A new branch-and-bound algorithm for the Maximum Weighted Clique Problem

Pablo San Segundo

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

Fabio Furini

Université Paris-Dauphine, PSL Research University, CNRS, 75016 Paris, France
fabio.furini@dauphine.fr

Jorge Artieda

Universidad Politécnica de Madrid (UPM), Madrid, Spain
Center of Automation and Robotics (CAR), Madrid, Spain
jorge.artieda@upm.es

Abstract

We study the Maximum Weighted Clique Problem (MWCP), a generalization of the Maximum Clique Problem in which weights are associated with the vertices of a graph. The MWCP calls for determining a complete subgraph of maximum weight. We design a new combinatorial branch-and-bound algorithm for the MWCP, which relies on an effective bounding procedure. The size of the implicit enumeration tree is largely reduced via a tailored branching scheme, specifically conceived for the MWCP. The new bounding function extends the classical MWCP bounds from the literature to achieve a good compromise between pruning potential and computing effort. We perform extensive tests on random graphs, graphs from the literature and real-world graphs, and we computationally show that our new exact algorithm is competitive with the state-of-the-art algorithms for the MWCP in all these classes of instances.

Keywords: Maximum Weighted Clique Problem, Branch-and-Bound Algorithm, Computational Results.

1. Introduction

Given a simple undirected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, a subset $C \subseteq V$ of vertices is called a *clique* of G if any two vertices of C are connected by an edge in E. The *Maximum Clique Problem* (MCP) calls for determining the largest clique of

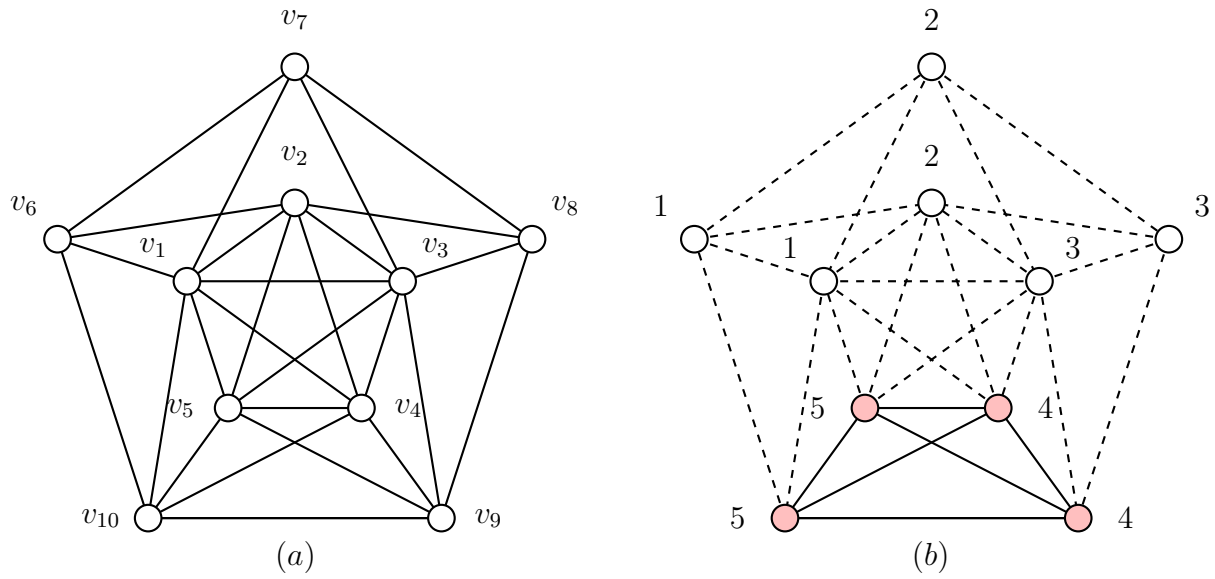

Figure 1: An example graph G of 10 vertices (part (a)) and the maximum weighted clique ($C = \{v_4, v_5, v_9, v_{10}\}$) of value $\omega(G, w) = 18$ (part (b)).

G. It is a fundamental $\mathcal{NP}$-hard problem, very challenging to solve from a computational viewpoint. It has been covered by a large body of literature, and it still is heavily studied. The size of the maximum clique is called the *clique number* of G and it is denoted by $\omega(G)$.

In this paper, we propose an exact branch-and-bound algorithm for the *Maximum Weighted Clique Problem* (MWCP). Let $w : V \to \mathbb{R}^+$ be a weight function; the MWCP calls for determining a clique C of G maximizing $w(C) = \sum_{v \in C} w(v)$, where $w(v)$ is the weight of a vertex $v \in V$, and $w(C)$ is the weight of a clique C. We denote by $\omega(G, w)$ the optimal MWCP solution value and call it the *weighted clique number* of G.

Let us consider the example graph of Figure 1 with ten vertices; in part (a) of the figure we depict the graph as well as a numbering of its vertices; in the part (b), we report the weights of the vertices. The value $\omega(G, w)$ of this graph is 18, as determined by the weight of the clique $C = \{v_4, v_5, v_9, v_{10}\}$. In part (b) of the figure, the vertices of the maximum weighted clique are colored in red and the edges with both endpoints in C are drawn with solid lines (the remaining edges show up dashed).

1.1. Literature review

Both the maximum clique and the maximum weighted clique problems are fundamental problems in graph theory and combinatorial optimization. Many are their practical applications, which span different fields. Examples can be found in computer vision [24, 38], coding theory[1], robotics [26], bioinformatics [2], combinatorial auctions [37] and network analysis [8]. The importance of determining large cliques is witnessed by the large body of literature

[1] `https://oeis.org/A265032/a265032.html`

developed for the MCP and MWCP. A recent algorithmic survey can be found in [36]; see also [34, 27, 28, 25, 29, 31, 32, 14, 15, 16, 17, 6, 11, 10, 22] for effective recent exact algorithms for both problems. The referenced algorithms are all branch-and-bound frameworks, which combine an enumeration scheme (that can be traced back to [3]) with strong upper and lower bounds.

As far as upper bounds for the MCP are concerned, since the seminal work of [5] most of the state-of-the-art exact algorithms employ the greedy sequential vertex coloring bound. Specifically, the number of colors of any legal coloring of a graph provides an upper bound on the size of the maximum clique (this bound has also been extended to the MWCP). Moreover, this bound has been improved via a number of post-optimization procedures. In [34], a local search procedure (called *reNumber*) is designed with the goal of obtaining stronger bounds and of reducing the number of colors prescribed by the greedy sequential coloring heuristic. More recently, greedy *infrachromatic* bounds have been introduced in the literature. The term infrachromatic first appeared in this context in [29], to refer to the potential of a maximum clique bound of being tighter than the chromatic number of the graph. The first infrachromatic algorithm is, to the best of our knowledge, the MaxSAT-based algorithm proposed in [18] (before the term infrachromatic was invented). Examples of recent infrachromatic algorithms are [29, 32, 14, 15]. It is worth noting that while the bounds described in [29, 32] are relatively simple to implement, the bounds described in [14, 15] are more complex and use MaxSAT technology. Finding large cliques in massive graphs has also received a lot of attention in the literature. Specific techniques are described in [31], enhanced by a bitstring representation and pre-processing of the massive graphs.

Determining good upper bounds for MWCP is generally more difficult than for the MCP, since the vertex weights give an additional degree of freedom to the problem. A well-known greedy color-based bound for MWCP is the sum of the weights of the vertices with maximum weight in each independent color set. Unfortunately, this bound has limited pruning ability in the general case. In [10], a stronger bound is designed by determining a set of overlapping independent color sets "covering" the weights of the vertices (see Section 2.1 for further details). The procedure proposed in [10] generates the independent color sets incrementally, aiming at determining a "good" covering of the vertex weights and, accordingly, a strong bound.

MaxSAT-based upper bounds for the MCP have also been extended to the MWCP, see e.g., [6] and [11]. The most recent algorithm of this type is described in [12]. However, these bounding functions are complex, and it is hard to implement them effectively.

To conclude this brief survey, we mention the work [17], in which the authors introduce the notion of a *weight cover* bounding function. The proposed bounding function determines a set of overlapping independent color sets as in [10]. However, it employs a more sophisticated heuristic to drive the generation of the independent color sets. Specifically, this algorithm takes into account both the graph structure and the weights of the vertices which have to be covered.

1.2. Paper contribution

This article describes a new efficient combinatorial branch-and-bound (B&B) algorithm for MWCP called `BBMCW` (BB stands for bitstring, MC for maximum clique and W for weighted). We design a new incremental fast-bounding procedure, which is able to effectively reduce the number of the branching nodes required by the implicit enumeration scheme. The new exact algorithm relies on effective data-encoding structures as well as an effective heuristic which is able to find feasible solutions of good quality required to initialize `BBMCW`. Extensive computational tests show that `BBMCW` is competitive with the state-of-the-art exact algorithms for the MWCP. Finally, we test `BBMCW` to compute the *Fractional Chromatic Number* of a graph, using `BBMCW` as the pricing algorithm for the *Column-Generation* phase (see Section 4.3). Also in this test-bed, we show that considerable speed-ups can be achieved when compared to the state-of-the-art algorithm.

The paper is structured as follows. Section 2 presents the new combinatorial MWCP branch-and-bound algorithm and Section 3 develops an example to illustrate how the algorithm operates. Extensive computational experiments are reported and discussed in Section 4. In Section 5, we present some conclusions.

1.3. Notation

We end this section by introducing the notation used in the remainder of the article. The complement graph $\overline{G} = (V, \overline{E})$ is a graph where $\overline{E} = \{u, v \in V, v \neq u : uv \notin E\}$. A subset of vertices $I \subseteq V$ is called an *independent set* if it is a clique in $\overline{G}$ or, equivalently, if no two vertices of I are connected by an edge in E. We also introduce the collection $\mathscr{I}$ of the all independent sets of G. Finally, let $N(u) = \{v \in V | uv \in E\}$ denote the *neighborhood* of a vertex $u \in V$ and $\overline{N}(u) = V \setminus (N(u) \cup \{u\})$ denote the *anti-neighborhood* of u.

2. The new combinatorial branch-and-bound algorithm

We present in this section the new combinatorial branch-and-bound (B&B) algorithm for the MWCP. Our implicit enumeration scheme is based on the concept of *Branching Set* and *Pruned Set* of vertices of the graph, and it belongs to the family of effective B&B algorithms proposed for the MCP and MWCP; see e.g., [25, 29, 32, 15, 11, 17].

Before the execution of the exact algorithm, a pre-preprocessing phase computes a heuristic solution C_h (maximal clique) as well as a lower bound $lb = w(C_h)$ (see Section 2.5).

During search, and at each node of the branching tree, we have a (non-maximal) clique $\hat{C} \subseteq V$ (feasible solution) and a *subproblem graph* $\hat{G} = (\hat{V}, \hat{E})$. This subproblem graph is the subgraph induced by the intersection of the neighboorhoods of the vertices in $\hat{C}$:

$$\hat{V} = \bigcap_{v \in \hat{C}} N(v), \quad \text{and} \quad \hat{E} = E(\hat{V}).$$

By construction, $\hat{V}$ is the set of vertices that can potentially be added to clique $\hat{C}$ in subsequent nodes of the branching tree (child nodes), and in each leaf of the branching tree clique $\hat{C}$ is maximal. At the root node of the branching tree, the clique is an empty set ($\hat{C} = \emptyset$) and the subproblem graph is the original graph ($\hat{G} = G$). The child nodes are created by selecting a vertex in $\hat{G}$ and including it into clique $\hat{C}$ in a repetitive fashion. During the implicit enumeration, if $w(\hat{C}) > lb$, the incumbent solution and the value lb are updated accordingly.

Only a subset of the nodes of $\hat{G}$ has to be considered as branching vertices to be fixed in $\hat{C}$ in order to create the child nodes of a given subproblem graph, thanks to the following property (also used, e.g., in [34, 27, 25]). Precisely, the vertex set of the subproblem graph $\hat{G}$ can be partitioned into two sets: the (subproblem) *Branching Set* B of candidate vertices for branching, and the (subproblem) *Pruned Set* $P = \hat{V} \setminus B$ of vertices, which do not have to be fixed in $\hat{C}$ in the current subproblem [2]. For a pair $(\hat{G}, lb)$ we want to determine a set $P \subseteq \hat{V}$ for which a MWCP upper bound $UB(P)$ is smaller than or equal to $lb - w(\hat{C})$. The upper bound is computed for graph $\hat{G}[P]$, which is the graph induced by the vertices of P. The Pruned and Branching Sets can be defined as follows:

$$P = \arg\max_{\hat{P} \subseteq \hat{V}} \left\{ |\hat{P}| : lb - w(\hat{C}) \geq UB(\hat{P}) \geq \omega(\hat{G}[\hat{P}], w) \right\}, \quad \text{and} \quad B = \hat{V} \setminus P. \tag{1}$$

Precisely, the set P is ideally the largest subset of vertices of $\hat{V}$ such that the defining property of P holds. Since determining an optimal set P can be computationally expensive, P is determined in a heuristic fashion, see Section 2.1. The optimal MWCP solution value of a specific subproblem is then either lb, or determined in one of the child subproblems that results when branching on the vertices in B. The computation of $UB(P)$ is oriented to reduce the size of the set B efficiently (see Section 2.1).

The bounding function $UB(P)$ can be designed *incrementally* and the vertices can be transferred from the set B to the set P as long as $UB(P) \leq lb - w(\hat{C})$. For each subproblem, initially the set B corresponds to $\hat{V}$ and P is the empty set. Similar ideas have also been used, e.g., in [25, 29, 32, 15, 16]. The key element of these families of branch-and-bound algorithms lies in the way the Pruned Set P is determined using the bounding function $UB(P)$. In the following section, we describe how our new and effective bounding function has been designed.

