65	0.74	0.82
	`MANN`	4	45	1,194.8	3,321	0.93	0.98	0.999
	`p_hat`	15	300	800.0	1,500	0.24	0.49	0.75
	`san`	15	200	346.7	1,000	0.50	0.73	0.90
		86						
`CSPLIB`	`aim`	48	472	909.8	2,016	0.91	0.93	0.96
	`B`	25	529	627.0	729	0.72	0.74	0.75
	`comp`	25	330	616.4	1,050	0.88	0.93	0.96
	`D`	25	320	1,824.0	7,200	0.86	0.87	0.89
	`ehi`	25	2,079	2,144.5	2,205	0.95	0.95	0.95
	`geom`	25	1,000	1,000.0	1,000	0.88	0.90	0.91
	`lat`	25	613	3,023.0	6,961	0.97	0.98	0.99
	`RB2`	25	450	773.2	1,150	0.82	0.85	0.88
		223						
`MISCLIB`	`evil`	20	120	182.6	253	0.87	0.94	0.98
	`mon`	3	343	528.0	729	0.79	0.81	0.84
	`vc`	78	153	1,501.8	7,400	0.82	0.96	0.9995
	`sud`	50	729	729.0	729	0.63	0.63	0.63
		151						
`BHOSHLIB`	`frb`	41	450	1,086.1	4,000	0.82	0.87	0.93

procedures `FiltCOL` (§2.3.1) and `FiltSAT` (§2.3.2) for $(\kappa + 1)$-partite branching nodes, and (*iv*) `CliSAT` without both components.

The table shows that `CliSAT` requires both of its components to solve to proven optimality the 67 instances of the dataset. Removing one or both of the components leads to a number of instances remaining unsolved within the time limit. Specifically, if the filtering component, i.e., the procedures `FiltCOL` and `FiltSAT`, is removed, only 40 instances out of the possible 67 are solved, whereas if the component `SATCOL` is removed, 57 instances are solved. A first conclusion to be drawn is, consequently, that the filtering component has more impact on the overall performance of `CliSAT` than its counterpart `SATCOL`.

Furthermore, and according to the reported results, the impact of the filtering component of `CliSAT` is smaller on the `DIMACS` dataset than on the other 3 datasets. This might be explained by the fact that, in the instances tested from the `DIMACS` dataset, the average gap between the clique number and the color-based bound is larger than in the other 3 datasets (with the exception, possibly, of the `keller` instance). Consequently, the probability of finding $(\kappa + 1)$-partite nodes in the shallow levels of the branching tree is lower. It is worth noting, that the incremental nature of `CliSAT`'s branching scheme also favours the appearance $(\kappa + 1)$-partite nodes and might be one explanation for the "good" overall performance of `CliSAT` when combined with the `FiltCOL`\`FiltSAT` component.

Table 2 also shows that the component `SATCOL` is not dominated by the `FiltCOL`\`FiltSAT`

Table 2: Analysis of the main components of the algorithm CliSAT over a subset of 67 instances from the 501 dataset. The time limit (*tl*) was set to 1,800 seconds.

					CliSAT					
			CliSAT		no SATCOL		no FiltCOL\FiltSAT		no both	
categ.	fam.	#inst.	#opt	time [s]	#opt	time [s]	#opt	time [s]	#opt	time [s]
DIMACS	dsjc	1	1	86.2	1	101.3	1	93.5	1	88.2
	keller	1	1	23.1	1	111.4	1	208.3	1	293.7
	MANN	1	1	5.5	1	89.3	1	7.7	1	75.9
	p_hat	2	2	42.5	2	216.6	2	56.3	2	187.9
	san	1	1	39.1	1	60.4	1	44.6	1	51.2
		6	6		6		6		6	
CSPLIB	aim	5	5	460.7	1	1,464.2	1	21.1	1	115.2
	B	5	5	73.6	5	73.2	5	525.4	5	521.3
	comp	5	5	0.1	5	0.1	5	0.1	5	0.1
	D	5	5	222.6	5	259.4	4	532.4	4	529.2
	ehi	5	5	37.4	1	1,796.2	0	tl	0	tl
	geom	5	5	1.8	5	1.8	5	25.2	5	24.9
	lat	5	5	208.1	5	209.6	0	tl	0	tl
	RB2	5	5	15.6	5	15.7	4	33.6	4	33.7
		40	40		32		24		24	
MISCLIB	evil	5	5	28.3	3	818.9	3	44.8	3	154.1
	mon	1	1	148.7	1	229.4	1	224.5	1	276.2
	vc	5	5	140.3	5	197.7	0	tl	0	tl
	sud	5	5	0.7	5	0.7	2	759.3	2	748.0
		16	16		14		6		6	
BHOSHLIB	frb	5	5	69.1	5	70.7	4	112.9	4	114.3
Total		67	67		57		40		40	

component. For example, in the families MANN, p_hat, aim and evil, removing SATCOL leads to a degradation in the performance of CliSAT greater than if the other component is removed. It is the combined effect of both components that causes the excellent performance of CliSAT regarding the tested instances. In addition, removing both components can lead to a large degradation of the performance of CliSAT. In the case of the family keller, this degradation is greater than one order of magnitude. In the case of the families ehi, lat and vc, when both components are removed (also when just the FiltCOL\FiltSAT component is removed) CliSAT is unable to solve any instance within the time limit, whereas it solves all of them efficiently when both components are executed. Finally, it is worth pointing out that in a small number of cases, such as, e.g., the aim family, individual components may also have a negative effect in the overall performance of the algorithm.

3.3. Comparison between CliSAT *and* MoMC *over structured instances*

We compare in detail the performance of CliSAT against the exact combinatorial BnB algorithm MoMC [12]. MoMC is the most recent and successful SAT-based algorithm for the MCP to the best of our knowledge. In this section, we consider for comparison purposes the 501 structured instance dataset described in §3.1. The results obtained are reported in the Table 3. The tables show aggregated results by families for the categories (datasets) DIMACS, CSPLIB, MISCLIB and BHOSHLIB respectively, reporting the number of instances solved (#opt) and the average and maximum times in seconds spent by the 2 algorithms to solve the instances to proven optimality. We fixed the time limit to 15 days for families of instances which either had been consistently employed in the recent literature for similar purposes, i.e., the DIMACS and BHOSHLIB datasets, or we considered relevant, i.e., `evil`, `mon` and `vc`. For example, the `evil` family is interesting because, as mentioned in §3.1, the creators claim it to be harder than `BHOSHLIB`. For the `CSPLIB` dataset and the Sudoku family (`sud`), the time limit was reduced to $1,800$ seconds for practical purposes.

Table 3: Performance comparison of the algorithms `CliSAT` and `MoMC` for the entire strucured dataset of 501 instances: *i*) 86 `DIMACS` instances (time limit 15 days); *ii*) 223 `CSPLIB` instances (time limit $1,800$ sec); *iii*) 151 `MISCLIB` instances (time limit 15 days for the `evil`, `mon` and `vc` families, $1,800$ sec for the `sud` family) ; *iv*) 41 `BHOSHLIB` instances (time limit 15 days).

