A new branch-and-filter exact algorithm for binary constraint satisfaction problems

Pablo San Segundo\textsuperscript{1}

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

Fabio Furini

\textit{Istituto di Analisi dei Sistemi ed Informatica “Antonio Ruberti” – Consiglio Nazionale delle Ricerche (IASI-CNR), Rome, Italy. fabio.furini@cnr.it}

Rafael León

\textit{Universidad Camilo José Cela (UCJC), Science and Technology School, Madrid, Spain. rleon@ucjc.edu}

\section*{Abstract}

A binary constraint satisfaction problem (BCSP) consist in determining an assignment of values to variables which is compatible with a set of constraints. The problem is called binary because the constraints involve only pairs of variables. The BCSP is a cornerstone problem in Constraint Programming (CP), appearing in a very wide range of real-world applications. In this work, we develop a new exact algorithm which effectively solves the BCSP by reformulating it as a $k$-clique problem on the underlying microstructure graph representation. Our new algorithm exploits the cutting-edge branching scheme of the state-of-the-art maximum clique algorithms combined with two filtering phases in which the domains of the variables are reduced. Our filtering phases are based on coloring techniques and on heuristically solving an associated boolean satisfiability (SAT) problem. In addition, the algorithm initialization phase performs a reordering of the microstructure graph vertices which produces an often easier reformulation to solve. We carry out an extensive computational campaign on a benchmark of almost 2000 instances which encompasses numerous real and synthetic problems from the literature. The performance of the new algorithm is compared against two SAT-based solvers and three general purpose CP solvers. Our tests reveal that the new algorithm significantly outperforms all the others in several classes of BCSP instances.

\textit{Keywords:} Binary Constraint Satisfaction Problems, Constraint Programming, Exact Algorithm, Computational Experiments.

\textsuperscript{1}Corresponding author
1. Introduction

A constraint satisfaction problem (CSP), in its general form, asks for an assignment of values to the variables of the problem respecting a set of constraints. It is a central topic in Constraint Programming (CP) with many practical applications due to its broad representational scope: location of facilities, scheduling, car sequencing, vehicle routing and many others, see, e.g., [9] and part II of [54] for a survey.

Formally, we are given a set \( X = \{x_1, x_2, \ldots, x_k\} \) of \( k \) variables, each of which is associated to a finite set of values \( D(x_i) \), called the domain of the variable. Without loss of generality (w.l.o.g.), we assume \( D(x_i) \) is a subset of \( \mathbb{Z} \). In addition, we are given a set of \( q \) constraints \( C = \{c_1, c_2, \ldots, c_q\} \), and each constraint \( c_j \) is a pair \((X(c_j), R(c_j))\) where \( X(c_j) \subseteq X \) is the subset of variables involved in the constraint. This set of variables is called the scope of the constraint and \( R(c_j) \) is a set of tuples, called the relation of the constraint. Each tuple \( R(c_j) \in \mathbb{Z}^{X(c_j)} \) is composed of \( |X(c_j)| \) values allowed by the constraint for the variables in its scope. In other words, each tuple represents a feasible combination of variable values. The CSP calls for finding a feasible assignment of values to each one of the variables which satisfies all the constraints. If no solution exists, the problem is said to be inconsistent or unsatisfiable. Determining whether a CSP has a solution is \( \mathcal{NP} \)-complete, since the problem admits the boolean satisfiability (SAT) problem as a special case, the archetypal \( \mathcal{NP} \)-complete decision problem. The state-of-the-art exact algorithms to solve CSPs are general purpose solvers based on CP and SAT techniques. A review of these algorithms is provided in Section 1.2, as well as general purpose CSP solvers.

An important feature of the CSP is the number of variables involved in its constraints. Specifically, the arity of a constraint \( c_j, j = 1, 2, \ldots, q \), is the number of variables in its scope \( |X(c_j)| \), and the arity of a CSP is the maximum arity over its constraints. A CSP of arity two is known as a binary constraint satisfaction problem (BCSP). W.l.o.g., in the remainder of the paper, we assume that the BCSP is normalized, i.e., there is at most one constraint for each pair of variables. It is worth noticing that, if a relation \( R(c) \) of a constraint \( c \in C \) is the empty set, the CSP instance is unsatisfiable; therefore we assume that \( R(c) \neq \emptyset \), for every constraint \( c \in C \).

It is worth mentioning that any CSP can be transformed into a BCSP via a polynomial time reduction called binarization. This reduction can be performed in different ways, and we refer the interested reader to [20, 53, 67]. For this reason the BCSP is also \( \mathcal{NP} \)-complete. All the binarization techniques have typically the negative effects of increasing the number of variables and/or the size of the variable domains. Naturally, the size of the problem is also reflected in the computational effort to solve the instances. Therefore, even if theoretically the BCSP is equivalent to the CSP, in this article we focus on the BCSP and on exact algorithms to solve it.

An example of a real-world application of the BCSP is the decision version of the Train Platforming Problem (TPP), see e.g., [12]. Considering a railway station, the TPP input consists in a set of trains, each of which is associated to a set of possible patterns, describing a stopping platform, along with arrival and departure times. The railway station operational
constraints forbid the assignment of patterns to trains if this implies occupying the same platform at the same time interval. A TPP solution consists of assigning a pattern to each train which is compatible with the patterns assigned to the other trains (if such a solution exists). In this context, the trains correspond to the BCSP variables and the patterns determine the variable domains. The set of operational constraints defines the set of BCSP constraints and the tuples are the pairs of feasible patterns for each pair of trains. Other BCSP applications can be found, for example, in frequency assignment problems (see e.g., [47]), where the variables are the communication links and the variable domains correspond to the frequencies. Geographically close links interfere with each other, and therefore the BCSP constraints reflect the feasible assignment of the frequencies to the communication links.

In this paper, we design a new exact algorithm for the BCSP based on the graph transformation which reduces the problem to the $k$-clique problem ($k$-CP) on a $k$ partite graph. This graph is known as the microstructure of the BCSP, see [32], where this transformation has been initially proposed. Given a simple undirected graph $G = (V,E)$ with $n = |V|$ vertices and $m = |E|$ edges, a subset $K \subseteq V$ of vertices is called a clique if any two vertices of $K$ are connected by an edge. The $k$-CP calls for determining if a clique of size $k$ exists in $G$. It is worth mentioning that finding a $k$-clique for a $k$-partite graph, such as the microstructure graph, is clearly $NP$-complete. The reduction from the $k$-CP (one of Karp’s 21 $NP$-complete problems [34]) to the $k$-CP in a $k$-partite graph works as follows. From a graph $G = (V,E)$ and a value $k$ construct the graph $G'$ with vertex set $V \times \{1,2,\ldots,k\}$ and edges between vertices $(v,i)$ and $(w,j)$ if and only if $i \neq j$ and $v \neq w$ and $\{v,w\} \in E$. By construction $G'$ is $k$-partite and it contains a $k$-clique if and only if $G$ contains a $k$-clique.

To the best of our knowledge, the state-of-the-art exact algorithms for the BCSP are the general purpose CSP solvers described in Section 1.2. These solvers are used as terms of comparison to evaluate the computational performance of our newly developed exact algorithm (see Section 4). For the $k$-CP instead, we mention that in the literature there are algorithms to enumerate all $k$-cliques in $k$-partite graphs. We refer the interested reader to e.g., [28], [45] and the references therein. Since our problem consists only in determining if a $k$-clique exists, such enumeration algorithms are not designed for our purpose.

1.1. Reduction of the BCSP to the $k$-CP: the microstructure graph

The reduction of the BCSP to the $k$-CP works as follows. Given an instance $I$ of the BCSP, we construct a $k$-partite graph $G(I) = (V,E)$, called the microstructure graph, in which the vertices are the following variable-value pairs:

$$V = \bigcup_{i=1,2,\ldots,k} V_i = \left\{ (x_i,a) : a \in D(x_i) \right\}. \tag{1}$$

In other words, the vertex set $V = \{V_1, V_2, \ldots, V_k\}$ of $G(I)$ is partitioned into $k$ subsets which we denote the layers of the graph, each one corresponding to a variable of the original BCSP. In addition, we assume that the vertices are sorted according to the layers of the graph.

We denote $c(i,j)$ the constraint having $x_i$, $x_j$ as its scope ($1 \leq i < j \leq k$). The edge set $E$ of $G(I)$ is defined according to the BCSP constraints, i.e., there is an edge between two vertices if the two endpoints map to a pair of variable-values allowed by the constraints. Formally:
\[ E = \left\{ ((x_i, a), (x_j, b)) : \ 1 \leq i < j \leq k, \ (a, b) \in R(c(i, j)) \right\}. \] (2)

It is easy to see that, by construction, \( G(\mathcal{I}) \) is a \( k \)-partite graph, since the vertices of a layer corresponding to a BCSP variable form an independent set (a subset of pairwise non-adjacent vertices). Accordingly, any \( k \)-clique in the microstructure graph is composed of exactly one vertex per layer and, consequently, it corresponds to a feasible BCSP solution (a feasible assignment of values to the BCSP variables from their domain). Precisely, there is a one-to-one correspondence between the set of feasible solutions to the \( k \)-CP and the set of BCSP feasible solutions. Accordingly, unsatisfiability of the BCSP implies the non-existence of a feasible solution to the \( k \)-CP, and vice versa. We denote \( x_i^* \) the value assigned to the variable \( x_i \) \((i = 1, 2, \ldots, k)\) associated to a \( k \)-clique \( K^* \subseteq V \). For the \( K^* \) we also use the notation \( \{v(1), v(2), \ldots, v(k)\} \), where \( v(i) \in V_i \) is the vertex of \( K^* \) in layer \( i \). Accordingly, the feasible BCSP solution associated to \( K^* \) is: \( x_i^* = f(v(i)) \) \((i = 1, 2, \ldots, k)\), where the function \( f(v(i)) \) returns the value from the domain \( D(x_i) \) of the variable \( x_i \) associated to the vertex \( v(i) \).

Example 1. In Figure 1, we depict the microstructure graph \( G(\mathcal{I}) \) resulting from the reduction of a BCSP instance \( \mathcal{I} \) with \( k = 4 \), i.e., \( X = \{x_1, x_2, x_3, x_4\} \) and variable domains \( D(x_1) = \{1, 3, 5\}, D(x_2) = \{1, 2, 3\}, D(x_3) = \{5, 6\} \) and \( D(x_4) = \{1, 2, 3\} \). This BCSP instance has \( q = 6 \) constraints \( C = \{c_1, c_2, c_3, c_4, c_5, c_6\} \), with scopes \( X(c_1) = \{x_1, x_2\}, X(c_2) = \{x_1, x_3\}, X(c_3) = \{x_1, x_4\}, X(c_4) = \{x_2, x_3\}, X(c_5) = \{x_2, x_4\}, X(c_6) = \{x_3, x_4\} \) and relations \( R(c_1) = \{(1, 1), (1, 2), (3, 3), (5, 3)\}, R(c_2) = \{(1, 5), (3, 6)\}, R(c_3) = \{(1, 1)\}, R(c_4) = \{(1, 5), (2, 5), (2, 6), (3, 6)\}, R(c_5) = \{(1, 1)\} \) and \( R(c_6) = \{(5, 1), (6, 2), (6, 3)\} \). Figure 1 shows the partitioned microstructure graph composed of 4 layers, i.e., \( V_1, V_2, V_3 \) and \( V_4 \). These layers are represented in the picture by dashed rectangles. On top of the vertices we report the corresponding variable-value pairs. The edges represent compatible pairs of variable-values according to the constraints. The vertices depicted in red correspond to a clique \( K^* \) of size 4, which is a feasible solution to the \( k \)-CP. This solution, given by the clique \( K^* = \{v_1, v_4, v_7, v_9\} \), corresponds to a partial assignment of values to variables, and the children nodes are created by assigning values to one additional variable in a compatible manner with the constraints. When all possible feasible variable-value combinations have been examined, the algorithm backtracks to the previously assigned variable. This simple scheme has been enhanced overtime with many techniques, such as nogood recording, conflict-driven backjumping, variable and value ordering heuristics and restarts (see, e.g., [54]).

A fundamental technique to improve the performance of backtracking algorithms is to prevent local inconsistency, which is a partial variable-value assignment, satisfying the constraints,
Figure 1: A BCSP instance \( \mathcal{I} \) with \( k = 4 \) variables, \( X = \{x_1, x_2, x_3, x_4\} \) and \( q = 6 \) constraints. The picture shows the microstructure graph \( G(\mathcal{I}) \) obtained by the reduction from the BCSP to the \( k \)-CP. In red we report a clique \( K^* \) of size 4 given by the vertices \( \{v_1, v_4, v_7, v_9\} \), which corresponds to the feasible solution \( x_1^* = 1, \ x_2^* = 1, \ x_3^* = 5 \) and \( x_4^* = 1 \).

but not supporting any global feasible solution. Naturally, these situations can lead to unproductive search and deteriorates the performance of the exact algorithms.