2.1. Outline of the bounding function

Our bounding function is based on the following MWCP model. Given a subproblem graph $\hat{G} = (\hat{V}, \hat{E})$, and by introducing a binary variable x_v for each vertex $v \in \hat{V}$ (which takes value 1 if and only if vertex v is chosen in the clique), a valid Integer Linear Programming (ILP)

[2] Vertices in P could make part of the Branching Set in deeper levels of the branching tree.

formulation for MWCP reads as follows:

$$\omega(\hat{G}, w) = \max \sum_{v \in \hat{V}} w(v) \cdot x_v \tag{2}$$

$$\sum_{v \in I} x_v \leq 1 \qquad I \in \mathscr{I}, \tag{3}$$

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

The objective function (2) maximizes the weights of the vertices in the clique. Constraints (3) impose that at most one vertex be selected from each independent set (an exponential family of constraints, one constraint for each independent set $I \in \mathscr{I}$).

Strong MWCP upper bounds can be obtained by computing feasible solutions of the dual of the Linear Programming (LP) relaxation of formulation (2)-(4)[3]. Given a (small) collection of independent sets $\tilde{\mathscr{I}} \subseteq \mathscr{I}$ (covering all the vertices of $\hat{G}$ at least once) and associating a non-negative continuous variable π_I with each of them, the dual of the LP relaxation of formulation (2)-(4), restricted to $\tilde{\mathscr{I}}$, reads as follows:

$$\omega(\hat{G}, w) \leq \min \sum_{I \in \tilde{\mathscr{I}}} \pi_I \tag{5}$$

$$\sum_{I \in \tilde{\mathscr{I}}: v \in I} \pi_I \geq w(v) \qquad v \in \hat{V}, \tag{6}$$

$$\pi_I \geq 0 \qquad I \in \tilde{\mathscr{I}}. \tag{7}$$

The objective function (5) minimizes the sum of the variable values, and constraints (6) impose to cover the weight $w(v)$ of each vertex $v \in \hat{V}$.

Any feasible solution $\tilde{\pi}$ of formulation (5)-(7) provides a valid upper bound for $\omega(\hat{G}, w)$ according to:

$$UB(\tilde{\mathscr{I}}, \tilde{\pi}) := \sum_{I \in \tilde{\mathscr{I}}} \tilde{\pi}_I \geq \omega(\hat{G}, w). \tag{8}$$

From equation (8), it is clear that the quality of the MWCP upper bound strongly depends on the collection of independent sets $\tilde{\mathscr{I}}$ and on the feasible solution $\tilde{\pi}$ of formulation (5)-(7). Several are the combinatorial ways of computing MWCP upper bounds based on feasible solutions of this formulation; see e.g., [10] and [17], where two effective procedures for computing such a bound are proposed.

In particular, strong MWCP upper bounds can be heuristically determined by covering the vertex set $\hat{V}$ with independent sets. Given a cover $\mathscr{C}$ of k independent sets: $\mathscr{C} = \{I_1, \ldots, I_k\}$, where each vertex of $\hat{G}$ is covered by one or more independent sets in $\mathscr{C}$, we define the

[3]Computing an optimal solution is $\mathcal{NP}$-hard, since the separation of constraints (6) requires the solution of the Maximum Weighted Independent Set Problem.

load $l(I, v)$ as the fraction of the weight $w(v)$ of a vertex $v \in \hat{V}$ covered by the independent set $I \in \mathscr{C}$. It immediately follows that a sufficient condition to have a feasible solution of formulation (5)-(7) is:

$$\sum_{I \in \mathscr{C}: v \in I} l(I, v) = w(v), \quad v \in \hat{V}. \tag{9}$$

Given a distribution of load values satisfying equation (9), and by defining:

$$\tilde{\mathscr{I}} := \mathscr{C} \text{ and } \tilde{\pi}_j := \hat{l}(I_j), \quad j = 1, 2, \ldots, k, \text{ where } \hat{l}(I) := \max_{v \in I}\{l(I, v)\}, \tag{10}$$

we obtain the following MWCP upper bound, called the *Covering Bound* in what follows:

$$UB(\mathscr{C}, \hat{l}) := \sum_{j=1}^{k} \hat{l}(I_j) \geq \omega(\hat{G}, w). \tag{11}$$

The Covering Bound is a valid bound also in the case of a "partial" cover $\tilde{\mathscr{C}}$ which corresponds to a cover of a subset $P \subseteq \hat{V}$ of the vertices of the graph $\hat{G}$ (the Pruned Set), and (9) holds for the vertices in P. Clearly, in this case the bound is valid for the subgraph $\hat{G}[P]$ induced by the vertices of P:

$$UB(\tilde{\mathscr{C}}, \hat{l}, P) := \sum_{j=1}^{k} \hat{l}(I_j) \geq \omega(\hat{G}[P], w). \tag{12}$$

We now describe the core ideas of the procedure we have designed to compute the Covering Bound of the family (12), used in our B&B algorithm to determine the Pruned Set P. We denote the new procedure by SFA (the acronym of the first letters of the surname of the authors) in the remainder of this article. We propose to compute the bound incrementally, building the partial cover $\tilde{\mathscr{C}}$ and the loads l via a two-phase computation.

First Phase. The SFA bounding procedure starts heuristically building a partial coloring: formally, a *partial coloring* of $\hat{G}$ is a partition of a subset $P \subseteq \hat{V}$ of vertices into k independent sets (where k represents the number of colors): $\tilde{\mathscr{P}} = \{I_1, \ldots, I_k\}$. The remaining vertices $B = \hat{V} \setminus P$ are non-colored. The sets P and B become the initial Pruning and Branching Sets which are further processed during the second phase. See Section 2.3 for further details on the sequential greedy coloring procedure. By defining $I(v)$ as the unique independent set in $\tilde{\mathscr{P}}$ containing the vertex $v \in P$, we have:

$$\tilde{\mathscr{I}} := \tilde{\mathscr{P}} \quad \text{and} \quad l(I(v), v) := w(v), \quad v \in P. \tag{13}$$

In practice, the greedy sequential coloring procedure stops when the Covering Bound (12) is below $lb - w(\hat{C})$ and no other vertices can be colored without exceeding it. Further details on this important point are given in Section 2.3.

Second Phase. In the second phase, we process one by one the remaining vertices $u \in B$ with the goal of extending the partial coloring $\tilde{\mathscr{P}}$ (or equivalently $\tilde{\mathscr{I}}$) to a partial cover $\tilde{\mathscr{C}}$. In other words, the goal is precisely to (potentially) reduce the branching set B. During the computation of SFA, the independent sets $I \in \tilde{\mathscr{I}}$ are sorted according to non-increasing loads l; that is, for any two nodes v_r, $v_s \in I$, if $r < s$ then $l(I, v_r) \geq l(I, v_s)$. Let $N(u, \hat{G}) = \{v \in \hat{V} | uv \in \hat{E}\}$ denote the *neighborhood* of u in the subproblem graph $\hat{G}$, and $\bar{N}(u, \hat{G}) = \hat{V} \setminus (N(u, \hat{G}) \cup \{u\})$ denote the *anti-neighborhood* of u. Specifically, we propose to build the partial cover $\tilde{\mathscr{C}}$ (or, equivalently, extend $\tilde{\mathscr{I}}$) by applying the following three (independent set) *splitting rules* denoted by R1, R2 and R3:

- **R1** (*insertion rule*): this rule is triggered when an independent set $I \in \tilde{\mathscr{I}}$ exists such that $I \subseteq \bar{N}(u, \hat{G})$ and $w(u) \leq \hat{l}(I)$. In this case, the vertex u can be added to the independent set I and its weight $w(u)$ can be fully covered by the current loads l of I. Thus we set:

$$I{:=}I \cup \{u\}, \quad l(I, u){:=}w(u), \quad \text{and} \quad P{:=}P \cup \{u\}. \tag{14}$$

  It is easy to see that the bound (12) on the graph $\hat{G}[P]$ induced by P remains unchanged.

- **R2** (*one-split rule*): this rule is triggered when an independent set $I \in \tilde{\mathscr{I}}$ exists such that $I \subseteq \bar{N}(u, \hat{G})$ and $w(u) > \hat{l}(I)$. The vertex u can be added to the independent set I but its weight $w(u)$ cannot be fully covered by the current loads l of I. In this case we set:

$$I{:=}I \cup \{u\}, \quad l(I, u){:=}\hat{l}(I), \quad \text{and} \quad P{:=}P \cup \{u\}. \tag{15}$$

  In addition, a new independent set $I_u{:=}\{u\}$ is created and added to $\tilde{\mathscr{I}}$ as follows:

$$\tilde{\mathscr{I}}{:=}\tilde{\mathscr{I}} \cup I_u, \quad \text{and} \quad l(I_u, u){:=}w(u) - \hat{l}(I). \tag{16}$$

  The bound (12) on the graph $\hat{G}[P]$ induced by P increases in this case by $w(u) - \hat{l}(I)$.

- **R3** (*two-split rule*): this rule is triggered when no independent sets $I \in \tilde{\mathscr{I}}$ exists such that $I \subseteq \bar{N}(u, \hat{G})$. Let $I = \{v_1, \ldots, v_{|I|}\} \in \tilde{\mathscr{I}}$ be an independent set sorted by non-increasing vertex loads, and let v_j be the first vertex adjacent to vertex u according to the ordering, see Section 2.3 for further details about the independent set partitioning procedure. Finally, let $l(I, v_1){:=}\hat{l}(I){:=}\alpha$.

  A first new independent set $\tilde{I}{:=}\{v_1, v_2, \ldots, v_{j-1}\} \cup \{u\}$ is created and added to $\tilde{\mathscr{I}}$ as follows:

$$\tilde{\mathscr{I}}{:=}\tilde{\mathscr{I}} \cup \tilde{I}, \quad l(\tilde{I}, v_i){:=}l(I, v_i) - l(I, v_j) \quad i = 1, \ldots, j-1, \tag{17}$$

$$l(\tilde{I}, u){:=} \min\{\alpha - l(I, v_j), w(u)\}, \quad \text{and} \quad P{:=}P \cup \{u\}.$$

A second new independent set $I_u := \{u\}$ is created and added to $\tilde{\mathscr{I}}$ as follows:

$$\tilde{\mathscr{I}} := \tilde{\mathscr{I}} \cup I_u, \quad \text{and} \quad l(I_u, u) := w(u) - l(\tilde{I}, u). \tag{18}$$

And, finally, the loads l of the independent set I are modified as follows:

$$l(I, v_i) := \begin{cases} l(I, v_j) & \text{if } i < j \\ \\ l(I, v_i) & \text{otherwise} \end{cases} \qquad i = 1, \ldots, |I|. \tag{19}$$

The load $\hat{l}$ of each independent set of the split are:

$$\hat{l}(I) := l(v_j), \quad \hat{l}(\tilde{I}) := \alpha - l(v_j), \quad \text{and} \quad \hat{l}(I_u) := \max\{w(u) - \hat{l}(\tilde{I}), 0\}.$$

Clearly, if $\hat{l}(I_u) = 0$, the weight of u is fully covered by I and $\tilde{I}$, and I_u can be discarded.

The bound (12) on the graph $\hat{G}[P]$ induced by P remains unchanged if $\hat{l}(\tilde{I}) \geq w(u)$; otherwise, it increases by $w(u) - \hat{l}(\tilde{I})$.

Moreover, since the splitting rules R2 and R3 can potentially increase the bound $UB(\tilde{\mathscr{C}}, \hat{l}, P)$ (12), they are both only triggered when $UB(\tilde{\mathscr{C}}, \hat{l}, P)$ remains smaller than or equal to $lb - w(\hat{C})$ (preserving in this way a feasible P set); see Section 2.4 for further details.

Compared with other bounds, such as the one proposed in [6], our computational tests show that SFA gives a good compromise between computational effort and pruning potential (see Section 4.5). In addition, the fact that the bound can be computed incrementally divides the computational effort in a more efficient way than if SFA was designed for the full subproblem. Specifically, each vertex transferred from B to P is expected to prune the search space. Similar advantages have been described in the literature for a few recent MCP exact algorithms, e.g., in [15] and [25]. The particular design of the heuristic rules is inspired, e.g., by [6], [10] and [17].

It is worth mentioning that the incremental nature of SFA makes it possible to stop the computation after a vertex fails to be transferred from set B to set P. However, in our SFA implementation we stop after all the vertices in B are examined once for each subproblem.

2.2. Example of the splitting rules of the SFA bounding function

In this section, we demonstrate how the three proposed splitting rules operate using the graph of Figure 1. In this example $\hat{G} = G$. For the sake of simplicity, we consider $\tilde{\mathscr{I}}$ to be made up of a single independent set I. This is the situation in which the first phase of SFA has determined a single color. Moreover, in this section, we simply denote the assignment symbol $:=$ as $=$, to simplify the notation.

- **Application of rule R1**: Consider $I = \{v_4, v_8\}$ with $\hat{l}(I) = w(v_4) = 4$. The weights of the two vertices are $w(v_4) = 4 = l(I, v_4)$ and $w(v_8) = 3 = l(I, v_8)$. We consider now vertex v_6. Since $I \subset \bar{N}(v_6)$ and $w(v_6) = 1 \leq \hat{l}(I)$, the R1 rule updates the unique independent set I to $I = \{v_4, v_8, v_6\}$ which now covers the weight $w(v_6)$ as well, by setting $l(I, v_6)$ to 1. The bound (12) on the graph induced by $P = \{v_4, v_6, v_8\}$ is equal to 4 ($|\tilde{\mathscr{I}}| = 1$).