			`CliSAT`			`MoMC`		
				time [sec]			time [sec]	
categ.	family	#inst.	#opt	avg.	max.	#opt	avg.	max.
`DIMACS`	`brock`	12	12	827.7	3,652.2	12	**500.7**	1,867.1
	`C`	7	3	**9,263.2**	27,660.5	3	11,689.1	34,936.7
	`c-fat`	7	7	0.05	0.1	7	**0.03**	0.1
	`dsjc`	7	5	**17.5**	86.2	5	22.9	112.3
	`gen`	5	5	**0.1**	0.1	5	0.4	1.1
	`ham`	6	5	**0.1**	0.1	5	5.0	24.5
	`john`	5	3	0.10	0.2	3	**0.03**	0.1
	`keller`	3	2	**11.6**	23.1	2	78.9	157.7
	`MANN`	4	4	**90,361.5**	361,440.5	4	242,661.6	970,637.5
	`p_hat`	15	14	1,215.8	16,289.6	14	**1,082.2**	14,453.0
	`san`	15	15	**2.8**	39.1	15	3.7	51.5
		86	75			75		
`CSPLIB`	`aim`	48	**47**	133.5	1,459.7	20	215.8	1,620.4
	`B`	25	25	**33.3**	146.5	25	82.1	349.0
	`comp`	25	**25**	0.1	0.2	22	0.5	1.1
	`D`	25	**25**	51.7	858.4	22	18.3	191.5
	`ehi`	25	**25**	18.9	139.2	24	171.3	273.5
	`geom`	25	25	**0.4**	4.9	25	2.7	12.0
	`lat`	25	**16**	65.8	528.1	8	49.3	158.2
	`RB2`	25	**24**	153.0	1,514.6	22	64.4	662.5
		223	212			168		
`MISCLIB`	`evil`	20	**20**	4,176.4	54,828.4	17	13,721.2	165,792.0
	`mon`	3	**3**	22,722.8	68,019.1	2	209.3	415.9
	`vc`	78	**76**	7,262.6	406,912.1	63	62.9	1,463.0
	`sud`	50	**50**	1.5	16.9	1	1.6	1.6
		151	149			83		
`BHOSHLIB`	`frb30-15`	5	5	**0.1**	0.1	5	0.3	0.4
	`frb35-17`	5	5	**0.2**	0.4	5	1.0	1.5
	`frb40-19`	5	5	**1.0**	2.9	5	3.3	5.8
	`frb45-21`	5	5	**31.1**	100.7	5	76.8	168.1
	`frb50-23`	5	5	**612.8**	2,534.4	5	1,400.6	5,932.0
	`frb53-24`	5	5	**861.6**	1,557.3	5	1,758.0	3,415.1
	`frb56-25`	5	5	**19,907.5**	53,642.3	5	45,209.2	121,379.9
	`frb59-26`	5	5	**73,261.5**	108,058.4	5	146,109.4	257,586.4
	`frb100-40`	1	0	tl	-	0	tl	-
		41	40			40		
Total		501	476			366		

According to Table 3, `CliSAT` consistently outperforms `MoMC` solving more instances than its counterpart or spending less time, on average, when both algorithms solve the same number of instances to proven optimality. It is worth mentioning that there is no family in which `MoMC` solves more instances than `CliSAT`. In detail, `CliSAT` solves within the time limit 75 `DIMACS` instances out of a possible 86, 212 `CSPLIB` instances out of a possible 223, 149 `MISCLIB` instances out of a possible 151 and 40 out of the 41 instances of the `BHOSHLIB` dataset. Overall, it is able to solve to proven optimality 476 instances out of a possible 501. In contrast, `MoMC` solves the same number of instances as `CliSAT` from the `DIMACS` and `BHOSHLIB` datasets, but its performance drops to 168 instances from `MISCLIB`, and 83 from `CSPLIB`. Altogether, `MoMC` manages to solve within the time

limit 366 instances out of the possible 501, i.e., 110 instances less than CliSAT.
One possible explanation for the difference in the number of instances solved from the CSPLIB dataset, and also the evil and sud families (MISCLIB), is the pruning ability of the new filtering component of CliSAT. This is because, in these instances, there is more probability of finding $(\kappa + 1)$-partite branching nodes in the shallow levels of the tree.

Also from the reported results in the Table, the evil family is much harder to solve than the BHOSHLIB dataset for MoMC, in accordance with what is claimed in the literature, see [38]. However this is not the case for CliSAT, which manages to solve the 20 evil instances within the time limit. CliSAT also outperforms MoMC in the BHOSHLIB dataset, e.g., it is more than twice as fast in the instances from the family frb59-26. With respect to the 50 Sudoku instances, the difference in performance in favour of CliSAT is even more acute, solving all the instances in an average time of 1.5 seconds, whereas MoMC is able to solve just one instance. In addition, Table 2 clearly shows that the filtering component of CliSAT is the major cause of its good performance on the sud family.

To end the section, we highlight that even though CliSAT outperforms MoMC in most of the families, there are exceptions. Specifically, in the (hard) families brock and p_hat from the DIMACS dataset, MoMC significantly outperforms CliSAT. The brock family is very sensitive to initial pre-processing, so it is difficult to relate the poor performance of CliSAT on this family with its algorithmic components. In the case of p_hat, the computing performance of CliSAT is reasonably close to MoMC. The other 2 cases in which MoMC outperforms CliSAT are the family c-fat and 3 instances of the family john. These are easy instances solved by both algorithms in less than 1 second, and therefore not representative enough, in our opinion, to draw any conclusion.

3.4. Comparison between CliSAT *and* MoMC *over uniform random instances*

We also compare the algorithms CliSAT and MoMC over a set of 270 Erdös-Rényi random $G(n, p)$ graphs of different sizes ($n = |V| \in \{150, 200, 300, 500, 1000, 3000, 5000, 10000, 15000\}$) and edge densities (see Table 4 for the specific density values tested). These uniform random graphs are created according to a given probability (equal to the desired edge density value) of existence of an edge between any pair of vertices. Similar graphs are commonly used for testing clique algorithms; precisely, the testbed employed is the same as the one used in [31]. For each of the 27 different classes of random graphs considered, we run 10 instances with similar features. All instances were solved to optimality by both algorithms within the time limit, with the exception of the instances from the class $G(15000, 0.1)$, for which MoMC reported a memory problem in all 10 cases.

According to Table 4, CliSAT also outperforms MoMC in this testbed (even without taking into account the class $G(15000, 0.1)$). In detail, CliSAT is, on average, faster than MoMC in 21 classes out of a possible 27. The bigger differences in favour of CliSAT occur in the dense graphs of order 300 and 500. For example, CliSAT is more than twice as fast in the classes $G(300, 0.8)$ and $G(500, 0.7)$ than its counterpart. From the table, it can also be observed that the difference in performance between both algorithms become less acute as the order of the graphs increase for $n \geq 1000$ (which have low densities).

