European Journal of Operational Research 299 (2022) 448–467

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

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

Pablo San Segundo[a,*], Fabio Furini[b], Rafael León[c]

[a] Universidad Politécnica de Madrid (UPM), Centre for Automation and Robotics (CAR), Madrid, Spain
[b] Department of Computer, Control and Management Engineering Antonio Ruberti, Sapienza University of Rome, Rome, Italy
[c] Researcher, Madrid, Spain

ARTICLE INFO

Article history:
Received 29 December 2020
Accepted 9 September 2021
Available online 16 September 2021

Keywords:
Combinatorial optimization
Binary constraint satisfaction problems
Constraint programming
Exact algorithm
Computational experiments

ABSTRACT

A binary constraint satisfaction problem (BCSP) consists in determining an assignment of values to variables that 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 colouring 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 that produces an often easier reformulation to solve. We carry out an extensive computational campaign on a benchmark of almost 2000 instances, encompassing numerous real and synthetic problems from the literature. The performance of the new algorithm is compared against four 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.

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., Brailsford, Potts, Smith, & Oper (1999) and part II of Rossi, Beek, & Walsh (2006) 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 the 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 the variables that 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.

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

* Corresponding author.
E-mail addresses: pablo.sansegundo@upm.es (P. San Segundo), fabio.furini@uniroma1.it (F. Furini), rleon@ucjc.edu, rafael.leon@coit.es (R. León).

https://doi.org/10.1016/j.ejor.2021.09.014

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 reduction called *binarization*. This reduction can be performed in different ways, and we refer the interested reader to Rossi, Petrie, & Dhar (1990); Samaras & Stergiou (2005); Stergiou & Walsh (1999). 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 in practice. Accordingly, even though the BCSP is equivalent to the CSP, in practice there can be large differences in algorithms designed specifically for one or the other. In this article we mainly focus on the BCSP and on exact algorithms to solve it.

Many combinatorial problems can be recast as BCSPs. We provide some examples from different contexts in what follows. One example of a real-world application of the BCSP is the decision version of the Train Platforming Problem (TPP), see e.g., Caprara, Galli, & Toth (2011). 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 that 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., Murphey, Pardalos, & Resende, 1999), 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 that reduces the problem to the *k-clique problem* (k-CLP) on a k partite graph. This graph is known as the *microstructure* of the BCSP, see Jégou (1993), where this transformation was 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-CLP 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 $\mathcal{NP}$-complete. The reduction from the k-CLP (one of Karp's 21 $\mathcal{NP}$-complete problems Karp, 1972) to the k-CLP 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. The polynomial time transformation based on the microstructure graph can also be used to reduce the k-CLP to the BCSP, showing that the latter problem is also $\mathcal{NP}$-complete.

To the best of our knowledge, the state-of-the-art exact algorithms for the BCSP are the general purpose solvers for CSPs 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-CLP 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., Grünert, Irnich, Zimmermann, Schneider, & Wulfhorst (2002); Mirghorbani & Krokhmal (2013) 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. Finally, it is worth pointing out that efficient algorithms for the decision version of the maximum clique problem (k-CLQ) are based on tailoring maximum clique algorithms, such as Li, Fang, Jiang, & Xu (2018a); San Segundo, Coniglio, Furini, & Ljubić (2019a); San Segundo, Furini, & Artieda (2019b); San Segundo, Lopez, & Pardalos (2016b); San Segundo, Nikolaev, & Batsyn (2015); San Segundo, Nikolaev, Batsyn, & Pardalos (2016).

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

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

$$V = \bigcup_{i=1,2,\ldots,k} V_i \quad \text{where} \quad V_i = \Big\{ (x_i, a) : \quad a \in D(x_i) \Big\}.$$

In other words, the vertex set $V = \{V_1, V_2, \ldots, V_k\}$ of $G(\mathcal{I})$ is partitioned into k subsets which we denote the *layers* of the graph, each one corresponding to a variable of the original BCSP (in the literature, the layers are also called the *parts* of a k-partite graph). 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 $G(\mathcal{I})$ 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:

$$E = \Big\{ \big((x_i, a), \ (x_j, b)\big) : \quad 1 \leq i < j \leq k, \quad (a, b) \in R\big(c(i, j)\big) \Big\}.$$

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-CLP and the set of BCSP feasible solutions. Accordingly, unsatisfiability of the BCSP implies the non-existence of a feasible solution to the k-CLP, 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 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: $x_i^* = f\big(v(i)\big)$ ($i = 1, 2, \ldots, k$), where the function $f\big(v(i)\big)$ returns the value from the domain $D(x_i)$ of the variable x_i associated to the vertex $v(i)$.

Example 1 In Fig. 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 $R(c_6) = \{(5, 1), (6, 2), (6, 3)\}$. Fig. 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. We report the corresponding variable-value pairs above the vertices. The edges represent compatible pairs of variable-values according to the constraints. The vertices depicted in red

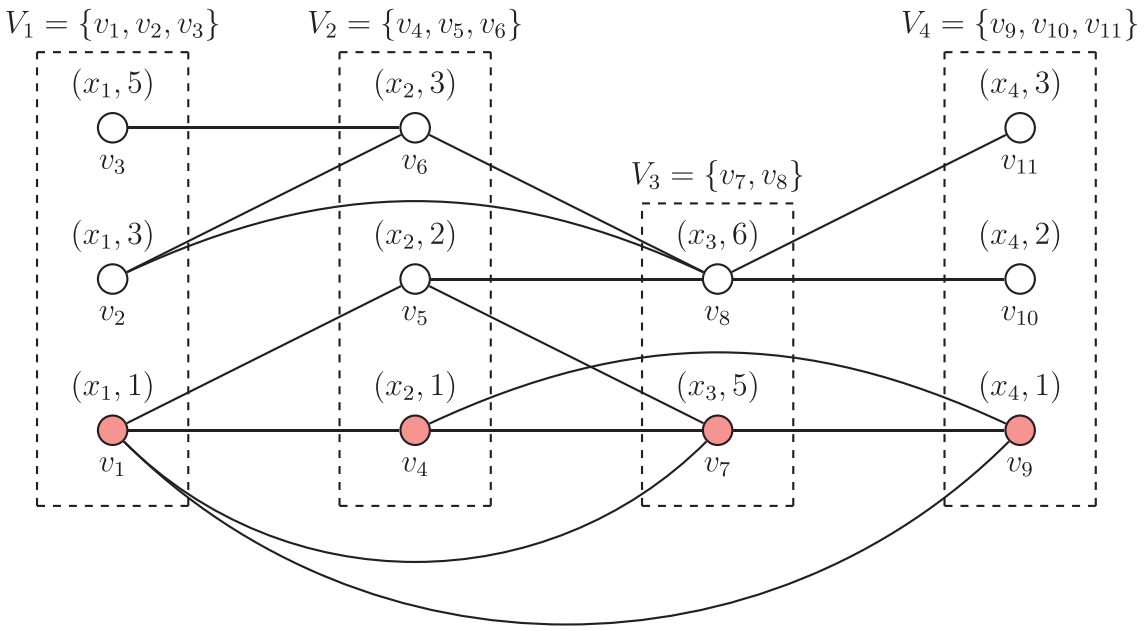

Fig. 1. A BCSP instance $\mathcal{I}$ with $k = 4$ variables, $X = \{x_1, x_2, x_3, x_4\}$ and $q = 6$ constraints. The picture shows the microstucture graph $G(\mathcal{I})$ obtained by the reduction from the BCSP to the k-CLP. In red we report a clique K^* of size 4 given by the vertices $\{v_1, v_4, v_7, v_9\}$, which corresponds to the feasible solution $x_1^* = 1$, $x_2^* = 1$, $x_3^* = 5$ and $x_4^* = 1$. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

correspond to a clique of size 4, i.e., a feasible solution to the k-CLP. This solution, given by the clique $K^* = \{v_1, v_4, v_7, v_9\}$, maps to the feasible BCSP solution: $x_1^* = 1$, $x_2^* = 1$, $x_3^* = 5$ and $x_4^* = 1$.

1.2. Literature review on CSP exact algorithms and solvers

Literature on CP exact algorithms used to solve CSPs has been very prolific since its inception in the 60s. It includes surveys, e.g., Brailsford et al. (1999); Buscemi & Montanari (2008), and books, e.g., Dechter (2003); Marriott & Stuckey (1998); Rossi et al. (2006); Tsang (1993). The main engine of CP-based exact algorithms relies on *backtracking* (branching) interleaved with *constraint propagation* techniques. In a basic backtracking algorithm, each node of the branching tree 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., Rossi et al., 2006).

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, 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*. The seminal 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) (Montanari, 1974), but higher consistency levels exist, such as *path-consistency* (Mackworth, 1977) and *k-consistency* (Freuder, 1982). Depending on the specific problem constraints, CP-based algorithms can also achieve weaker forms of consistency than AC, such as *forward checking* (Golomb & Baumert, 1965; Haralick & Elliott, 1980), *directional arc-consistency* (Dechter & Pearl, 1988) and *unit propagation* (Davis & Putnam, 1960). We refer the reader to the chapters 3 and 4 in Rossi et al. (2006) for a detailed description on this subject.

In the last two decades, much of the research in this area has been devoted to the study of global constraints (non-binary CSPs) together with the development of efficient specialized propagators. Some examples of different types of global constraints in the literature are *all-different* (Régin, 1994), *channeling*, *at-most*, *at-least*, *counting*, *lexicographic*, *sorting*, *table*, the enumeration being by no means conclusive. We refer the reader to the Chapter 7 of Rossi et al. (2006) for an interesting overview on the topic. Additionally, the MiniZinc catalogue (https://www.minizinc.org/doc-2.5.3/en/lib-globals.html) contains an extensive list of available global constraints for modelling in the MiniZinc format. A detailed review on n-ary constraints and propagators goes beyond the scope of this paper.

Classical constraint propagators originally conceived for the BCSP have also 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., Samaras & Stergiou (2005).

Concerning the microstructure graph representation of the BCSP, another recent ongoing stream of (theoretical) work has been concerned with the generalization of the microstructure graph to non-binary CSPs, see, e.g., Cohen (2003); Mouelhi, Jégou, & Terrioux (2014). 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 graph, see Cooper, Jeavons, & Salamon (2010); Dechter & Pearl (1989); Jégou & Terrioux (2015); Mouelhi et al. (2014); Naanaa (2020).

We end this literature review with two comments on existing solvers for the CSP. Firstly, 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. Secondly, 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 solver – CP-SAT (henceforth `OR-tools`), the lazy clause solver `Chuffed` as well as the solvers `PicatSAT` (henceforth `Picat`) and `sCOP`. Examples of purely CP-based solvers are `Gecode` (GECODE, 2016), `Minion` (Gent, Jefferson, & Miguel, 2006), ILOG – CP `Optimizer` (IBM, 2017), `choco`, `Mistral` and `Concrete`. The list is by no means exhaustive.