- **Application of rule R2**: Consider $I = \{v_8, v_6\}$, with $\hat{l}(I) = w(v_8) = 3$. The weights of the two vertices are $w(v_8) = 3 = l(I, v_8)$ and $w(v_6) = 1 = l(I, v_6)$. We consider now vertex v_4. The situation is different from the previous case, because $w(v_4) = 4 > \hat{l}(I)$, and thus the weight of v_4 cannot be covered simply by inserting v_4 in I. By applying rule R2, we update the independent set I to $I = \{v_4, v_8, v_6\}$ and we set $l(I, v_4) = \hat{l}(I) = 3$. We then create a new independent set $I_{v_4} = \{v_4\}$ and we set $l(I_{v_4}, v_4)$ to $w(v_4) - \hat{l}(I) = 4 - 3 = 1$ ($\hat{l}(I_{v_4}) = 1$). The bound (12) on the graph induced by $P = \{v_4, v_6, v_8\}$ is again equal to 4 ($|\tilde{\mathscr{I}}| = 2$).

- **Application of rule R3**: Consider $I = \{v_8, v_6\}$ (as for R2) but we consider now vertex v_{10} with $w(v_{10}) = 5$ and $\alpha = \hat{l}(I) = w(v_8) = 3 = l(I, v_8)$. The vertex v_6 is the first vertex adjacent to v_{10} in I, so applying rule R3 does not change the independent set I but updates its load $l(I, v_8)$ to 1 and, accordingly, $\hat{l}(I) = w(v_6) = 1$. The rule then creates two new independent sets: $\tilde{I} = \{v_8, v_{10}\}$ and $I_{v_{10}} = \{v_{10}\}$ with:

$$l(\tilde{I}, v_8) = l(\tilde{I}, v_{10}) = \alpha - w(v_6) = 3 - 1 = 2, \quad \text{and} \quad \hat{l}(\tilde{I}) = 2,$$
$$l(I_{v_{10}}, v_{10}) = w(v) - l_{\tilde{I}}(v_{10}) = 5 - 2 = 3, \quad \text{and} \quad \hat{l}(I_{v_{10}}) = 3.$$

  The bound (12) on the graph induced by $P = \{v_6, v_8, v_{10}\}$ is equal to 6 ($|\tilde{\mathscr{I}}| = 3$).

The rules R1-R3 provide thus a methodology to obtain an upper bound with the goal of enlarging the Pruned Set P, at the same time.

2.3. Greedy sequential coloring procedure

At each node of the implicit enumeration scheme, a greedy sequential coloring procedure is executed to determine $\tilde{\mathscr{P}} = \{I_1, \ldots, I_k\}$. In this way, we obtain a *partial coloring* of the vertices of the subproblem graph $\hat{G}$ using k colors and a set of loads for the vertices in $\tilde{\mathscr{P}}$. This procedure tries to color the vertices in $\hat{V}$ by examining them in the order determined during pre-processing (see Section 2.5). As a result, it computes a partition of a subset $P \subseteq \hat{V}$ of vertices into k independent sets. Once the initial Pruned Set P is determined, it is then incrementally enlarged by applying the splitting rules described in Section 2.1.

The greedy sequential coloring procedure is outlined in Algorithm 1. `COLOR` receives as input a subproblem graph $\hat{G}$, the incumbent solution value lb and the weight $w(\hat{C})$ of the clique $\hat{C}$. This procedure is inspired by the greedy sequential independent-set coloring algorithm

described in [27] for the MCP, which incrementally computes maximal independent sets. It is implemented with an outer and an inner loop. Each execution of the inner loop computes a new independent set I, which is then added to $\tilde{\mathscr{P}}$. Specifically, a vertex $v \in \hat{V}$ is added to I if the bound (12) is lower than or equal to $lb - w(\hat{C})$ (see Step 8). The outer loop initializes the auxiliary data structures. With a slight abuse of notation, we define $\hat{l}(I) = 0$ if I is the empty set.

At the end of the procedure, `COLOR` also returns the gap value g between the bound (12) and $lb - w(\hat{C})$, defined as follows: $g = lb - w(\hat{C}) - UB(\tilde{\mathscr{P}}, \hat{l}, P)$. This gap is used as a budget to trigger the splitting rules R2 and R3, as long as the bound $UB(\tilde{\mathscr{P}}, \hat{l}, P)$ increases by no more than the gap.

Algorithm 1: `COLOR`$(\hat{G},\ lb,\ w(\hat{C}))$

Input: A graph $(\hat{G}, w)$. A solution value $lb \leq \omega(\hat{G}, w)$. The weight $w(\hat{C})$ of the clique $\hat{C}$. `// The order of` $\hat{V}$ `is determined during pre-processing`

Output: A feasible partial coloring $\tilde{\mathscr{P}} = \{I_1, \ldots, I_k\}$, the loads l for the vertices in $\tilde{\mathscr{P}}$, and a gap value g.

```
1  B ← V̂, k ← 0, ub ← 0
2  repeat
3      k ← k + 1
4      B_k ← B, I_k ← ∅
5      repeat
6          v ← the first vertex in B_k
7          B_k ← B_k \ {v}
8          if w(v) ≤ l̂(I_k) or ub + w(v) − l̂(I_k) ≤ lb − w(Ĉ) then
9              B_k ← B_k \ N(v)
10             B ← B \ {v}
11             I_k ∪ {v}                          // sorted by non-increasing weight
12             if w(v) > l̂(I_k) then
13                 ub ← ub + w(v) − l̂(I_k)
14             end
15         end
16     until B_k = ∅
17 until I_k = ∅
18 k ← k − 1, P ← V̂ \ B
19 return P̃ = {I_1, ..., I_k} , l, g ← lb − w(Ĉ) − ub
```

`COLOR` has a worst-case time complexity of $O(|\hat{V}|^2)$. However, compared with the greedy sequential independent-set coloring function described in [27], `COLOR` inserts the vertices in the independent sets according to non-increasing weight (see Step 11), so that each insertion is done in $O(|I|)$. Preserving vertices ordered in this way allows to efficiently compute the triggering conditions of the splitting rules, as well as the new independent set produced by the splitting rule R3. In practice, it is possible to reduce this extra complexity by setting a maximum cardinality threshold for every independent set I of $\tilde{\mathscr{P}}$. We comment on this optimization strategy in Section 2.6.

2.4. Reducing the branching set

The output of procedure COLOR provides a partial coloring $\tilde{\mathscr{P}}$ which determines our initial Pruned Set P; In other words, we obtain the vertices in $\tilde{\mathscr{P}}$ and the Branching Set $B = \hat{V} \setminus P$. We recall that the search procedure is only branching on vertices in B and it examines child subproblems by fixing vertices in $\hat{C}$ from set B. We now propose two bounding schemes: *the single independent set cover* and the *multiple independent set cover*, using the splitting rules R1, R2 and R3 from subsection 2.1 to reduce the size of this initial Branching set B.

2.4.1. Single independent set cover

We consider the problem of transferring a vertex v from the Branching set B to the Pruned set P, thus pruning the search space. This problem is equivalent to determining if $\omega(\hat{G}[P \cup \{v\}], w) \leq lb - w(\hat{C})$, given that $\omega(\hat{G}[P], w) \leq lb - w(\hat{C})$ by construction of P. The procedure COVER, described in Algorithm 2, outlines an efficient procedure for this task.

COVER receives as input a k independent set cover $\tilde{\mathscr{C}}$ of the Pruned Set P, where the vertices of the independent sets are sorted by non-increasing loads. It is worth mentioning that for the first call to COVER, $\tilde{\mathscr{C}}$ is the partial coloring $\tilde{\mathscr{P}}$ computed by COLOR. The template of COVER also contains the loads l of $\tilde{\mathscr{C}}$, the gap value g and a vertex $v \in B$ that is not in the cover. The procedure attempts to insert v into $\tilde{\mathscr{C}}$ while, at the same time, keeping the bound (12) lower than or equal to $lb - w(\hat{C})$. If COVER is successful, the input cover $\tilde{\mathscr{C}}$ is modified according to the splitting rules R1, R2 and R3 (see Section 2.1). For the single independent set cover described in this Section, each transfer attempt works with a single independent set $I \in \tilde{C}$.

We recall that firing the bounding rules R2 and R3 increases the number of independent sets of the cover by 1 and (at most) 2, respectively. In order to avoid an explosion of the size of $\tilde{\mathscr{C}}$, we do not allow more than one new independent set for each transferred vertex from the set B to the set P in any subproblem. Thus, procedure COVER does not include the singleton set $I_v = \{v\}$ when rules R2 and R3 are triggered; consequently, the transferred vertex v is covered by a single independent set in the new cover, independently of the splitting rule that is fired. The P defining property is assured by Steps 11 and 20, limiting the increment of the bound $UB(\hat{\mathscr{C}}, \hat{l}, P)$ to the available gap value g. We further recall that R1 is the only rule that guarantees that the bound will not change, but it can seldom be applied.

To better understand how COVER operates, we now summarize the structural properties of $\tilde{\mathscr{C}}$ when each splitting rule is succesful. In what follows, we denote by $\tilde{\mathscr{C}}_0$ the input cover to procedure COVER and by $\tilde{\mathscr{C}}$ the new cover computed by the procedure. Also let $\alpha(I)$ denote the maximum load $\hat{l}(I)$ of the independent set I of $\tilde{\mathscr{C}}_0$. Let I_j be the candidate independent set used to trigger the splitting rules.

- **When R1 is applied to (v, I_j):** There is no increment of the bound $UB(\tilde{\mathscr{C}}_0, \hat{l}, P)$. Also, the cardinality of the cover remains unchanged ($|\tilde{\mathscr{C}}| = |\tilde{\mathscr{C}}_0|$) and so do the loads. The load assigned to the transferred vertex v is $w(v)$, because it is covered completely by I_j.

- **When R2 is applied to (v, I_j):** The bound $UB(\tilde{\mathscr{C}_0}, \hat{l}, P)$ is increased by $w(v) - \alpha(I_j)$. The cardinality of the cover remains unchanged as in the previous rule, and so do the loads. Finally, the load assigned to the transferred vertex v is now $l(I_j, v) = \alpha(I_j)$ (see Step 12), which is the part of the weight of v that is covered. The remaining weight is accounted by the gap value.
- **When R3 is applied to (v, I_j):** The bound $UB(\tilde{\mathscr{C}_0}, \hat{l}, P)$ increases by $\max\{w(v) - (\alpha(I_j) - l(I_j, v_l)), 0\}$, where v_l is a vertex adjacent to the vertex v with maximum load in I_j. In other words, it is the first vertex adjacent to v according to the ordering of I_j. The independent set I_j remains unchanged, but the loads of the vertices preceding[4] v_l are changed to $l(I_j, v_l)$. Moreover, the cardinality of the cover increases by one: $|\tilde{\mathscr{C}}| = |\tilde{\mathscr{C}_0}| + 1$. The new independent set $\tilde{I}_j$ contains all the vertices in I_j preceding v_l, as well as v. The loads of these vertices in $\tilde{I}_j$, except v, are reduced by $\alpha(I_j) - l(I_j, v_l)$. Finally, the load of the vertex v is set to $l(\tilde{I}_j, v) = \min\{\alpha(I_j) - l(I_j, v_l), w(v)\}$ in Step 24.

2.4.2. Multiple independent set cover

The procedure `COVER`, outlined in the previous section, works with a single independent set of the input cover $\tilde{\mathscr{C}}$ to determine if a vertex $v \in B$ can be added to $\tilde{\mathscr{C}}$ (and the Pruned Set P) according to the defining property of P (see (1)). We now describe an additional efficient procedure, called `MCOVER`, which works with multiple independent sets. `MCOVER` is outlined in Algorithm 3; it has the same template as `COVER` and operates as follows.

`MCOVER` examines one by one the independent sets in the cover $\tilde{\mathscr{C}}$ and, each time an independent set $I \in \tilde{\mathscr{C}}$ triggers a splitting rule, the procedure determines the maximum part of the weight $w(v)$ that I can cover (which depends on the specific rule), and increments a variable Δ. The goal of `MCOVER` is to identify a subset of independent sets $\tilde{\mathscr{I}} \subseteq \tilde{\mathscr{C}}$ such that it covers $w(v)$ completely using Δ, and, at the same time, preserves the validity of P. This is evaluated by the expression $\Delta + g \geq w(v)$ in Step 7, which is relative to the triggering of R1 and R2, and in Step 15, relative to the triggering of R3. If such a subset $\tilde{\mathscr{I}}$ is found, `MCOVER` returns `TRUE` to indicate that v can be transferred to P without incrementing the bound (12) by more than the gap value g (and g is updated accordingly); otherwise `MCOVER` returns `FALSE` after having examined all the indepedent sets of $\tilde{\mathscr{C}}$. It is worth noticing that $\tilde{\mathscr{C}}$ is not modified during the operation of `MCOVER` for efficiency reasons. In addition, we do not split the independent sets in $\tilde{\mathscr{I}}$ (following the rules R1, R2 and R3) to avoid an explosion of the cardinality of $\tilde{\mathscr{C}}$. Specifically, the algorithm returns the set $\tilde{\mathscr{I}}$, which is removed from the input cover $\tilde{\mathscr{C}}$ in subsequent calls to `MCOVER` when examining the remaining vertices in B.