Constraint propagation procedures try to prevent local inconsistency. To this end, CP-based algorithms execute constraint propagation techniques in the nodes of the branching tree, thereby enforcing a certain consistency level. In the past, the majority of the constraint propagation procedures were designed for the BCSP, i.e., for binary constraints. In practice, efficient CP-based algorithms for the BCSP typically enforce up to the consistency level denoted arc-consistency (AC)\[46\], but higher consistency levels exist, such as path-consistency \[43\] and \( k \)-consistency \[23\]. Depending on the specific problem constraints, CP-based algorithms can also achieve weaker forms of consistency than AC, such as forward checking \[27, 29\], directional arc-consistency \[19\] and unit propagation \[18\]. We refer the reader to the chapters 3 and 4 in \[54\] for a detailed description on this subject.

A recent stream of work has been the study of global constraints (non-binary CSPs) together with the development of efficient specialized propagators, such as the algorithm described in \[52\] for the all-different constraint. In addition, classical constraint propagators originally conceived for the BCSP have been extended for constraints of any arity. For example, arc-consistency is typically referred to as generalized arc-consistency for the \( n \)-ary case. Moreover, binary encodings of non-binary CSPs have also been studied and some specialized arc-consistency propagators and algorithms for such encodings have been proposed in the literature, see, e.g., \[55\].
Concerning the microstructure graph representation of the BCSP, another recent ongoing stream of (theoretical) work has been concerned with the generalization of the microstructure to non-binary CSPs, see, e.g., [14, 22]. Of interest to this stream has been the study of tractable classes of CSPs, i.e., those that can be recognized and solved in polynomial time exploiting the specific topology of the microstructure, see [16, 22, 48, 20, 33].

We end this literature review with two comments on existing CSP solvers. In first place, today’s modern solvers implement efficient propagators for many global constraints and can solve both binary and non-binary CSPs. To the best of our knowledge, there are no state-of-the-art solvers tailored specifically for the BCSP. In second place, boolean satisfiability (SAT) modules are now becoming increasingly popular. The SAT modules can either be called selectively inside the CP-based solvers for specific constraints, or be the engine driving the algorithm (given an adequate compilation to SAT). Examples of today’s successful SAT-based solvers are the OR-Tools: CP-SAT [49], PicatSAT and sCOP. Examples of purely CP-based solvers are Gecode [2], Minion [25], ILOG CP Optimizer [3], Choco, Mistral and Concrete. The list is by no means exhaustive.

To the best of our knowledge, the state-of-the-art exact methods to solve BSCPs are CP-based solvers and SAT-based solvers. For these reasons, in the computational section, we compare the performance of our new exact algorithm for the BCSP against some of the best general purpose CSP solvers (see the computational Section 4).

1.3. Main contributions and outline of the paper

In this paper, we design and test a new exact algorithm for the BCSP, based on a reduction of the problem to the $k$-CP on the underlying microstructure graph described in Section 1.1. The newly developed exact algorithm effectively relies on the state-of-the-art branching scheme of the maximum-clique problem algorithms. In order to improve the computational performance, the algorithm makes use of two different filtering phases. The first filtering phase, called COL-filtering, is based on coloring the microstructure graph with the aim of pruning the branching nodes and/or filtering vertices of the layers. The second filtering phase, called SAT-filtering, is based on an associated SAT-problem which is solved heuristically via unit propagation and failed literal inference techniques. In the initialization phase, the algorithm performs a preprocessing stage, called pre-filtering, in which the size of the microstructure graph is effectively reduced before the branching phase of the algorithm is performed. Finally, also in the initialization phase, the algorithm performs a second preprocessing stage, called re-partitioning, in which the BCSP is reformulated by reordering the vertices of the microstructure graph. This reformulation is surprisingly capable of improving the performance of the algorithm in several classes of instances.

A central contribution of this paper is to empirically show that, by reducing the BCSP to the $k$-CP, an effective algorithm can be designed. Such an algorithm, which is based on techniques inspired by the cutting edge algorithms for the maximum clique problem, is shown to be more effective than several CSP solvers for a number of families of BCSP instances. In details, the algorithm significantly outperforms three general purpose CP-based solvers and two general purpose SAT-based solvers. We tested 1,895 instances from the literature originated from benchmark problems of different nature. These instances can be found in the libraries of CSP...
instances and they have been used for benchmarking general purpose solvers. Our extensive computational results show that many classes of BCSP instances can be effectively solved by the new exact algorithm.

The remainder of the paper is structured as follows. Section 2 is devoted to the new exact algorithm for the BCSP. In Section 2.1, we describe the branching operations of the algorithm, and in Sections 2.2 and 2.3, we describe its filtering procedures. Section 3 describes the re-partitioning procedure and it presents some additional algorithmic improvements, the overall algorithm as well as some implementation details. Section 4 reports the results of the extensive computational campaign comparing against SAT-based and CP-based general purpose solvers. Finally, in Section 5 we draw some conclusions and we comment on future lines of work.

2. The new exact branch-and-filter algorithm

In this section, we present the new branch-and-filter (B&F) algorithm for the \( k \)-CP on the microstructure graph \( G(I) \). The proposed B&F algorithm is inspired by the recent state-of-the-art algorithms for the maximum-clique problem (MCP) and its variants, see, e.g., [15, 39, 40, 56, 57, 58, 62, 60, 63, 66, 64]. This family of combinatorial exact algorithms are based on the \( n \)-ary branching scheme proposed in [13], where at each node of the branching tree the children nodes are created by adding one vertex at a time to a partial clique solution. This branching scheme is particularly effective since, each time a vertex is added, its non-adjacent vertices are discarded and the graph is reduced. The subproblem graph associated to a node corresponds to the graph induced by the intersection of the neighborhoods of the vertices in the partial clique associated to the node (see §2.1 for a formal definition). The term “filter” in B&F refers to removing vertices from the subproblem graphs with the aim of pruning the nodes and/or make children nodes easier. It is worth noting that filtering vertices in our B&F algorithm is equivalent to deleting the corresponding values from their variable domains.

We denote \( \textbf{Bfilt} \) our new exact B&F algorithm. The initial letter ‘B’ in the name refers to the fact that \( \textbf{Bfilt} \) makes extensive use of bitstrings and efficient bitmasking operations during the execution of the algorithm. In the following sections we describe the \( \textbf{Bfilt} \) algorithm and its main components.

Before launching \( \textbf{Bfilt} \), a pre-filtering procedure is carried out to reduce the microstructure graph \( G(I) = (V, E) \) by filtering those vertices which cannot be part of any \( k \)-clique. This procedure removes the vertices which are inconsistent with a particular layer. Specifically, for each layer \( i = \{1, \ldots, k\} \), it removes any vertex \( v \notin V_i \) such that \( V_i \cup \{v\} \) is an independent set. Such vertices correspond to values of the BCSP which have no support in the variable \( x_i \). The pre-filtering procedure is effective in reducing the microstructure graph before the branching phase of \( \textbf{Bfilt} \) is executed. Consider for example the microstructure graph of Figure 1. The vertex \( v_{11} \) of layer \( V_4 = \{v_9, v_{10}, v_{11}\} \) can be removed since it is not adjacent to any of the vertices of the first layer \( V_1 = \{v_1, v_2, v_3\} \).

2.1. The branching scheme of \( \textbf{Bfilt} \)

In this section we describe the way the B&F tree of \( \textbf{Bfilt} \) is generated. A branching node is created by selecting a layer, and one of its vertices, and by adding it to the clique \( K \)
constructed by the branching operations (at the root node, $\hat{K} = \emptyset$). By selecting a vertex in the layer, the branching operation, de facto, assigns a value to the corresponding BCSP variable.

We denote $N(v)$ the neighbourhood of a vertex $v \in V$, i.e., the set of its adjacent vertices. For a subset of vertices $U \subseteq V$ of a graph $G$, we denote $G[U] = (U, E[U])$ the graph induced by $U$, where $E[U] \subseteq E$ contains those edges with both endpoints in $U$. The effect of the branching on a vertex of a layer is to create a suproblem graph $\hat{G} = (\hat{V}, \hat{E})$ induced by the intersection of the neighbourhoods of the vertices $v \in \hat{K}$. As a by product, all the the remaining vertices of the branching layer are removed. Formally, the vertices and edges of $\hat{G}$ are:

$$\hat{V} = \bigcap_{v \in \hat{K}} N(v), \quad \text{and} \quad \hat{E} = E[\hat{V}].$$  \hfill (3)

At each node of the B&F tree the corresponding subproblem graph $\hat{G}$ has the vertex set partitioned into $\alpha = k - |\hat{K}|$ layers, i.e., $\hat{V} = \{\hat{V}_1, \hat{V}_2, \ldots, \hat{V}_\alpha\}$. Formally:

$$\hat{V}_j = \left\{ v \in V_{i(j)} : (v, u) \in E, u \in \hat{K} \right\}, \quad j = 1, 2, \ldots, \alpha,$$  \hfill (4)

where $i(j)$ is the index of the original layer in the microstructure graph of the layer $j$ in the subproblem graph $^2$. Precisely, the vertex set $\hat{V}_j$ is composed of the vertices of the original layer $i(j)$ which are neighbors to every vertex in $\hat{K}$. As a consequence of the branching operations described above, each node of the tree is associated to the pair $(\hat{K}, \hat{G})$, where $\hat{K}$ is the clique of size $|\hat{K}| \leq k$ constructed by the branching operations and $\hat{G}$ is the corresponding subproblem $\alpha$-partite graph.

The branching scheme of Bfilt is an $n$-ary branching scheme. It selects one layer $j$ of $\hat{V}$ for branching, and then creates $|\hat{V}_j|$ branching nodes by adding to $\hat{K}$ each vertex $v \in \hat{V}_j$, one per node. It is worth noticing that it is sufficient, for the algorithm to be complete, to branch on the vertices corresponding to one layer only. This is due to the fact that, in order to build a $k$ clique, a vertex per layer is necessary.

A first fathoming condition for backtracking which allows to end the branching recursion is given by the following observation:

**Observation 1.** Given a pair $(\hat{K}, \hat{G})$, the corresponding branching node can be fathomed if $\hat{V}_j = \emptyset$, for any $j \in \{1, 2, \ldots, \alpha\}$.

This fathoming condition represents the fact that if one layer of the subproblem graph becomes empty, a clique of size $k$ cannot be constructed. By the nature of the subproblem graph, indeed one vertex per layer has to be selected in order to construct a clique of size $k$. This condition is equivalent to reaching an empty domain for the variable of the corresponding layer.

---

$^2$At the root node of the B&F tree, $\hat{G}$ is the microstructure graph.
Finally, if $|\hat{K}| = k$, the B&F algorithm stops and we reconstruct the solution to the original BCSP instance associated to $\hat{K}$. In this case the original BCSP instance is satisfiable. Alternatively, if the recursion ends without reaching this condition, the original BCSP instance is unsatisfiable.

In this paragraph, we show an example of one branching operation at the root node for the BCSP instance depicted in Figure 1. In this example, we describe the effect of branching by selecting the layer 1 and its first vertex $v_1 \in V_1$. The incumbent clique $\hat{K}$ becomes $\{v_1\}$ and $\alpha = 3$. The resulting subproblem 3-partite graph $\hat{G}$ is composed of 3 layers: $\hat{V}_1 = \{v_4, v_5\}, \hat{V}_2 = \{v_7\}, \hat{V}_3 = \{v_9\}$. In this example, $j = 1, 2, 3$ and the function $i(j)$ provides the mapping between the indexes of the layers of the subproblem graph to the layers of the original graph, i.e., $i(1) = 2, i(2) = 3$ and $i(3) = 4$. Since none of the layers are empty, the branching node associated to this subproblem cannot be fathomed.

2.2. The color-filtering phase of $Bfilt$

In this section we describe the color-filtering phase carried out by the algorithm $Bfilt$. Given a node $(\hat{K}, \hat{G})$ associated to the subproblem graph $\hat{G}$ and the clique $\hat{K}$, the color-filtering phase is executed first, before the SAT-filtering phase (described in Section 2.3) and the branching. It aims at fathoming the node or, if this is not possible, filtering some of its vertices.

We observe that if a vertex $v$ in a layer $\hat{V}_j$ ($j = 1, 2, \ldots, \alpha$) exists such that its neighbourhood does not contain any of the vertices of a different layer $l \neq j$, then the vertex can be filtered, i.e., it can be removed from $\hat{G}$. This is due to the fact that if this vertex is part of a clique, then this clique cannot contain any vertex from $\hat{V}_l$, i.e., it cannot be an $\alpha$-clique. This reasoning is summarized in the following observation.