A recent improvement employed by efficient state-of-the-art solvers such as, e.g., `Chuffed`, is the *lazy clause generation*, a technique first described in Ohrimenko, Stuckey, & Codish (2009). It relies on a hybrid approach to CP that combines features of finite constraint propagation and Boolean satisfiability. In the details, constraint propagators are replaced by clause generators that record the reason for the propagator step as new clauses (typically denoted *nogoods*). Generating these clauses eagerly at the beginning of the search is typically impractical, specially in the case of variables with large domains. For this reason they are generated dynamically, and then exploited with efficient SAT techniques.

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 solvers for the CSP (see the computational results in 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-CLP 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 colour-filtering, is based on colouring 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 that 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-CLP, 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 solvers for the CSP for a number of families of BCSP instances. In details, the algorithm significantly outperforms three general purpose CP-based solvers and four general purpose SAT-based solvers. We tested 1895 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.

An additional contribution of this work is to establish a link between classical constraint propagator algorithms and the clique-based filtering phases employed by `Bfilt+`. Specifically, we show in Section 2.2 that the level of consistency reached by the colour-filtering phase is directional arc consistency, a consistency level below arc consistency. We also relate the level of consistency achieved by the SAT-filtering phase of `Bfilt+` with singleton consistency, see Section 2.3.

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-partionining 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-CLP on the microstructure graph $G(\mathcal{I})$. The proposed B&F algorithm is inspired by the recent state-of-the-art algorithms for the *maximum-clique problem* (MCLP) and its variants, see, e.g., Coniglio, Furini, & San Segundo (2020); Li et al. (2018a); Li, Liu, Jiang, Manyá, & Li (2018b); San Segundo et al. (2019b, 2016b); San Segundo, Matia, Rodriguez-Losada, & Hernando (2013); San Segundo et al. (2015); San Segundo, Rodríguez-Losada, & Jiménez (2011); San Segundo et al. (2016); San Segundo & Tapia (2014). This family of combinatorial exact algorithms are based on the n-ary branching scheme proposed in Carraghan & Pardalos (1990), 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 `Bfilt` our new exact B&F algorithm. The initial letter 'B' in the name refers to the fact that `Bfilt` makes extensive use of *bitstrings* and efficient *bitmasking* operations during the execution of the algorithm. In the following sections we describe `Bfilt` and its main components.

Before launching `Bfilt`, a *pre-filtering* procedure is carried out to reduce the microstructure graph $G(\mathcal{I}) = (V, E)$ by filtering those vertices that cannot be part of any k-clique. This procedure removes the vertices that 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 that have no support in the variable x_i. The pre-filtering procedure is effective in reducing the microstructure graph before the branching phase of `Bfilt` is executed. Consider for example the microstructure graph of Fig. 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 Bfilt

In this section we describe the way the B&F tree of Bfilt is generated. A branching node is created by selecting a layer, and one of its vertices, and by adding it to the clique $\hat{K}$ constructed by the branching operations. 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 remaining vertices of the branching layer are removed. Formally, the vertices and edges of $\hat{G}$ are:

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

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:

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

where $i(j)$ is the index of the original layer in the microstructure graph of the layer j in the subproblem graph. Precisely, the vertex set $\hat{V}_j$ is composed of the vertices of the original layer $i(j)$ that 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 α-partite graph. At the root node of the B&F tree, $\hat{G}$ corresponds to the original microstructure graph $G(\mathcal{I})$ and $\hat{K} = \emptyset$.

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.

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 Fig. 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 colour-filtering phase of Bfilt

In this section we describe the colour-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 colour-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 $\hat{V}_j$ ($j \in \{1, 2, \ldots, \alpha\}$) 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 a clique of size α. 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 (colours). It does so by sequentially examining the vertices following a predefined order and builds one independent set at a time, see San Segundo et al. (2011), where it is used inside an MCLP algorithm. We denote FILT-ISEQ the modified version of ISEQ employed by Bfilt, which is explained in the following paragraph.

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 $\mathscr{P}(\hat{V})$, one at a time. Additionally, and during execution, FILT-ISEQ stores a label $l(I)$ for every independent set $I \in \mathscr{P}(\hat{V})$. 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 every 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 $\hat{G}$. In other words, the vertex v can be filtered since it is not connected to any of the vertices in I and thus cannot make part of any α-clique, i.e., the value associated to v is not supported by any value of the original layer $l(I)$. 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 $\mathscr{P}(\hat{V})$. 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 FILT-ISEQ manages to partition the remaining unfiltered vertices of $\hat{G}$ in less than α independent sets (colours), the corresponding branching node can be fathomed. This fact is summarized in the following observation.

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

If the node of the B&F tree is not fathomed after the execution of FILT-ISEQ, we keep track of the reduced $\hat{G}$ and denote it

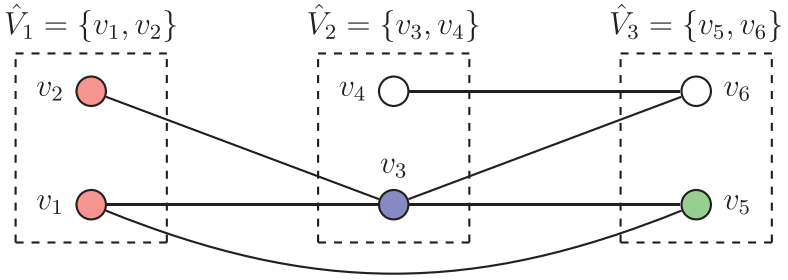

Fig. 2. The subproblem graph $\hat{G}$ used in Example 2. The colorued vertices reflect the colouring obtained by FILT−ISEQ on the reduced subproblem graph $\tilde{G} = \hat{G}[\hat{V} \setminus \{v_4, v_6\}]$.

$\tilde{G} = (\tilde{V}, E[\tilde{V}])$. Furthermore, we set its layers $\tilde{V}$ to each one of the independent sets (colours) in the collection $\mathscr{P}(\hat{V})$. Subsequently, $\tilde{G}$ becomes the input graph of the ensuing SAT-based filtering phase. It is worth noting that, at the end of its execution, FILT−ISEQ computes a colouring of the reduced graph $\tilde{G}$. A colouring corresponds to an assignment of colours to the vertices of a graph in such a way that two adjacent vertices receive different colours and it can also be seen as a partition of the vertex set into independent sets.

Example 2 In Fig. 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 $G(\mathcal{I})$ 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 colour-filtering phase. At the start of the procedure, FILT−ISEQ sets B to $\hat{V}$ and the collection of independent sets $\mathscr{P}(\hat{V})$ is the empty set. In the first iteration, 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. 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 $\mathscr{P}(\hat{V})$, and the filtered vertices of $\hat{G}$ are $\{v_4\}$ and $\{v_6\}$. In the last two iterations, 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 $\tilde{G}$ has the set of vertices $\tilde{V} = \{v_1, v_2, v_3, v_5\}$ corresponding to the coloured vertices in the figure. Finally, we note that $\mathscr{P}(\hat{V}) = \{I_1, I_2, I_3\}$ is an independent set partition of $\tilde{V}$ (also a colouring of $\tilde{G}$), and its independent sets are each one of the layers of $\tilde{G}$.

The subproblem graph $\hat{G}$ is, *de facto*, a microstructure graph and its associated BCSP has α variables and $\beta = \frac{\alpha\,(\alpha-1)}{2}$ constraints (one for each pair of variables). Each variable i, with $i \in \{1, \ldots, \alpha\}$, has a domain size equal to $|\hat{V}_i|$, i.e., the number of vertices in the layer associated to the variable i. The colour-filering phase can be seen as a constraint propagator procedure for the aforementioned BCSP and, therefore, it is relevant to analyse the domain consistency level it achieves. For this purpose we introduce the following definitions related to *arc consistency* (Mackworth, 1977).

Two variables are said to be *arc consistent*, if each value of both variable domains is compatible with at least one value of the domain of the other variable. A BCSP is called arc consistent if each pair of variables is arc consistent. A pair of variables (i, j), with $i, j \in \{1, \ldots, \alpha\}$, is said to be *directional arc consistent* (Dechter & Pearl, 1988), if each value of the domain of the variable j is compatible with at least one value of the domain of the variable i. For a given ordering of the variables, a BCSP is said to be directional arc consistent if every pair of variables respecting the ordering is directional arc consistent. The same notions can also be applied to the layers of a microstructure graph: *i*) two layers are arc consistent if each vertex of both layers is connected to at least one vertex of the other layer; *ii*) a pair (i, j) of layers, with $i, j \in \{1, \ldots, \alpha\}$, is directional arc consistent if every vertex of the layer j is connected to at least one vertex of the layer i.

Starting from the microstructure graph $\hat{G}$, the colour-filtering phase generates a new microstructure graph $\tilde{G}$ that is associated to a directional arc consistent BCSP with respect to the lexicographical order of the variables. Therefore, the level of consistency achieved by the colour-filtering phase is directional arc consistency (DAC). This fact is summarized in the following observation, expressed in terms of the microstructure graph $\tilde{G}$:

Observation 4. The BCSP problem associated to the microstructure graph $\tilde{G}$ is directional arc consistent, i.e., every pair (i, j) of layers, with $i, j \in \{1, \ldots, \alpha\}$ and $i < j$, is directional arc consistent.

This is due to the fact that, after colouring a layer $i \in \{1, \ldots, \alpha\}$, each vertex in the layer $j > i$ that is not connected to at least one vertex in the layer i is filtered. From a theoretical perspective, it is worth noting that DAC is a lower level of consistency compared to classical (full) arc consistency (AC). The worst-case time complexity of the best procedure from the literature that imposes DAC in the BCSP associated to $\hat{G}$ is $O(\beta \cdot \Phi^2)$, see, e.g, chapter 3 of Rossi et al. (2006), where Φ is $\max_{i=1,\ldots,\alpha}\{|\hat{V}_i|\}$, i.e., the maximum size of the variable domains. The worst-case time complexity of the colour-filtering phase is $O(|\hat{V}|^2)$, since the colouring heuristic examines, for each vertex, all those preceding it in lexicographical order. This is the same complexity as enforcing DAC, since $|\hat{V}| \leq \alpha \cdot \Phi$.

For the example of Fig. 2, the BCSP problem associated to the new microstructure graph $\tilde{G} = \hat{G}[\hat{V} \setminus \{v_4, v_6\}]$ determined by FILT−ISEQ, is directional arc consistent with respect to the lexicographical order of the layers. However, it is not (fully) arc consistent since the pair of variables associated to the layers $\hat{V}_1$ and $\hat{V}_3$, i.e., variables 1 and 3, are not arc consistent as the vertex $v_2 \in \hat{V}_1$ is not connected to any vertex in $\hat{V}_3$.

We conclude this section by briefly mentioning that the colour-filtering phase can be extended to enforce AC on the BCSP associated to $\tilde{G}$. This can be done by repeating the execution of FILT−ISEQ according to increasing and decreasing order of the vertices until the graph $\tilde{G}$ remains unchanged. At the end of each execution, the graph $\tilde{G}$ becomes the new input graph $\hat{G}$ for the next iteration, if some of the vertices are filtered. When this procedure stops, the BCSP associated to $\tilde{G}$ is arc consistent. Preliminary extensive tests showed that the computational overhead of achieving AC in this manner is not compensated by the increased filtering capabilities. This is due to the fact that, in the tested instances, most of the vertices are filtered during the first execution of FILT−ISEQ.