As far as specific implementation details are concerned, we highlight the following. The algorithm starts with $\Delta = 0$. Steps 4-11 are intended for the case in which the rules R1 or R2 are applicable. This is the case in which vertex v is non-adjacent to all the vertices in the examined independent set $I \in \tilde{\mathscr{C}}$. In this situation, either the remaining weight $d = w(v) - g - \Delta$ of v is less than or equal to $\hat{l}(I)$, which corresponds to the application of

[4] The vertices of a stable set I are sorted by non-increasing loads $l(I, v)$.

Algorithm 2: COVER ($\tilde{\mathscr{C}} = \{I_1, \ldots, I_k\}$, l, g, v)

Input: A cover $\tilde{\mathscr{C}}$ with loads l, a gap value g and a vertex v not in $\tilde{\mathscr{C}}$.
// all $I = \{v_1, v_2, \ldots, v_m\} \in \tilde{\mathscr{C}}$ are sorted by loads $\big(l(v_1) \geq l(v_2) \geq \cdots \geq l(v_m)\big)$.

Output: A (potential) new cover $\tilde{\mathscr{C}}$ and a (potential) new gap value g.

1 $j \leftarrow 0$
2 **repeat**
3 $token \leftarrow false$
4 $j \leftarrow j + 1$
5 **if** *R1(v, I_j) is applicable* **then**
6 $I_j \leftarrow I_j \cup \{v\}$ // $I_j = \{v_1, \ldots, v, \ldots, v_m\}$
7 $l(I_j, v) \leftarrow w(v)$
8 $token \leftarrow true$
9 **else if** *R2(v, I_j) is applicable* **then**
10 **if** $w(v) - \hat{l}(I_j) \leq g$ **then**
11 $g \leftarrow g - (w(v) - \hat{l}(I_j))$
12 $l(I_j, v) \leftarrow \hat{l}(I_j)$
13 $I_j \leftarrow I_j \cup \{v\}$ // $I_j = \{v, v_1, \ldots, v_m\}$
14 $token \leftarrow true$
15 **end**
16 **else if** *R3(v, I_j) is applicable and token = false* **then**
 // $I_j = \{v_1, \ldots, v_{l-1}, v_l, v_{l+1}, \ldots, v_m\}$
17 $v_l \leftarrow$ first vertex in I_j such that $v_l \in N(v)$
18 $\alpha \leftarrow \hat{l}(I_j)$, $\beta \leftarrow \alpha - l(I_j, v_l)$
19 **if** $w(v) - \beta \leq g$ **then**
20 $g \leftarrow g - \max\{w(v) - \beta, 0\}$
21 $l(I_j, v_k) \leftarrow l(I_j, v_l), 1 \leq k < l$
22 $\tilde{I}_j \leftarrow \{v_1, v_2, \ldots, v_{l-1}\} \cup \{v\}$
23 $l(\tilde{I}_j, v_k) \leftarrow l(I_j, v_k) - l(I_j, v_l), 1 \leq k < l$
24 $l(\tilde{I}_j, v) \leftarrow \min\{\alpha - l(I_j, v_l), w(v)\}$ // $\hat{l}(\tilde{I}_j) = \beta$
25 $\tilde{\mathscr{C}} \leftarrow \tilde{\mathscr{C}} \cup \tilde{I}_j$
26 $token \leftarrow true$
27 **end**
28 **end**
29 **until** $token = true$ *or* $j = k$
30 **return** $\tilde{\mathscr{C}}$, g

R1, or $d > \hat{l}(I)$, which corresponds to the application of R2. In both cases, I can cover at most $\hat{l}(I)$. On the other hand, Steps 12-19 consider the application of R3. In this situation, the maximum weight that can be covered by I is $\hat{l}(I) - l(I, v_l)$, where, as in COVER, v_l denotes a vertex in I with maximum load belonging to the neighbor set of v.

Algorithm 3: `MCOVER` $(\tilde{\mathscr{C}} = \{I_1, \ldots, I_k\},\ l,\ g,\ v)$

Input: A cover $\tilde{\mathscr{C}}$ with loads l, a gap value g and a vertex v not in $\tilde{\mathscr{C}}$.
`// all` $I = \{v_1, v_2, \ldots, v_m\} \in \tilde{\mathscr{C}}$ `are sorted by loads` $\big(l(v_1) \geq l(v_2) \geq \cdots \geq l(v_m)\big)$.

Output: A subset of indepedendent sets $\tilde{\mathscr{I}}$ that cover $w(v)$. A (potential) new gap value g. TRUE is returned if the algorithm is successful; FALSE otherwise.

$\Delta \leftarrow 0$, $j \leftarrow 0$, $token \leftarrow false$
repeat
  $j \leftarrow j + 1$
  if $\bar{N}(v) \supseteq I_j$ **then**
    $\Delta \leftarrow \Delta + \hat{l}(I_j)$
    $\tilde{\mathscr{I}} \leftarrow \tilde{\mathscr{I}} \cup I_j$
    if $\Delta + g \geq w(v)$ **then**
      $token \leftarrow true$
      $g \leftarrow g - \max\{\Delta - w(v), 0\}$
    end
  else
    $v_l \leftarrow$ first vertex in I_j such that $v_l \in N(v)$
    $\Delta \leftarrow \Delta + (\hat{l}(I_j) - l(I_j, v_l))$
    $\tilde{\mathscr{I}} \leftarrow \tilde{\mathscr{I}} \cup I_j$
    if $\Delta + g \geq w(v)$ **then**
      $token \leftarrow true$
      $g \leftarrow g - \max\{\Delta - w(v), 0\}$
    end
  end
until $token = true$ *or* $j = k$
if *token* **then**
  return $\tilde{\mathscr{I}}$, g, *TRUE*
else return $\emptyset$, g, *FALSE*

2.4.3. Determining the branching set

By making use of the procedures `COLOR`, `COVER` and `MCOVER`, we further define the procedure `GenBranchSet`, which determines a (small) Branching Set B for a given subproblem graph $\hat{G}$, such that $\omega(\hat{G}, w)$ is less than or equal to $lb - w(\hat{C})$.

Procedure `GenBranchSet` is outlined in Algorithm 4. It starts by calling the procedure `COLOR` to compute a partial coloring $\tilde{\mathscr{P}}$ of G such that $UB(\tilde{\mathscr{P}}, \hat{l}, P) \leq lb - w(\hat{C})$. As described in previous sections, this determines an initial Branching Set B composed by the vertices in $\hat{V}$, which remain uncolored. After that, `GenBranchSet` calls `COVER` and `MCOVER` in succession for every remaining vertex in B. Each time any one of these two functions is succesful, the examined vertex from set B is transferred to the Pruned set P and a new vertex in B is selected for examination.

The vertices are always selected from the set B in reverse order with respect to the way they are computed by `COLOR`, and they are treated according to non-decreasing load. This is

because vertices with a lower weight have a higher probability of being covered by $\tilde{\mathscr{C}}$. At the end of the procedure a (small) set of vertices with (high) weights are left in the Branching Set B for every subproblem.

Algorithm 4: `GenBranchSet` $(\hat{G}, lb, w(\hat{C}))$

Input: A graph $(\hat{G}, w)$. An incumbent solution value lb. A weight $w(\hat{C})$ of a clique $\hat{C}$.
Output: A Branching Set B of vertices such that $\omega(G[V \setminus B], w) \leq lb - w(\hat{C})$.

```
1  (C̃, l, g) ← COLOR(Ĝ, lb, w(Ĉ))
2  B = {b_1, ..., b_|B|} ← V \ C̃        // vertices in B are sorted by non-increasing loads
3  if B = ∅ then
4  |  return B
5  end
6  for b_i := b_|B| downto b_1 do
7  |  (Ĩ, g) ← COVER(C̃, l, g, b_i)
8  end
9  if B = ∅ then
10 |  return B
11 end
12 for b_i := b_|B| downto b_1 do
13 |  (Ĩ, g, success) ← MCOVER(C̃, l, g, b_i)
14 |  C̃ ← C̃ \ Ĩ
15 |  if sucess = true then B ← B \ {b_i}
16 end
17 return B
```

2.5. The outline of the algorithm

The design of our new recursive B&B algorithm `BBMCW` for the MWCP is outlined in Algorithm 5 and is inspired by recent state-of-the-art MCP solvers, such as [25, 15]. The first call to `BBMCW` requires some pre-processing, as follows. It is well established that pre-processing is critical for efficiency in many B&B frameworks, see [30] for an in-depth discussion on pre-processing concerning the MCP.

A first consideration is how the vertices are sorted initially. `BBMCW` employs a new adaptive initial vertex ordering procedure inspired by the work of [14] and [30]. The procedure always computes two orderings for the vertices in the preprocessing phase: one based on vertex degree and the other on an independent set partition of the vertex set V.

First Order. The degree-based ordering is described as follows. Let $V = \{v_1 \prec v_2 \prec \cdots \prec v_n\}$ be the original ordering of the vertices of the graph G. The procedure `BBMCW` sorts the vertices of G as follows. First, a vertex v_j, $1 \leq j \leq n$ with minimum degree in V becomes the last vertex v_n in the new ordering. Next, another vertex with minimum degree in $V \setminus \{v_j\}$ is placed at v_{n-1} and so on, until all vertices are sorted. This is a typical degenerate ordering

(so called because the sorting criterion is applied dynamically to the vertices that remain to be sorted, and thus "degenerates" as the ordering proceeds) used by state-of-the art solvers for the MCP , see [30] for a recent in-depth study of different vertex orderings for the MCP. In addition, if vertices have the same degree we select the one with smallest weight.

Second Order. The overarching idea of the independent set-based ordering is to compute an independent set partition of V with a *small* number of independent sets (colors), and sort the vertices according to this partition, i.e., by non-decreasing color number. An independent set ordering for the MCP was originally described in [14]. In this work we use the `COLOUR_SORT` procedure described in [30].

Once the two vertex orderings have been computed, our initial sorting procedure selects the one which produces the smallest branching set at the root node. Finally, we compute a graph G_0 isomorphic to G with such ordering, and use a bitstring representation of G_0 in memory. The advantages of both orderings, as well as the bitstring encoding, are well established in literature for the MCP, see, e.g., [27, 28, 30].

We obtain a "good" feasible solution C_0 using the heuristic AMTS (acronym for Adaptive Multi-start Tabu Search), see [35]. This solution is the initial incumbent solution C_{max} used by the algorithm. It is worth noticing that AMTS is used only by `BBMCW`; the other exact algorithms tested in our computational results have their own heuristic initialization phase. A first root Branching Set B_0 is computed by the call `GetBranchSet`(G_0, $|C_0|$, $\emptyset$). Finally, vertices in B_0 are sorted by non-increasing weights so that the more promising vertices with high weights are examined first at the root node (see Step 1). At the end of the pre-processing phase, the first call to the algorithm is `BBMCW`((G_0, w), V, $\emptyset$, C_0, B_0).

2.6. Further implementation details

We highlight that `BBMCW` uses the sequence determined by the initial ordering of vertices to compute the greedy sequential coloring heuristic in procedure `COLOR`. This favours the production of large partial colorings, as described in [27, 28], specifically in the shallower levels of the branching tree.

In addition, `BBMCW` employs the following optimization related to the size of the independent sets of the cover $\tilde{\mathscr{C}}$ of each subproblem. We recall that, in `COVER`, the vertex that triggers rule R3 for a pair ($v \in B$, $I \in \tilde{\mathscr{C}}$) is the first vertex adjacent to v in I (we recall that vertices in I are ordered by non-increasing loads, see Section 2.3). It is clear that the probability that m vertices in I precede this first neighbor of v decreases with m. Thus, we conceive a modified `COLOR` procedure that only stores a subset of the vertices for each independent set $I \in \tilde{\mathscr{C}}$, precisely those with the highest weights and, therefore, more pruning potential. In our experiments, the best compromise between efficiency and pruning capability was obtained when the size of the color sets was restricted to a maximum of 3 vertices. It is easy to tailor the design of the procedures `COVER` and `MCOVER` for this optimization.

We end this section with a consideration on the way the vertices in the subproblems analyzed are sorted during tree traversal. `BBMCW` uses a bitstring representation of the input graph and is able to efficiently preserve the relative order of vertices determined at root in all subsequent

Algorithm 5: BBMCW$((G, w), \hat{V}, \hat{C}, C_{\max}, B)$

Input: A graph $(G = (V, E), w)$. A subproblem $\hat{V} \subseteq V$. A (feasible solution) clique $\hat{C}$. An incumbent solution $C_{\max}$. An ordered Branching Set $B = \{b_1, b_2, \ldots, b_{|B|}\} \subseteq \hat{V}$ sorted by non-increasing loads.

Output: A maximum weighted clique of G in $C_{\max}$.

```
1  for b := b_1 to b_|B| do
2      V̂_b ← V̂ ∩ N(b)
3      if V̂_b ≠ ∅ then
4          B_b ← GenBranchSet(G[V̂_b], w(C_max), w(Ĉ))
5          if B_b ≠ ∅ then
6              Ĉ ← Ĉ ∪ {b}
7              C_1 ← BBMCW((G, w), V̂_b, Ĉ, C_max, B_b)
8              if w(C_1) > w(C_max) then C_max ← C_1
9              Ĉ ← Ĉ \ {b}
10         end
11     end
12     V̂ ← V̂ \ {b}
13 end
14 return C_max
```

subproblems. Specifically, the vertices of the (bit encoded) vertex set $\hat{V}_b$ in step 2 of algorithm 5 are stored always according to the initial order. During pre-processing, BBMCW additionally computes a mapping between the initial order and the weight-based order of vertices that is required by the Branching set in Algorithm 4. With this mapping, it is always possible to sort any (bit encoded) set of vertices according to vertex weight in linear time on the size of the set. This optimization detail is used in step 2 of Algorithm 4 to efficiently determine the order of vertices.