**Observation 2.** If a vertex $v \in \hat{V}_j$ of $\hat{G}$ exists such that $N(v) \cap \hat{V}_l = \emptyset$, for any layer $l \in \{1, 2, \ldots, \alpha\}, l \neq j$, then the vertex $v$ can be removed from $\hat{G}$.

From the perspective of the BCSP problem, such a vertex corresponds to a variable value that is not supported by another variable, i.e., it is not compatible with any of the (non-filtered) values of another variable domain.

In order to efficiently find such vertices, $Bfilt$ executes a modified version of the greedy independent set sequential heuristic ($ISEQ$). We recall that $ISEQ$ computes a partition of the vertex set of a graph into independent sets (colors). It does so by sequentially examining the vertices following a predefined order and builds one independent set at a time, see [57], where it is used inside an MCP algorithm. We denote $FILT-ISEQ$ the modified version of $ISEQ$ employed by $Bfilt$, which is explained in the following.

In its initialization phase, $FILT-ISEQ$ creates a duplicated set $B$ of the vertices of $\hat{V}$ and then iteratively constructs a collection of independent sets $\mathcal{P}(\hat{V})$, one at a time. Additionally, and during execution, $FILT-ISEQ$ stores a label $l(I)$ for each independent set $I \in \mathcal{P}(\hat{V})$. This label corresponds to the layer index in the microstructure graph of the first vertex added to $I$. In order to build $I$ (starting from the empty set), it examines each vertex $v \in B$ and checks whether $I \cup \{v\}$ is an independent set. If this is the case, it further tests whether the layer $i$ of $v$ in the microstructure graph, is the same as the label of $I$, i.e., $i = l(I)$. If both
conditions hold, \( v \) is removed from \( B \) and added to \( I \). If, on the other hand, \( i \neq l(I) \), \( v \) is removed from \( B \) and according to Observation 2, filtered from the graph \( \hat{G} \). If \( I \cup \{v\} \) is not an independent set, vertex \( v \) remains in \( B \) and the next vertex is examined. Each time an independent set \( I \) is constructed, it is added to the collection \( \mathcal{P}(\hat{V}) \). The procedure halts when \( B \) is the empty set.

Since at most one vertex in each independent set can be part of any clique in \( \hat{G} \), if \( \text{FILT-ISEQ} \) manages to partition the remaining unfiltered vertices of \( \hat{G} \) in less than \( \alpha \) independent sets (colors), the corresponding branching node can be fathomed. This fact is summarized in the following observation.

**Observation 3.** After the execution of \( \text{FILT-ISEQ} \), if the number of independent sets of \( \mathcal{P}(\hat{V}) \) is less than \( \alpha \), the corresponding branching node can be fathomed.

If the node of the B&F tree is not fathomed after the execution of \( \text{FILT-ISEQ} \), we keep track of the reduced \( \hat{G} \) and denote it \( \hat{G} = (\hat{V}, E[\hat{V}]) \). Furthermore, we set its layers \( \hat{V} \) to each one of the independent sets (colors) in the collection \( \mathcal{P}(\hat{V}) \). Subsequently, \( \hat{G} \) becomes the input graph of the ensuing SAT-based filtering phase. It is worth noting that, at the end of its execution, \( \text{FILT-ISEQ} \) computes a coloring of the reduced graph \( \hat{G} \).

**Example 2.** In Figure 2, we depict the subproblem graph \( \hat{G} \) with \( \alpha = 3 \) and layers: \( \hat{V}_1 = \{v_1, v_2\} \), \( \hat{V}_2 = \{v_3, v_4\} \) and \( \hat{V}_3 = \{v_5, v_6\} \). The corresponding layers of the microstructure graph \( G(I) \) have the same indexes. The edges of the graph \( \hat{G} \) are also depicted in the figure. Using this demonstration graph, we illustrate the operation of the color-filtering phase. At the start of the procedure, \( \text{FILT-ISEQ} \) sets \( B \) to \( \hat{V} \) and the collection of independent sets \( \mathcal{P}(\hat{V}) \) is the empty set. In the first iteration, \( \text{FILT-ISEQ} \) computes the independent set \( I_1 \). The first vertex added to \( I_1 \) is \( v_1 \), after which the label \( l(I_1) \) is set to 1, since \( v_1 \) belongs to the layer \( V_1 \) of the microstructure graph. The procedure continues by adding \( v_2 \) to \( I_1 \), since it also belongs to the layer \( V_1 \), but the next selected vertex \( v_3 \) is skipped as it is a neighbour of both \( v_1 \) and \( v_2 \). \( \text{FILT-ISEQ} \) further continues by selecting and filtering vertex \( v_4 \), because, while the vertex can be added to \( I_1 \), it does not belong to the layer \( V_1 \). Vertex \( v_5 \) is further skipped because it is adjacent to \( v_1 \) and, finally, vertex \( v_6 \) is filtered for the same reason as \( v_4 \), i.e., it does not belong to layer \( V_1 \). The resulting independent set at the end of this iteration is \( I_1 = \{v_1, v_2\} \), which is added to \( \mathcal{P}(\hat{V}) \), and the filtered vertices of \( \hat{G} \) are \( \{v_4\} \) and \( \{v_6\} \). In the last two iterations, \( \text{FILT-ISEQ} \) computes the independent sets \( I_2 = \{v_3\} \) and \( I_3 = \{v_5\} \). The node of the B&F tree is not fathomed according to Observation 3, since the number of independent sets is 3, the same as the number of layers in \( \hat{G} \). The resulting reduced graph \( \hat{G} \) has the set of vertices \( \hat{V} = \{v_1, v_2, v_3, v_5\} \), which correspond to the colored vertices in the figure. Finally, we note that \( \mathcal{P}(\hat{V}) = \{I_1, I_2, I_3\} \) is an independent set partition of \( \hat{V} \) (also a coloring of \( \hat{G} \)), and its independent sets are each one of the layers of \( \hat{G} \).

2.3. The SAT-filtering phase of Bfilt

After the color-filtering phase is over and before the branching, Bfilt executes the SAT-filtering phase. This phase works on the reduced graph \( \hat{G} \) and it has the same two goals of

\footnote{A coloring corresponds to an assignment of colors to the vertices of a graph in such a way that two adjacent vertices receive different colors; it is a partition of the vertex set into independent sets.}
Figure 2: The subproblem graph $\tilde{G}$ used in Example 2. The colored vertices reflect the coloring obtained by \textsc{Filt-ISeq} on the reduced subproblem graph $\tilde{G} = \tilde{G}[V \setminus \{v_4, v_6\}]$. 

the color-filtering phase: i) fathoming a node in case the procedure manages to prove that an $\alpha$-clique does not exist in $\tilde{G}$, and ii) filtering individual vertices which cannot be part of any $\alpha$ clique.

The SAT-filtering phase exploits a reduction of the $\alpha$-CP to the boolean satisfiability (SAT) problem, where $\alpha$ is the number of layers of the graph $\tilde{G} = (\tilde{V}, E[\tilde{V}])$. This reduction operates as follows. Each vertex $v \in \tilde{V}$ is associated to a boolean variable $y_v \in \{0, 1\}$. A literal of a boolean variable $y$ refers to its value, i.e., either value 1 (denoted $y$, the positive literal) or value 0 (denoted $\bar{y}$, the negative literal). A clause is a finite collection of literals linked by logical operators (e.g., $\lor$, $\land$, $\neg$). A unit clause refers to a clause with only one literal. A conjunctive normal form (CNF) boolean formula is a conjunction of clauses, each of which is a disjunction of literals. The SAT problem associated to the $\alpha$-CP is defined by a CNF boolean formula which has two types of clauses: non-edge clauses and layer clauses. It calls for an interpretation of the variables \footnote{An assignment of values to each one of the variables.} that satisfies all its clauses. The non-edge clauses, associated to the non-adjacent pairs of vertices in $\tilde{G}$, contain only negative literals and they read as follows:

$$h_{uv} \equiv (\bar{y}_u \lor \bar{y}_v) \quad uv \in E[\tilde{V}].$$  \hspace{1cm} (5)

The layer clauses, associated to the layers of $\tilde{G}$, contain only positive literals and they read as follows:

$$s_j \equiv (y_{v(1)} \lor y_{v(2)} \lor \ldots \lor y_{v(g)}) \quad j = 1, 2, \ldots, \alpha,$$  \hspace{1cm} (6)

where for each layer $j$, the function $v(l)$ returns the vertex in the microstructure graph of the $l$-th vertex in the layer $j$, and $g = |\tilde{V}_j|$. We denote $S(\tilde{G})$ the SAT problem obtained by the reduction of the $\alpha$-CP in $\tilde{G}$.

The subproblem graph $\tilde{G}$ contains a clique of size $\alpha$ if and only if the associated SAT problem $S(\tilde{G})$ is satisfiable. It is therefore sufficient to check that $S(\tilde{G})$ is unsatisfiable to prove that the subproblem graph $\tilde{G}$ does not contain a clique of size $\alpha$. Accordingly, the following observation provides a fathoming condition of a branching node:

\textbf{Observation 4.} \textit{Given a subproblem graph $\tilde{G}$, if the associated SAT problem $S(\tilde{G})$ is unsatisfiable, the corresponding branching node can be fathomed.}
Solving \( S(\bar{G}) \) in every node of the B&F tree can be computationally challenging. In an attempt to efficiently prove only unsatisfiability, \texttt{Bfilt} employs unit propagation (UP) and failed literal detection (FL). These procedures are typically used by state-of-the-art SAT algorithms and, recently, also by MCP algorithms (see e.g., [42]). We refer the interested reader to e.g., [18], where these techniques have been proposed. We adapt these two procedures for the specific requirements of \( S(\bar{G}) \). Specifically, the UP procedure exploits the fact that layer unit clauses can only be satisfied by setting to 1 the corresponding variables. Our UP procedure starts by determining the set of unit clauses in \( S(\bar{G}) \); if no unit clauses are found it terminates. The UP procedure then selects a unit clause, fixes to 1 the corresponding boolean variable and fixes to 0 the variables in the corresponding non-edge clauses (5). As a result of this operation, every layer clause with all except one variable remaining unassigned is added to the initial set of unit clauses. In the next step, UP selects a remaining unit clause and the procedure is repeated until every unit clause has been selected or until a layer clause is found such that all its variables are set to 0. In the latter case, the corresponding SAT problem \( S(\bar{G}) \) is unsatisfiable, and the node of the tree is fathomed according to Observation 4. Once the UP procedure terminates, all variables set to 0 during the execution are filtered from the graph \( \bar{G} \), since they cannot make part of any \( \alpha \)-clique in \( \bar{G} \). This filtering condition is summarized as follows.

**Observation 5.** After the execution of the UP procedure, the vertices associated to the boolean variables set to 0 can be deleted from the graph \( \bar{G} \).

**Example 3.** We report in this paragraph the operations of UP according to the example graph \( \bar{G} \) of Figure 3. This graph has the following \( \alpha = 3 \) layers: \( \bar{V}_1 = \{v_1, v_2\}, \bar{V}_2 = \{v_3, v_4, v_5\} \) and \( \bar{V}_3 = \{v_6\} \). The edges of the graph are also reported in the figure. The SAT problem \( S(\bar{G}) \) encoding \( \bar{G} \) is composed of the following clauses. Its non-edge clauses (5) are:

\[
\begin{align*}
    h_{v_1v_2} &\equiv (\bar{y}_{v_1} \lor y_{v_2}), & h_{v_3v_4} &\equiv (\bar{y}_{v_3} \lor y_{v_4}), & h_{v_1v_5} &\equiv (\bar{y}_{v_1} \lor y_{v_5}), \\
    h_{v_2v_6} &\equiv (\bar{y}_{v_2} \lor y_{v_6}), & h_{v_3v_4} &\equiv (\bar{y}_{v_3} \lor y_{v_4}), & h_{v_3v_5} &\equiv (\bar{y}_{v_3} \lor y_{v_5}), \\
    h_{v_3v_6} &\equiv (\bar{y}_{v_3} \lor y_{v_6}), & h_{v_4v_5} &\equiv (\bar{y}_{v_4} \lor y_{v_5}), & h_{v_5v_6} &\equiv (\bar{y}_{v_5} \lor y_{v_6}).
\end{align*}
\]

and its layer clauses (6) are:

\[
\begin{align*}
    s_1 &\equiv (y_{v_1} \lor y_{v_2}), & s_2 &\equiv (y_{v_3} \lor y_{v_4} \lor y_{v_6}), & s_3 &\equiv (y_{v_6}).
\end{align*}
\]

In the example, the layer clause \( \bar{V}_3 \) corresponding to layer 3 is a unit clause, so UP sets \( y_{v_6} \) to 1. It follows from the non-edge clauses which contain \( y_{v_6} \), i.e., \( h_{v_2v_6}, h_{v_3v_6} \) and \( h_{v_5v_6} \), that the boolean variables \( y_{v_2}, y_{v_3}, y_{v_5} \) have to be set to 0. As a consequence, the layer clause corresponding to layer 1 becomes a unit clause. In the next iteration, UP sets the boolean variable \( y_{v_1} \) to 1 and according to its non-edge clauses, sets \( y_{v_4} \) to 0. At this point, the layer clause \( s_2 \) has all its variables set to 0 so the SAT problem \( S(\bar{G}) \) is unsatisfiable. According to Observation 4, the node is fathomed. In Figure 3, the vertices which correspond to boolean variables set to 1 by UP are coloured in red. The vertices associated to boolean variables set to 0 are coloured in grey.