2.3. The SAT-filtering phase of Bfilt

After the colour-filtering phase is over and before the branching, Bfilt executes the *SAT-filtering phase*. This phase works on the reduced graph $\tilde{G}$ and it has the same two goals of the colour-filtering phase: *i*) fathoming a node in case the procedure manages to prove that a clique of size α does not exist in $\tilde{G}$, and *ii*) filtering individual vertices that cannot be part of any clique of size α.

The SAT-filtering phase exploits a reduction of the α-CLP to the *boolean satisfiability* (SAT) problem, where α 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., $\vee$, $\wedge$, $\neg$). 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 α-CLP is defined by a CNF boolean formula with two types of clauses: *non-edge clauses* and *layer clauses*. It calls for an interpretation of the variables, an assignment of values to each one of the variables, that satisfies all its clauses. The complement graph of a graph G is denoted $\overline{G} = (V, \overline{E})$ and $\overline{E} = \{(u, v) : (u, v) \notin E\}$. The non-edge clauses, associated to the non-adjacent pairs of vertices in $\tilde{G}$, contain only negative literals and they read as follows:

$$h_{uv} \equiv \left(\bar{y}_u \vee \bar{y}_v \right) \qquad (u, v) \in \overline{E}[\tilde{V}]. \tag{1}$$

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

$$s_j \equiv \left(y_{v(1)} \vee y_{v(2)} \vee \ldots \vee y_{v(g)} \right) \qquad j = 1, 2, \ldots, \alpha, \tag{2}$$

where, for each layer j, the function $v(l)$ returns the vertex in the microstructure graph of the lth vertex in the layer j, and $g = |\tilde{V}_j|$. We denote $S(\tilde{G})$ the SAT problem obtained by the reduction of the α-CLP in $\tilde{G}$. Solving to optimality $S(\tilde{G})$ by using a SAT solver may be very time consuming, due to the $\mathcal{NP}$-completeness of the problem. For this reason, we design the following procedure with the aim of either heuristically detecting the unsatisfiability of $S(\tilde{G})$ or filtering some of the vertices of $\tilde{G}$.

The subproblem graph $\tilde{G}$ contains a clique of size α 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 α. Accordingly, the following observation provides a fathoming condition of a branching node:

Observation 5. Given a subproblem graph $\tilde{G}$, if the associated SAT problem $S(\tilde{G})$ is unsatisfiable, the corresponding branching node can be fathomed.

Solving $S(\tilde{G})$ in every node of the B&F tree can be computationally challenging. In an attempt to efficiently prove only unsatisfiability, 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 MCLP algorithms (see e.g., Li & Quan, 2010). We refer the interested reader to e.g., Davis & Putnam (1960), where these techniques have been proposed. We adapt these two procedures for the specific requirements of $S(\tilde{G})$. 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 $S(\tilde{G})$; 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 (1). 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 $S(\tilde{G})$ is unsatisfiable, and the node of the tree is fathomed according to Observation 5. Once the UP procedure terminates, all variables set to 0 during the execution are filtered from the graph $\tilde{G}$, since they cannot make part of any clique of size α in $\tilde{G}$. This filtering condition is summarized as follows.

Observation 6. After the execution of the UP procedure, the vertices associated to the boolean variables set to 0 can be deleted from the graph $\tilde{G}$.

Example 3 We report in this paragraph the operations of UP according to the example graph $\tilde{G}$ of Fig. 3. This graph has the following $\alpha = 3$ layers: $\tilde{V}_1 = \{v_1, v_2\}$, $\tilde{V}_2 = \{v_3, v_4, v_5\}$ and $\tilde{V}_3 = \{v_6\}$. The edges of the graph are also reported in the figure. The SAT problem $S(\tilde{G})$ encoding $\tilde{G}$ is composed of the following clauses. Its non-edge clauses (1) are:

$$h_{v_1v_2} \equiv (\bar{y}_{v_1} \vee \bar{y}_{v_2}), \quad h_{v_1v_4} \equiv (\bar{y}_{v_1} \vee \bar{y}_{v_4}), \quad h_{v_1v_5} \equiv (\bar{y}_{v_1} \vee \bar{y}_{v_5}),$$

$$h_{v_2v_6} \equiv (\bar{y}_{v_2} \vee \bar{y}_{v_6}), \quad h_{v_3v_4} \equiv (\bar{y}_{v_3} \vee \bar{y}_{v_4}), \quad h_{v_3v_5} \equiv (\bar{y}_{v_3} \vee \bar{y}_{v_5}),$$

$$h_{v_3v_6} \equiv (\bar{y}_{v_3} \vee \bar{y}_{v_6}), \quad h_{v_4v_5} \equiv (\bar{y}_{v_4} \vee \bar{y}_{v_5}), \quad h_{v_5v_6} \equiv (\bar{y}_{v_5} \vee \bar{y}_{v_6}).$$

and its layer clauses (2) are:

$$s_1 \equiv (y_{v_1} \vee y_{v_2}), \quad s_2 \equiv (y_{v_3} \vee y_{v_4} \vee y_{v_5}), \quad s_3 \equiv (y_{v_6}).$$

In the example, the layer clause $\tilde{V}_3$ corresponding to layer 3 is a unit clause, so UP sets y_{v_6} to 1. It follows from the non-edge clauses containing 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 $S(\tilde{G})$ is unsatisfiable. According to Observation 5, the node is fathomed. In Fig. 3, the vertices that 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 the assignment is propagated according to UP, a layer clause with every variable set to 0 is attained. If this is the case, the variable y_v 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 Bfilt when FL is restricted to layer clauses with two literals. The filtering operations of FL are summarized by the following observation.

Observation 7. After the execution of the FL procedure, the vertices that generate an empty layer clause via UP can be deleted from the graph $\tilde{G}$.

The SAT-filtering phase of Bfilt interleaves FL with UP. Initially, procedure 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 UP, are processed. If the node of the B&F tree is not fathomed after the termination of UP, Bfilt then calls procedure 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 UP is then called upon

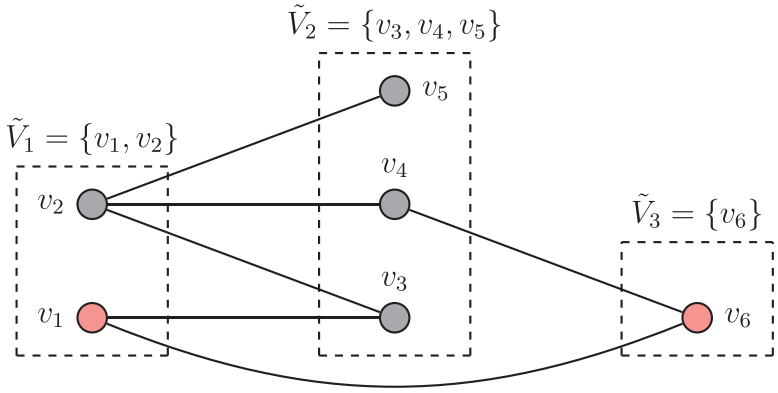

Fig. 3. The subproblem graph $\tilde{G}$ of Example 3. Red vertices are associated to boolean variables set to 1 by UP. Vertices in grey are filtered from $\tilde{G}$ since they are associated to boolean variables set to 0 by UP.

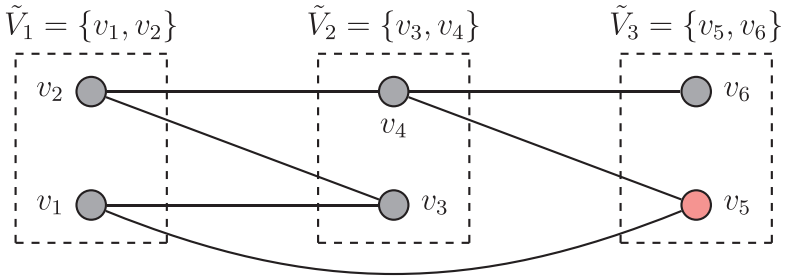

Fig. 4. The subproblem graph $\tilde{G}$ of Example 4. The vertex coloured in red corresponds to a failed literal. In grey the vertices of the boolean variables that participated in the reasoning of the SAT-filtering phase of `Bfilt`. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

the latter. The procedure continues until either $S(\tilde{G})$ remains unchanged after the execution of FL or unsatisfiabiliy of $S(\tilde{G})$ can be proved.

Example 4 We show the operations carried out by `Bfilt` during the SAT-filtering phase, explaining in detail the operations carried out by FL. In Fig. 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 UP terminates immediately. However, the layer clause s_3 contains two literals so 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 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. UP is then called for this unit clause, and, as a result, it sets each variable of the layer clause s_1 to 0, since v_6 is not adjacent to any variable in s_1. According to Observation 5 the node is fathomed. In Fig. 4, the vertex coloured in red is associated to the boolean variable of the failed literal and we depict in grey the vertices that took part in the reasoning.

The SAT-filtering phase, as the colour-filtering phase, can be seen as a constraint propagator procedure for the BCSP associated to the microstructure graph $\tilde{G}$, and so it can be analysed in terms of the domain consistency level it achieves. Specifically, the level propagated by UP is below AC. This can be seen by the fact that the encoding of the subproblem α-CLP to SAT in this phase uses the *direct encoding*, i.e., one literal per vertex (also variable value in the associated BCSP); and classical unit propagation in this setting is known to be weaker than AC, see, e.g. [68]. We recall that the procedure FL attempts to heuristically filter vertices by first setting their corresponding boolean variable to 1 and then executing UP on the resulting subproblem to detect unsatisfiability. Therefore, the consistency level FL achieves corresponds to (a subset of) singleton UP consistency (SUP), which is above AC. FL does not achieve full SUP because it is applied selectively to those layers in $\tilde{G}$ which have two vertices. The consistency level of the (full) SAT-filtering phase is determined by FL.

With respect to complexity, UP runs in worst-case time $O(|\tilde{V}|^2)$, which is comparable to the complexity of classical unit propagation in SAT solvers, while FL runs in $O(|\tilde{V}|^3)$, since it fixes individual vertices and then executes UP in the remaining subproblem.

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

In this section, we present the additional algorithmic improvements of `Bfilt` designed to 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, if the size of the new partition is larger than k, but within a given gap (see next section), `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 (MCLP), see, e.g., San Segundo, Lopez, Batsyn, Nikolaev, & Pardalos (2016a) or Section 3 of Li, Jiang, & Manyà (2017). According to San Segundo et al. (2016a), the *colour-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 Li, Fang, & Xu (2013); San Segundo et al. (2016a), which computes a colour-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. Since the instances of our benchmark are quite dense (see Section 4), the maximum independent set problems solved during the SORT procedure are expected to be easy. From a computational perspective, maximum independent sets are typically easy on dense graphs, as observed in several papers, see e.g., Li et al. (2017); San Segundo et al. (2016), also in the context of interdiction problems, see, e.g., Furini, Ljubić, Martin, & San Segundo (2019); Furini, Ljubić, San Segundo, & Zhao (2021). This is due to several factors, the two main ones are: *i*) dense graphs contain small independent sets and, for this reason, it is easy to find good-quality heuristics solutions; *ii*) in dense graphs, the upper bounds used to reduce the branching tree are expected to be tight.