3. Algorithm demonstration

We present in this section a demonstration graph to illustrate how the proposed bound is used by the algorithm BBMCW. We consider the graph depicted in Figure 2, part (a), with $n = 7$ and $m = 14$, together with weights: $w(v_0) = 1, w(v_1) = 2, \ldots, w(v_6) = 7$. The algorithm, during pre-processing, finds an initial clique v_0, v_1, v_3, v_6 of size $w(\{v_0, v_1, v_3, v_6\}) = 14$ and branches on vertex v_3 (thus $\hat{C} = \{v_3\}$). The child subproblem, determined by the neighborhood $\{v_0, v_1, v_2, v_4, v_5, v_6\}$ of v_3, is shown in Figure 2, part (b), and the (initial) gap value g is defined as $g = lb - w(\hat{C}) = 14 - w(v_3) = 10$.

We now trace the work of the procedure GenBranchSet, described in Algorithm 4, for the child subproblem $\hat{V} = \{v_0, v_1, v_2, v_4, v_5, v_6\}$. The call to COLOR made by GenBranchSet computes the independent set partition $\tilde{\mathscr{P}} = \{I_1, I_2\}$ shown in Table 1. As can be seen, the bound (12) for $\tilde{\mathscr{P}}$ is $\hat{l}(I_1) + \hat{l}(I_2) = 6 + 3 = 9$, and no other vertex $v \in \hat{V}$ may be added to $\tilde{\mathscr{P}}$ such that the bound (12) for the enlarged partition is less or equal to the initial gap value 10. Consequently,

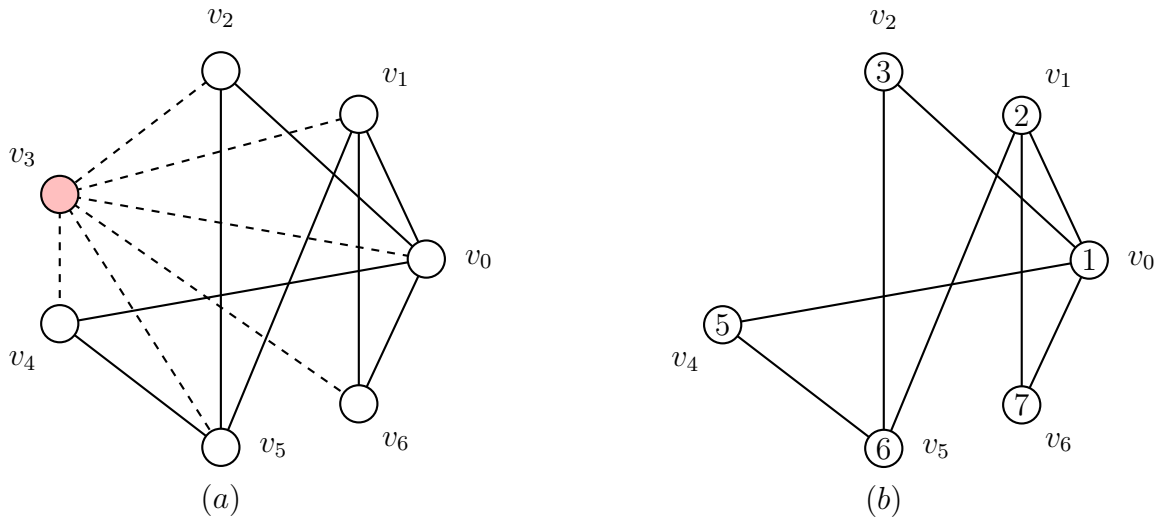

Figure 2: In part (a) of the Figure, we depict a demonstration graph $\hat{G}$ where we show in red the vertex v_3 that is examined by the branching scheme. In part (b), we show the corresponding child subproblem graph $\hat{G}[N(v_3)]$, together with the vertex weights.

the new gap value is $g = 10 - 9 = 1$. The initial Pruned Set P for the subproblem contains the vertices of $\tilde{\mathscr{P}}$ where $P = \{v_0, v_1, v_2, v_5\}$, and the Branching Set is $B = \hat{V} \setminus P = \{v_4, v_6\}$.

$I_1 = \{v_5, v_0\}$	$l(I_1, v_5) = 6,\ l(I_1, v_0) = 1$
$I_2 = \{v_2, v_1\}$	$l(I_2, v_2) = 3,\ l(I_2, v_1) = 2$

Table 1: A partial coloring of the graph in Figure 2, part (b).

At this point, the call to `COVER` in `GenBranchSet` fails to transfer any vertex from B to P, because it is not possible to cover the weight of either vertex v_4 or v_6 in B using the single independent set cover of I_1 or I_2. Consider, for example, vertex v_4. Applying rule R2 to the pair (v_4, I_2) increments the bound by $w(v_4) - w(v_2) = 5 - 3 = 2$, thus exceeding the unit gap. Similar reasonings can be made for the other rules and pairs.

Having failed `COVER`, the procedure `GenBranchSet` calls `MCOVER`, which succeeds in transferring vertex v_6 from set B to set P as follows:

- *By applying R3(v_6, I_1)*: The first adjacent vertex in I_1 to vertex v_6 is v_5, so R3 is triggered. As a result, $l(I_1, v_5) = l(I_1, v_1) = 1$ and the new independent set split according to rule R3 is $\tilde{I}_1 = \{v_5, v_6\}$, where $l(I_1, v_5) = l(I_1, v_6) = 5$. `MCOVER` does not compute the split, but instead updates Δ with the maximun weight units that I_1 can cover, which is 5. At this point, this is not enough to cover $w(v_6) = 7$.
- *By applying R3(v_6, I_2)*: The first adjacent vertex in I_2 to vertex v_6 is v_2, so again R3 is triggered. As a result, $l(I_2, v_2) = l(I_1, v_1) = 2$ and the new independent set split according to rule R3 is $\tilde{I}_2 = \{v_2, v_6\}$, where $l(\tilde{I}_2, v_2) = l(\tilde{I}_2, v_6) = 1$. As noted in the previous case, `MCOVER` does not compute the split but instead increments Δ by the

maximum weight that I_2 can cover which is 1, so $\Delta = 5 + 1 = 6$. This value of Δ is still not enough to cover $w(v_6) = 7$ by itself.

- The gap value g: The remaining unit weight $w(v_6) - \Delta$ can be accounted by the gap value, which is 1. The gap value is reduced by this one unit to $g = g - 1 = 0$.

The independent set cover $\tilde{\mathscr{C}} = \{I_1, \tilde{I}_1, I_2, \tilde{I}_2\}$, which corresponds to the operations done by MCOVER to determine the transfer of vertex $v_6 \in B$ to the Pruned Set P, is shown in Table 2.

$I_1 = \{v_5, v_0\}$	$l(I_1, v_5) = 6,\ l(I_1, v_0) = 1$
$\tilde{I}_1 = \{v_5, v_6\}$	$l(\tilde{I}_1, v_5) = 5,\ l(\tilde{I}_1, v_6) = 5$
$I_2 = \{v_2, v_1\}$	$l(I_2, v_2) = 2,\ l(I_1, v_1) = 2$
$\tilde{I}_2 = \{v_2, v_6\}$	$l(\tilde{I}_2, v_2) = 1,\ l(\tilde{I}_2, v_6) = 1$

Table 2: A partial independent set cover of the graph in Figure 2, part (b).

We conclude this section by comparing our bounding technique with the principal bounding technique proposed in [10]. This bounding technique is the greedy algorithm proposed to compute a *weighted clique cover*. We denote the algorithm WCC for brevity; moreover, this procedure is the core of the branch-and-bound algorithm called MWSS and described in detail in [10]. In our case, we are interested in a *weighted independent set cover* and we will describe WCC adapted to the latter case. In addition, we will use WCC, as in [10], to determine the branching set B (and, accordingly, the pruned set $P = \hat{V} \setminus B$). More precisely, given a graph $\hat{G} = (\hat{V}, \hat{E})$ and a set of weights on the vertices $w(v)$ ($v \in \hat{V}$), WCC incrementally covers the vertex weights using maximal cliques (independent sets in our case).

Before the execution of the algorithm, an initialization step sets the residual weight $\tilde{w}(v)$ of each vertex $v \in \hat{V}$ to its original weight $w(v)$. WCC then iteratively determines the vertex $\bar{v}$ with the smallest residual weight $\tilde{w}(\bar{v})$ and greedily builds a maximal independent set $\tilde{I}$ containing it. Once the set $\tilde{I}$ has been constructed, $\pi_{\tilde{I}}$ (see Formulation (2)-(4)) is set to the (current) residual weight of the vertex $\bar{v}$ ($\pi_{\tilde{I}} = \tilde{w}(\bar{v})$) and all the residual weights of the vertices in $\tilde{I}$ are decreased by $\pi_{\tilde{I}}$. We note that, by construction, $\tilde{w}(\bar{v}) = 0$. Moreover, a vertex $\bar{v}$ can only be used as seed for a new independent set if and only if its residual weight plus the sum of the π variable (as established by the previous independent sets) does not exceed the initial gap value $lb - w(\hat{C})$. If no vertex with positive residual weight meets this condition, the algorithm stops and the pruned set P is determined by the union of vertices that have residual weights equal to 0 ($B = \hat{V} \setminus P$). Alternatively, if there are no vertices with positive residual weights that can be the seed of a new independent set, the algorithm stops and outputs an empty branching set B.

We show now how WCC works on the graph in Figure 2, (part (b)), under the same assumptions as for BBMCW (the initial gap g is set to 10) . To the best of our knowledge, no additional detail on how to build the maximal cliques (independent sets in our case) is given in [10]. For this reason, we make the assumption that the vertices are considered in order of non-increasing residual weight (in a sequential greedy fashion). We recall that the initial residual weights of the vertices in the graph are:

$$\tilde{w}(v_0) = 1,\ \tilde{w}(v_1) = 2,\ \tilde{w}(v_2) = 3,\ \tilde{w}(v_4) = 5,\ \tilde{w}(v_5) = 6,\ \tilde{w}(v_6) = 7.$$

WCC determines a first independent set $I_1 = \{v_0, v_5\}$ (the seed $\bar{v} = v_0$) and sets π_{I_1} to $\tilde{w}(\bar{v}) = 1$. After updating, the residual weights of the vertices in I_1 become $\tilde{w}(v_0) = 0$ and $\tilde{w}(v_5) = 5$. A second independent set $I_2 = \{v_1, v_2, v_4\}$ is then computed with $\pi_{I_2} = 2$. The residual weights of the vertices in I_2 become $\tilde{w}(v_1) = 0, \tilde{w}(v_2) = 1, \tilde{w}(v_4) = 3$. A third independent set $I_3 = \{v_2, v_4, v_6\}$ is computed and $\pi_{I_3} = \tilde{w}(v_2) = 1$. The residual weights of the vertices in I_3 become $\tilde{w}(v_2) = 0, \tilde{w}(v_4) = 2, \tilde{w}(v_6) = 6$. A fourth independent set $I_4 = \{v_4, v_6\}$ is computed and $\pi_{I_4} = 2$. The residual weights of the vertices in I_4 become $\tilde{w}(v_4) = 0, \tilde{w}(v_6) = 4$. Finally, a fifth independent set $I_5 = \{v_5, v_6\}$ is computed and $\pi_{I_6} = 4$. The residual weights of the vertices in I_5 become $\tilde{w}(v_6) = 0$, $\tilde{w}(v_5) = 1$. WCC then stops since the residual weight of v_5 is equal to 1 and $\sum_{i=1}^{5} \pi_{I_i} = 10$. The branching set B computed by WCC is $B = \{v_5\}$, and the pruned set $P = \{v_0, v_1, v_2, v_4, v_6\}$. In the example, the algorithm BBMCW obtains a different branching set $B = \{v_4\}$. In this case, the size of the two branching sets coincides, but, as shown in the next section, our new algorithm computationally outperforms MWSS.

4. Computational results

We conducted extensive tests to computationally evaluate the ideas presented in this paper and the new combinatorial branch-and-bound algorithm for the MWCP. The hardware used in the experiments was a 20-core Intel(R) Xeon(R) CPU E5-2690 v2@3.00GHz, with 128GB of main memory, and running a 64-bit Linux OS. All algorithms presented in the previous sections are implemented in C++, compiled using gcc 4.8.4 (with –o3 optimizations) and run on a single core of the machine. The performance of the algorithm dfmax, commonly used for calibration between different machines, is 0.189, 1.155 and 4.369 seconds for the benchmark graphs r300.5, r400.5 and r500.5, respectively. In each experiment, the CPU time is reported in seconds.

The goal of this computational section is to compare the performance of the new exact MWCP algorithm, called BBMCW, with state-of-the-art algorithms from the literature. We consider the following four algorithms (where either the source code or the binaries were available):

- MWSS[5]: the exact algorithm for the Maximum Weighted Stable Set Problem presented in [10] (tested as a MWCP algorithm on the complemented graph). Stable set and Independent set are used as synonyms.
- MWCLQ[6]: the exact algorithm for the MWCP presented in [6].
- WLMC: the exact algorithm for the MWCP described in [11].
- TSM-MWC: the very recent exact algorithm for the MWCP described in [12]

The choice of algorithms requires some clarification. We note that the algorithm WLMC, see [11], uses MaxSAT reasoning as the the algorithm MWCLQ, but its overarching bounding scheme is directed at reducing a Branching set, in a similar fashion as BBMCW. In [11], WLMC is described as an algorithm for large and massive vertex-weighted graphs but, as can be seen in the

[5] https://github.com/heldstephan/exactcolors
[6] A Linux release was provided by the one of the developers

reported results, it also performs well on graphs of small and medium size. The TSM-MWC algorithm presented in [12] is also based on MaxSAT reasoning but uses a more sophisticated branching strategy than WLMC. The source codes of both WLMC and TSM-MWC are publicly available[7].