We describe in this paragraph our FL procedure. Consider a non-unit layer clause and one of its variables. We say the (positive) literal \( y_v \) is failed if, when the variable is set to 1 and
Figure 3: The subproblem graph $\tilde{G}$ of Example 3. Red vertices are associated to boolean variables set to 1 by $\text{UP}$. Vertices in grey are filtered from $\tilde{G}$ since they are associated to boolean variables set to 0 by $\text{UP}$.

\[ \tilde{V}_1 = \{v_1, v_2\} \]
\[ \tilde{V}_2 = \{v_3, v_4, v_5\} \]
\[ \tilde{V}_3 = \{v_6\} \]

the assignment is propagated according to $\text{UP}$, a layer clause with every variable set to 0 is attained. If this is the case, the variable $y_{v_5}$ is fixed to 0, and its corresponding vertex $v$ can be filtered from $\tilde{G}$. Checking this condition for all the variables in layer clauses can be computationally challenging. A good compromise between computational overhead and filtering power is achieved by $\text{Bfilt}$ when $\text{FL}$ is restricted to layer clauses with two literals. The filtering operations of $\text{FL}$ are summarized by the following observation.

**Observation 6.** After the execution of the $\text{FL}$ procedure, the vertices which generate an empty layer clause via $\text{UP}$ can be deleted from the graph $\tilde{G}$.

The SAT-filtering phase of $\text{Bfilt}$ interleaves $\text{FL}$ with $\text{UP}$. Initially, procedure $\text{UP}$ is called upon $S(\tilde{G})$ and, as a result, all layer unit clauses, as well as those that have become unit clauses when some of its variables are set to 0 during the execution of $\text{UP}$, are processed. If the node of the B&F tree is not fathomed after the termination of $\text{UP}$, $\text{Bfilt}$ then calls procedure $\text{FL}$. If, as a result of the latter, a failed literal is detected, its layer clause becomes unit clauses when the variable is set to 0, so $\text{UP}$ is then called upon the latter. The procedure continues until either $S(\tilde{G})$ remains unchanged after the execution of $\text{FL}$ or unsatisfiability of $S(\tilde{G})$ can be proved.

**Example 4.** We show the operations carried out by $\text{Bfilt}$ during the SAT-filtering phase, explaining in detail the operations carried out by $\text{FL}$. In Figure 4 we depict a subproblem graph $\tilde{G}$ with $\alpha = 3$ layers: $\tilde{V}_1 = \{v_1, v_2\}$, $\tilde{V}_2 = \{v_3, v_4\}$ and $\tilde{V}_3 = \{v_5, v_6\}$. The edges of the graph are also reported in the figure. In this example there are no unit clauses, so procedure $\text{UP}$ terminates immediately. However, the layer clause $s_3$ contains two literals so $\text{FL}$ attempts to fail both positive literals of $s_3$. We consider first the variable $y_{v_5} \in s_3$. Setting $y_{v_5}$ to 1 and propagating this assignment according to $\text{UP}$ leads to the layer clause $s_2$ having all its variables set to 0, in a similar way as in Example 3. This proves that the positive literal of $y_{v_5}$ is failed. Consequently, the variable is set to 0 and the layer clause $s_3$ becomes unit, since $y_{v_6}$ is now the only variable that remains unassigned. $\text{UP}$ is then called for this unit clause, and, as a result, it sets every variable of the layer clause $s_1$ to 0, since $v_6$ is non-adjacent to every variable in $s_1$. According to Observation 4 the node is fathomed. In Figure 4, the vertex colored in red is associated to the boolean variable of the failed literal and we depict in grey
the vertices which took part in the reasoning.

Figure 4: The subproblem graph $\tilde{G}$ of Example 4. The vertex coloured in red corresponds to a failed literal. In grey the vertices of the boolean variables that participated in the reasoning of the SAT-filtering phase of $Bfilt$.

\[ \tilde{V}_1 = \{v_1, v_2\} \]
\[ \tilde{V}_2 = \{v_3, v_4\} \]
\[ \tilde{V}_3 = \{v_5, v_6\} \]

3. The additional algorithmic improvements of $Bfilt$: the algorithm $Bfilt^+$

In this section, we present the additional algorithmic improvements of $Bfilt$ which further enhance its computational performance. We recall that $Bfilt$ works on the microstructure graph which is, by nature, a $k$-partite graph. It is worth mentioning, however, that $Bfilt$ can also work with any $k$ partition of the microstructure graph. By analysing preliminary extensive computational results, we noticed that working on the original partition is, for some classes of instances, not the most effective choice (see Section 4.3). This is due to the fact that the original order of the vertices in the microstructure graph does not correspond to the most effective order used by the state-of-the-art maximum clique algorithms. To face this problem, we propose two algorithmic enhancements that enrich the algorithm $Bfilt$, de facto producing an improved version denoted $Bfilt^+$. In a nutshell, we attempt to reorder the vertices of the microstructure graph according to a new partition of size $k$ and execute $Bfilt$ on this reformulation of the original BCSP instance. This new re-partition procedure is described in Section 3.1. If the size of the new partition is too large, $Bfilt^+$ resorts to the initial partition of the microstructure graph and executes $Bfilt$. Finally, for the intermediate cases $Bfilt^+$ executes the maximum clique procedure described in Section 3.3.

3.1. The re-partition procedure of $Bfilt^+$

The re-partition procedure attempts to find a new partition of size $k$ (and accordingly a new ordering) of the vertices of the microstructure graph that is more effective for clique-based algorithms such as $Bfilt^+$. In addition, the procedure de facto reformulates the original BCSP instance by computing new layers, i.e., a new (independent set) partition of the microstructure graph. The ordering of the vertices is a very significant factor for the computational performance of exact algorithms for the maximum clique problem (MCP), see, e.g., [61] or Section 3 of [38]. According to [61], the COLOR_SORT ordering is to be preferred when the graph admits an independent set partition of size close to its clique number. Consequently, we design for $Bfilt^+$ a SORT procedure, inspired in [61, 37], which computes a COLOR_SORT ordering as follows.

The procedure SORT reorders the vertices of the microstructure graph according to the (independent set) partition obtained by computing maximum/maximal independent sets in
the microstructure graph. The key observation is that computationally hard \( k \)-CP instances are typically dense and, therefore, computing maximum independent sets is often easier. The computed independent sets will be the new layers employed by \texttt{Bfilt}. In order to effectively reduce the size of the branching tree of \texttt{Bfilt+}, vertices are ordered by \texttt{SORT} according to the following two criteria: \( i \) the layers (independent sets) of the new partition are sorted according to non-decreasing size and \( ii \) inside each layer, vertices are sorted according to non-increasing vertex degree. The first criteria aims at selecting a branching layer with a small number of vertices. The second criteria aims at examining first those vertices in the branching layer which are more likely to be part of a \( k \)-clique.

To compute each maximum independent set, \texttt{Bfilt+} executes the state-of-the-art MCP algorithm [63] on the complement of the microstructure graph. In the case when finding maximum independent sets is computationally challenging, the \texttt{SORT} procedure computes maximal independent sets within a time limit of 0.1 seconds.

Thanks to extensive preliminary tests, we noticed that the \texttt{SORT} procedure is not always able to find new partition of size equal to \( k \). Accordingly, we change the partition and reorder the vertices of the original microstructure graph if and only if the \texttt{SORT} procedure is able to find a new partition of size exactly \( k \). We denote the new \( k \)-partition \( \{\mathcal{V}_1, \ldots, \mathcal{V}_k\} \) and denote the new \( k \) partite graph \( G(\mathcal{I}) = (\mathcal{V}, \mathcal{E}) \). It is worth mentioning that if a partition of size strictly smaller than \( k \) is found, then the BCSP instance is unsatisfiable. The vertices and the edges of \( G(\mathcal{I}) \) are the same as \( G(\mathcal{I}) \), only the layers differ. By construction, each layer \( j = 1, 2, \ldots, k \) of \( G(\mathcal{I}) \) is a maximum (or maximal) independent set in the subgraph \( G(\mathcal{I})[\bigcup_{i=1}^{j} \mathcal{V}_i] \). In case a new partition of size \( k \) is found, \texttt{Bfilt+} executes \texttt{Bfilt} on \( G(\mathcal{I}) \). If, instead, the \texttt{SORT} procedure fails and the new partition is of size strictly greater than \( k + 10 \) then \texttt{Bfilt+} executes \texttt{Bfilt} on the original microstructure graph \( G(\mathcal{I}) \). Finally, if the new partition is of size strictly greater than \( k \) and smaller than \( k + 10 \), then \texttt{Bfilt+} executes the maximum clique procedure described in Section 3.3.

**Example 5.** We describe in this paragraph the operations executed by the \texttt{SORT} procedure to compute the \texttt{COLOR\_SORT} ordering for the microstructure graph of Figure 1. For this graph, the procedure is able to find a new partition of size \( k = 4 \). Figure 5 illustrates the new ordering of the vertices, as well as the new layers. In the example, the degrees of the vertices of the microstructure graph are: \( \text{deg}(V) = \{4, 2, 1, 3, 3, 3, 4, 5, 3, 1, 1\} \), e.g., \( \text{deg}(v_1) = 4 \), \( \text{deg}(v_2) = 2 \).

The maximum independent set of the microstructure graph, and first independent set of the new partition, is \( \{v_9, v_5, v_2, v_{11}, v_{10}, v_3\} \), where the vertices are sorted according to the ordering. The second maximum independent set in the graph induced by the remaining vertices is \( \{v_8, v_7\} \), the next one is \( \{v_1, v_6\} \) and the last one is the singleton \( \{v_4\} \). Finally, layers are sorted according to non-decreasing size, i.e., in reverse order as they are computed. The resulting \texttt{COLOR\_SORT} ordering is: \( v_4 < v_1 < v_6 < v_8 < v_7 < v_9 < v_5 < v_2 < v_{11} < v_{10} < v_3 \), and the new layers, \( \mathcal{V}_1 = \{v_4\}, \mathcal{V}_2 = \{v_1, v_6\}, \mathcal{V}_3 = \{v_8, v_7\}, \mathcal{V}_4 = \{v_9, v_5, v_2, v_{11}, v_{10}, v_3\} \), are depicted in Figure 5 in different colors. For each layer in the figure, the vertices at the bottom come first in the ordering.

### 3.2. Color-filtering additional enhancements

In this paragraph, we consider a further enhancement of the color-filtering phase. The key idea is that the procedure \texttt{FILT\_ISEQ} can be applied to any (layer) partition of size \( k \) of the
subproblem graph $\tilde{G}$, as long as its vertices are processed according to the layers (see Section 2.2, where the FILT-ISEQ procedure is described). We illustrate this enhancement via the following example. Consider the new partition of the subproblem graph depicted in Figure 2 into the following three layers: $\hat{V}_1 = \{v_1, v_6\}$, $\hat{V}_2 = \{v_3, v_4\}$ and $\hat{V}_3 = \{v_2, v_5\}$. The graph $\tilde{G}$ computed by FILT-ISEQ according to the new layers $\hat{V}$, has the (reduced) set of vertices $\tilde{V} = \{v_1, v_3, v_5, v_6\}$, since $v_6$ would be selected before $v_2$ and, $v_2$ is not adjacent to any of the vertices in the independent set $\hat{V}_1$. It is worth noting that the FILT-ISEQ procedure can be applied more than once with the goal of further reducing the subproblem graph. In the example, by executing FILT-ISEQ in reverse order (selecting vertices from last to first index number) on the graph $\tilde{G}$, $\tilde{V} = \{v_1, v_2, v_3, v_5\}$ (filtered by the first execution of FILT-ISEQ), the vertex $v_2$ is removed from $\tilde{G}$, since it is not adjacent to the independent set determined by the singleton $v_5$. It should be pointed that the new graph, which is determined by the vertices $\{v_1, v_3, v_5\}$, cannot be reduced further, since these vertices form a clique of size three.