The computed independent sets by the SORT procedure become the new layers of the microstructure graph. In order to effectively reduce the size of the branching tree of Bfilt+, vertices are ordered by SORT according to the following two criteria: *i*) the layers (independent sets) of the new partition are sorted by non-decreasing size and *ii*) inside each layer, vertices are sorted by 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 that are more likely to be part of a k-clique.

To compute each maximum independent set, Bfilt+ executes the state-of-the-art MCLP algorithm (San Segundo et al., 2016) on the complement of the microstructure graph. In the case when finding maximum independent sets is computationally challenging, the SORT procedure computes maximal independent sets (using the same algorithm) within a time limit of 0.1 seconds. It is worth mentioning that Bfilt+ solves an optimisation problem (maximum or maximal independent set) in the preprocessing of a satisfaction problem. This operation is not performed very often since optimisation problems are in general harder than satisfaction problems. Surprisingly, in our case the optimization-based repartitioning improves the overall computational performance of Bfilt+. As explained above, this is possible since the maximum (maximal) independent set problem is typically easy on dense instances, whereas the maximum clique problem is expected to be difficult.

Thanks to extensive preliminary tests, we noticed that the SORT procedure is not always able to find a 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 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 $\mathcal{G}(\mathcal{I}) = (\mathcal{V}, \mathcal{E})$. 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 $\mathcal{G}(\mathcal{I})$ are the same as $G(\mathcal{I})$, only the layers differ. By construction, each layer $j = 1, 2, \ldots, k$ of $\mathcal{G}(\mathcal{I})$ is a maximum (or maximal) independent set in the subgraph $\mathcal{G}(\mathcal{I})[\cup_{i=1}^{j} \mathcal{V}_i]$. In case a new partition of size k is found, Bfilt+ executes Bfilt on $\mathcal{G}(\mathcal{I})$. If, instead, the SORT procedure fails and the new partition is of size strictly greater than $k + 10$ then Bfilt+ executes 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 Bfilt+ executes the maximum clique procedure described in Section 3.3.

Example 5 We describe in this paragraph the operations executed by the SORT procedure to compute the colour-sort ordering for the microstructure graph of Fig. 1. For this graph, the procedure is able to find a new partition of size $k = 4$. Fig. 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: $\texttt{deg}(V) = \{4, 2, 1, 3, 3, 3, 4, 5, 3, 1, 1\}$, e.g., $\deg(v_1) = 4$, $\deg(v_2) = 2$. 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 colour-sort ordering is: $v_4 \prec v_1 \prec v_6 \prec v_8 \prec v_7 \prec v_9 \prec v_5 \prec v_2 \prec v_{11} \prec v_{10} \prec v_3$, 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 Fig. 5 in different colours. For each layer in the figure, the vertices at the bottom come first in the ordering.

It is worth mentioning that the repartitioning procedure does not exploit specific a priori knowledge about the structure of the constraints. Consequently, we design the repartitioning relying only on maximum (maximal) independent sets. This repartitioning strategy is shown to be computationally effective, see Section 3.1. However, different repartitions, i.e., different reformulations of the BCSP problems, can be obtained by exploiting the specific nature of the constraints, such as, e.g., the all-different constraint. Bfilt+ does not attempt to exploit this information to build the repartition. However, we believe that it could be a promising line of future research.

3.2. Color-filtering additional enhancements

In this paragraph, we consider a further enhancement of the colour-filtering phase. The key idea is that the procedure FILT−ISEQ can be applied to any (layer) partition of size k of the subproblem graph $\hat{G}$, 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 Fig. 2 into the following three layers: $\hat{\mathcal{V}}_1 = \{v_1, v_6\}, \hat{\mathcal{V}}_2 = \{v_3, v_4\}$ and $\hat{\mathcal{V}}_3 = \{v_2, v_5\}$. The graph $\tilde{G}$ computed by FILT−ISEQ according to the new layers $\hat{\mathcal{V}}$, 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 $\hat{\mathcal{V}}_1$. 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

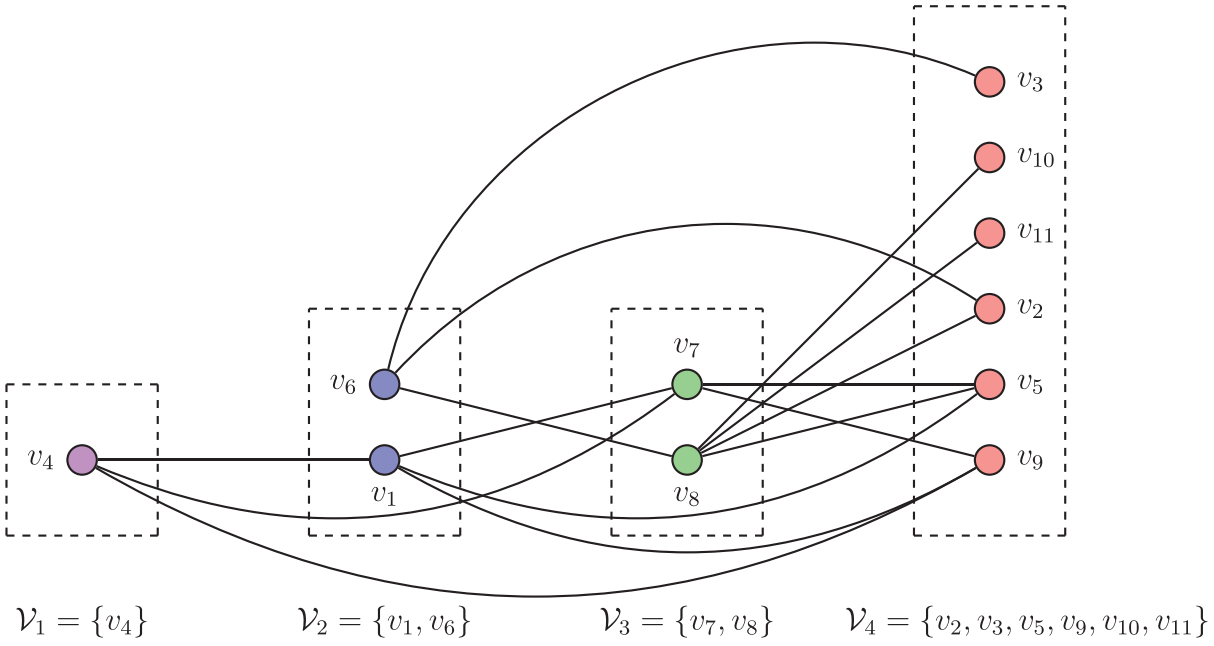

Fig. 5. The reordered microstructure graph $\mathcal{G}(\mathcal{I})$ of the example depicted in Fig. 1 after the execution of the SORT procedure. The graph $\mathcal{G}(\mathcal{I})$ is isomorphic to the graph $G(\mathcal{I})$ but it exhibits a new partition of size $k = 4$.

vertices $\{v_1, v_3, v_5\}$, cannot be reduced further, since these vertices form a clique of size three.

Specifically, Bfilt+ employs the colour-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 $\hat{G}$, 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 executed when the re-partition procedure determines a partition of size ℓ with $k < \ell \leq k + 10$. In this case, the SORT procedure reorders the vertices of the microstructure graph determining a new ℓ-partite graph denoted $\mathcal{G}_\ell = (\mathcal{V}_\ell, \mathcal{E}_\ell)$. The vertices and the edges of $\mathcal{G}_\ell$ are the same as $G(\mathcal{I})$, only the partition, the number of layers and the ordering of the vertices is different. Following the standard notation, we denote $\omega(\mathcal{G}_\ell)$ the *clique number* of the graph $\mathcal{G}_\ell$, i.e., the size of the largest clique in $\mathcal{G}_\ell$.

We recall that Bfilt cannot be executed since it works only on k-partite graphs. Therefore, in order to solve the k-CLP on $\mathcal{G}_\ell$, we adapt the state-of-the-art maximum clique algorithms (Li et al., 2018a; San Segundo et al., 2016) 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 colour 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.

3.3.1. Branching scheme

For a given node of the branching tree, the subproblem graph $\tilde{\mathcal{G}}_\ell = (\tilde{\mathcal{V}}_\ell, \tilde{\mathcal{E}}_\ell)$ 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 $\hat{K}$ after the colour-filtering phase is executed. The branching is in line with the classical scheme of MCLP algorithms and it relies on partitioning the branching candidate set $\tilde{\mathcal{V}}_\ell$ into two sets, denoted the *Branching Set* B and, its complement, the *Pruned Set* $P := \tilde{\mathcal{V}}_\ell \setminus B$. At each node of the branch-and-bound tree, we carry out a $|B|$-ary branching operation, thereby creating a tree node per vertex $v \in B$, by individually adding each vertex $v \in B$ to $\hat{K}$. A branching tree node is pruned if the Branching set B is empty. The vertices in P are never selected as branching vertices at the current tree node. The largest Pruned Set P (and the corresponding smallest Branching Set B) can be obtained by solving the following problem:

$$P := \arg\max_{\bar{P} \subseteq \tilde{\mathcal{V}}} \left\{ |\bar{P}| : (k-1) - |\hat{K}| \geq \omega(\tilde{\mathcal{G}}_\ell[\bar{P}]) \right\} \text{ and } \quad B := \tilde{\mathcal{V}}_\ell \setminus P, \tag{3}$$

where $(k-1)$ is the size of the largest clique (assumed) known at the start of the procedure. The key idea is to construct P in such a way that any subset of its vertices would not suffice, when added to $\hat{K}$, to produce a clique of size k. This implies that to find a clique of size k, one has to add to $\hat{K}$ at least one of the vertices in B. This is precisely what our branch-and-bound algorithm will do as it generates, for each $v \in B$, a child node containing the partial clique $\hat{K} \cup \{v\}$. For constructing the pruned set P, we look, in principle, for the largest set P such that:

$$|\hat{K}| + \omega(\tilde{\mathcal{G}}_\ell[P]) \leq k - 1,$$

where $\omega(\tilde{\mathcal{G}}_\ell[P])$ denotes the clique number of the graph induced by P. In other words, the condition states that the graph induced by the vertices in $\hat{K} \cup P$ does not contain a clique of size k. The larger the set P, the smaller the number of vertices in B and, thus, the smaller the number of child nodes created by the branching operation at the current node. Since determining the largest set P can be computationally difficult, the branching scheme determines P heuristically using two different bounds on $\omega(\tilde{\mathcal{G}}_\ell[P])$.

3.3.2. Bounding procedure

A first upper bound on $\omega(\tilde{\mathcal{G}}_\ell[P])$ is obtained by colouring $\tilde{\mathcal{G}}_\ell$ with ISEQ, the greedy sequential independent-set heuristic that inspires FILT−ISEQ, see Section 2.2. Precisely, the set P is composed of the set of vertices belonging to the first $(k-1)-|\hat{K}|$ 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 San Segundo et al., 2015), which operates on the colouring of $\tilde{\mathcal{G}}_\ell$. The outline of such procedure is to determine a collection $\mathcal{U}$ of independent sets in the colouring such that no clique of size $|\mathcal{U}|$ exists in the subgraph of $\tilde{\mathcal{G}}_\ell$ induced by the vertices in $\mathcal{U}$. Every time one such subset $\mathcal{U}$ is found, the upper bound provided by the colouring is reduced by one unit. The infrachromatic bounding function used by Bfilt+ is the MaxSAT-based bounding procedure of Li et al. (2018a), enhanced with the efficient bitstring data structures employed by Bfilt+.