We divide this computational Section into five subsections, in which we test the performance of BBMCW in different categories of instances and analyse its bounding function. In Section 4.1 we present the results on "hard" DIMACS and BHOSHLIB instances. In Section 4.2 we present the results on random-graph instances, with up to 15,000 vertices and different edge densities. Section 4.3 reports tests carried out to compute the Fractional Coloring Number of a graph, using the classical instances typically used for the *Vertex Coloring Problem*. In Section 4.4 we analyse the behaviour of the algorithm over four real-world datasets. Finally, Section 4.5 reports the impact of the different individual bounding schemes that are combined in BBMCW.

4.1. DIMACS and BHOSHLIB instances

The majority of the structured graphs tested are taken from the DIMACS 2 benchmark set of instances, see [1]; in addition we consider also several frb graphs from the BHOSHLIB[8] benchmark set. These two datasets, DIMACS 2 and BHOSHLIB, are commonly used for evaluating MCP algorithms; see e.g., [30, 29, 15, 6] , and comprise non-weighted graphs. We generate the vertex weights according to the function $w(v_i) = (i \mod 200) + 1$, a method originally proposed in [23], which has been employed in several recent works for the MWCP; see e.g., [17, 10, 6, 11]. From the two benchmarks, we reported those instances which were non-trivial but "tractable". More precisely, we selected those graphs that are solved to proven optimality in more than 0.1s and less than 5h by at least one of the algorithms considered in this work. We thus identify 45 graphs for this set of tests.

Table 3 reports the names of the selected instances, together with their corresponding size ($|V|$), number of edges ($|E|$), edge density ($\mu(G)$) and clique number ($\omega(G)$). The latter is obviously an upper bound on the size (maximum number of vertices) of any clique solution to the MWCP. The table also reports the optimal MWCP solution value ($\omega(G, w)$), as well as the computing times spent by the five algorithms we considered: BBMCW, MWSS, MWCLQ, WLMC and TSM-MCW. In the case of BBMCW, the table further provides the initial incumbent solution value (lb_0) determined during preprocessing. As explained in Section 2.5, this solution was obtained using the AMTS heuristic. The time spent by AMTS to compute lb_0 for each graph is reported in the column with the same name. In addition, the best computing time of the five algorithms is highlighted in bold for each instance. With respect to actual setup of the runs, the time limit was set to 5 hours for all the algorithms.

According to Table 3, the new algorithm BBMCW is the fastest algorithm for 21 out of the 45 reported graphs, while second best is the algorithm TSM-MWC, which is the fastest for 19 of the graphs. The other algorithms are far behind, the third being WLMC, which is the fastest in two

[7] https://home.mis.u-picardie.fr/~cli/EnglishPage.html
[8] http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm

cases. On the other hand, TSM-MWC is able to solve the full set to proven optimality, while the new algorithm BBMCW cannot solve within the 5h time limit the graphs *gen400_p0.9_55*, *gen400_p0.9_75* and *hamming10-2*. With respect to individual families, BBMCW performs well in the *brock*, *dsjc* and *frb* datasets, while it is outperformed by TSM-MWC and WLMC in a significative number of instances from the *p_hat*, *san* and *gen* families. It is worth noting that the algorithm MWSS is several orders of magnitude faster than all the other algorithms over the latter instance.

As far as the incumbent solution value lb_0 is concerned, the table shows that the AMTS heuristic is able to provide very good initial feasible solutions. Specifically, AMTS is able to compute the optimal solution value in 34 out of the 45 instances, and the difference with the optimal solution value is larger than 100 units in just two graphs.

In order to give a graphical representation of the relative performance of the different exact algorithms considered in this section, we report the performance profile of Figure 3. For each instance, we compute a normalized time τ as the ratio of the computing time of the considered algorithm over the minimum computing time for solving the instance to optimality. For each value of τ in the horizontal axis, the vertical axis reports the percentage of the instances for which the corresponding algorithm spent at most τ times the computing time of the fastest algorithm. The curves start from the percentage of instances in which the corresponding algorithm is the fastest and, at the right end of the chart, we can read the percentage of instances solved by a specific algorithm. The best performances are graphically represented by the curves in the upper part of Figure 3.

The performance profile visually confirms that BBMCW and TSM-MWC are the two best algorithms for the set of instances reported. The remaining three algorithms are clearly outperformed. To note, BBMCW is the fastest in 46% of the instances while TSM-MWC is the fastest in 42% of the graphs. On the other hand, BBMCW is able to solve to optimality 94% of the graphs, while TSM-MWC is able to solve to optimality all of the graphs. It is worth noting that TSM-MWC requires time ratios of $\tau = 300$ and $\tau = 900$ to complete this task and there are instances where it is more than two orders of magnitude slower than the fastest algorithm.

4.2. Random instances

We tested a set of 270 Erdős-Rényi random $G(n, p)$ graphs of different sizes ($n = |V| \in \{150, 200, 300, 500, 1000, 3000, 5000, 10000, 15000\}$) and different edge densities (see Table 4 for the specific density values tested). These 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 MCP and MWCP algorithms. Specifically, the concrete test bed is the same as the one used in [32]. For each class of random graphs, we created 10 instances with similar features. The vertex weights were generated using the same distribution used for the DIMACS and BHOSHLIB graphs (see 4.1).

For this set of experiments we also test the performance of one state-of-the art commercial Integer Linear Programming (ILP) solver. Specifically, we test CPLEX 12.7.0 in single-thread mode (called just CPLEX in what follows) to solve the standard *edge formulation* of the MWCP

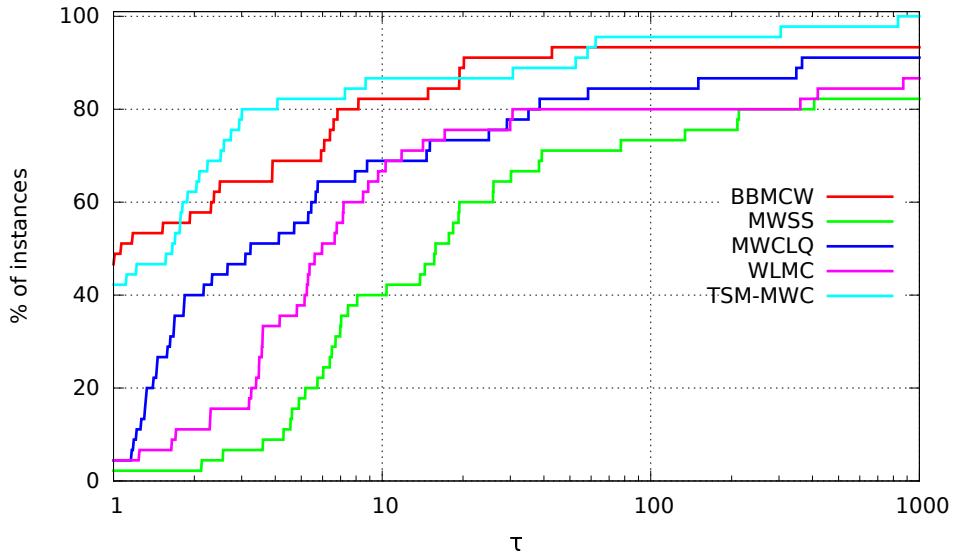

Figure 3: Performance profile on selected "hard" DIMACS and BHOSHLIB instances.

(all `CPLEX` parameters were set to their default values). The edge formulation is an ILP where $\omega(G, w) = \max \sum_{v \in V} w(v) \cdot x_v$ over x in $CLIQ(G)$, that is, the set of vectors of $\mathbb{R}^V$ satisfying:

$$CLIQ(G) = \left\{ x_u + x_v \leq 1, uv \in \overline{E}, \;\; x_v \in \{0,1\}, v \in V \right\}. \tag{20}$$

Table 4 reports, for each subset of 10 instances with similar features, their size ($|V|$) and their edge density ($\mu(G)$). It then reports the maximum and minimum clique number ($\omega(G)$) of the instances. The last four columns provide the average CPU time required by the algorithms `BBMCW`, `MWSS`, `MWCLQ`, `CPLEX`, `WLMC` and `TSM-MWC` respectively. For each row of the table we report in bold text the average CPU time required by the fastest algorithm.

The results presented in Table 4 demonstrate that `BBMCW` is characterized by the best overall computational performance for this test bed of random instances. In particular, of the 27 families of uniform random graphs reported, `BBMCW` is the fastest algorithm for 19 cases, whereas the second best algorithm, `TSM-MWC`, is the fastest for 6 of them. Notably, `TSM-MWC` performs best in the families of small dense graphs with 150 and 200 vertices and edge density values ($\mu(G)$) ranging from 0.8 up to 0.95. The remaining algorithms are the fastest for, at most, a single family. For the large instances that have 500 or more vertices, `BBMCW` always performs best and achieves speedups averaging $\approx 3\times$ with respect to the second best algorithm `TSM-MWC`.

4.3. Computing the fractional chromatic number of a graph using `BBMCW`

The *chromatic number* of a graph G, denoted by $\chi(G)$, is the minimum number of independent sets (or equivalently colors) in any legal coloring of G. The *Vertex Coloring Problem* (VCP) calls for determining the chromatic number of the graph. The *fractional chromatic number*

$\chi_f(G)$ corresponds to the optimal solution value of the Linear Programming (LP) relaxation of the VCP formulation proposed in [21] and it is $\mathcal{NP}$-hard to compute (see [7]).

Let $A \in \{0,1\}^{|V|\cdot|\mathscr{I}|}$ be a binary matrix with $a_{vI} = 1$ if and only if the vertex $v \in V$ belongs to independent set $I \in \mathscr{I}$, and let e be the all-one vector of $|V|$ components. Upon introducing a binary variable ξ_I for each independent set $I \in \mathscr{I}$, equal to 1 if and only if the independent set I is chosen, the LP relaxation of the VCP formulation proposed in [21], featuring exponentially many variables (or columns), reads as follows:

$$\chi_f(G) = \min_{\xi \geq 0} \left\{ \sum_{I \in \mathscr{I}} \xi_I : A\xi \geq e \right\}. \tag{21}$$

Formulation (21) is typically solved by *Column Generation* (CG) (we refer the interested reader to [4] for further details on CG). Starting from Formulation (21) initialized with a subset of columns $\tilde{\mathscr{I}}$ admitting a feasible solution, known as *Restricted Master Problem* (RMP), CG iteratively solves a *Pricing Problem* (PP) to generate columns with a strictly negative reduced cost to be added to $\tilde{\mathscr{I}}$. RMP is then reoptimized, and the procedure is iterated until no more columns with a negative reduced cost exist. When the algorithm terminates, the fractional chromatic number $\chi_f(G)$ is determined.

To compute the strong VCP lower bound $\chi_f(G)$, one of the state-of-the-art algorithms is presented in [10][9]. The computational results of [10] show that this implementation is among the fastest in the literature. In particular, this algorithm relies on the exact solution of a series of MWSSPs using `MWSS`. The PP corresponds then to a MWSSP in which the objective function weights are the dual variable values of the RMP constraints.

To compare the performance of `BBMCW` to `MWSS` as PP algorithms, we consider the standard set of instances [10] typically used for testing algorithms for the VCP (see e.g., [19]). We discard each of the instances for which $\chi_f(G)$ can be computed in less than 1 sec or cannot be computed within a time limit of 1 hour. We determine in this way a set of 49 instances.

In Table 5, we report the results of this set of tests. The table reports the name of each selected instance, with the corresponding number of vertices ($|V|$), number of edges ($|E|$) and edge density ($\mu(G)$). The table then shows the rounded-up value of the fractional chromatic number of the graph ($\lceil \chi_f(G) \rceil$). As previously mentioned, this value provides a strong lower bound for the *chromatic number* $\chi(G)$. Finally, the table shows the total computing time and the time spent to solve the PPs, for `BBMCW` and `MWSS`, respectively. The difference between the total and the pricing time corresponds to the time necessary to solve the RMP, using the Linear Programming solver of `CPLEX`.

Table 5 clearly demonstrates that `BBMCW` compares favourably with `MWSS` for this set of instances. For each instance, we report in bold text the time required by the fastest algorithm. Using `BBMCW`, the bound can be computed for 47 out of 49 instances, and, using `MWSS`, for

[9] code available at `https://github.com/heldstephan/exactcolors`

[10] `http://mat.gsia.cmu.edu/COLOR/instances.html`

46 instances. In particular, using MWSS the fractional chromatic number can be computed faster in 12 instances; for the remaining 37, BBMCW is faster. It is worth mentioning that a large portion of the computational time is taken by PP compared to the time taken by the solution of RMP. These tests demonstrate that very good computational performance can be achieved by BBMCW when the vertex weights correspond to the dual variable values of the RMP constraints. These weights are typically not correlated with any particular structural property of the graph and they evolve during the CG algorithm. As far as the impact of edge density on performance is concerned, the table shows that BBMCW consistently outperforms MWSS when the density is high, contrary to what is experienced for low density values ($\approx \mu(G) \leq 0.1$).