Specifically, Bfilt+ employs the color-filtering enhancement in case a new partition of size $k$ is found by the re-partition procedure. With the aim of reducing and filtering the subproblem graph $\tilde{G}$, Bfilt+ executes the FILT-ISEQ procedure using both partitions and processing the vertices also in reverse order. If two calls to FILT-ISEQ are executed consecutively, the second call operates on the filtered subproblem graph of the first call. The final filtered graph becomes then the final $\tilde{G}$ graph for the SAT-filtering phase (see section 2.3).
3.3. The maximum clique procedure of Bfilt+

In this section, we describe the extension of Bfilt which is executed when the re-partition procedure determines a partition of size $\ell$ with $k < \ell \leq k + 10$. In this case, the SORT procedure reorders the vertices of the microstructure graph determining a new $\ell$-partite graph which we call $G_\ell = (V_\ell, E_\ell)$. The vertices and the edges of $G_\ell$ are the same as $G(I)$, only the partition, the number of layers and the ordering of the vertices is different.

We recall that Bfilt cannot be executed since it works only on $k$-partite graphs. Therefore, in order to solve the $k$-CP on $G_\ell$, we adapt the state-of-the-art maximum clique algorithms [39, 63] to exploit the knowledge of the original $k$-partition. In outline, the tailoring is based on the following two points: i) the initial lower and upper bounds on the clique number of the graph are set to $k - 1$ and $k$ respectively, i.e., the algorithm assumes a $k - 1$ clique exists and stops whenever a clique of size $k$ is found; ii) in every node of the branching tree, the maximum clique procedure executes the color filtering enhancement using the original $k$-partition with the original vertex order and in reverse direction (as described in the previous section). The aim is to reduce the subproblem graph and/or prune the node. In the following, we outline the main operations of the maximum clique procedure of Bfilt+, i.e., the branching scheme and the bounding procedure.

Branching scheme. For a given node of the branching tree, the subproblem graph $\tilde{G}_\ell = (\tilde{V}_\ell, \tilde{E}_\ell)$ is obtained in the same way as the one described in Sections 2.1 and 2.2. Precisely, this graph contains the common neighborhood of the vertices of the partial clique $\tilde{K}$ after the col-filtering phase is executed. The branching is in line with the classical scheme of MCP algorithms and it relies on partitioning the branching candidate set $\tilde{V}_\ell$ into two sets, denoted the Branching Set $B$ and, its complement, the Pruned Set $P := \tilde{V}_\ell \setminus B$. The largest Pruned Set $P$ (and the corresponding smallest Branching Set $B$) can be obtained by solving the following problem:

$$P := \arg \max_{P \subseteq \tilde{V}} \left\{|P| : (k - 1) - |\tilde{K}| \geq \omega(\tilde{G}_\ell[P])\right\} \quad \text{and} \quad B := \tilde{V}_\ell \setminus P$$

where $(k - 1)$ is the size of the largest clique (assumed) known at the start of the procedure. By construction, $P$ is the largest subset of $\tilde{V}_\ell$ which, by itself, cannot contain a clique of size $k$ (solution to the problem), so the branching can be restricted to the vertices of the Branching Set $B$. A branching tree node is pruned if the Branching set $B$ is empty. Since determining the largest set $P$ can be computationally difficult, the branching scheme determines $P$ heuristically using two different bounds on $\omega(\tilde{G}_\ell[P])$.

Bounding Procedure. A first upper bound on $\omega(\tilde{G}_\ell[P])$ is obtained by coloring $\tilde{G}_\ell$ with ISEQ, the greedy sequential independendent set heuristic which inspires FILT-ISEQ, see Section 2.2. Precisely, the set $P$ is composed of the set of vertices belonging to the first $(k - 1) - |\tilde{K}|$ independent sets computed by the (ISEQ) procedure. If the Branching Set $B$ is not empty, the procedure attempts to further reduce $B$ by employing an infrachromatic bounding procedure (a term first employed in [60]), which operates on the coloring of $\tilde{G}_\ell$. The outline of such procedure is to determine a collection $U$ of independent sets in the coloring such that no clique of size $|U|$ exists in the subgraph of $\tilde{G}_\ell$ induced by the vertices in $U$. Every time one such subset $U$ is found, the upper bound provided by the colouring is reduced by one unit. The
infra-chromatic bounding function used by \texttt{Bfilt+} is the MaxSAT-based bounding procedure of Li et al. [39], enhanced with the efficient bitstring data structures employed by \texttt{Bfilt+}.

### 3.4. Implementation details

We end this section commenting on relevant implementation details. \texttt{Bfilt+} uses bitstrings to encode the adjacency matrix in memory, as well as the sets of vertices $\hat{V}$ and $\tilde{V}$ at every node of the branch-and-filter tree. The encoding allows for a number of critical operations of the color-filtering, SAT-filtering and branching phases to be efficiently computed with bitmasks, and is inspired by recent efficient bitstring clique algorithms, see, e.g. [63, 66, 65]. It is also worth mentioning that \texttt{Bfilt} does not use dynamic variable/value branching heuristics, a typical feature of efficient constraint programming solvers. As explained in previous sections, the ordering of vertices is static and determined during the initialization phase. Vertices, i.e., variable-value pairs, are selected for branching lexicographically by layers in all the nodes.

### 4. Computing experience

In this section, we assess the computational performance of the new branch-and-filter algorithm \texttt{Bfilt+} presented in this work. The purpose of this computational study is threefold: \textit{i)} to evaluate the computational performance of \texttt{Bfilt+}, as well as the impact of its main components (see Sections 4.2 and 4.3); \textit{ii)} to compare \texttt{Bfilt+} against state-of-art algorithms available for the BCSP (see Section 4.4) and \textit{iii)} to assess the impact of the number of BCSP variables on the computing performance (see Section 4.5).

#### 4.1. Experimental setting and testbed of instances

All the experiments have been performed on a 20-core Intel(R) Xeon(R) CPU E5-2690 v2@3.00GHz, equipped with 128 GB of main memory and running a 64 bit Linux operating system. The source code was compiled with gcc 5.4.0 and the -o3 optimization flag. In all the tests, a time limit of 600 seconds was set for each run.

All the BCSP instances that we consider for benchmarking are publicly available online in a dedicated server [1], denoted \textit{XCSP3 server} in what follows. The instances are provided in the \textit{XCSP3} format, a recent XML-based format designed to represent constrained programming problems [8]. Our \textit{XCSP3} parser is able to read BCSP instances modelled by general constraints, both extensionally (the set of allowed/disallowed tuples is explicit) and intensionally (the set of allowed/disallowed tuples is represented by a function). We consider as testbed every BCSP instance in the \textit{XCSP3} server compatible with our parser, such that the number of vertices of the microstructure graph (number of BCSP variable values) is not greater than 8,000. The testbed comprises 1,895 instances of real and synthetic problems, which we group in 17 categories. Table 1 reports information concerning the BCSP instances, i.e., for each category, the average number of variables, compatible tuples and domain sizes. For each feature, we provide the minimum, the maximum and the average values. Table 2 reports the information of the associated microstructure graph, i.e., the number of vertices and edge density (in percentage). The entire testbed is available upon request to the authors.

The first 15 categories reported in the Tables 1 and 2, correspond each to a different class of problems described in the literature, i.e., a collection of families of instances originating from
the same problem or from the same parametrized generator. The terms category and class are used interchangeably for these 15 categories hereafter. The last two categories, supersolutions (ssol) and miscellaneous (misc), group different problem instances. Specifically, the ssol category contains extensions of different problem instances with the property that, if a variable-value pair in a solution is disallowed, the solution can be repaired by re-assigning a new variable-value pair, see [30] for a more detailed analysis. The misc category groups problem instances that contain less than 10 instances.

The imposed threshold of 8,000 vertices on the size of the microstructure graphs filters a small subset of the BCSP instances in the XCSP3 server compatible with our XCSP3 parser. The following 10 classes: B (50 instances), Bla (37 instances), comp (90 instances), D (700 instances), dflat (100 instances), ehi (200 instances), frb (80 instances), geom (100 instances), lat (100 instances) and RB2 (300 instances), are considered in full. On the other hand, the XCSP3 server provides 51 Haystack (hay) instances, of which we consider a subset of 16, 19 Knights (kni) instances, of which we consider 10, 18 instances of Queens-Knights (qk), of which we consider 12, 69 Rlfap (rlfap) instances, of which we consider 17 and 34 RoomMate (rm) instances, of which we consider 12. With respect to the ssol category, it contains 38 instances out of a possible 330. Finally, the individual families marc, lard and QueenAttacking, each containing 10 instances, are reduced to 6, 6 and 7 instances respectively and, as a consequence, fall under the misc category.

The testbed includes synthetic random models of varying difficulty (B, D, RB2 and frb) as well as quasi-random models (comp and geom). Other examples of synthetic instances are the finding-the-needle-in-a-haystack class hay, the board games classes qk and kni, the lattice classes lat and the SAT-based classes ehi and dflat. The testbed also contains instances derived from real problems, such as the radio link frequency assignment problem (rlfap). For the interested reader, a detailed description of the instances is provided in the Appendix 6.1.

4.2. Evaluation of the color-filtering and SAT-filtering phases of Bfilt+

In this section, we evaluate the computing impact of the main components of the Bfilt+ algorithm, i.e., the (enhanced) color-filtering and SAT-filtering phases. To establish the comparison, we consider a subset of 225 instances from our 1,895 instance testbed. The subset is composed of 25 instances from each one of the following 9 classes: B, Bla, comp, D, ehi, frb, geom, lat and RB2. A first set of results is reported in Table 3, considering four variants of Bfilt+. The first variant is our reference Bfilt+. The second variant, called Bfilt+ no COL filt., is without the color-filtering phase. The third variant, called Bfilt+ no SAT filt., is without the SAT-based filtering phase. Finally, the last variant, called Bfilt+ no SAT/no COL filt., executes neither of the two filtering phases. Each row of the table shows, for each one of four algorithmic variants, the number of instances solved to optimality (columns #opt) and the average CPU time spent by each algorithm (measured in seconds) on the corresponding class. In the averages we only consider instances solved within the time limit of 600 seconds.

According to Table 3, the proposed Bfilt+ algorithm outperforms the other variants, solving 209 out of the 225 instances tested. The impact of the color-filtering and SAT-based filtering phases by themselves is as follows: the no COL filt. variant solves 207 instances and the
Table 1: Features of the 17 categories of our BCSP instance testbed [1].

<table>
<thead>
<tr>
<th>categories</th>
<th>#</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
<th>number of variables</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
<th>number of tuples</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
<th>domain size</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>50</td>
<td>23</td>
<td>27</td>
<td>25.0</td>
<td>100,694</td>
<td>198,666</td>
<td>146,539.4</td>
<td>23.0</td>
<td>27.0</td>
<td>25.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bla</td>
<td>37</td>
<td>64</td>
<td>208</td>
<td>117.2</td>
<td>211,379</td>
<td>26,291,636</td>
<td>6,176,596.8</td>
<td>10.5</td>
<td>35.3</td>
<td>19.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comp</td>
<td>90</td>
<td>33</td>
<td>105</td>
<td>63.2</td>
<td>47,970</td>
<td>531,000</td>
<td>226,943.3</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>700</td>
<td>40</td>
<td>40</td>
<td>40.0</td>
<td>44,944</td>
<td>22,823,756</td>
<td>4,161,833.6</td>
<td>8.0</td>
<td>180.0</td>
<td>51.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dflat</td>
<td>100</td>
<td>2237</td>
<td>2237</td>
<td>2237.0</td>
<td>28,100,632</td>
<td>28,102,060</td>
<td>28,101,380.1</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ehi</td>
<td>200</td>
<td>297</td>
<td>315</td>
<td>306.0</td>
<td>2,051,275</td>
<td>2,314,937</td>
<td>2,183,080.6</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frb</td>
<td>80</td>
<td>30</td>
<td>59</td>
<td>46.0</td>
<td>83,083</td>
<td>1,049,583</td>
<td>509,821.2</td>
<td>15.0</td>
<td>26.0</td>
<td>21.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>geom</td>
<td>100</td>
<td>50</td>
<td>50</td>
<td>50.0</td>
<td>434,373</td>
<td>456,163</td>
<td>447,848.9</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hay</td>
<td>16</td>
<td>16</td>
<td>361</td>
<td>153.5</td>
<td>1,770</td>
<td>23,389,245</td>
<td>4,727,964.3</td>
<td>4.0</td>
<td>19.0</td>
<td>11.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kni</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>6.6</td>
<td>21,840</td>
<td>10,569,744</td>
<td>2,006,599.6</td>
<td>64.0</td>
<td>625.0</td>
<td>295.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lat</td>
<td>100</td>
<td>100</td>
<td>625</td>
<td>337.5</td>
<td>181,827</td>
<td>24,074,218</td>
<td>7,812,842.7</td>
<td>6.1</td>
<td>11.1</td>
<td>8.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qk</td>
<td>12</td>
<td>13</td>
<td>20</td>
<td>20.0</td>
<td>43,288</td>
<td>4,095,405</td>
<td>1,111,847.2</td>
<td>29.5</td>
<td>125.0</td>
<td>68.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB2</td>
<td>300</td>
<td>30</td>
<td>50</td>
<td>40.0</td>
<td>83,330</td>
<td>581,244</td>
<td>303,739.8</td>
<td>15.0</td>
<td>23.0</td>
<td>19.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlfap</td>
<td>17</td>
<td>28</td>
<td>400</td>
<td>136.0</td>
<td>455,586</td>
<td>29,776,814</td>
<td>10,703,797.8</td>
<td>17.4</td>
<td>44.0</td>
<td>35.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rm</td>
<td>12</td>
<td>4</td>
<td>100</td>
<td>23.4</td>
<td>24,192,034</td>
<td>24,192,034</td>
<td>1,844,187.9</td>
<td>3.0</td>
<td>69.1</td>
<td>18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssol</td>
<td>38</td>
<td>8</td>
<td>100</td>
<td>33.1</td>
<td>276</td>
<td>26,159,948</td>
<td>12,121,350.9</td>
<td>4.0</td>
<td>246.0</td>
<td>148.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td>33</td>
<td>10</td>
<td>650</td>
<td>147.8</td>
<td>1,225</td>
<td>20,593,125</td>
<td>6,185,383.7</td>
<td>1.0</td>
<td>89.0</td>
<td>43.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grand total</td>
<td>1895</td>
<td>4</td>
<td>2237</td>
<td>205.9</td>
<td>24</td>
<td>29,776,814</td>
<td>4,407,482.7</td>
<td>1.0</td>
<td>625.0</td>
<td>33.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Features of the microstructure graphs $G(I)$ of the 17 categories in our BCSP instance testbed.