3.4. Implementation details

We end this section commenting on relevant implementation details. Bfilt+ uses bitstrings to encode the problem and the data structures employed by our algorithm also in the nodes of the branch-and-filter tree. The encoding allows for a number of critical operations of the colour-filtering, SAT-filtering and branching phases to be efficiently computed with bitmasks, and is inspired by recent bitstring clique algorithms, see, e.g., San Segundo et al. (2019b); San Segundo et al. (2016). Specifically, bitmasking operations allow to manipulate bits in chunks of 64. Bfilt+ employs this encoding to represent the sets of vertices of the microstructure graph, where each bit refers to a specific vertex. Bitmasking allows to reduce the computing time of the operations on subsets of vertices by a factor of 64, such as the union or intersection operations. For example, the operations necessary to determine the subproblem graph (see Section 2.1) require computing the common neighborhood of a set of vertices. The latter computation can be done very efficiently using bitmasking. Another example can be found in the colour-filtering phase, see Section 2.2. Precisely, FILT−ISEQ uses bitmasking to test whether a vertex can be added to an independent set efficiently, reducing by a factor of 64 the computation time of a standard procedure. It is also worth mentioning that 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. Computational experiments

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

4.1. Experimental setting and dataset 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.

In this work we consider BCSP instances in the XCSP3 format, a recent XML-based format designed to represent (binary) CSPs (Boussemart, Lecoutre, Audemard, & Piette, 2016). Several state-of-the-art CSP solvers are able to directly read this format. Moreover, a library of (binary) CSP instances in XCSP3 format is available online in a dedicated server (http://xcsp.org/series). This library contains many different BCSPs in both *extensional* (the set of allowed/disallowed tuples is explicit) and *intensional* (the set of allowed/disallowed tuples is represented by a function) forms. In order to run Bfilt+ on the XCSP3 instances, we developed a dedicated parser to obtain the associated microstructure graph. We encode the latter using two files: *i*) a first file in DIMACS format (http://dimacs.rutgers.edu/programs/challenge/) containing the list of edges of the microstructure graph and *ii*) a second file representing the layers. Our parser is available in a GitHub repository (https://github.com/psanse/Bfilt), and it is able to convert a large subset of the binary instances in the XCSP3 library.

Starting from the XCSP3 dataset, we obtained a testbed of 1895 BCSP instances of real and synthetic problems, grouped in 17 *categories*. We converted all the instances which can be read by our parser and which generate a microstructure graph with no more than 8000 vertices. Table 1 reports information concerning the dataset, 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). It also reports, for each category, the percentage of instances that are satisfiable (column *sat*), unsatisfiable (column *unsat*) or unknown (column *fail*), i.e., unsolved by all the tested solvers within the time limit. All these instances are available in XCSP3 format at http://xcsp.org/series.

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 Hebrard, Hnich, & Walsh (2004) for a more detailed analysis. The misc category groups problem classes that contain less than 10 instances.

The imposed threshold of 8000 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 dataset includes synthetic random models of varying difficulty (B, D, RB2 and frb) as well as quasi-random models (comp

Table 1
Features of the 17 categories of the BCSP instances tested in this work.

		Number of variables			Number of tuples			Domain size		
Categories	#	min	max	avg.	min	max	avg.	min	max	avg.
B	50	23	27	25.0	100,694	198,666	146,539.4	23.0	27.0	25.0
Bla	37	64	208	117.2	211,379	26,291,636	6,176,596.8	10.5	35.3	19.7
comp	90	33	105	63.2	47,970	531,000	226,943.3	10.0	10.0	10.0
D	700	40	40	40.0	44,944	22,823,756	4,161,833.6	8.0	180.0	51.4
dflat	100	2237	2237	2237.0	28,100,632	28,102,060	28,101,380.1	3.4	3.4	3.4
ehi	200	297	315	306.0	2,051,275	2,314,937	2,183,080.6	7.0	7.0	7.0
frb	80	30	59	46.0	83,083	1,049,583	509,821.2	15.0	26.0	21.3
geom	100	50	50	50.0	434,373	456,163	447,848.9	20.0	20.0	20.0
hay	16	16	361	153.5	1,770	23,389,245	4,727,964.3	4.0	19.0	11.5
kni	10	5	9	6.6	21,840	10,569,744	2,006,599.6	64.0	625.0	295.2
lat	100	100	625	337.5	181,827	24,074,218	7,812,842.7	6.1	11.1	8.7
qk	12	13	30	20.0	43,288	4,095,405	1,111,847.2	29.5	125.0	68.1
RB2	300	30	50	40.0	83,230	581,244	303,739.8	15.0	23.0	19.0
rlfap	17	28	400	136.0	455,586	29,776,814	10,703,797.8	17.4	44.0	35.8
rm	12	4	100	23.4	24	19,120,341	1,844,187.9	3.0	69.1	18.3
ssol	38	8	100	33.1	276	26,159,948	12,121,350.9	4.0	246.0	148.6
misc	33	10	650	147.8	1225	20,593,125	6,185,383.7	1.0	89.0	43.4
Grand total	1895	4	2237	205.9	24	29,776,814	4,407,482.7	1.0	625.0	33.1

Table 2
Features of the microstructure graphs $G(\mathcal{I})$ for the BSCP tested instances in this work, and satisfiability information of the corresponding BCSP instances.

		Number of vertices			Edge density (%)			Satisfiability (%)		
Categories	#	min	max	avg	min	max	avg.	sat	unsat	fail
B	50	529	729	627.0	72.1	74.9	73.5	46.0	54.0	0.0
Bla	37	674	7334	2705.9	93.2	97.8	95.6	0.0	100.0	0.0
comp	90	330	1050	632.2	88.4	96.4	92.8	11.1	88.9	0.0
D	700	320	7200	2057.1	86.2	88.6	87.2	49.7	50.3	0.0
dflat	100	7511	7511	7511.0	99.6	99.6	99.6	100.0	0.0	0.0
ehi	200	2079	2205	2142.0	95.0	95.3	95.1	0.0	100.0	0.0
frb	80	450	1534	1013.3	82.2	89.3	86.6	66.3	0.0	33.7
geom	100	1000	1000	1000.0	87.0	91.3	89.7	92.0	8.0	0.0
hay	16	64	6859	2254.0	87.8	99.4	97.1	0.0	100.0	0.0
kni	10	320	5625	2033.6	40.4	67.2	51.6	0.0	100.0	0.0
lat	100	613	6961	3242.4	96.9	99.4	98.5	80.0	20.0	0.0
qk	12	384	3750	1558.0	58.2	59.3	58.6	0.0	100.0	0.0
RB2	300	450	1150	786.7	82.4	88.0	85.4	76.3	17.0	6.7
rlfap	17	1232	7820	3844.0	60.1	99.1	82.2	23.5	76.5	0.0
rm	12	12	6910	974.7	36.4	81.4	59.5	58.3	41.7	0.0
ssol	38	32	7872	4858.4	55.6	95.6	84.1	26.3	73.7	0.0
misc	33	50	7832	3690.1	48.2	100.0	76.9	66.7	30.3	3.0
Grand total	1895	12	7872	2114.9	36.4	100.0	88.4	51.6	45.9	2.5

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 dataset 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 A.1.

The language in which the BCSP instances are written and modelled is a critical issue. In the literature there are two main languages: *i*) the MiniZinc format (https://www.minizinc.org) and *ii*) the more recent XCSP3 format, described at the beginning of this section. To the best of our knowledge, an automatic translator between the two languages is unavailable, although it has been the subject of discussion in the literature, see, e.g., Morara, Mauro, & Gabbrielli (2011). Developing such a tool clearly goes beyond the scope of this paper, due to the intrinsic difficulties of this operation.

The presence of the two formats (MiniZinc and XCSP3) has an impact on the usage of the CSP solvers, since they are typically only able to read one of them. Since all our BCSP instances are in XCSP3 format, we can directly use those solvers which can parse this format (see Section 4.4). In order to run those solvers which are only able to read the MiniZinc format, we developed a converter from our microstructure-graph encoding to (extensional) MiniZinc format as follows: *i*) binary constraints in the XCSP3 format represented extensionally, i.e., by explicit enumeration of compatible or incompatible tuples of values, are compiled similarly in MiniZinc using the "table constraint", *ii*) binary constraints in the XCSP3 format represented intensionally, i.e., by predicates or functions, are compiled in two steps. In a first step, an extensional representation of each intensional constraint is derived based on the graph transformation required by Bfilt+. In a second step, the latter representation is compiled to MiniZinc according to *i*). The converter can be downloaded from the URL https://github.com/psanse/Bfilt.

It is worth mentioning that some problem classes may have been binarized before the inclusion in the XCSP3 database, e.g., the lat family (which can be more naturally modelled by the all-different constraint) or the Blackhole family. It is possible that the computational performance of the solvers may degrade by such a binarization, and a systematic analysis of this effect would be an interesting future line of research. Unfortunately, since *n*-ary

Table 3
Effect of the colour-filtering and SAT-filtering phases of Bfilt+ for a selected subset of instances.

		Bfilt+		Bfilt+ no COL filt.		Bfilt+ no SAT filt.		Bfilt+ no SAT/no COL filt.	
Categories	#	#opt	time [s]	#opt	time [s]	#opt	time [s]	#opt	time [s]
B	25	25	38.1	25	37.7	25	71.6	22	108.0
Bla	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

equivalent models for the vast majority of the instances in our dataset are unavailable, and a translation would require an additional heavy manipulation of the instances, this analysis goes beyond the scope of this work, which focuses, instead, on the evaluation of the performance of the newly developed exact algorithm Bfilt+. Notwithstanding the fact that some of the instances have been binarized, our dataset of 1895 problems is a large and representative set of BCSPs. Considering the diversity of the 17 families tested, see Tables 1 and 2, and the fact that these instances have been proposed in the literature for benchmarking purposes, we believe that our dataset is adequate to establish a fair computational comparison between Bfilt+ and the state-of-the-art solvers.

4.2. Evaluation of the colour-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) colour-filtering and SAT-filtering phases. To establish the comparison, we consider a subset of 225 instances from our 1895 instance dataset. 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 colour-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 colour-filtering and SAT-based filtering phases by themselves is as follows: the *no COL filt.* variant solves 207 instances and the *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.