3.5. Comparison with algorithms designed for sparse real-world graphs

As mentioned in the introductory section, the algorithm CliSAT is tailored to solve hard dense graphs of small and medium order, i.e. $|V| \leq 25,000$. Existing algorithms for sparse large and

Table 4: Comparison between the algorithms `CliSAT` and `MoMC` over 270 uniform random graphs. In all the instances with $15,000$ vertices, `MoMC` reported a memory problem (indicated by "-").

			clique number $\omega(G)$			time [sec]	
$\|V\|$	$d(G)$	#inst.	min.	av.	max.	`CliSAT`	`MoMC`
150	0.7	10	16.00	16.50	17.00	**0.06**	0.05
150	0.8	10	22.00	22.90	24.00	0.08	**0.07**
150	0.9	10	35.00	37.10	41.00	**0.10**	**0.10**
150	0.95	10	53.00	54.50	57.00	0.05	**0.01**
200	0.7	10	18.00	18.20	19.00	**0.12**	0.17
200	0.8	10	24.00	25.10	26.00	**0.63**	1.11
200	0.9	10	39.00	40.70	42.00	**3.56**	3.73
200	0.95	10	60.00	62.30	64.00	**0.40**	0.49
200	0.98	10	91.00	94.70	98.00	0.05	**0.02**
300	0.6	10	15.00	15.40	16.00	**0.25**	0.37
300	0.7	10	20.00	20.20	21.00	**2.02**	5.03
300	0.8	10	28.00	28.50	30.00	**41.32**	98.18
500	0.4	10	10.00	10.70	11.00	**0.13**	0.23
500	0.5	10	13.00	13.30	14.00	**0.67**	1.36
500	0.6	10	17.00	17.00	17.00	**9.09**	12.86
500	0.7	10	22.00	22.40	23.00	**335.06**	728.74
500	0.994	10	261.00	266.20	276.00	**0.06**	0.65
1,000	0.2	10	7.00	7.50	8.00	**0.09**	0.17
1,000	0.3	10	9.00	9.20	10.00	**0.40**	0.78
1,000	0.4	10	12.00	12.00	12.00	**3.54**	5.15
1,000	0.5	10	15.00	15.00	15.00	**80.13**	113.47
3,000	0.1	10	6.00	6.40	7.00	**0.31**	0.98
3,000	0.2	10	9.00	9.00	9.00	**4.41**	5.19
5,000	0.1	10	7.00	7.00	7.00	**1.31**	3.18
5,000	0.2	10	9.00	9.10	10.00	62.87	**59.90**
10,000	0.1	10	7.00	7.40	8.00	21.62	**21.00**
15,000	0.1	10	8.00	8.00	8.00	**126.10**	-

massive real-world graphs, such as, e.g., [42, 30, 9] (see also the introductory section), exploit the specific topology of such networks, e.g. the fact that the clique number is usually "close" to the graph's *degeneracy* $\gamma(G)$. We recall that the degeneracy of a graph G (also known as the graph's k-core) is the maximum integer $\gamma(G)$ such that a subgraph G' of G exists with minimum degree $\delta(G')$ greater or equal than $\gamma(G)$. It follows that $\gamma(G) + 1$ is an upper bound on the clique number $\omega(G)$ of the graph. Such algorithms rely heavily on *kernelization*, i.e., a pre-processing stage in which the original input network is replaced by a smaller network called a *kernel*, and other reduction techniques inspired in the vertex cover problem, see, e.g., [9], that are employed in the nodes of a combinatorial *branch-and-reduce* tree.

The aim of this section is to establish an approximate (not exhaustive) performance comparison between `CliSAT` and the state-of-the-art algorithms for real-world graphs. For this purpose, we have selected the algorithms `dOmega` [42] and `BBMCSP` [30]. Both algorithms employ kernelization, but while `dOmega` is a branch-and-reduce algorithm, `BBMCSP` is a branch-and-bound algorithm, inspired in [26], that employs a tailored sparse bitstring encoding of the input graph.
The reported results are shown in the Tables 5 and 6. Table 5 presents results for the 86 instances of the `DIMACS` dataset, while Table 6 studies 20 real-world networks with less than $150,000$ vertices. The choice of instances in the latter case is motivated by the memory requirements of `CliSAT`, which are too large for massive graphs since it stores the full adjacency matrix in memory to operate efficiently with vertex neighbourhoods using bitmasks. Table 5 shows aggregated results for each family of instances, including the number of instances solved to proven optimality (#opt)

and the average and maximum times spent by the 3 algorithms. Table 6 shows the number of vertices and edges, the degeneracy ($\gamma(G)$), the clique number($\omega(G)$) and the time spent by the 3 algorithms for the 20 instances reported. In all the tests the time limit was fixed at $1,800$ seconds.

Table 5: Performance comparison of the algorithm `CliSAT` with the algorithms `BBMCSP` and `dOmega`, designed for real-world graphs, over the 86 instances of the `DIMACS` dataset. The time limit was set to $1,800$ seconds.

		CliSAT			BBMCSP			dOmega		
			time [sec]			time [sec]			time [sec]	
family	#inst.	#opt	avg.	max.	#opt	avg.	max.	#opt	avg.	max.
`brock`	12	**9**	164.8	1220.1	8	99.1	285.6	4	14.3	39.5
`C`	7	**2**	64.5	128.9	1	0.0	0.0	1	2.0	2.0
`c-fat`	7	**7**	0.1	0.1	**7**	0.0	0.0	**7**	0.0	0.0
`dsjc`	7	**5**	17.5	86.2	5	33.9	168.2	4	46.1	180.9
`gen`	5	**5**	0.1	0.1	2	0.6	0.8	2	1098.0	1470.3
`ham`	6	**5**	0.1	0.1	**5**	0.0	0.1	**5**	6.4	29.4
`john`	5	**3**	0.1	0.2	**3**	0.0	0.1	**3**	5.3	15.8
`keller`	3	**2**	11.6	23.1	1	0.0	0.0	1	3.6	3.6
`MANN`	4	**3**	1.9	5.5	**3**	78.8	236.1	**3**	12.6	37.3
`p_hat`	15	**13**	56.2	629.6	12	149.2	1605.8	8	163.3	555.9
`san`	15	**15**	2.8	39.1	**15**	6.9	75.4	4	135.0	467.3
Total	86	69			62			42		

The results are consistent with the expected behaviour of `CliSAT`. `CliSAT` clearly outperforms both real-world algorithms, i.e, `dOmega` and `BBMCSP`, in the `DIMACS` dataset. Precisely, `CliSAT` manages to solve 69 instances out of a possible 86 within the time limit, whereas `BBMCSP` solves 62 and `dOmega` is only capable of solving 42. On the other hand, `CliSAT` is outperformed by both `BBMCSP` and `dOmega` in the 20 real-world instance dataset. According to Table 6, kernelization is specially strong for the harder instances, i.e., the large networks with millions of edges such as, e.g., `soc-BlogCatalog` and `soc-buzznet`. It is also worth pointing out that the reduction techniques employed by `dOmega` in the search tree would seem to be less efficient in those cases when the gap between the graph's degeneracy and its clique number is large, such as, e.g., `soc-BlogCatalog`, `soc-buzznet` and `soc-LiveMocha`. In contrast, the algorithm `BBMCSP` is not affected by this fact, possibly because it relies on standard maximum clique techniques during search tailored for sparse graphs, see [30] for further details.