<table>
<thead>
<tr>
<th>categories</th>
<th>#</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
<th>number of vertices</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
<th>edge density (%)</th>
<th>min</th>
<th>max</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>50</td>
<td>529</td>
<td>729</td>
<td>627.0</td>
<td>72.1</td>
<td>74.9</td>
<td>73.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bla</td>
<td>37</td>
<td>674</td>
<td>7334</td>
<td>2705.9</td>
<td>93.2</td>
<td>97.8</td>
<td>95.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comp</td>
<td>90</td>
<td>330</td>
<td>1050</td>
<td>632.2</td>
<td>88.4</td>
<td>96.4</td>
<td>92.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>700</td>
<td>320</td>
<td>7200</td>
<td>2057.1</td>
<td>86.2</td>
<td>88.6</td>
<td>87.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dflat</td>
<td>100</td>
<td>7511</td>
<td>7511</td>
<td>7511.0</td>
<td>99.6</td>
<td>99.6</td>
<td>99.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ehi</td>
<td>200</td>
<td>2079</td>
<td>2205</td>
<td>2142.0</td>
<td>95.0</td>
<td>95.3</td>
<td>95.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frb</td>
<td>80</td>
<td>450</td>
<td>1534</td>
<td>1013.3</td>
<td>82.2</td>
<td>89.3</td>
<td>86.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>geom</td>
<td>100</td>
<td>1000</td>
<td>1000</td>
<td>1000.0</td>
<td>87.0</td>
<td>91.3</td>
<td>89.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hay</td>
<td>16</td>
<td>64</td>
<td>6859</td>
<td>2254.0</td>
<td>87.8</td>
<td>99.4</td>
<td>97.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kni</td>
<td>10</td>
<td>320</td>
<td>5625</td>
<td>2033.6</td>
<td>40.4</td>
<td>67.2</td>
<td>51.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lat</td>
<td>100</td>
<td>613</td>
<td>6961</td>
<td>3242.4</td>
<td>96.9</td>
<td>99.4</td>
<td>98.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qk</td>
<td>12</td>
<td>384</td>
<td>3750</td>
<td>1558.0</td>
<td>58.2</td>
<td>59.3</td>
<td>58.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB2</td>
<td>300</td>
<td>450</td>
<td>1150</td>
<td>786.7</td>
<td>82.4</td>
<td>88.0</td>
<td>85.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlfap</td>
<td>17</td>
<td>1232</td>
<td>7820</td>
<td>3844.0</td>
<td>60.1</td>
<td>99.1</td>
<td>82.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rm</td>
<td>12</td>
<td>12</td>
<td>6910</td>
<td>974.7</td>
<td>36.4</td>
<td>81.4</td>
<td>59.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssol</td>
<td>38</td>
<td>32</td>
<td>7872</td>
<td>4858.4</td>
<td>55.6</td>
<td>95.6</td>
<td>84.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td>33</td>
<td>50</td>
<td>7832</td>
<td>3690.1</td>
<td>48.2</td>
<td>100.0</td>
<td>76.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grand total</td>
<td>1895</td>
<td>12</td>
<td>7872</td>
<td>2114.9</td>
<td>36.4</td>
<td>100.0</td>
<td>88.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no SAT filt. variant solves 205. In contrast, the impact of removing both components is very significant, i.e., the last variant solves 184 instances, 25 less than Bfilt+. As far as the
computing time is concerned, Bfilt+ is also on average the fastest one.

Table 3: Computing impact of the color-filtering and SAT-filtering phases of Bfilt+.

| categories | # | Bfilt+ | | Bfilt+ | | Bfilt+ | | Bfilt+ |
|------------|---|-------|---|-------|---|-------|---|
|             |   | #opt  | time [s] | no COL filt. | | #opt  | time [s] | no SAT filt. | | #opt  | time [s] | no SAT/no COL filt. |
| B          | 25 | 25    | 38.1 | 25 | 37.7 | 25 | 71.6 | 22 | 108.0 |
| B1a        | 25 | 25    | 7.9  | 25 | 7.9  | 25 | 8.0  | 25 | 7.5  |
| comp       | 25 | 25    | 0.2  | 25 | 0.2  | 25 | 0.2  | 25 | 0.2  |
| D          | 25 | 25    | 23.4 | 24 | 31.5 | 25 | 29.7 | 24 | 37.8 |
| ehi        | 25 | 25    | 5.5  | 25 | 5.5  | 25 | 5.5  | 23 | 3.7  |
| frb        | 25 | 19    | 29.3 | 19 | 43.3 | 19 | 75.6 | 14 | 63.2 |
| geom       | 25 | 25    | 0.5  | 25 | 0.5  | 25 | 1.2  | 25 | 2.7  |
| lat        | 25 | 18    | 34.6 | 18 | 35.0 | 15 | 95.1 | 10 | 10.3 |
| RB2        | 25 | 22    | 39.2 | 21 | 21.9 | 21 | 54.5 | 16 | 23.5 |
| Grand total| 225 | 209   | 18.8 | 207 | 19.1 | 205 | 33.7 | 184 | 27.1 |

Table 4: Percentage of instances in which the additional features of Bfilt+ are used.

| categories | # | Bfilt+ | | Bfilt→G(I) | | Bfilt→G(\ell) | | max clique proc.→G_\ell |
|------------|---|-------|---|----------|---|----------|---|
| B          | 50 | 0.0   | 100.0 | 0.0 |
| B1a        | 37 | 0.0   | 0.0   | 100.0 |
| comp       | 90 | 0.0   | 1.1   | 98.9 |
| D          | 700 | 0.1  | 87.6  | 12.3 |
| dflat      | 100 | 100.0 | 0.0   | 0.0 |
| ehi        | 200 | 23.5  | 0.0   | 76.5 |
| frb        | 80  | 0.0   | 95.0  | 5.0 |
| geom       | 100 | 0.0   | 1.0   | 0.0 |
| hay        | 16  | 0.0   | 100.0 | 0.0 |
| kni        | 10  | 0.0   | 100.0 | 0.0 |
| lat        | 100 | 0.0   | 59.0  | 41.0 |
| qk         | 12  | 0.0   | 100.0 | 0.0 |
| RB2        | 300 | 0.0   | 97.0  | 3.0 |
| rlfap      | 17  | 0.0   | 35.3  | 64.7 |
| rm         | 12  | 25.0  | 8.3   | 66.7 |
| ssol       | 38  | 7.9   | 0.0   | 92.1 |
| misc       | 33  | 21.2  | 33.3  | 45.5 |
| Grand total| 1,895 | 8.5 | 65.8 | 25.7 |

4.3. Evaluation of the re-partitioning and maximum clique improvements of Bfilt+
In this section, we evaluate two additional algorithmic improvements of Bfilt, i.e., the re-partition procedure computed during initialization, see Section 3.1, and the maximum clique procedure, see Section 3.3.

Table 4 reports, for the 17 categories of our testbed, the percentage of instances in which the aforementioned features are employed. Specifically, the column Bfilt→G(I) refers to the execution of the procedure Bfilt on the microstructure graph G(I), the column Bfilt→G(\ell) refers to the execution of Bfilt considering the new partition of k layers provided
Table 5: Evaluation of the additional improvements of Bfilt+. The algorithmic variant Bfilt → G(\mathcal{I}) does not repartition the layers of the microstructure graph G(\mathcal{I}). The last two columns report the computing performance of the state-of-the-art MCP solvers MoMC [38] and BBMCX [63].

<table>
<thead>
<tr>
<th>categories</th>
<th>#</th>
<th>Bfilt+ #opt</th>
<th>time (s)</th>
<th>Bfilt → G(\mathcal{I}) #opt</th>
<th>time (s)</th>
<th>MoMC [38] #opt</th>
<th>time (s)</th>
<th>BBMCX [63] #opt</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>25</td>
<td>25</td>
<td>38.1</td>
<td>25</td>
<td>32.4</td>
<td>23</td>
<td>104.7</td>
<td>23</td>
<td>109.4</td>
</tr>
<tr>
<td>B1a</td>
<td>25</td>
<td>25</td>
<td>7.9</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>comp</td>
<td>25</td>
<td>25</td>
<td>0.2</td>
<td>25</td>
<td>0.1</td>
<td>22</td>
<td>1.0</td>
<td>22</td>
<td>16.8</td>
</tr>
<tr>
<td>D</td>
<td>25</td>
<td>25</td>
<td>23.4</td>
<td>18</td>
<td>19.9</td>
<td>22</td>
<td>36.0</td>
<td>22</td>
<td>31.6</td>
</tr>
<tr>
<td>ehi</td>
<td>25</td>
<td>25</td>
<td>5.5</td>
<td>25</td>
<td>0.8</td>
<td>24</td>
<td>350.6</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>frb</td>
<td>25</td>
<td>19</td>
<td>29.3</td>
<td>12</td>
<td>99.1</td>
<td>16</td>
<td>74.6</td>
<td>12</td>
<td>99.3</td>
</tr>
<tr>
<td>geom</td>
<td>25</td>
<td>25</td>
<td>0.5</td>
<td>24</td>
<td>17.0</td>
<td>25</td>
<td>4.9</td>
<td>25</td>
<td>21.1</td>
</tr>
<tr>
<td>lat</td>
<td>25</td>
<td>18</td>
<td>34.6</td>
<td>18</td>
<td>14.8</td>
<td>8</td>
<td>104.8</td>
<td>8</td>
<td>5.9</td>
</tr>
<tr>
<td>RB2</td>
<td>25</td>
<td>22</td>
<td>39.2</td>
<td>16</td>
<td>52.1</td>
<td>21</td>
<td>65.8</td>
<td>16</td>
<td>62.7</td>
</tr>
<tr>
<td>Grand total</td>
<td>225</td>
<td>209</td>
<td>18.8</td>
<td>163</td>
<td>23.8</td>
<td>161</td>
<td>94.2</td>
<td>128</td>
<td>49.6</td>
</tr>
</tbody>
</table>

by the re-partitioning procedure, and the column max clique proc. → \mathcal{G}_\ell corresponds to the execution of the maximum clique procedure considering a partition of \ell > k layers. According to the table, in more than 91% of the instances the re-partition procedure is able to compute a useful new partition, and in \approx 66% of the cases the new partition is of size k. We recall that the latter case corresponds to a reformulation of the original BBCSP instance which, to the best of our knowledge, has not been reported in the literature. The fact that this reformulation is used by Bfilt+ in the majority of cases is a further contribution of this work.