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 dataset, the percentage of instances in which the aforementioned features are employed. Specifically, the column Bfilt $\rightarrow G(\mathcal{I})$ refers to the execution of the procedure Bfilt on the microstructure graph $G(\mathcal{I})$, the column Bfilt $\rightarrow \mathcal{G}(\mathcal{I})$ refers to the execution of Bfilt considering the new partition of k layers provided by the re-partitioning procedure, and the column *max clique proc.* $\rightarrow \mathcal{G}_\ell$ 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 BCSP instance that, 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 $\rightarrow G(\mathcal{I})$ 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 MCLP solvers: MoMC (Li et al., 2017), and BBMCX (San Segundo et al., 2016). Table 5 reports the number of instances solved and the average CPU time spent by the four algorithms, i.e., Bfilt+, Bfilt $\rightarrow G(\mathcal{I})$, 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 MCLP solvers, determining the satisfiability of 209 of the instances, while Bfilt $\rightarrow G(\mathcal{I})$ and MoMC are only able to solve 163 and 161 respectively, and BBMCX can only solve 128. It is worth mentioning that only Bfilt+ is able to solve all the instances from the Blackhole family (bla), while the other 3 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 7 publicly available state-of-the-art solvers for BCSPs. Four of the 7 solvers are based on the reduction of the BCSP to a CNF-SAT problem which is solved by a state-of-the-art SAT algorithm. We denote them *SAT-based* solvers in the remainder of the paper and we provide a brief description in what follows.

1. *PicatSAT 2.8* (Picat) (Zhou, Kjellerstrand, & Fruhman, 2015) (http://picat-lang.org/): A SAT-based solver that uses the Picat Prolog-like rule-based language. In the tests, we employ the version that performed best in the main track—CSP, sequential—of the (last held) XCSP3 2019 competition (http://xcsp.org/competition).

Table 4
Percentage of instances from the entire dataset of 1895 instances in which the additional features of Bfilt+ are used.

		Bfilt+		
Categories	#	Bfilt $\rightarrow G(\mathcal{I})$	Bfilt $\rightarrow \mathcal{G}(\mathcal{I})$	max clique proc. $\rightarrow \mathcal{G}_\ell$
B	50	0.0	100.0	0.0
Bla	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	100	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
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	1895	8.5	65.8	25.7

Table 5
Evaluation of the additional algorithmic improvements of Bfilt+ over a selected subset of instances. The algorithmic variant Bfilt $\rightarrow G(\mathcal{I})$ 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 MCLP solvers MoMC (Li et al., 2017) and BBMCX (San Segundo et al., 2016).

		Bfilt+		Bfilt $\rightarrow G(\mathcal{I})$		MoMC (Li et al., 2017)		BBMCX (San Segundo et al., 2016)	
Categories	#	#opt	time (s)	#opt	time (s)	#opt	time (s)	#opt	time (s)
B	25	25	38.1	25	32.4	23	104.7	23	109.4
Bla	25	25	7.9	0	–	0	–	0	–
comp	25	25	0.2	25	0.1	22	1.0	22	16.8
D	25	25	23.4	18	19.9	22	36.0	22	31.6
ehi	25	25	5.5	25	0.8	24	350.6	0	-
frb	25	19	29.3	12	99.1	16	74.6	12	99.3
geom	25	25	0.5	24	17.0	25	4.9	25	21.1
lat	25	18	34.6	18	14.8	8	104.8	8	5.9
RB2	25	22	39.2	16	52.1	21	65.8	16	62.7
Grand total	225	209	18.8	163	23.8	161	94.2	128	49.6

2. *sCOP[order + MapleCOMSPS]* (sCOP) (https://tsoh.org/sCOP/): A SAT-based constraint programming solver written in the *Scala* language. In the tests we use the latest publicly available version https://tsoh.org/sCOP/, that 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 (http://www.satcompetition.org/).
3. *Chuffed 0.10.14* (Chuffed) (https://github.com/chuffed/chuffed): It is a lazy clause solver written in C++ which has taken part in recent MiniZinc Challenges (https://www.minizinc.org/challenge.html). The solver dynamically transforms a CSP problem into a SAT problem in a "lazy" fashion, i.e., without translating a priori the full model.
4. *OR-Tools 8.2.8710/CP-SAT* (OR-tools) (https://developers.google.com/optimization): it is the open source software suite developed by Google, which includes a SAT-based module for constraint programming problems. It has won the gold medal consistently in the most recent MinZinc Challenges.

The other 3 solvers are classical constraint programming solvers, denoted *CP-based* in the remainder of the paper and we provide a brief description in what follows.

1. *Choco-solver 4.10.4* (choco) (https://choco-solver.org/): 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.
2. *Concrete 3.12.3* (Concrete) (https://github.com/concrete-cp/concrete): Concrete is a CP-based solver written in Scala. In the tests, we use the version that took part in the XCSP3 2019 competition.
3. *Mistral 2.0* (Mistral) (https://github.com/ehebrard/Mistral-2.0): 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.

It is worth noting that choco, Concrete, Mistral, Picat and sCOP are directly able to parse the XCSP3 format and they are run on the instances downloaded from the XCSP3 server without requiring any pre-processing. However, the solvers Chuffed and OR-tools are only able to read instances in the MiniZinc format. Consequently, to test these solvers, we compiled all the instances to the MiniZinc format as described in Section 4.1.

Table 6 reports the computing performance of the 5 solvers that can parse the XCSP3 format, together with Bfilt+, over our 1895 instance dataset. The first column of the table reports the name and number of instances in parenthesis. For each one of the algorithms and the 17 categories in the dataset, 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 that performs best, determining satisfiability in 1783 instances out of the possi-

Table 6
Computational comparison of Bfilt+ against two SAT-based and three CP-based solvers over the entire dataset of 1895 instances.

	Bfilt+			Picat			choco		
		CPU time (s)			CPU time (s)			CPU time (s)	
Categories	#opt	avg.	std. dev.	#opt	avg.	std. dev.	#opt	avg.	std. dev.
B(50)	50	40.6	54.9	8	312.4	187.0	21	201.9	142.6
Bla(37)	37	9.3	18.8	30	234.0	169.4	10	1.0	0.0
comp(90)	90	0.2	0.2	90	0.1	0.0	90	0.5	0.0
D(700)	695	24.4	64.7	691	57.5	74.9	700	15.9	28.0
dflat(100)	94	160.1	114.7	100	10.0	5.8	100	3.8	2.7
ehi(200)	200	4.4	2.5	200	0.4	0.0	200	0.7	0.1
frb(80)	52	23.2	44.7	37	82.3	149.8	36	81.9	148.7
geom(100)	100	0.6	1.2	100	19.1	41.9	100	13.1	36.2
hay(16)	4	58.8	116.3	11	65.4	166.4	16	0.7	0.2
kni(10)	10	2.4	2.1	10	7.1	9.4	10	1.3	1.2
lat(100)	70	32.8	98.6	100	2.8	4.6	71	51.9	136.1
qk(12)	8	3.0	2.7	12	6.0	7.6	12	12.6	27.8
RB2(300)	278	46.3	109.6	211	80.7	122.5	203	76.2	122.7
rlfap(17)	15	20.0	50.6	17	8.6	7.0	17	0.8	0.2
rm(12)	12	1.6	5.1	12	0.1	0.1	12	2.8	6.6
ssol(38)	36	55.7	58.9	34	3.1	4.9	33	18.4	52.0
misc(33)	32	22.2	86.7	31	41.6	49.2	31	9.4	13.3
Grand total (1895)	1783	30.9	78.1	1694	44.3	86.3	1662	24.4	69.4

	Mistral			Concrete			sCOP		
		CPU time (s)			CPU time (s)			CPU time (s)	
Categories	#opt	avg.	std. dev.	#opt	avg.	std. dev.	#opt	avg.	std. dev.
B(50)	16	265.7	174.1	3	111.0	130.3	12	288.9	162.1
Bla(37)	10	0.4	0.0	10	14.9	0.7	30	63.6	43.9
comp(90)	90	0.1	0.0	90	3.0	0.2	90	3.0	0.4
D(700)	700	35.8	45.4	531	119.4	144.8	686	64.7	91.5
dflat(100)	100	10.2	12.5	99	74.8	83.3	100	11.3	0.5
ehi(200)	200	0.3	0.0	200	4.8	1.0	200	6.3	0.2
frb(80)	34	73.7	153.2	25	101.9	139.7	40	75.1	127.6
geom(100)	100	27.8	82.7	89	13.8	50.2	100	12.0	20.9
hay(16)	2	0.9	1.2	2	15.0	17.8	16	1.0	0.2
kni(10)	6	70.9	93.5	10	3.2	0.9	10	20.4	37.6
lat(100)	71	26.5	91.8	69	40.1	83.5	100	6.8	6.1
qk(12)	11	106.2	151.6	12	3.6	1.1	12	2.2	0.3
RB2(300)	184	88.3	141.8	121	71.0	124.6	208	67.9	107.2
rlfap(17)	17	0.4	0.3	17	6.0	2.9	17	2.7	0.6
rm(12)	12	0.7	1.5	12	4.5	5.3	12	1.9	1.1
ssol(38)	34	1.7	5.1	34	26.1	47.7	34	2.2	0.8
misc(33)	29	35.2	83.7	29	51.3	115.8	30	30.5	36.2
Grand total (1895)	1616	35.0	79.1	1353	66.7	117.1	1697	42.9	81.7

ble 1895. The SAT-based solvers perform second best, sCOP} solving 1697 instances, and Picat three instances less in similar time. From the group of CP-based solvers, choco is the algorithm that solves the largest number of instances (precisely 1662) 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 determine 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 A.1), in which Bfilt+ shows speed-ups of around 3× with respect to the rest of competitors. Bfilt+ also performs very effectively in geom, where it is orders 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 Table 2, all the instances of dflat are of the satisfiable type, and dispose of a huge number of variables (precisely 2237) 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 dataset 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.

Table 7 reports the performance of the solvers Chuffed and OR-tools, together with Bfilt+, over the entire instance

Table 7
Computational comparison of Bfilt+ against the SAT-based solvers Chuffed and OR-tools over the entire 1895 dataset of instances.

	Bfilt+			Chuffed			OR-tools		
		CPU time (s)			CPU time (s)			CPU time (s)	
categories	#opt	avg.	std. dev.	#opt	avg.	std. dev.	#opt	avg.	std. dev.
B(50)	50	40.6	54.9	4	152.0	251.6	0	-	-
Bla(37)	37	9.3	18.8	10	0.3	0.0	37	5.5	8.5
comp(90)	90	0.2	0.2	90	0.3	0.1	90	0.2	0.1
D(700)	695	24.4	64.7	698	42.6	57.4	162	202.2	173.9
dflat(100)	94	160.1	2.5	100	2.2	0.5	94	130.2	132.7
ehi(200)	200	4.4	44.7	200	0.6	0.0	200	0.8	0.1
frb(80)	52	23.2	1.2	32	67.5	122.8	10	76.1	54.4
geom(100)	100	0.6	2.1	94	29.4	85.6	43	37.9	71.1
hay(16)	4	58.8	98.6	8	3.2	7.7	16	1.0	1.1
kni(10)	10	2.4	2.7	10	2.6	4.0	10	4.6	7.4
lat(100)	70	32.8	109.6	100	9.0	26.0	84	35.6	104.8
qk(12)	8	3.0	58.9	12	1.4	1.9	12	2.2	3.0
RB2(300)	278	46.3	50.6	177	90.0	138.1	81	185.3	156.9
rlfap(17)	15	20.0	5.1	17	0.8	0.4	17	3.8	6.4
rm(12)	12	1.6	114.7	12	2.5	6.6	12	2.3	6.1
ssol(38)	36	55.7	116.3	34	5.3	2.8	32	46.4	93.8
misc(33)	32	22.2	86.7	30	11.4	11.8	30	15.2	16.7
Grand total (1895)	1783	30.9	78.1	1628	32.6	71.7	930	73.0	132.3

dataset. This comparison is reported in a separate table since the solvers are run on the instances compilated to the MiniZinc format produced by our parser, as described in Section 4.1. The structure of this table is the same as the one of Table 6. According to the reported results, both algorithms are clearly outperformed by Bfilt+. Specifically, out of the 1895 instances, Chuffed solves 1628 within the time limit, while OR-tools is only able to solve 930. A possible explanation for the poor performance of both Chuffed and OR-tools on BCSP instances can be related to the fact that these solvers are specifically tailored for n-ary CSPs. Moreover, since these solvers are run on extensional models generated by our parser, see Section 4.1, another explanation might be connected to the fact that their sophisticated SAT-compilation/solving routines are probably not fully exploited.