4.4. Real-world instances

We also experiment with a number of real-world instances derived from practical applications and mentioned in [20] as being potentially interesting to test MWCP algorithms. Specifically, we consider the following four datasets:

- *Kidney-Exchange schemes* (KES) [20]: Kidney-exchange schemes exist in several countries to determine a way of pairing donors to patients with end-stage renal disease. Typically, a patient enters the scheme along with a friend or family member who is willing to donate to that patient but is unable to due to some medical incompatibility. These two participants form a *donor-patient* pair. An *exchange* is established according to medical constraints and involves two or more donor-patient pairs. In the 100 graphs of this dataset, vertices are exchanges, the weights represent medical scores and there is an edge between two exchanges if they have no participants in common.

- Error-correcting codes (ECC): The graphs of this set derive from coding theory, specifically from a problem proposed by Östergåard in [22]. Given a length n, a distance d, a weight w, and a permutation group Π, the problem calls for determining a maximum cardinality set C of binary vectors (codes) of length n and Hamming weight w, such that each pair of codes in C is separated at least by a Hamming distance d. Furthermore, for every permutation $\sigma \in \Pi$ and for every code $c \in C$, it happens that $\sigma(c) \in C$. This is due to the fact that a permutation of a code is another code also in C. This problem can be reduced to solving a MWCP on a graph such that the vertices represent each orbit of the permutation group, the weight of each vertex is the size of the corresponding orbit and there is an edge between two vertices if and only if all pairs of members of the two orbits are separated by at least a Hamming distance d. The dataset contains 15 graphs and was originally proposed in [22].

- Winner Determination Problem (WDP): In auctions where combined bids on sets of items are allowed, as opposed to single item bids, the same item can be part of two or more different bids. Determining an allocation of items that maximizes the auctioneer's revenue is called the *Winner Determination Problem*, see [33]. An instance of the WDP can be modelled as a MWCP where each vertex of the graph represents a (combined) bid, a vertex weight is the value of the corresponding bid, and there is an edge between two vertices if the bids do not share a common item. In the tests, we consider the set

of 499 problem instances originally generated by [13]. The set is publicly available via cspLib [11].

- Research Excellence Framework (REF): The Research Excellence Framework is an impact evaluation which analyzes the research work of British higher education institutions. It was first used in the year 2014 to evaluate the six year period 2008–2013, and the next iteration of REF will be in 2021. As explained in [20], the rules for REF2021 establish that an academic can be evaluated over a maximum of 4 papers in a given period of time (typically 4 years). In the evaluated units from universities, such as schools or departments, each academic submits six authored papers which are then ranked by the management of the unit (with numbers from 1 to 4). This generates $\binom{6}{4}$ possible selections, each one of them with a combined value that ranges from four to 16. In the final submission for REF2021, each unit accepts only one selection for each member of the staff, and selections that contain co-authored papers are not allowed. Allocating the (4-paper) sets that maximizes the unit's combined ranking can be reduced to a MWCP that resembles the Winner Determination Problem. In this case, each vertex is a four-paper combination submitted by a member of the staff (bidder); its weight is the combined ranking of the proposal (the value the bid), and there is an edge between two vertices if the two bids have no co-authors in common. The dataset used for the tests contains 129 graphs of this type.

The full datasets KES, ECC and REF, and a subset of WDP, have been brought together by the authors of [20] in DIMACS format [12].

Table 6 reports the results obtained for the proposed new algorithm BBMCW, together with the algorithms TSM-MWC and WLMC, averaged over all the graphs of each of the four datasets considered. For each dataset, the table reports the number of instances (#), together with the computing time in seconds (*t[s]*) and the instances solved to optimality by the three algorithms. In every graph we considered a maximum computing time of 600 seconds, and cells in boldface indicate the best result in the corresponding dataset. According to the table, both BBMCW and TSM-MWC solve to optimality the same number of instances; BBMCW is the fastest in the datasets KES and REF while TSM-MWC is the best in the two remaining datasets, WDP and ECC. The algorithm WLMC performs comparably to TSM-MWC except in the auction dataset, where it is slower and closer to BBMCW.

4.5. Impact of the new bounding procedure

We also study the new bounding procedure in terms of the number of recursive calls to the algorithm and the total computing time on a subset of 10 DIMACS graphs. For the analysis we compared BBMCW with and without individual parts of the bounding function: i) the multiple independent set cover bounding function MCOVER described in Section 2.4.2 and ii) the single independent set cover function COVER described in Section 2.4.1. The specific choice of the graphs spans over 6 different families to be as representative as possible.

[11] http://www.csplib.org/Problems/prob063/
[12] https://doi.org/10.5281/zenodo.816293

Table 7 reports the results of this analysis. The first two columns show the number of vertices $|V|$ and edges $|E|$ of each problem instance, and the remaining columns report the number of recursive calls (*#calls*) and the computing time in seconds (*t[s]*) spent by the corresponding three algorithms. Cells in boldface indicate the best results for each graph.

From the table, it becomes clear that both `COVER` and `MCOVER` have a significant impact on the overall performance of `BBMCW`. A first consideration is that `BBMCW` is always faster, and produces smaller branch-and-bound trees, than the other two variants. When removing `MCOVER`, the algorithm shows an average reduction in speed of more than a 30% with respect to `BBMCW`, reaching up to nearly a 70% for the graph *p_hat500-3*. Moreover, when the bound derived from the independent set partition is considered, the algorithm becomes on average more than three times slower (we note that it is more than 5 times slower in *p_hat500-3*).

Finally, we also measure the impact of the heuristic `AMTS` employed by `BBMCW` to determine an initial solution. The results appear in the last column of Table 7. The column reports the time taken by `BBMCW` when `AMTS` is substituted by the construction of a clique in a greedy sequential way, taking vertices by non-increasing degree (and non-increasing weight in case of a tie). From the table, it can be seen that the improvement in the overall performance of the algorithm `BBMCW` with the heuristic `AMTS` is never more than a 36%.

5. Conclusions and future research

In this manuscript, we have studied the Maximum Weight Clique Problem (MWCP), an important well studied problem in graph theory. For the MWCP, we describe (and implement) a new combinatorial Branch-and-Bound algorithm, which employs a novel bound inspired by a prior independent set cover bound. In addition, the algorithm is designed so that it exploits bitstring representation and color partitioning, following recent literature for MCP. The extensive computational results provided in the manuscript prove that the new B&B algorithm improves on previous state-of-the-art exact approaches reported in the literature.

Concerning future research, it is interesting to consider further enhancements of the proposed algorithm so as to improve its performance on the very dense graphs. Specifically, we are planning to test the local search scheme (*reNumber*) proposed in [34] for the MCP to attempt to improve the initial independent set partition of the bounding procedure. Another interesting attempt would be to include some form of incremental MaxSAT reasoning in the algorithm, as in [6, 12, 17]. Finally, we also consider worth investigating the conflict-directed clause learning (CDCL) reasoning scheme proposed in the work [9], which has been published during the reviewing process of this manuscript. This new SAT-based approach seems very promising for clique solvers.

6. Acknowledgments

This work has been partially funded by the Spanish Ministry of Science, Innovation and Universities through the project COGDRIVE (DPI2017-86915-C3-3-R). The authors would

also like to thank Stefan Held for making the source code for the MWSS algorithm available online, and Chu-Min Li and Hua Jiang for providing us with a Linux release for MWCLQ.

						AMTS		BBMCW	MWSS	MWCLQ	WLMC	TSM-MWC
	$\|V\|$	$\|E\|$	$\mu(G)$	$\omega(G)$	$\omega(G,w)$	lb_0	t. [s]	t. [s]	t. [s]	t. [s]	t. [s]	t. [s]
brock400_1	400	59,723	0.75	27	3,422	3,422	1.51	**84.0**	387.6	102.1	441.0	93.9
brock400_2	400	59,786	0.75	29	3,350	3,350	1.48	99.5	452.2	**99.4**	533.1	120.6
brock400_3	400	59,681	0.75	31	3,471	3,471	0.65	75.9	410.6	83.6	344.3	**71.5**
brock400_4	400	59,765	0.75	33	3,626	3,626	1.45	**40.4**	209.2	64.1	573.8	121.4
brock800_1	800	207,505	0.65	23	3,121	3,121	2.80	**830.4**	5,840.5	1,163.8	7,371.4	1,567.3
brock800_2	800	208,166	0.65	24	3,043	3,043	2.78	**1,260.3**	8,188.5	1,660.9	8,382.5	2,133.6
brock800_3	800	207,333	0.65	25	3,076	3,076	2.98	**990.5**	6,919.8	1,293.6	7,103.2	1,761.7
brock800_4	800	207,643	0.65	26	2,971	2,970	2.80	**1,341.3**	8,565.2	1,694.4	9,092.4	2,214.1
C2000.5	2,000	999,836	0.50	16	2,466	2,466	9.30	**6,735.2**	t.l.	9,813.1	t.l.	14,063.3
C250.9	250	27,984	0.90	44	5,092	5,092	0.74	68.0	251.7	31.8	72.5	**17.5**
dsjc500.5	500	62,624	0.50	13	1,725	1,725	0.03	**0.5**	3.2	0.8	2.8	1.2
dsjc1000.5	1,000	249,826	0.50	15	2,186	2,186	0.13	**41.4**	309.0	69.7	219.3	74.7
gen200_p0.9_44	200	17,910	0.90	44	5,043	5,043	0.01	4.2	20.8	5.5	2.5	**0.7**
gen200_p0.9_55	200	17,910	0.90	55	5,416	5,382	0.00	1.6	12.3	2.2	2.4	**0.7**
gen400_p0.9_55	400	71,820	0.90	55	6,718	6,718	0.97	t.l.	t.l.	6,822.3	t.l.	**5,161.2**
gen400_p0.9_75	400	71,820	0.90	75	8,006	7,980	0.79	14,483.2	t.l.	11,841.2	10,329.1	**337.8**
hamming10-2	1,024	518,656	0.99	512	50,512	50,512	2.28	t.l.	**0.1**	973.0	1,257.8	38.6
keller5	776	225,990	0.75	27	3,317	3,219	2.21	**3,973.4**	17,077.8	t.l.	t.l.	10,868.4
MANN_a27	378	70,551	0.99	126	12,283	12,268	0.02	**0.5**	28,829.2	t.l.	0.9	4.6
MANN_a45	1,035	533,115	1.00	345	34,265	34,174	15.88	t.l.	t.l.	t.l.	**334.4**	1,362.8
p_hat300-3	300	33,390	0.74	36	3,774	3,774	0.01	0.7	6.8	1.9	1.2	**0.4**
p_hat500-2	500	62,946	0.50	36	3,920	3,920	0.03	0.4	8.0	1.8	0.5	**0.3**
p_hat500-3	500	93,800	0.75	50	5,375	5,375	1.21	173.0	2,483.7	682.0	99.1	**11.7**
p_hat700-2	700	121,728	0.50	44	5,290	5,290	0.05	2.3	197.5	36.2	2.2	**0.9**
p_hat700-3	700	183,010	0.75	62	7,565	7,563	1.26	487.9	t.l.	8,751.7	258.6	**25.1**
p_hat1000-1	1,000	122,253	0.24	10	1,514	1,514	0.09	**0.2**	2.6	0.4	0.6	0.5
p_hat1000-2	1,000	244,799	0.49	46	5,777	5,777	2.51	106.6	5,448.6	1,933.9	**5.3**	13.3
p_hat1500-1	1,500	371,746	0.33	12	1,619	1,619	0.21	**1.5**	21.1	3.3	5.3	2.7
p_hat1500-2	1,500	284,923	0.25	65	7,360	7,360	3.67	11,153.1	t.l.	8,429.5	6,807.0	**576.1**
san200_0.9_2	200	17,910	0.90	60	6,082	5,835	0.01	1.2	5.4	1.2	3.6	**0.2**
san200_0.9_3	200	17,910	0.90	44	4,748	4,748	0.01	8.5	38.4	11.8	6.9	**2.2**
san400_0.7_1	400	55,860	0.70	40	3,941	3,641	0.03	9.3	2.9	2.7	2.6	**1.1**
san400_0.7_2	400	55,860	0.70	30	3,110	3,110	0.03	7.5	6.9	3.8	11.6	**3.2**
san400_0.7_3	400	55,860	0.70	22	2,771	2,771	0.02	**1.2**	12.6	5.0	8.7	2.5
san400_0.9_1	400	71,820	0.90	100	9,776	9,776	2.25	449.0	1,268.3	992.2	1,975.4	**65.9**
san1000	1,000	250,500	0.50	15	1,716	1,716	0.11	**1.0**	3.5	145.0	1.2	7.0
sanr200_0.9	200	17,863	0.90	42	5,126	5,126	0.81	10.3	24.6	4.8	5.1	**1.6**
sanr400_0.5	400	39,984	0.50	13	1,835	1,835	0.02	**0.1**	1.1	0.2	0.7	0.3
sanr400_0.7	400	55,869	0.70	21	2,992	2,992	0.02	**13.5**	66.0	19.5	75.6	21.1
monotone_0.7	343	46,305	0.79	19	1,998	1,998	0.02	16.9	66.5	321.1	106.5	**11.0**
frb30-15-1	450	83,198	0.82	30	2,990	2,990	1.44	**122.4**	4,693.4	199.4	t.l.	3,753.1
frb30-15-2	450	83,151	0.82	30	3,006	3,006	1.38	156.8	962.7	**24.5**	10,254.0	1,424.1
frb30-15-3	450	83,216	0.82	30	2,995	2,988	1.30	**22.7**	1,752.3	106.6	8,163.3	1,189.6
frb30-15-4	450	83,194	0.82	30	3,032	3,032	1.34	**16.9**	2,265.4	148.5	14,711.3	1,052.8
frb30-15-5	450	83,231	0.82	30	3,011	3,003	0.02	**1.9**	751.1	46.2	9,802.2	1,541.5

Table 3: Results on "hard" DIMACS and BHOSHLIB instances. Time limit is fixed at 5h.