To measure the computing impact of the re-partitioning procedure, we test the Bfilt → G(\mathcal{I}) algorithmic variant, which is executed on the original layer partition. To measure the impact of the maximum clique procedure, we also compare with the state-of-the-art MCP solvers: MoMC [38], and BBMCX [63]. Table 5 reports the number of instances solved and the average CPU time spent by the four algorithms, i.e., Bfilt+, Bfilt → G(\mathcal{I}), MoMC and BBMCX, for each one of the 9 instance classes of our 225 instance subset testbed. According to the table, Bfilt+ clearly outperforms both MCP solvers, determining the satisfiability of 209 of the instances, while Bfilt → G(\mathcal{I}) and MoMC are only able to solve 163 and 161 respectively, and BBMCX can only solve 128. It is worth mentioning that only Bfilt+ was able to solve the instances from the Black Hole class, while all the other algorithms were unable to solve any instance. The reported results provide empirical evidence of the efficiency of the combined repartitioning and maximum clique procedures employed by Bfilt+.

### 4.4. Comparison of Bfilt+ against state-of-the-art BCSP solvers

In this section, we compare the new algorithm Bfilt+ with 5 publicly available state-of-the-art solvers for the entire testbed of instances. All the solvers provide a parser for the XCSP3 format. Two of the 5 solvers are SAT-based, i.e., they reduce the BCSP to CNF-SAT and call a state-of-the-art SAT solver to determine satisfiability. The other 3 solvers are classical constraint programming solvers, which we denote CP-based in the remainder.
• PicatSAT 2.8 (Picat) \cite{70}: A SAT-based solver that uses the Picat Prolog-like rule-based language. In the tests, we employ the version which performed best in the main track—CSP, sequential—of the (last held) XCSP3 2019 competition\cite{6}.

• sCOP[order + MapleCOMSPS] (sCOP): A SAT-based constraint programming solver written in the Scala language. In the tests we use the latest publicly available version\cite{7}, which came first in the standard track—CSP, sequential—of the XCSP3 2018 competition. The term order in the name refers to the type of encoding to SAT, while the term MapleCOMSPS refers to the state-of-the-art award-winning SAT solver from the SAT 2016 competition\cite{8}.

• Choco-solver 4.10.4 (choco)\cite{9}: Choco-solver is a CP-based solver written in Java. To the best of our knowledge, we are using in the tests an improved version than the one which took part in the XCSP3 2019 competition.

• Concrete 3.12.3 (Concrete)\cite{10}: Concrete is a CP-based solver written in Scala. In the tests, we use the version that took part in the XCSP3 2019 competition.

• Mistral 2.0 (Mistral)\cite{11}: Mistral is a CP-based solver written in C++. In the tests we use the release that came fourth in the main track—CSP, sequential—of the XCSP3 2018 competition.

Table 6 reports the computing performance of the 5 solvers, together with Bfilt+, over our 1,895 instance testbed. For each one of the algorithms and the 17 categories in the testbed, the table provides the number of instances solved (#opt), the average CPU time (avg.) spent on the instances solved and the standard deviation (std. dev.).

According to the table, Bfilt+ is the algorithm which performs best, proving satisfiability in 1,783 instances out of the possible 1,895. The SAT-based solvers perform second best, sCOP solving 1,697 instances, and Picat three instances less in similar time. From the group of CP-based solvers, choco is the algorithm which solves the largest number of instances (precisely 1,662) and spends less CPU time. Mistral and Concrete are outperformed by the other four algorithms, the latter determining satisfiability in 430 cases less than Bfilt+, and spending around double the time. Concerning individual classes, Bfilt+ is very effective in B, Bla, frb, RB2, as well as in the two categories ssol and misc, where it solves more instances than any other algorithm. To take one example, Bfilt+ manages to prove satisfiability for the 50 instances of B, while choco, the second best solver for the class, is only able to solve 21. Also worth noting is the case of the (hard) frb series, generated from the model RB (see the Appendix Section 6.1), in which Bfilt+ shows speed-ups of around 3 \times with respect to the rest of competitors. Bfilt+ also performs very effectively in geom, where it is orders

\footnote{\url{shorturl.at/bfj1D}}
\footnote{\url{http://xcsp.org/competition}}
\footnote{\url{https://tsoh.org/sCOP/}}
\footnote{\url{http://www.satcompetition.org/}}
\footnote{\url{https://choco-solver.org/}}
\footnote{\url{https://github.com/concrete-cp/concrete}}
\footnote{\url{https://github.com/ehebrard/Mistral-2.0}}
of magnitude faster than the other algorithms. A possible explanation for the successful performance of Bfilt+ lies in its re-partitioning phase, since, as shown in Table 4, Bfilt+ always uses the new partition over the families B, Bla, frb, RB2 and geom.

In contrast, Bfilt+ performs poorly in the class dflat with respect to all the other algorithms, solving 94 instances out of a possible 100 and spending around 20 times more time than Picat and around 40 times more time than choco. A possible explanation might be the specific topology of the corresponding microstructure graphs. According to our tests, all the instances of dflat are of the satisfiable type, and dispose of a huge number of variables (precisely 2,237) with small average domain sizes of less than 4 values in most cases. It would seem that the Bfilt+ competitors are able to find a feasible assignment of values to the variables quickly by means of good variable selection heuristics, whereas Bfilt+ uses a static variable selection heuristic and is better oriented towards filtering than towards finding “good” assignments. Interestingly, the dflat series is the only class in the testbed where Bfilt+ always operates with the original partition. With respect to the other categories, both SAT-based solvers outperform Bfilt+ in the lat and hay classes. In the former, Bfilt+ is unable to solve the largest family of the lat class with 625 variables. Concerning the class hay, Bfilt+ appears to be unable to crack its carefully crafted structure, and scales poorly.

For further clarification, we also report in Figure 6 the performance profile of the 6 algorithms. The graphical representation is constructed as follows. We compute the normalized time $\tau$ as the ratio of the computing time of each algorithm (which is $\infty$ if the instance is not solved to optimality) over the minimum computing time taken for all the algorithm we tested. For each value of $\tau$ on the horizontal axis, the vertical axis reports the percentage of instances for which the corresponding algorithm spent at most $\tau$ times the computing time of the fastest algorithm. The interpretation of the chart at both ends of the horizontal axis is in this way. At $\tau = 1$, the value of the curves is equal to the percentage of instances in which the corresponding algorithm is the fastest one. At the right-end, i.e., the largest value of $\tau$, each curve corresponds to the percentage of instances solved by the specific algorithm. In the performance profile, the best performance is achieved by the algorithms whose curves appear higher in the chart. In the case of Figure 6, this corresponds to the algorithm Bfilt+, which is fastest in more than 50% of the instances (left-end of the figure), and also solves the largest amount, i.e., slightly over 94%, as shown by its curve in the right-end. The SAT-based solvers initially solve less instances than the CP-based solvers choco and mistral, but they gradually overtake them in the more difficult instances. Specifically, Picat and sCOP are solving more than 80% of the instances, slightly less than choco, within two orders of magnitude of the best algorithm ($\tau = 100$), while Mistral, only solves slightly above 70%. Within the time limit, however, Picat and sCOP prove the satisfiability of slightly over 89% of the instances, while choco solves slightly over 87% and Mistral around 85%. Concerning the CP-based algorithms, choco is the best performing one, being the fastest in around 10% of the instances.

We end the section by showing in Figure 7 the computing time boxplots of the 6 algorithms. In the figure, we graphically show the time (in logarithmic scale) spent by each algorithm through their quartiles; the lines extending vertically from the boxes indicate the variability outside the upper and lower quartiles. Above the upper quartile, the outliers are plotted as individual points. Figure 7 further evidences the superior computing times of Bfilt+, and is
consistent with the other results reported in the section.

Table 6: Computational comparison of Bfilt+ against 2 SAT-based and 3 constraint programming solvers over the instance testbed.

<table>
<thead>
<tr>
<th>categories</th>
<th>#opt</th>
<th>CPU time (s)</th>
<th>#opt</th>
<th>CPU time (s)</th>
<th>#opt</th>
<th>CPU time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>avg.</td>
<td>std. dev.</td>
<td></td>
<td>avg.</td>
<td>std. dev.</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>40.6</td>
<td>54.9</td>
<td>8</td>
<td>312.4</td>
<td>187.0</td>
</tr>
<tr>
<td>Bla</td>
<td>37</td>
<td>9.3</td>
<td>18.8</td>
<td>30</td>
<td>234.0</td>
<td>169.4</td>
</tr>
<tr>
<td>comp</td>
<td>90</td>
<td>0.2</td>
<td>0.2</td>
<td>90</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>D</td>
<td>695</td>
<td>24.4</td>
<td>64.7</td>
<td>691</td>
<td>57.5</td>
<td>74.9</td>
</tr>
<tr>
<td>dflat</td>
<td>94</td>
<td>160.1</td>
<td>114.7</td>
<td>100</td>
<td>10.0</td>
<td>5.8</td>
</tr>
<tr>
<td>ehi</td>
<td>200</td>
<td>4.4</td>
<td>2.5</td>
<td>200</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>frb</td>
<td>52</td>
<td>23.2</td>
<td>44.7</td>
<td>37</td>
<td>82.3</td>
<td>149.8</td>
</tr>
<tr>
<td>geom</td>
<td>100</td>
<td>0.6</td>
<td>1.2</td>
<td>100</td>
<td>19.1</td>
<td>41.9</td>
</tr>
<tr>
<td>hay</td>
<td>4</td>
<td>58.8</td>
<td>116.3</td>
<td>11</td>
<td>65.4</td>
<td>166.4</td>
</tr>
<tr>
<td>kni</td>
<td>10</td>
<td>2.4</td>
<td>2.1</td>
<td>10</td>
<td>7.1</td>
<td>9.4</td>
</tr>
<tr>
<td>lat</td>
<td>70</td>
<td>32.8</td>
<td>98.6</td>
<td>100</td>
<td>2.8</td>
<td>4.6</td>
</tr>
<tr>
<td>qk</td>
<td>8</td>
<td>3.0</td>
<td>2.7</td>
<td>12</td>
<td>6.0</td>
<td>7.6</td>
</tr>
<tr>
<td>RB2</td>
<td>278</td>
<td>46.3</td>
<td>109.6</td>
<td>211</td>
<td>80.7</td>
<td>122.5</td>
</tr>
<tr>
<td>rlfap</td>
<td>15</td>
<td>20.0</td>
<td>50.6</td>
<td>17</td>
<td>8.6</td>
<td>7.0</td>
</tr>
<tr>
<td>rm</td>
<td>12</td>
<td>1.6</td>
<td>5.1</td>
<td>12</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>ssol</td>
<td>36</td>
<td>55.7</td>
<td>58.9</td>
<td>34</td>
<td>3.1</td>
<td>4.9</td>
</tr>
<tr>
<td>misc</td>
<td>32</td>
<td>22.2</td>
<td>86.7</td>
<td>31</td>
<td>41.6</td>
<td>49.2</td>
</tr>
<tr>
<td>Grand total</td>
<td>1,783</td>
<td>30.9</td>
<td>78.1</td>
<td>1,694</td>
<td>44.3</td>
<td>86.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>categories</th>
<th>#opt</th>
<th>CPU time (s)</th>
<th>#opt</th>
<th>CPU time (s)</th>
<th>#opt</th>
<th>CPU time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>avg.</td>
<td>std. dev.</td>
<td></td>
<td>avg.</td>
<td>std. dev.</td>
</tr>
<tr>
<td>Mistral</td>
<td></td>
<td></td>
<td></td>
<td>Concrete</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>16</td>
<td>265.7</td>
<td>174.1</td>
<td>3</td>
<td>111.0</td>
<td>130.3</td>
</tr>
<tr>
<td>Bla</td>
<td>10</td>
<td>0.4</td>
<td>0.0</td>
<td>10</td>
<td>14.9</td>
<td>0.7</td>
</tr>
<tr>
<td>comp</td>
<td>90</td>
<td>0.1</td>
<td>0.0</td>
<td>90</td>
<td>3.0</td>
<td>0.2</td>
</tr>
<tr>
<td>D</td>
<td>700</td>
<td>35.8</td>
<td>45.4</td>
<td>531</td>
<td>119.4</td>
<td>144.8</td>
</tr>
<tr>
<td>dflat</td>
<td>100</td>
<td>10.2</td>
<td>12.5</td>
<td>99</td>
<td>74.8</td>
<td>83.3</td>
</tr>
<tr>
<td>ehi</td>
<td>200</td>
<td>0.3</td>
<td>0.0</td>
<td>200</td>
<td>4.8</td>
<td>1.0</td>
</tr>
<tr>
<td>frb</td>
<td>34</td>
<td>73.7</td>
<td>153.2</td>
<td>25</td>
<td>101.9</td>
<td>139.7</td>
</tr>
<tr>
<td>geom</td>
<td>100</td>
<td>27.8</td>
<td>82.7</td>
<td>89</td>
<td>13.8</td>
<td>50.2</td>
</tr>
<tr>
<td>hay</td>
<td>2</td>
<td>0.9</td>
<td>1.2</td>
<td>2</td>
<td>15.0</td>
<td>17.8</td>
</tr>
<tr>
<td>kni</td>
<td>6</td>
<td>70.9</td>
<td>93.5</td>
<td>10</td>
<td>3.2</td>
<td>0.9</td>
</tr>
<tr>
<td>lat</td>
<td>71</td>
<td>26.5</td>
<td>91.8</td>
<td>69</td>
<td>40.1</td>
<td>83.5</td>
</tr>
<tr>
<td>qk</td>
<td>11</td>
<td>106.2</td>
<td>151.6</td>
<td>12</td>
<td>3.6</td>
<td>1.1</td>
</tr>
<tr>
<td>RB2</td>
<td>184</td>
<td>88.3</td>
<td>141.8</td>
<td>121</td>
<td>71.0</td>
<td>124.6</td>
</tr>
<tr>
<td>rlfap</td>
<td>17</td>
<td>0.4</td>
<td>0.3</td>
<td>17</td>
<td>6.0</td>
<td>2.9</td>
</tr>
<tr>
<td>rm</td>
<td>12</td>
<td>0.7</td>
<td>1.5</td>
<td>12</td>
<td>4.5</td>
<td>5.3</td>
</tr>
<tr>
<td>ssol</td>
<td>34</td>
<td>1.7</td>
<td>5.1</td>
<td>34</td>
<td>26.1</td>
<td>47.7</td>
</tr>
<tr>
<td>misc</td>
<td>29</td>
<td>35.2</td>
<td>83.7</td>
<td>29</td>
<td>51.3</td>
<td>115.8</td>
</tr>
<tr>
<td>Grand total</td>
<td>1,616</td>
<td>35.0</td>
<td>79.1</td>
<td>1,353</td>
<td>66.7</td>
<td>117.1</td>
</tr>
</tbody>
</table>
4.5. Impact of the number of variables on the computing performance