3.6. Comparison with additional MCP exact approaches

In order to provide a broader picture of the performance of `CliSAT`, we provide a comparison against integer linear programming (ILP) formulations, solved by a general purpose ILP solver, and 3 additional effective combinatorial branch-and-bound algorithms from the literature.

Let x_v be a binary variable taking value 1 if and only if vertex $v \in V(G)$ belongs to the maximum clique. The natural ILP formulation for the MCP reads as follows:

$$\omega(G) = \max \sum_{u \in V} x_u \tag{9a}$$

$$x_u + x_v \leq 1, \qquad \forall (u,v) \in \overline{E}(G), \tag{9b}$$

$$x_u \in \{0,1\}, \qquad \forall u \in V(G). \tag{9c}$$

The objective function (9a) corresponds to the total number of vertices of the maximum clique. Constraints (9b) impose that at most one vertex from each pair of non-adjacent vertices is selected.

Table 6: Performance comparison of the algorithm `CliSAT` with 2 state-of-the-art algorithms designed for real-world networks, over a subset of 20 instances (with $|V| < 150,000$) from the `DIMACS10`, `SNAP` and `Social Networks` collections.

						`BBMCSP`	`dOmega`	`CliSAT`
source	name	$\lvert V\rvert$	$\lvert E\rvert$	$\gamma(G)$	$\omega(G)$	time [sec]	time [sec]	time [sec]
`SNAP`	`p2p-Gnutella24`	26,518	65,369	5	4	**0.0**	**0.0**	0.1
`SNAP`	`Cit-HepTh`	27,769	352,285	37	23	**0.1**	0.3	0.2
`DIMACS10`	`delaunay_n15`	32,768	98,274	4	4	**0.0**	**0.0**	0.2
`SNAP`	`Cit-HepPh`	34,546	420,877	30	19	**0.2**	**0.2**	0.2
`DIMACS10`	`cond-mat-2005`	40,421	175,691	29	30	**0.0**	**0.0**	0.2
`DIMACS10`	`fe-body`	45,087	163,734	6	6	**0.0**	**0.0**	0.3
`DIMACS10`	`t60k`	60,005	89,440	2	2	**0.0**	0.1	0.5
`DIMACS10`	`wing`	62,032	121,544	3	3	0.1	**0.0**	0.5
`DIMACS10`	`delaunay_n16`	65,536	196,575	4	4	**0.1**	**0.1**	0.6
`SNAP`	`soc-Epinions1`	75,879	405,740	67	23	**0.2**	0.6	0.9
`DIMACS10`	`fe-tooth`	78,136	452,591	7	5	**0.2**	0.3	1.1
`SNAP`	`soc-Slashdot0902`	82,168	504,230	55	27	**0.2**	0.4	1.1
`Social`	`soc-BlogCatalog`	88,784	2,093,195	221	45	**3.9**	186.5	236.8
`DIMACS10`	`fe_rotor`	99,617	662,431	8	5	**0.3**	0.4	2.5
`Social`	`soc-buzznet`	101,163	2,763,066	153	31	**3.2**	43.0	10.1
`Social`	`soc-LiveMocha`	104,103	2,193,083	92	15	**2.1**	4.0	4.3
`DIMACS10`	`598a`	110,971	741,934	8	7	0.5	**0.3**	2.3
`DIMACS10`	`delaunay_n17`	131,072	393,176	4	4	0.2	**0.1**	2.4
`DIMACS10`	`fe-ocean`	143,437	409,593	4	2	**0.2**	**0.2**	2.9
`DIMACS10`	`144`	144,649	1,074,393	9	7	**0.7**	**0.6**	4.6
Total						**12.3**	237.2	271.4

It is well known that the linear programming (LP) relaxation of this formulation provides a very weak upper bound $\geq |V|/2$. In line with what is typically done in the literature to strengthen Constraints (9b), we consider a collection $\mathscr{C}$ of independent sets of the graph G, covering all the pairs of non-adjacent vertices $\{u,v\} \in \overline{E}(G)$. We therefore can replace Constraints (9b) by:

$$\sum_{u \in I} x_u \leq 1, \qquad \forall I \in \mathscr{C}. \tag{10}$$

Constraints (10) impose that no more than a single vertex is selected from each independent set $I \in \mathscr{C}$. Different heuristic procedures can be used for creating $\mathscr{C}$; we employ the one proposed in [2].

We use the IBM CPLEX Optimizer version 12.8, one of the state-of-the-art commercial solvers, to tackle the ILP model (9) enhanced by Constraints (10). According to extensive preliminary experiments, these constraints have a positive impact on the performance of the solver. The solver also generates several additional valid inequalities of type (10) (as well as several other families of general purpose valid inequalities) during the execution of its branch-and-cut scheme to further strengthen the LP relaxation of the formulation. For a fair comparison against the branch-and-bound algorithms, the solver is run in single-thread mode (with default parameters). We denote this methodology to solve the MCP based on an ILP formulation as `CPLEX` in the remainder of this section.

We now briefly introduce 3 additional combinatorial branch-and-bound algorithms for the MCP from the literature that are tested in this work:

- `IncMC2`[13]: An incremental SAT-based solver in the lines of `MoMC`, but which was developed some time earlier.

- `BBMCX`[28]: An incremental SAT-based solver, denoted *infrachromatic* (see also §1.2), whose reasoning scheme is restricted to determining (a subset of) conflicting independent sets of cardinality 3. The algorithm also employs a similar bit-encoding as `CliSAT` to represent the graph and sets of vertices in memory.
- `CLIQUER`[20]: To the best of our knowledge, the first successful exact MCP solver that employs the *Russian Doll Search* (RDS) branching scheme.

Figure 7: Performance profile of the algorithm `CliSAT` and other 5 state-of-the-art algorithms over the entire dataset of 501 instances. The time limit was fixed at $1,800$ seconds.