				BBMCW	MWSS	MWCLQ	CPLEX	WLMC	TSM-MWC
$\|V\|$	$\mu(G)$	$\omega(G)$	#	t. [s]	t. [s]	t. [s]	t. [s]	t. [s]	t. [s]
150	0.7	16-17	10	**0.01**	0.06	0.02	5.40	0.05	0.03
150	0.8	23	10	**0.05**	0.19	0.09	4.90	0.13	0.07
150	0.9	35-38	10	0.41	1.05	0.41	1.49	0.27	**0.13**
150	0.95	51-58	10	0.41	0.33	0.35	0.34	0.08	**0.07**
150	0.98	75-84	10	0.01	**0.00**	0.02	0.00	0.02	0.06
200	0.7	17-18	10	**0.06**	0.33	0.12	60.47	0.25	0.13
200	0.8	25-26	10	0.82	2.59	0.80	81.88	1.97	**0.67**
200	0.9	40-42	10	14.96	46.36	13.09	43.10	9.71	**2.47**
200	0.95	58-66	10	17.41	37.81	20.22	2.12	1.69	**0.52**
200	0.98	90-103	10	0.62	0.05	0.23	**0.01**	0.08	0.24
300	0.6	15-16	10	**0.10**	0.53	0.15	445.90	0.61	0.28
300	0.7	20-21	10	**0.75**	2.74	0.85	935.03	4.08	1.10
300	0.8	28-29	10	13.78	53.23	12.35	2431.00	56.52	**11.05**
500	0.4	10-11	10	**0.07**	0.62	0.13	tl	0.34	0.24
500	0.5	13	10	**0.39**	2.83	0.67	tl	2.15	0.97
500	0.6	17	10	**3.81**	19.04	5.10	tl	23.36	6.58
500	0.7	22-23	10	**70.04**	331.13	76.27	tl	467.00	106.53
1000	0.2	7-8	10	**0.05**	0.81	0.15	tl	0.18	0.23
1000	0.3	9-10	10	**0.32**	4.13	0.64	tl	1.28	0.95
1000	0.4	12	10	**2.95**	28.83	5.25	tl	12.78	7.21
1000	0.5	15	10	**42.27**	290.92	63.56	tl	225.80	76.94
3000	0.1	6-7	10	**0.24**	10.48	1.14	tl	0.75	1.01
3000	0.2	9	10	**4.26**	80.06	8.37	tl	11.87	10.39
5000	0.1	7	10	**1.54**	95.09	4.89	tl	3.40	5.63
5000	0.2	9-10	10	**53.84**	875.23	93.24	tl	116.24	87.97
10000	0.1	7-8	10	**22.87**	1408.91	49.56	tl	30.13	61.69
15000	0.1	8	10	**132.73**	7250.20	273.18	tl	149.46	259.98

Table 4: Results on random instances. Time limit is fixed at 1h.

	$\|V\|$	$\|E\|$	$\mu(G)$	$\lceil\chi_f(G)\rceil$	BBMCW t[s] tot	BBMCW t[s] pricing	MWSS t[s] tot	MWSS t[s] pricing
1-Insertions_5	202	1227	0.06	3	250.36	242.21	**52.03**	**44.22**
2-FullIns_5	852	12201	0.03	5	**12.53**	**5.74**	72.54	65.43
2-Insertions_4	149	541	0.05	3	305.63	301.98	**18.51**	**14.76**
3-FullIns_4	405	3524	0.04	6	**1.66**	**1.07**	4.12	3.46
3-Insertions_4	281	1046	0.03	3	t.l.	t.l.	**423.18**	**383.96**
4-FullIns_4	690	6650	0.03	7	**6.01**	**2.88**	22.11	18.77
5-FullIns_4	1085	11395	0.02	8	**270.47**	**255.02**	650.22	635.45
DSJC1000.9	1000	449449	0.90	215	**341.33**	**202.63**	3191.66	3061.14
DSJC125.1	125	736	0.09	5	741.27	738.68	**44.29**	**41.87**
DSJC125.5	125	3891	0.50	16	**2.36**	**1.85**	3.83	3.33
DSJC250.5	250	15668	0.50	26	**36.36**	**28.77**	184.66	176.64
DSJC250.9	250	27897	0.90	71	**5.07**	**4.47**	9.40	8.93
DSJC500.5	500	62624	0.50	43	**2522.41**	**2381.04**	t.l.	t.l.
DSJC500.9	500	112437	0.90	123	**30.91**	**24.19**	154.21	148.32
DSJR500.1c	500	121275	0.97	85	**8.73**	**8.43**	40.27	39.97
DSJR500.5	500	58862	0.47	122	**21.35**	**18.67**	94.86	93.04
flat300_20_0	300	21375	0.48	20	**387.39**	**322.87**	1439.75	1380.04
flat300_26_0	300	21633	0.48	26	**393.57**	**349.12**	2352.34	2308.34
flat300_28_0	300	21695	0.48	28	**136.06**	**118.29**	881.03	863.05
fpsol2.i.1	496	11654	0.09	65	**2.65**	**2.13**	11.15	10.65
fpsol2.i.2	451	8691	0.09	30	**4.19**	**3.39**	16.27	15.73
fpsol2.i.3	425	8688	0.10	30	**4.20**	**3.35**	14.66	13.91
homer	561	1628	0.01	13	**2.89**	**1.30**	9.39	7.92
inithx.i.1	864	18707	0.05	54	**44.70**	**11.91**	158.47	153.65
inithx.i.2	645	13979	0.07	31	**11.01**	**7.53**	74.76	69.49
inithx.i.3	621	13969	0.07	31	**6.53**	**5.50**	69.36	66.60
latin_square_10	900	307350	0.76	90	**108.90**	**49.15**	670.20	608.60
le450_25a	450	8260	0.08	25	1006.87	1002.37	**118.63**	**114.21**
le450_25b	450	8263	0.08	25	t.l.	t.l.	**1071.29**	**1067.70**
le450_5d	450	9757	0.10	5	**181.31**	**181.30**	t.l.	t.l.
mug100_1	100	166	0.03	4	336.86	334.61	**93.72**	**91.67**
mug100_25	100	166	0.03	4	168.27	166.59	**71.33**	**69.36**
mug88_1	88	146	0.04	4	80.70	79.55	**22.40**	**21.30**
mug88_25	88	146	0.04	4	30.38	29.30	**18.55**	**17.47**
mulsol.i.1	197	3925	0.20	49	**1.11**	**1.01**	1.75	1.53
mulsol.i.3	184	3916	0.23	31	1.31	1.28	**1.08**	**1.01**
mulsol.i.4	185	3946	0.23	31	1.44	1.40	**1.28**	**1.08**
myciel7	191	2360	0.13	5	**2.47**	**1.75**	3.33	2.68
queen10_10	100	1470	0.30	10	**3.19**	**2.64**	4.92	4.37
queen11_11	121	1980	0.27	11	**9.20**	**8.21**	13.87	12.98
queen12_12	144	2596	0.25	12	**41.51**	**39.63**	67.42	65.60
queen13_13	169	3328	0.23	13	**234.69**	**231.10**	303.05	299.73
queen14_14	196	4186	0.22	14	**1564.04**	**1558.14**	1922.45	1916.36
r1000.1c	1000	485090	0.97	96	**57.72**	**54.13**	773.80	770.50
r1000.5	1000	238267	0.48	234	**268.40**	**211.79**	2556.25	2508.37
r250.1c	250	30227	0.97	64	**1.35**	**1.32**	1.88	1.85
r250.5	250	14849	0.48	65	**3.88**	**3.54**	6.41	6.15
school1_nsh	352	14612	0.24	14	**3369.82**	**2433.05**	t.l.	t.l.
zeroin.i.1	211	4100	0.19	49	**1.67**	**1.62**	1.94	1.72

Table 5: Comparing the performance of BBMCW and MWSS as pricing algorithms in computing the fractional cromatic number $\chi_f(G)$. Time limit is fixed at 1h.

		BBMCW		TSM-MWC		WLMC	
Group	#	t[s]	# solved	t[s]	# solved	t[s]	# solved
ECC	15	19.71	**15**	**10.01**	**15**	11.31	**15**
KES	100	**2.04**	**81**	5.20	**81**	6.85	79
REF	129	**0.98**	**106**	1.94	**106**	2.69	**106**
WDP	499	17.70	**499**	**4.50**	**499**	13.92	**499**

Table 6: Results on real-world instances. Time limit is fixed at 600s.

			BBMCW		BBMCW no MCOVER		BBMCW no COVER no MCOVER		BBMCW no AMTS
	$\|V\|$	$\|E\|$	# calls	t[s]	# calls	t[s]	# calls	t[s]	t[s]
brock400_1	400	59723	7,869,547	84.0	13,793,678	115.1	24,941,238	235.3	93.5
brock400_2	400	59786	9,751,100	99.5	16,059,030	132.7	29,055,021	278.1	105.2
brock800_1	800	207505	41,490,556	830.4	75,834,270	1,155.8	124,128,767	2,079.6	931.6
brock800_2	800	208166	70,463,326	1,260.3	125,328,373	1,693.0	208,705,792	3,160.2	1,344.2
C250.9	250	27984	7,224,238	68.0	12,108,657	92.6	29,441,985	336.8	76.9
dsjc1000.5	1000	249826	1,967,325	41.4	3,084,468	52.3	4,618,631	68.7	42.9
p_hat500-3	500	93800	10,384,545	173.0	24,101,240	293.3	58,841,327	1,043.1	181.1
san200_0.9_3	200	17910	1,161,350	8.5	1,792,078	10.3	4,084,597	28.5	11.6
frb30-15-1	450	83198	25,081,348	122.4	29,937,620	126.6	52,624,228	215.1	140.2
frb30-15-2	450	83151	31,237,012	156.8	40,591,578	179.3	78,815,232	311.5	201.0

Table 7: Impact on performance of the (individual) bounding procedures employed by BBMCW.

References

[1] 2nd dimacs implementation challenge - np hard problems: Maximum clique, graph coloring, and satisfiability, 2017. URL `http://dimacs.rutgers.edu/Challenges/`.

[2] S. Butenko and W. E. Wilhelm. Clique-detection models in computational biochemistry and genomics. *European Journal of Operational Research*, 173(1):1–17, 2006.

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

[4] J. Desrosiers and M. E. Lübbecke. *A Primer in Column Generation*, pages 1–32. Springer US, Boston, MA, 2005.

[5] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Algorithms - ESA 2002, 10th Annual European Symposium, Proceedings*, pages 485–498, 2002.

[6] Z. Fang, C. Li, and K. Xu. An exact algorithm based on maxsat reasoning for the maximum weight clique problem. *J. Artif. Intell. Res.*, 55:799–833, 2016.

[7] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[8] T. Gschwind, S. Irnich, F. Furini, and R. W. Calvo. Social network analysis and community detection by decomposing a graph into relaxed cliques. Technical report, University of Mainz, 2016. Submitted.

[9] E. Hebrard and G. Katsirelos. Conflict directed clause learning for maximum weighted clique problem. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1316–1323. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[10] S. Held, W. J. Cook, and E. C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Math. Program. Comput.*, 4(4):363–381, 2012.

[11] H. Jiang, C. Li, and F. Manyà. An exact algorithm for the maximum weight clique problem in large graphs. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, USA.*, pages 830–838, 2017.

[12] H. Jiang, C. Li, Y. Liu, and F. Manyà. A two-stage maxsat reasoning approach for the maximum weight clique problem. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February, 2018*, pages 1338–1346, 2018.

[13] H. C. Lau and Y. G. Goh. An intelligent brokering system to support multi-agent web-based 4th-party logistics. In *14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2002), 4-6 November 2002, Washington, DC, USA*, page 154, 2002.

[14] C. 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, USA*, pages 939–946, 2013.

[15] 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 & OR*, 84:1–15, 2017.

[16] 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.

[17] 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*, 2018.

[18] C. M. Li and Z. Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[19] E. Malaguti, M. Monaci, and P. Toth. An exact approach for the vertex coloring problem. *Discrete Optimization*, 8(2), 2011.

[20] C. McCreesh, P. Prosser, K. Simpson, and J. Trimble. On maximum weight clique algorithms, and how they are evaluated. In *CP*, volume 10416 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2017.

[21] A. Mehrotra and M. A. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996.

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

[23] W. J. Pullan. Approximating the maximum vertex/edge weighted clique using local search. *J. Heuristics*, 14(2):117–134, 2008.

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

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

[26] 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. *Appl. Intell.*, 32(3): 311–329, 2010.

[27] P. San Segundo, D. Rodriguez-Losada, and A. Jimenez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & OR*, 38(2):571–581, 2011.

[28] 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.

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

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

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

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

[33] T. Sandholm and S. Suri. BOB: improved winner determination in combinatorial auctions and generalizations. *Artif. Intell.*, 145(1-2):33–58, 2003.

[34] 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 *WALCOM: Algorithms and Computation, 4th International Workshop, Bangladesh, Proceedings*, pages 191–203, 2010.

[35] Q. Wu and J. Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. *J. Comb. Optim.*, 26(1):86–108, 2013.

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

[37] Q. Wu and J. Hao. Solving the winner determination problem via a weighted maximum clique heuristic. *Expert Syst. Appl.*, 42(1):355–365, 2015.

[38] D. Zhang, O. Javed, and M. Shah. Video object co-segmentation by regulated maximum weight cliques. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland,Proceedings, Part VII*, pages 551–566, 2014.