As far as the aggregate performance according to families of instances is concerned, the reported results show that Bfilt+ compares favourably against the other solvers. Specifically, for the 17 families in our dataset, Bfilt+ solves to optimality strictly more instances than all the other solvers in 5 families, i.e., B, frb, RB2, ssol and misc, and it ties with the best solvers in 6 families, i.e., Bla, comp, ehi, geom, kni and rm. In other words, Bfilt+ solves to optimality at least the same number of instances as those solved by the best solver for each specific family in 11 out of the 17 families. On the other hand, and regarding the SAT-based solvers, Picat outperforms Bfilt+ in 5 problem classes and it is beaten in 7, sCOP in 5 (beaten in 7), chuffed in 6 (beaten in 7) and or-tools in 4 (beaten in 8). Regarding the CP-based solvers, choco outperforms Bfilt+ in 6 families and it is beaten in 6, mist in 5 (beaten in 8) and conc in 3 (beaten in 10). These results show that, even though Bfilt+ has an overall excellent computational performance, it is outperformed in some problem classes by the state-of-the-art solvers. It is worth noting that by comparing Bfilt+ and choco in terms of the number of classes in which one beats the other, the results show a tie. However, as mentioned previously, Bfilt+ solves more instances in total and it shows a superior performance according to the performance profile, see, Fig. 6.

A graphical representation of the relative computing performance of Bfilt+ with respect to the other 7 solvers is provided by the performance profile shown in Fig. 6. We compute the normalized time τ as the ratio of the computing time of each algorithm (∞ if the instance is not solved to optimality) over the minimum computing time taken for all the algorithms we tested. For each value of τ on the horizontal axis, the vertical axis reports the percentage of instances for which the corresponding algorithm spent at most τ times the computing time of the fastest algorithm. The 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 τ, 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.

According to Fig. 6, the solver Bfilt+ is the one which performs best, being the fastest in more than 50% of the instances (left-end of the figure) and also solving the largest number of instances, i.e., slightly over 94% (right-end of the figure). The SAT-based solvers that are directly able to parse the XCSP3 format 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%. With respect to the two solvers that only parse the MiniZinc format, Chuffed solves initially more instances than the two best SAT-based solvers Picat and sCOP, but it is gradually outperformed by the other two, e.g. Chuffed solves around 75% of the instances for $\tau = 100$, while, as mentioned previously, Picat and sCOP solve more than 80%. Finally, OR-tools is the worst performing algorithm, solving slightly less than 50% of the instances within the time limit, and far below the rest of competitors. Concerning the CP-based algorithms, choco is the best performing one, being the fastest in around 5% of the instances.

We end the section by showing in Fig. 7 the computing time boxplots of the 8 algorithms. The figure depicts 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 quar-

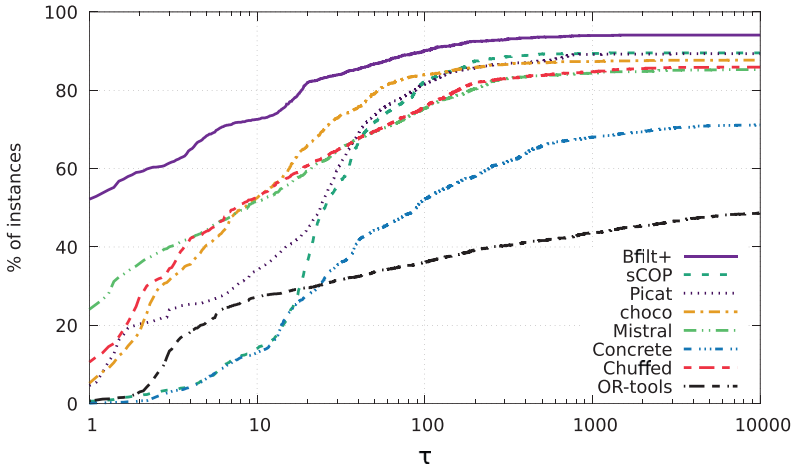

Fig. 6. Performance profile of `Bfilt+`, together with another 7 solvers, over the entire dataset of 1895 instances.

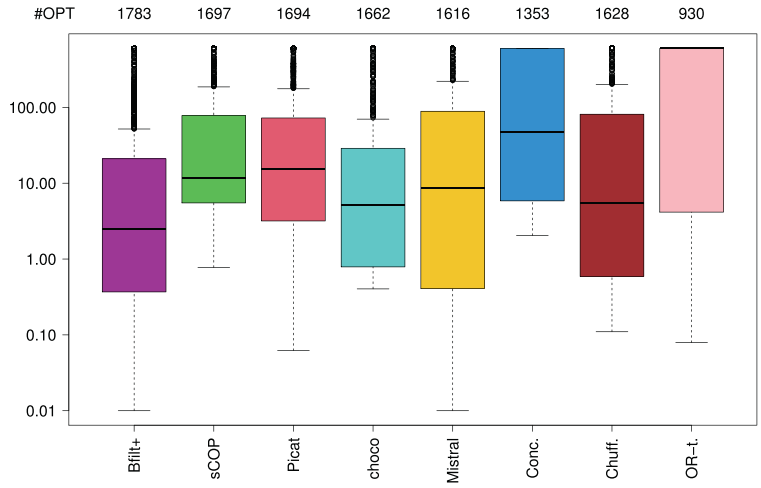

Fig. 7. Computing time boxplots of `Bfilt+` and the seven solvers over the entire dataset of 1895 instances. The *y*-axis shows in logarithmic scale the CPU times in seconds. On the top part of the figure, we report the total number of instances (#OPT) solved by each of the algorithms.

tile, the outliers are plotted as individual points. Fig. 7 further evidences the superior computing times of `Bfilt+`, and is consistent with the other results reported in the section.

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 per instance on the computing performance. As a general rule, the larger the number of variables, the harder the instances become. However, some algorithms are well-suited to specific classes and thus scale much better in those cases. These considerations can be observed in Fig. 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*-CLP on the underlying microstructure graph. The new exact algorithm, 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 colour-filtering, is based on colouring 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 algorith-

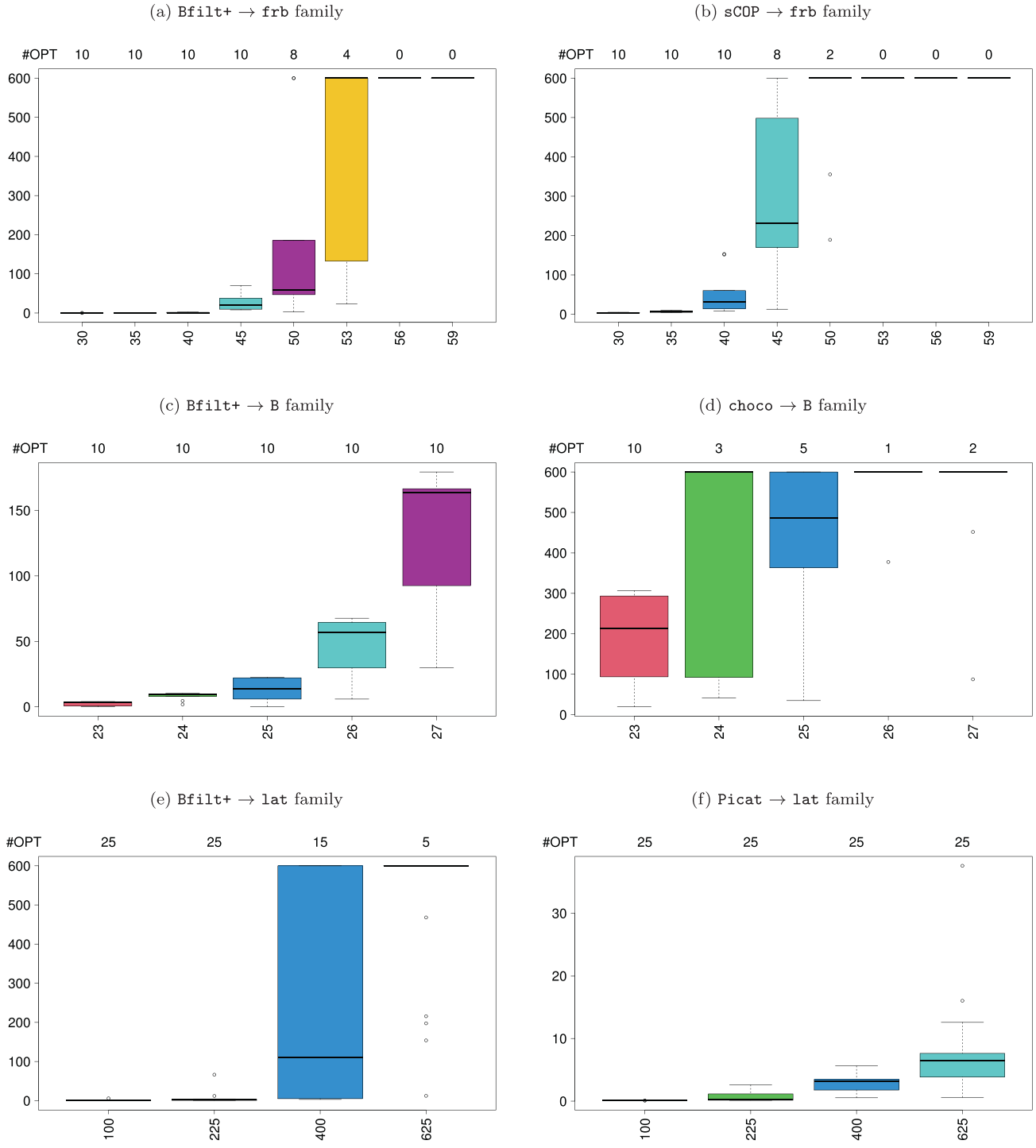

Fig. 8. A comparison of the computing performance of Bfilt+ with the best solver for three classes of BSCP instances: in parts (a) and (b) we compare Bfilt+ against sCOP for the frb class; in parts (c) and (d) we compare Bfilt+ against choco for the B class; in parts (e) and (f) we compare Bfilt+ against 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..

mic 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 7 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., Hespe, Lamm, Schulz, & Strash (2020); San Segundo et al. (2016b). 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

Dechter & Pearl (1989); Rossi et al. (1990); Stergiou & Walsh (1999). 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, and the extension of Bfilt+ for these non-binary problems.