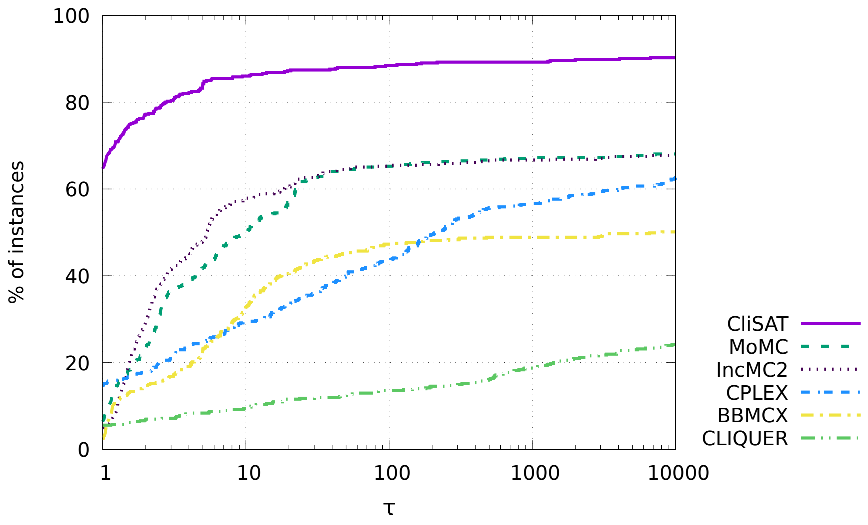
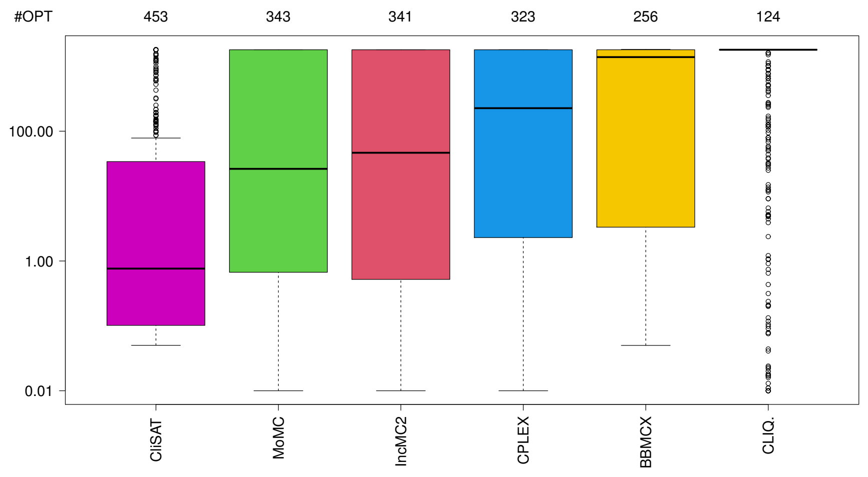


Figure 7 shows the performance profile of the 3 algorithms described above plus `CPLEX`, together with `CliSAT` and `MoMC`, over the 501 structured instance dataset. The performance profile is constructed in the following way. We compute the normalized time τ as the ratio of the computing time of each algorithm (which is ∞ if the instance is not solved to optimality within the time limit, here set to $1,800$ seconds) over the minimum computing time spent considering the 6 algorithms tested. For each value of τ on the horizontal axis, the vertical axis reports the percentage of instances for which the corresponding algorithm spent at most τ times the computing time of the fastest algorithm. The chart interpretation at both ends of the horizontal axis is as follows. At $\tau = 1$, the value of the curves is equal to the percentage of instances which the corresponding algorithm solves to optimality in less time. At the right-end, i.e., the largest value of τ, each curve corresponds to the percentage of instances solved to optimality by the specific algorithm. Consequently, in the performance profile the best performance is achieved by those algorithms whose curves appear highest in the chart, "wrapping" the other curves.

According to Figure 7, the best performing algorithm is, clearly, `CliSAT`, which is the fastest in more than 63% of the instances (left-end of the figure), and also solves the largest amount, i.e., slightly over 90% (as shown by the intersection of its curve in the right-end). The algorithms `MoMC` and `IncMC2` are the second best performers according to the figure, `MoMC` solving to optimality 2 more instances (343) than its counterpart `IncMC2` (341); this represents slightly more than 63% of the 501 dataset in both cases. The fourth performer is `CPLEX`, which initially solves around 18% of the instances, and shows the best slope as τ increases, solving more than 64% of the instances to optimality. The worst performing solvers according to the figure are `BBMCX`, which solves more than 51% of the instances and, finally, `CLIQUER`, which manages to solve slightly over 24%.

We end the section by showing in Figure 8 the computing time boxplots of the 6 algorithms. The figure plots the time (in logarithmic scale) spent by each algorithm through their quartiles; the lines extending vertically from the boxes indicate the variability outside the upper and lower quartiles. Above the upper quartile, the outliers (heterogeneous results) are plotted as individual points. Figure 8 evidences the superior computing times of `CliSAT`, and is consistent with the results reported in the performance profile.

Figure 8: Box plots of the performances of `CliSAT` and other 5 state-of-the-art algorithms over the entire dataset of 501 instances. The time limit was fixed at $1,800$ seconds.



4. Conclusions and future work

In this paper we present a very efficient combinatorial branch-and-bound exact algorithm `CliSAT` for the maximum clique problem. `CliSAT` combines all the recent state-of-the-art techniques with two new bounding procedures: (*i*) a filtering phase which exploits the notion of $(\kappa+1)$-partite branching nodes, i.e., nodes that are associated to a $(\kappa+1)$-partite graph and which require precisely a $(\kappa+1)$-clique to improve the incumbent solution; (*ii*) a partial maximum satisfiability-based procedure that prunes branching candidate vertices grouped according to independent sets, instead of individually. Our implementation has been extensively tested over a dataset of more than 700 instances from the literature, where it outperforms the state-of-the-art algorithms sometimes by several orders of magnitude.

A number of conclusions may be drawn from the tests. To begin with, empirical evidence suggests that the two new bounding techniques presented do not dominate each other and that the filtering phase of `CliSAT` is more effective in those instances where the gap between the chromatic number and the clique number is "small". Another conclusion is that, contrary to what is suggested in the recent paper entitled *Why is Maximum Clique Often Easy in Practice?* [42], the problem remains very hard in practice, as witnessed by the instances that could not be solved to proven optimality by any of the algorithms tested. Clearly, further breakthroughs will be required to solve these very hard instances. An open question is whether these breakthroughs will come in the form of new heuristics for partial maximum satisfiability or in some other form. Another open question is the impact that the incremental branching scheme of `CliSAT` has on the effectiveness of its filtering phase. Intuitively, it would seem that incremental branching favours the appearance of $(\kappa+1)$-partite branching nodes in the shallow levels of the branch-and-bound tree, which, in turn, might be pruned with higher probability by the filtering phase of `CliSAT`.

Acknowledgements

This work is partially funded by the Spanish *Agencia Estatal de Investigación* (PID2020-113096RB-I00 / AEI / 10.13039/501100011033) through the project ACOGES (COGnitive personal Assistance for Social Environments).