We end the section with an analysis of the impact of the number of variables of the instances on the computing performance. As a general rule, the larger the number of variables, the harder the instances become. However, some algorithms are better suited to specific classes, and may scale much better in those cases. These considerations can be observed in Figure 8, which compares Bfilt+ against the best performing solver over the instance classes frb, B and lat. Specifically, the figure shows boxplots of the computing times of Bfilt+ and sCOP over the frb class (figures (a) and (b)), boxplots of the computing times of Bfilt+ and choco over the B class (figures (c) and (d)) and, finally, boxplots of the computing times of Bfilt+ and Picat over the lat class (figures (e) and (f)).

In all cases, the boxplots show that the families become harder as the number of variables increase. With respect to the frb and B classes, Bfilt+ scales better than the other tested solvers. For example, Bfilt+ solves to optimality all the frb instances with 50 or less variables (except 2 instances with 50 variables), while sCOP can solve very few large instances (only 2 with 50 variables). In contrast, Bfilt+ scales worse than Picat in the lat class. Specifically, it cannot solve any of the instances with 625 variables, whereas Picat determines satisfiability in all of them.

5. Conclusions and future work

In this work, we present a new efficient algorithm for the BCSP based on a reduction of the problem to the k-CP on the underlying microstructure graph. The new exact algorithm,
Figure 7: Computing time boxplots of the 6 tested algorithms for the BCSP. The y-axis shows in logarithmic scale the CPU times in seconds. On the top part of the figure, we report the total number of instances (#OPT) solved by each of the algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>0.01</th>
<th>0.10</th>
<th>1.00</th>
<th>10.00</th>
<th>100.00</th>
<th>#OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bfilt+</td>
<td>1783</td>
<td>1697</td>
<td>1694</td>
<td>1662</td>
<td>1616</td>
<td>1353</td>
</tr>
<tr>
<td>sCOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Picat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>choco</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mistral</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

denoted Bfilt+, is inspired by the recent efficient techniques of state-of-the-art clique solvers. Its excellent computational performance is achieved thanks to several unique features that exploit the specific topology of the $k$-partite microstructure graph. Specifically, we propose two filtering phases: the first filtering phase, denoted color-filtering, is based on coloring the microstructure graph, while the second one, denoted SAT-filtering, is based on an associated SAT-problem, which is solved heuristically. Complementary to these phases, we have also proposed several algorithmic enhancements, the major one based on repartitioning the microstructure graph. Extensive tests, carried out over a benchmark of almost two thousands instances, computationally show that Bfilt+ significantly outperforms 5 general purpose Constraint Programming solvers, all of which have participated in recent challenges.

An interesting future line of research is to extend Bfilt+ for the case when the microstructure graph is very large or massive but sparse. Precisely, it would be interesting to enhance Bfilt+ with the recent techniques employed by clique solvers for massive sparse graphs, such as, e.g., [62, 31]. Finally, another interesting and very challenging line of research concerns the extension of Bfilt+ for general CSPs. As mentioned in the introduction, different binarization strategies have been proposed in the literature to solve CSPs of greater arity than two as a BCSP, see [20, 53, 67]. A future line of work is the characterization of those non-binary constraints and binarization methods for which a clique based algorithm would be efficient,
Figure 8: Computing performance, comparing Bfilt+ against the best solver for three classes of BSCP instances: in parts (a) and (b) we compare Bfilt+ against sCOP for the frb class; in parts (c) and (d) we compare Bfilt+ against choco for the B class; in parts (e) and (f) we compare Bfilt+ against Picat for the lat class. Results are grouped according to the number of variables of the families of instances that make up each class (x-axis). The y-axis shows CPU times measured in seconds.

(a) Bfilt+ → frb family

(b) sCOP → frb family

(c) Bfilt+ → B family

(d) choco → B family

(e) Bfilt+ → lat family

(f) Picat → lat family

and the extension of Bfilt+ for these non-binary problems.
We conclude the paper by pointing out the relation that the algorithm BFilt+ establishes between recent clique-solving color-based techniques and constraint propagation techniques, which pivots in the microstructure graph. An analysis of this connection from a theoretical perspective is out of the scope of this work, but should be the study of future research. We conjecture, for example, that the level of consistency reached by the color-filtering phase is directional arc consistency, a consistency level below AC described in detail in, e.g., [54], owing to the sequential nature of the procedure FIL-SEQ. Another conjecture is that the repeated application of FIL-SEQ, alternating between a lexicographical and a reverse lexicographical selection of vertices until a fixed point is reached, de facto propagates AC. Attempting to improve existing AC algorithms based on FIL-SEQ would also, in our opinion, deserve further consideration, taking into account the efficiency shown by BFilt+.

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).

References


6. Appendix

6.1. Detailed description of the classes of BCSP instances.

In what follows, we briefly describe the different categories that make up our testbed.

- Random models (B, D, RB2, frb), see, e.g., [24]: A constraint graph $G$ can be associated to a BCSP problem, in which the vertices represent variables and the edges represent constraints between the variables. Basic standard random CSP models described in the literature for benchmarking are labelled from letter A to D. They are parametrized by the tuple $\langle k, d, p_1, p_2 \rangle$, where $k$ is the number of variables, $d$ the uniform domain size, $p_1$ is a measure of the density of $G$ and $p_2$ is a measure of the tightness of the constraints. The generator for model B selects exactly $p_2 \times d^2$ inconsistent tuples for each edge, while the model D generator selects each one of the possible $d^2$ incompatible tuples with probability $p_2$. Besides the families B and D, our testbed contains the families frb and RB2 which derive from a revised model of B, denoted RB [69], which produces harder instances than the ones obtained from the original model; see [4] for the motivation behind the revision of the model. Incidentally, the frb family is also typically employed for benchmarking MCP solvers.

- Black Hole (Bla) [26]: This instance class derives from the Black Hole solitaire played with 52 cards. The goal of the game is to place all cards from three different piles into the Black Hole pile (BH), which initially holds a single card. The rules of the game allow cards to move from the three piles, and into the BH, if they are adjacent to the card in the BH according to their numbering.

- Quasi-random problems (comp, geom): The testbed contains two crafted classes of quasi-random instances. These are typically built using a pure random kernel to which a number of auxiliary fragments are added. The family comp is made up of 90 instances of this type, see [36]. The geom class of 100 instances was created by Rick Wallace and derives from a geometric problem. The edges of the constraint graph are prefixed as follows: i) a geometric distance value $\text{dist}$ (less than $\sqrt{2}$) is chosen randomly and ii) for each BCSP variable, a point inside a unit cube is mapped randomly and an edge is added to the constraint graph if the two points of the corresponding variables lie at a distance less or equal than $\text{dist}$. Once the constraint graph is computed, the constraint relations are determined using a random kernel.

- SAT-based problems (ehi, dflat): The ehi class derives from an encoding to CSP of two 100 problem series of 3-SAT unsatisfiable instances, i.e., ehi-85 and ehi-90, see [36]. The SAT-flat-dual (dflat) class derives from an encoding to CSP of the 3-coloring problem over a set of 3-colourable graphs (specifically, the flat200-479 series), see [17] for a description of this class of graphs. It is worth noting that the instances are compiled from a prior encoding of the problem to SAT. All the instances are satisfiable.

- Quasigroup problems (lat) [50, 5]: The name of the class refers to the fact that a quasigroup of order $m$ is also a latin square of size $m$, i.e., an $m$ by $m$ square matrix in which each element occurs exactly once in every row and column. Quasigroup problems are representative of structured random problems that are closer to real-life problems. The lat class contains 60 instances that derive from the quasigroup completion problem.
and another 40 instances from the \textit{quasigroup with holes problem}, for a total of 100 instances.

- Queens-Knights (\texttt{qk}): The Queens-Knights problem asks for placing on a chessboard of size $n \times n$, $q$ queens and $k$ knights such that no two queens attack each other and all knights form a cycle (when considering knight moves) [6]. The class contains two types, denoted \texttt{add} and \texttt{mul}. In the type \texttt{add}, a square of the chessboard can be shared by both a queen and a knight. In the type \texttt{mul}, this is not allowed.

- Knights (\texttt{kni}) [6]: A variant of the $n$-queens problem which considers knight constraints instead of queen constraints, see, e.g., [6].

- Haystack (\texttt{hay}): This class of unsatisfiable instances was created by Marc van Dongen \footnote{http://research.ucc.ie/profiles/D005/dongen}, and has been included in several CSP challenges, see, e.g., [35]. The instances are parameterized by their size $n$, with $n \times n$ variables each with $n$ domains. The constraint graph consists of $n$ clusters, a central one and $n - 1$ satellites, and each cluster is an $n$-clique. The outer clusters are connected to the central cluster by a single edge (constraint). The problems are designed in such a way that any value assignment to the variables in the center cluster has a corresponding associated outer cluster which is inconsistent. This cluster is called the haystack.

- Roommates (\texttt{rm}) [51]: This set of instances derives from the \textit{stable roommates problem} (SRP). In the SRP, a set of participants initially rank each other. The problem calls to find a \textit{stable matching}, i.e., a matching such that no two participants prefer each other to their matched partners.

- Radio link frequency assignment (\texttt{rlfap}) [11]: The problem of radio frequency assignment is to provide communication radio channels from a limited set, while minimizing the interferences suffered by those wishing to communicate. The \texttt{rlfap} class was initially derived from data from real networks and is composed of different series.

- Supersolution problems (\texttt{ssol}): As mentioned at the beginning of the section, this category encompasses instances that have been built by converting an original BCSP problem into a more constrained one with a reduced, and more \textit{robust}, solution set. The original problems considered include some scheduling and $N$-queens instances.

- Miscellaneous (\texttt{misc}): As mentioned at the beginning of the section, this category includes series which have less than 10 instances. The category holds, amongst others, the \texttt{marc} and \texttt{lard} synthetic families designed by Marc van Dongen, the \textit{queenAttacking} suite derived from the \textit{Queen Attacking problem} and the real-world suite \texttt{driverlogw} from the logistics domain, see, e.g., [7], for a more detailed description.

\subsection*{6.2. Impact of the number of variables on the computational performance – detailed analysis}

In this appendix, we include additional computing time boxplots of the 6 constraint programming algorithms considered in this work. Each one of the following 3 figures reports the results of the algorithm over one tested instance class, see Table 1 for the full list. Results for
each class are grouped according to families with the same number of variables (x-axis in all the figures). The y-axis shows CPU times measured in seconds.

Figure 9: Computational performance of the 6 exact algorithms for the BCSP over the B class of instances. The x-axis refers to the number of variables. The y-axis shows CPU times measured in seconds.
Figure 10: Computational performance of the 6 exact algorithms for the BCSP over the frb class of instances. The x-axis refers to the number of variables. The y-axis shows CPU computing times measured in seconds.
Figure 11: Computational performance of the 6 exact algorithms for the BCSP over the 1at class of instances. The x-axis refers to the number of variables. The y-axis shows CPU times measured in seconds.