Acknowledgements

The authors are grateful to three anonymous referees for their useful comments that helped us to improve both the quality and the contribution of this paper. The work has been partially funded by the Spanish Ministry of Science, Innovation and Universities through the projects COGDRIVE (DPI2017-86915-C3-3-R). This work is also partially funded by the Spanish Agencia Estatal de Investigación (PID2020-113096RB-I00 / AEI / 10.13039/501100011033) through the project ACOGES (COGnitive personal Assistance for Social Environments).

A1. Detailed description of the classes of BCSP instances

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

- Random models (B, D, RB2, frb), see, e.g., Gent, Macintyre, Prosser, Smith, & Walsh (2001): 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 dataset contains the families frb and RB2 that derive from a revised model of B, denoted RB (Xu & Li, 2000), that produces harder instances than the ones obtained from the original model; see Achlioptas et al. (1997) for the motivation behind the revision of the model. Incidentally, the frb family is also typically employed for benchmarking MCLP solvers.
- Black Hole (Bla) (Gent et al., 2007): 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 dataset 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 Lecoutre, Boussemart, & Hemery (2004). 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 *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 *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 Lecoutre et al. (2004). The *SAT-flat-dual* (dflat) class derives from an encoding to CSP of the 3-colouring problem over a set of 3-colourable graphs (specifically, the *flat200-479* series), see Culberson & Luo (1996) 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) (Achlioptas, Gomes, Kautz, & Selman, 2000; Pesant, Quimper, & Zanarini, 2012): 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 *quasigroup with holes problem*, for a total of 100 instances.
- Queens-Knights (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) (Boussemart, Hemery, Lecoutre, & Sais, 2004). The class contain two types, denoted add and mul. In the type add, a square of the chessboard can be shared by both a queen and a knight. In the type mul, this is not allowed.
- Knights (kni) (Boussemart et al., 2004): A variant of the n-queens problem that considers knight constraints instead of queen constraints.
- Haystack (hay): This class of unsatisfiable instances was created by Marc van Dongen http://research.ucc.ie/profiles/D005/dongen, and has been included in several CSP challenges, see, e.g., Lecoutre & Roussel (2018). 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 that is inconsistent. This cluster is called the haystack.
- Roommates (rm) (Prosser, 2014): This set of instances derives from the *stable roommates problem* (SRP). In the SRP, a set of participants initially rank each other. The problem calls to find a *stable matching*, i.e., a matching such that no two participants prefer each other to their matched partners.
- Radio link frequency assignment (rlfap) (Cabon, De Givry, Lobjois, Schiex, & Warners, 1999): 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 rlfap class was initially derived from data from real networks and is composed of different series.
- Supersolution problems (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 *robust*, solution set. The original problems considered include some scheduling and *N*-queens instances.
- Miscellaneous (misc): As mentioned at the beginning of the section, this category includes series that have less than 10 instances. The category holds, amongst others, the *marc* and *lard* synthetic families designed by Marc van Dongen, the *queenAttacking* suite derived from the *Queen Attacking problem* and the real-world suite *driverlogw* from the logistics domain, see, e.g., Boussemart, Hemery, & Lecoutre (2005), for a more detailed description.

References

Achlioptas, D., Gomes, C., Kautz, H., & Selman, B. (2000). Generating satisfiable problem instances. In *AAAI/IAAI, 2000* (pp. 256–261).

Achlioptas, D., Kirousis, L. M., Kranakis, E., Krizanc, D., Molloy, M. S., & Stamatiou, Y. C. (1997). Random constraint satisfaction: A more accurate picture. In *International conference on principles and practice of constraint programming* (pp. 107–120).

Boussemart, F., Hemery, F., & Lecoutre, C. (2005). Description and representation of the problems selected for the first international constraint satisfaction solver competition. In *Proceedings of the second international workshop on constraint propagation and implementation: 2* (pp. 7–26).

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *ECAI: 16* (p. 146).

Boussemart, F., Lecoutre, C., Audemard, G., & Piette, C. (2016). XCSP3: An integrated format for benchmarking combinatorial constrained problems. arXiv:1611.03398

Brailsford, S. C., Potts, C. N., Smith, B. M., & Oper, J. (1999). Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research, 119*(3), 557–581.

Buscemi, M. G., & Montanari, U. (2008). A survey of constraint-based programming paradigms. *Computer Science Review, 2*(3), 137–141.

Cabon, B., De Givry, S., Lobjois, L., Schiex, T., & Warners, J. P. (1999). Radio link frequency assignment. *Constraints, 4*(1), 79–89.

Caprara, A., Galli, L., & Toth, P. (2011). Solution of the train platforming problem. *Transportation Science, 45*(2), 246–257.

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

Cohen, D. A. (2003). A new class of binary CSPs for which arc-consistency is a decision procedure. In *International conference on principles and practice of constraint programming* (pp. 807–811).

Coniglio, S., Furini, F., & San Segundo, P. (2020). A new combinatorial branch-and-bound algorithm for the Knapsack problem with conflicts. *European Journal of Operational Research*. https://doi.org/10.1016/j.ejor.2020.07.023.

Cooper, M. C., Jeavons, P. G., & Salamon, A. Z. (2010). Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination. *Artificial Intelligence, 174*(9–10), 570–584.

Culberson, J. C., & Luo, F. (1996). Exploring the *k*-colorable landscape with iterated greedy. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 26*, 245–284.

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

Dechter, R., et al. (2003). *Constraint processing*. Morgan Kaufmann.

Dechter, R., & Pearl, J. (1988). Network-based heuristics for constraint-satisfaction problems. In *Search in artificial intelligence* (pp. 370–425). Springer.

Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence, 38*(3), 353–366.

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

Furini, F., Ljubić, I., Martin, S., & San Segundo, P. (2019). The maximum clique interdiction problem. *European Journal of Operational Research, 277*(1), 112–127.

Furini, F., Ljubić, I., San Segundo, P., & Zhao, Y. (2021). A branch-and-cut algorithm for the edge interdiction clique problem. *European Journal of Operational Research, 294*(1), 54–69.

GECODE (2016). Gecode toolkit. https://www.gecode.org.

Gent, I. P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., et al. (2007). Search in the patience game 'black hole'. *AI Communications, 20*(3), 211–226.

Gent, I. P., Jefferson, C., & Miguel, I. (2006). Minion: A fast scalable constraint solver. In *ECAI: 141* (pp. 98–102).

Gent, I. P., Macintyre, E., Prosser, P., Smith, B. M., & Walsh, T. (2001). Random constraint satisfaction: Flaws and structure. *Constraints, 6*(4), 345–372.

Golomb, S. W., & Baumert, L. D. (1965). Backtrack programming. *Journal of the ACM (JACM), 12*(4), 516–524.

Grünert, T., Irnich, S., Zimmermann, H.-J., Schneider, M., & Wulfhorst, B. (2002). Finding all *k*-cliques in *k*-partite graphs, an application in textile engineering. *Computers and Operations Research, 29*(1), 13–31.

Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*(3), 263–313.

Hebrard, E., Hnich, B., & Walsh, T. (2004). Super solutions in constraint programming. In *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming* (pp. 157–172).

Hespe, D., Lamm, S., Schulz, C., & Strash, D. (2020). WeGotYouCovered: The winning solver from the PACE 2019 challenge, vertex cover track. In H. M. Bücker, X. S. Li, & S. Rajamanickam (Eds.), *Proceedings of the SIAM workshop on combinatorial scientific computing, CSC 2020, Seattle, USA, February 11–13, 2020* (pp. 1–11). SIAM.

IBM (2017). ILOG CPLEX optimization studio 12.7.1: CP optimizer online documentation. http://ibm.biz/COS1271Documentation.

Jégou, P. (1993). Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In R. Fikes, & W. G. Lehnert (Eds.), *Proceedings of the 11th national conference on artificial intelligence, Washington, DC, USA, July 11–15, 1993* (pp. 731–736).

Jégou, P., & Terrioux, C. (2015). The extendable-triple property: A new CSP tractable class beyond BTP. In *AAAI* (pp. 3746–3754).

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations* (pp. 85–103). Springer.

Lecoutre, C., Boussemart, F., & Hemery, F. (2004). Backjump-based techniques versus conflict-directed heuristics. In *16th IEEE international conference on tools with artificial intelligence* (pp. 549–557). IEEE.

Lecoutre, C., & Roussel, O. (2018). Proceedings of the 2018 XCSP3 Competition. arXiv:1901.01830

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

Li, C., Fang, Z., & Xu, K. (2013). Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In *2013 IEEE 25th international conference on tools with artificial intelligence* (pp. 939–946).

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

Li, C., Liu, Y., Jiang, H., Manyá, F., & Li, Y. (2018b). A new upper bound for the maximum weight clique problem. *European Journal of Operational Research, 270*(1), 66–77.

Li, C. M., & Quan, Z. (2010). An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *AAAI: 10* (pp. 128–133).

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence, 8*(1), 99–118.

Marriott, K., & Stuckey, P. J. (1998). *Programming with constraints: An introduction*. MIT Press.

Mirghorbani, M., & Krokhmal, P. A. (2013). On finding *k*-cliques in *k*-partite graphs. *Optimization Letters, 7*(6), 1155–1165.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences, 7*, 95–132.

Morara, M., Mauro, J., & Gabbrielli, M. (2011). Solving xcsp problems by using gecode.

Mouelhi, A. E., Jégou, P., & Terrioux, C. (2014). Different classes of graphs to represent microstructures for CSPs. In *Graph structures for knowledge representation and reasoning* (pp. 21–38). Springer.

Murphey, R. A., Pardalos, P. M., & Resende, M. G. C. (1999). *Frequency assignment problems* (pp. 295–377). Boston, MA: Springer US.

Naanaa, W. (2020). New schemes for simplifying binary constraint satisfaction problems. *Discrete Mathematics and Theoretical Computer Science, DMTCS, 22*(1) ⟨hal-01731250v4⟩.

Ohrimenko, O., Stuckey, P. J., & Codish, M. (2009). Propagation via lazy clause generation. *Constraints, 14*(3), 357–391.

Pesant, G., Quimper, C.-G., & Zanarini, A. (2012). Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research, 43*, 173–210.

Prosser, P. (2014). Stable roommates and constraint programming. In *International conference on AI and OR techniques in constriant programming for combinatorial optimization problems* (pp. 15–28). Springer.

Régin, J. C. (1994). A filtering algorithm for constraints of difference in CSPs. In *AAAI: 94* (pp. 362–367).

Rossi, F., Beek, P. V., & Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Rossi, F., Petrie, C. J., & Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In *ECAI: 90* (pp. 550–556).

Samaras, N., & Stergiou, K. (2005). Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research, 24*, 641–684.

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

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

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

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

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

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

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

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

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

Stergiou, K., & Walsh, T. (1999). Encodings of non-binary constraint satisfaction problems. In *AAAI/IAAI* (pp. 163–168).

Tsang, E. (1993). *Foundations of constraint satisfaction*. Citeseer.

Xu, K., & Li, W. (2000). Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research, 12*, 93–103.

Zhou, N.-F., Kjellerstrand, H., & Fruhman, J. (2015). *Constraint solving and planning with Picat*. Springer.