References

[1] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[2] A. Bettinelli, V. Cacchiani, and E. Malaguti. A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, 29(3):457–473, 2017.

[3] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6): 375–382, 1990.

[4] S. Coniglio, F. Furini, and P. San Segundo. A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *Eur. J. Oper. Res.*, 289(2):435–455, 2021.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[6] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *European Symposium on Algorithms*, pages 485–498, 2002.

[7] F. Furini, I. Ljubic, S. Martin, and P. San Segundo. The maximum clique interdiction problem. *Eur. J. Oper. Res.*, 277(1): 112–127, 2019.

[8] F. Furini, I. Ljubic, P. San Segundo, and Y. Zhao. A branch-and-cut algorithm for the edge interdiction clique problem. *Eur. J. Oper. Res.*, 294(1):54–69, 2021.

[9] D. Hespe, S. Lamm, C. Schulz, and D. Strash. Wegotyoucovered: The winning solver from the pace 2019 challenge, vertex cover track. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 1–11, 2020.

[10] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[11] H. Jiang, C.-M. Li, Y. Liu, and F. Manya. A two-stage maxsat reasoning approach for the maximum weight clique problem. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] C. Li, H. Jiang, and F. Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017.

[13] C. Li, Z. Fang, H. Jiang, and K. Xu. Incremental upper bound for the maximum clique problem. *INFORMS Journal on Computing*, 30(1):137–153, 2018.

[14] C. Li, Y. Liu, H. Jiang, F. Manyà, and Y. Li. A new upper bound for the maximum weight clique problem. *European Journal of Operational Research*, 270(1):66–77, 2018.

[15] C.-M. Li and Z. Quan. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 344–351, 2010.

[16] C.-M. Li and Z. Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Twenty-fourth AAAI conference on artificial intelligence*, pages 128–133, 2010.

[17] C.-M. Li, Z. Fang, and K. Xu. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 939–946, 2013.

[18] E. Malaguti and P. Toth. A survey on vertex coloring problems. *Int. Trans. Oper. Res.*, 17(1):1–34, 2010.

[19] E. Maslov, M. Batsyn, and P. Pardalos. Speeding up branch and bound algorithms for solving the maximum clique problem. *Journal of Global Optimization*, 59(1):1–21, 2014.

[20] P. R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, 2002.

[21] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming.* Elsevier, 2006.

[22] P. San Segundo and J. Artieda. A novel clique formulation for the visual feature matching problem. *Applied Intelligence*, 43(2): 325–342, 2015.

[23] P. San Segundo and D. Rodriguez-Losada. Robust global feature based data association with a sparse bit optimized maximum clique algorithm. *IEEE Trans. Robotics*, 29(5):1332–1339, 2013.

[24] P. San Segundo and C. Tapia. Relaxed approximate coloring in exact maximum clique search. *Computers & Operations Research*, 44:185–192, 2014.

[25] P. San Segundo, D. Rodriguez-Losada, F. Matia, and R. Galan. Fast exact feature based data correspondence search with an efficient bit-parallel mcp solver. *Applied Intelligence*, 32(3):311–329, 2010.

[26] P. San Segundo, D. Rodríguez-Losada, and A. Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[27] P. San Segundo, F. Matia, D. Rodriguez-Losada, and M. Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, 2013.

[28] P. San Segundo, A. Nikolaev, and M. Batsyn. Infra-chromatic bound for exact maximum clique search. *Computers & Operations Research*, 64:293–303, 2015.

[29] P. San Segundo, A. Lopez, M. Batsyn, A. Nikolaev, and P. M. Pardalos. Improved initial vertex ordering for exact maximum clique search. *Applied Intelligence*, 45(3):868–880, 2016.

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

[31] P. San Segundo, A. Nikolaev, M. Batsyn, and P. M. Pardalos. Improved infra-chromatic bound for exact maximum clique search. *Informatica*, 27(2):463–487, 2016.

[32] P. San Segundo, S. Coniglio, F. Furini, and I. Ljubić. A new branch-and-bound algorithm for the maximum edge-weighted clique problem. *European Journal of Operational Research*, 278(1):76–90, 2019.

[33] P. San Segundo, F. Furini, and J. Artieda. A new branch-and-bound algorithm for the maximum weighted clique problem. *Computers & Operations Research*, 110:18–33, 2019.

[34] P. San Segundo, F. Furini, and R. León. A new branch-and-filter exact algorithm for binary constraint satisfaction problems. *European Journal of Operational Research*, 299(2):448–467, 2022.

[35] S. Shimizu, K. Yamaguchi, and S. Masuda. A branch-and-bound based exact algorithm for the maximum edge-weight clique problem. In *International Conference on Computational Science/Intelligence & Applied Informatics*, pages 27–47, 2018.

[36] F. Stentiford. Face recognition by the construction of matching cliques of points. *Electronic Imaging*, 2019(8):404–1, 2019.

[37] S. Szabó. Monotonic matrices and clique search in graphs. *Annales Univ. Sci. Budapest., Sect. Computatorica*, 41:307–322, 2013.

[38] S. Szabó and B. Zaválnij. Benchmark problems for exhaustive exact maximum clique search algorithms. *Informatica*, 43(2), 2019.

[39] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *International Workshop on Algorithms and Computation*, pages 191–203, 2010.

[40] E. Tomita, T. Akutsu, and T. Matsunaga. *Efficient algorithms for finding maximum and maximal cliques: Effective tools for bioinformatics.* IntechOpen, 2011.

[41] S. Trukhanov, C. Balasubramaniam, B. Balasundaram, and S. Butenko. Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. *Computational Optimization and Applications*, 56(1):113–130, 2013.

[42] J. L. Walteros and A. Buchanan. Why is maximum clique often easy in practice? *Operations Research*, 68(6):1866–1895, 2020.

[43] Q. Wu and J. Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. *Journal of Combinatorial Optimization*, 26(1):86–108, 2013.

[44] Q. Wu and J. Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3): 693–709, 2015.

[45] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint solving and planning with Picat.* Springer, 2015.

Appendix

Extended comparison with state-of-the-art algorithms

This section extends the performance comparison between the algorithms `CliSAT` and `MoMC` reported in the computational section of the article, providing details for individual instances from the `DIMACS` (Table 7) and `BHOSHLIB` (Table 8) datasets, as well as the `evil` family from `MISCLIB` and the `D` family from `CSPLIB` (Table 9). All the tables report the number of steps (number of recursive calls), and the time in seconds spent by both algorithms to prove optimality. Instances with an entry of 0 steps indicate that the corresponding algorithm was able to prove optimality during its initialization phase.

Table 7: Extended performance comparison between the algorithms `CliSAT` and `MoMC` over a subset of 38 `DIMACS` representative instances.

				`CliSAT`		`MoMC`	
name	$\|V\|$	$d(G)$	$\omega(G)$	steps	time [sec]	steps	time [sec]
brock200_1	200	0.75	21	6,459	**0.2**	72,181	0.6
brock400_1	400	0.75	27	4,781,237	137.6	10,462,110	**112.8**
brock400_2	400	0.75	29	2,560,166	**76.7**	8,521,285	93.5
brock400_3	400	0.75	31	851,004	**34.0**	8,357,671	67.4
brock400_4	400	0.75	33	303,585	**14.8**	2,384,115	19.0
brock800_1	800	0.65	23	107,277,894	3,652.2	235,564,546	**1,828.2**
brock800_2	800	0.65	24	66,145,240	2,929.1	223,408,604	**1,867.1**
brock800_3	800	0.65	25	49,625,642	1,867.4	108,221,563	**752.3**
brock800_4	800	0.65	26	24,581,677	**1,220.1**	175,510,646	1,266.8
C250.9	250	0.90	44	3,417,260	**128.9**	6,639,713	130.6
C2000.5	2,000	0.50	16	675,567,772	**27,660.5**	3,185,621	34,936.7
dsjc500.5	500	0.50	13	33,179	**0.9**	280,166	1.8
dsjc1000.5	1,000	0.50	15	3,029,020	**86.2**	17,047,147	112.3
gen400_p0.9_55	400	0.90	55	0	**0.1**	4,049	1.1
gen400_p0.9_65	400	0.90	65	1	**0.1**	3,154	0.4
gen400_p0.9_75	400	0.90	75	1	**0.1**	3,360	0.2
hamming10-2	1,024	0.99	512	0	**0.1**	131,840	24.5
keller5	776	0.75	27	222,737	**23.1**	7,821,773	157.7
MANN_a45	1,035	0.996	345	19,130	**5.5**	83,195	8.7
MANN_a81	3,321	0.999	1100	179,444,376	**361,440.5**	406,990,467	970,637.5
p_hat300-3	300	0.74	36	2,555	**0.2**	9,336	0.3
p_hat500-2	500	0.50	36	305	**0.1**	4,075	0.2
p_hat500-3	500	0.75	50	75,644	**6.6**	372,329	9.3
p_hat700-2	700	0.50	44	2,547	**0.3**	16,365	0.8
p_hat700-3	700	0.75	62	650,084	**78.4**	2,678,530	100.5
p_hat1000-1	1,000	0.24	10	5,244	**0.2**	66,268	0.4
p_hat1000-2	1,000	0.49	46	132,244	**13.6**	749,907	17.7
p_hat1000-3	1,000	0.74	68	110,015,398	16,289.6	358,413,660	**14,453.0**
p_hat1500-1	1,500	0.33	12	79,771	**2.0**	498,585	3.1
p_hat1500-2	1,500	0.25	65	3,580,545	629.6	14,955,067	**565.9**
san400_0.7_1	400	0.70	40	1,436	**0.1**	5,185	0.3
san400_0.7_2	400	0.70	30	1,006	**0.1**	882	0.2
san400_0.7_3	400	0.70	22	0	**0.1**	1,153	0.3
san400_0.9_1	400	0.90	100	1	**0.1**	6,591	0.3
san1000	1,000	0.50	15	2,196	**0.2**	17,651	1.3
sanr200_0.9	200	0.90	42	31,120	**1.2**	74,213	1.3
sanr400_0.5	400	0.50	13	8,228	**0.2**	62,483	0.4
sanr400_0.7	400	0.70	21	1,588,204	**39.1**	5,518,655	51.5

Concerning the choice of the sorting procedure during the execution of the initial preprocessing phase of `CliSAT`, we report the following. In the case of the instances reported from the `BHOSHLIB` dataset (Table 8) and the `D` family (Table 9), `CliSAT` always selects the ordering of vertices determined by `COLOR-SORT`. In the case of the `DIMACS` instances (Table 7), the choice is as follows: (*i*) `COLOR-SORT` is selected in 4 instances out of the 9 `brock` instances reported; (*ii*) `COLOR-SORT` is selected for the families `gen` and `keller`; (*iii*) `DEG-SORT` is selected in the remaining families, i.e. `c-fat`, `dsjc`, `hamming`, `MANN`, `p_hat` and `san`. Finally, in the case of the `evil` family (Table 9), `DEG-SORT` is invariably the choice.

Table 8: Extended performance comparison of the algorithms `CliSAT` and `MoMC` over the 41 instances of the `BHOSHLIB` dataset.The time limit (*tl*) was set to 15 days.

				`CliSAT`		`MoMC`	
name	$\|V\|$	$d(G)$	$\omega(G)$	steps	time [sec]	steps	time [sec]
`frb30-15-1`	450	0.82	30	0	**0.1**	918	0.3
`frb30-15-2`	450	0.82	30	0	**0.1**	985	0.3
`frb30-15-3`	450	0.82	30	0	**0.1**	1,095	0.4
`frb30-15-4`	450	0.82	30	0	**0.1**	1,155	0.4
`frb30-15-5`	450	0.82	30	126	**0.1**	981	0.4
`frb35-17-1`	595	0.84	35	0	**0.1**	2,158	1.2
`frb35-17-2`	595	0.84	35	643	**0.4**	2,928	1.5
`frb35-17-3`	595	0.84	35	0	**0.1**	1,409	0.7
`frb35-17-4`	595	0.84	35	102	**0.1**	1,427	0.7
`frb35-17-5`	595	0.84	35	221	**0.2**	1,727	0.9
`frb40-19-1`	760	0.86	40	0	**0.1**	2,816	2.1
`frb40-19-2`	760	0.86	40	121	**0.1**	1,700	1.3
`frb40-19-3`	760	0.86	40	0	**0.1**	2,660	2.2
`frb40-19-4`	760	0.86	40	4,140	**2.9**	10,881	5.8
`frb40-19-5`	760	0.86	40	1,938	**1.8**	8,399	5.0
`frb45-21-1`	945	0.87	45	2,434	**2.0**	102,403	94.0
`frb45-21-2`	945	0.87	45	42,513	**28.1**	61,069	60.4
`frb45-21-3`	945	0.87	45	19,794	**15.1**	29,443	34.1
`frb45-21-4`	945	0.87	45	13,243	**9.8**	26,477	27.6
`frb45-21-5`	945	0.87	45	113,072	**100.7**	158,421	168.1
`frb50-23-1`	1150	0.88	50	333,199	**315.4**	613,547	604.7
`frb50-23-2`	1150	0.88	50	117,948	**142.0**	214,297	268.2
`frb50-23-3`	1150	0.88	50	2,149,755	**2,534.4**	4,902,490	5,932.0
`frb50-23-4`	1150	0.88	50	6,401	**4.3**	12,354	14.8
`frb50-23-5`	1150	0.88	50	66,901	**68.0**	225,983	183.5
`frb53-24-1`	1272	0.88	53	1,271,221	**1,557.3**	1,981,396	3,415.1
`frb53-24-2`	1272	0.88	53	152,126	217.7	126,751	**152.0**
`frb53-24-3`	1272	0.88	53	747,174	**800.8**	977,570	1,511.1
`frb53-24-4`	1272	0.88	53	1,320,753	**1,555.4**	1,893,507	2,986.5
`frb53-24-5`	1272	0.88	53	168,563	**177.1**	428,286	725.4
`frb56-25-1`	1400	0.89	56	23,724,283	**33,772.0**	41,817,946	73,432.0
`frb56-25-2`	1400	0.89	56	1,290,189	**1,275.9**	1,355,963	1,982.4
`frb56-25-3`	1400	0.89	56	3,780,230	**4,218.6**	4,895,951	9,824.0
`frb56-25-4`	1400	0.89	56	4,668,820	**6,628.6**	13,978,117	19,427.5
`frb56-25-5`	1400	0.89	56	36,028,666	**53,642.3**	100,513,778	121,379.9
`frb59-26-1`	1534	0.89	59	62,599,285	**107,239.4**	138,978,307	257,586.4
`frb59-26-2`	1534	0.89	59	75,062,511	**108,058.4**	64,467,920	178,912.5
`frb59-26-3`	1534	0.89	59	35,806,103	**56,605.6**	53,434,151	112,232.2
`frb59-26-4`	1534	0.89	59	44,200,880	**76,077.7**	83,521,902	172,087.0
`frb59-26-5`	1534	0.89	59	15,343,706	18,326.1	4,451,221	**9,729.0**
`frb100-40`	4000	0.93	100	-	tl	-	tl

Table 9: Extended performance comparison between the algorithms `CliSAT` and `MoMC` over: *i*) the 20 instances of the `evil` family (`MISCLIB`) with a time limit (*tl*) set to 15 days); *ii*) the 25 instances of the `D` family (`CSPLIB`) with a time limit set to $1,800$ seconds.

				`CliSAT`		`MoMC`	
name	$\lvert V \rvert$	$d(G)$	$\omega(G)$	steps	time [sec]	steps	time [sec]
`evil-N120-p98-chv12x10`	120	0.92	20	2,782	**0.1**	392,883	1.2
`evil-N120-p98-myc5x24`	120	0.97	48	47	0.05	1,505	**0.02**
`evil-N121-p98-myc11x11`	121	0.93	22	4,675	**0.1**	572,074	1.3
`evil-N125-p98-s3m25x5`	125	0.89	20	13,728	**0.2**	606,103	1.4
`evil-N138-p98-myc23x6`	138	0.87	12	82,781	**0.6**	983,869	2.6
`evil-N150-p98-myc5x30`	150	0.97	60	112	0.05	2,148	**0.02**
`evil-N150-p98-s3m25x6`	150	0.90	24	121,391	**1.2**	14,702,870	36.7
`evil-N154-p98-myc11x14`	154	0.94	28	109,508	**1.2**	17,449,596	54.2
`evil-N180-p98-chv12x15`	180	0.94	30	2,156,387	**21.9**	1,582,875,479	4,863.0
`evil-N180-p98-myc5x36`	180	0.97	72	115	0.06	3,259	**0.04**
`evil-N184-p98-myc23x8`	184	0.90	16	2,195,219	**17.0**	54,927,834	183.5
`evil-N187-p98-myc11x17`	187	0.95	34	1,838,267	**19.5**	1,371,470,102	5,115.4
`evil-N200-p98-s3m25x8`	200	0.92	32	8,425,011	**101.3**	976,980,140	3,491.2
`evil-N210-p98-myc5x42`	210	0.98	84	236	**0.1**	4,219	0.1
`evil-N220-p98-myc11x20`	220	0.95	40	78,774,365	**889.3**	-	tl
`evil-N230-p98-myc23x10`	230	0.91	20	145,397,825	**1,237.0**	1,756,363,669	53,718.4
`evil-N240-p98-chv12x20`	240	0.95	40	1,160,983,608	**13,353.0**	-	tl
`evil-N240-p98-myc5x48`	240	0.97	96	138	**0.1**	5,248	**0.1**
`evil-N250-p98-s3m25x10`	250	0.93	40	893,359,445	**13,057.7**	1,311,951,648	165,792.0
`evil-N253-p98-myc11x23`	253	0.95	46	4,643,934,432	**54,828.4**	-	tl
`rand-2-40-8-753-010-04`	320	0.88	39	501	**0.6**	3,162	0.7
`rand-2-40-8-753-010-32`	320	0.89	40	0	**0.1**	2,144	0.4
`rand-2-40-8-753-010-60`	320	0.88	39	265	**0.4**	3,028	0.6
`rand-2-40-8-753-010-88`	320	0.88	39	615	**0.8**	4,528	1.1
`rand-2-40-11-414-020-00`	440	0.87	39	956	**1.0**	4,409	1.5
`rand-2-40-11-414-020-28`	440	0.87	39	828	**0.8**	5,363	1.5
`rand-2-40-11-414-020-56`	440	0.87	40	459	**0.4**	2,947	0.9
`rand-2-40-11-414-020-84`	440	0.87	40	246	**0.2**	1,902	0.6
`rand-2-40-16-250-035-12`	640	0.87	40	107	**0.1**	2,247	1.1
`rand-2-40-16-250-035-40`	640	0.87	39	3,979	**2.3**	7,657	4.6
`rand-2-40-16-250-035-68`	640	0.87	39	775	**0.6**	3,124	1.5
`rand-2-40-16-250-035-96`	640	0.87	39	2,634	**1.6**	5,873	4.1
`rand-2-40-180-84-090-24`	7200	0.88	40	311	**13.7**	-	tl
`rand-2-40-180-84-090-52`	7200	0.88	40	147,375	**858.4**	-	tl
`rand-2-40-180-84-090-80`	7200	0.88	39	91,718	**191.7**	-	tl
`rand-2-40-25-180-050-08`	1000	0.86	40	638	**0.5**	2,904	2.9
`rand-2-40-25-180-050-36`	1000	0.86	40	3,578	**1.5**	6,005	5.0
`rand-2-40-25-180-050-64`	1000	0.86	39	872	**0.5**	2,603	3.2
`rand-2-40-25-180-050-92`	1000	0.86	40	474	**0.4**	3,235	2.8
`rand-2-40-40-135-065-20`	1600	0.87	40	179	**0.8**	2,497	7.0
`rand-2-40-40-135-065-48`	1600	0.87	40	2,599	**2.5**	4,079	9.3
`rand-2-40-40-135-065-76`	1600	0.87	39	6,188	**4.9**	13,064	18.5
`rand-2-40-80-103-080-16`	3200	0.87	39	47,252	**137.8**	38,205	191.5
`rand-2-40-80-103-080-44`	3200	0.87	40	74,117	**61.8**	9,158	93.2
`rand-2-40-80-103-080-72`	3200	0.87	40	739	**9.9**	5067	52.0