Combining CP and ILP in a tree decomposition of bounded height for the sum colouring problem

Maël Minot, Samba Ndojh Ndiaye, Christine Solnon

Combining CP and ILP in a tree decomposition of bounded height for the sum colouring problem

M. Minot[1,3], S. N. Ndiaye[1,2], and C. Solnon[1,3]

[1] Université de Lyon – LIRIS
[2] Université Lyon 1, LIRIS, UMR5205, F-69622 France
[3] INSA-Lyon, LIRIS, UMR5205, F-69621, France
{mael.minot,samba-ndojh.ndiaye,christine.solnon}@liris.cnrs.fr

Abstract. The Sum Colouring Problem is an $\mathcal{NP}$-hard problem derived from the well-known graph colouring problem. It consists in finding a proper colouring which minimizes the *sum* of the assigned colours rather than the number of those colours. This problem often arises in scheduling and resource allocation. In this paper, we conduct an in-depth evaluation of ILP and CP's capabilities to solve this problem, with several improvements. Moreover, we propose to combine ILP and CP in a tree decomposition with a bounded height. Finally, those methods are combined in a portfolio approach to take advantage from their complementarity.

1 Introduction

The Sum Colouring Problem (SCP) is an $\mathcal{NP}$-hard problem derived from the well-known graph colouring problem. It consists in finding a proper colouring (*i.e.* an assignment of colours to vertices such that neighbour vertices have different colours) which minimizes the *sum* of the assigned colours rather than the number of those colours. This problem arises in a variety of real-world problems, especially in scheduling and resources allocation [21]. Many incomplete approaches have been proposed [12], whereas only few complete approaches have been proposed: mainly Integer Linear Programming (ILP) in [12], Branch and Bound (B&B), SAT and Constraint Programming (CP) in [22]. In this paper, we propose to conduct a more in-depth evaluation of ILP and CP's capabilities to solve the SCP, with several improvements. Moreover, we use tree decomposition to improve the solution process by decomposing SCPs into independent subproblems which are solved by ILP and CP, with promising results. Finally, we show that a portfolio approach can take advantage of the complementarity of the different approaches.

Section 2 defines the SCP and gives an overview of existing approaches. Sections 3 and 4 introduce improvements for CP and ILP models. Section 5 explains how CP and ILP may be combined by means of a tree decomposition, and Section 6 introduces a portfolio approach.

2 The sum colouring problem

An undirected graph $G = (V, E)$ is defined by a set V of nodes and a set $E \subseteq V \bigtimes V$ of edges. Each edge of G is an undirected pair of nodes. We note

$\deg(v)$ the degree of a vertex v, *i.e.* $\deg(v) = |\{u \in V, \{u,v\} \in E\}|$, and $\Delta(G)$ the largest degree in the graph, *i.e.* $\Delta(G) = \max\{\deg(v), v \in V\}$.

A legal (or "proper") k-colouring of a graph $G = (V, E)$ is a mapping $c : V \to [1, k]$ such that $\forall\{x, y\} \in E, c(x) \neq c(y)$. Classic graph colouring aims at finding a proper k-colouring that minimizes k, whereas the SCP aims at finding a proper k-colouring that minimizes the sum of assigned colours, *i.e.*, $\sum_{x \in V} c(x)$. The lowest achievable sum for G is called the *chromatic sum* and is denoted $\Sigma(G)$.

Existing bounds. In [30], it is shown that $\lceil \sqrt{8|E|} \rceil \leq \Sigma(G) \leq \lfloor \frac{3(|E|+1)}{2} \rfloor$. In [21], it is demonstrated that $\Sigma(G) \leq |V| + |E|$, and that an optimal sum colouring will never use strictly more than $\Delta(G) + 1$ colours. Finally, in [24,33] a lower bound is defined with respect to a clique decomposition of the graph.

Definition 1. *A* clique *is a subset of nodes which are all linked pairwise. A clique decomposition of a graph is a partition $\mathcal{C}$ of its vertices such that, for each set $C_i \in \mathcal{C}$, the subgraph induced by C_i is a clique.*

Given a clique decomposition $\mathcal{C}$ of G, we have $\Sigma(G) \geq \sum_{C_i \in \mathcal{C}} |C_i| \cdot (|C_i|+1)/2$, as all vertices in a same clique must have different colours.

Dominant colourings. Dominant colourings were discussed in [22]: a k-colouring may be seen as an ordered partition on the vertices of the coloured graph, such that the i-th set S_i contains all vertices using colour i (with $1 \leq i \leq k$). A colouring is dominated when the vertex colour sum can be lowered simply by reassigning the indices of these sets (*i.e.*, swapping colours) without actually altering the partition. The *dominant colouring* of a k-colouring c is the colouring obtained by ordering the partition defined by c by decreasing size of sets, so that S_1 is the largest set, and S_k the smallest. This dominant colouring has the lowest vertex colour sum among all possible colour swappings of c.

Incomplete approaches. Many incomplete approaches were used to find approximate solutions for the SCP. A review of most of these approaches may be found in [12]. It classifies main contributions in three classes: greedy algorithms [32,33], local search heuristics [3,7] and evolutionary algorithms [13,23,11]. None of these algorithms are able to reach all best known upper and lower bounds. The percentage of instances on which the best known upper bound is reached ranges from 46 % ([32,33]) to 90 % ([11]) on tested graphs. Such approaches can prove the optimality of a solution if the lowest upper bound happens to reach the highest lower bound. However, such proofs were only made on 21 instances out of 94, even when using all the bounds found by every methods in [12] simultaneously.

CP. A basic CSP model was proposed in [22]. For each node $u \in V$, this model includes a variable x_u whose domain is $D(x_u) = [1, \Delta(G) + 1]$. There is a disequality constraint for each edge of the graph, *i.e.*, $\forall\{u, v\} \in E, x_u \neq x_v$. The

objective is to minimize the sum of all variables, *i.e.*, $\sum_{u \in V} x_u$. This model was evaluated using Choco [14]. The results were not competitive with state-of-the-art approaches: solution times on rather easy instances were long [22].

B&B. In [22], a B&B approach is described. At each node of the search, a lower bound is obtained by computing a clique decomposition $\mathcal{C}$ of the subgraph induced by uncoloured vertices. However, instead of bounding the colour sum for each clique $C_i \in \mathcal{C}$ by $|C_i| \cdot (|C_i|+1)/2$ (as proposed in [24,33]) the authors bound it by the sum of the $|C_i|$ smallest available colours among the vertices of C_i. This new bound is tighter since it takes – to some degree – the availability of colours into account. Besides, each time a proper colouring is found, its corresponding dominant colouring is computed to improve the bound. This approach obtains better results than the basic CP model, but remains limited to small graphs.

SAT. Different SAT encodings for the SCP are described in [22]. They are experimentally compared (using different SAT solvers) with B&B and CP, on randomly generated graphs and on six DIMACS instances. On these instances, the proposed B&B and CP approaches are not competitive with the SAT portfolio ISAC [15], which obtains the best results. In this paper, we introduce new CP models which are competitive with these SAT models, and an ILP model which outperforms them on the six DIMACS instances for which results were given.

ILP. An ILP model was proposed in [31]. It associates a binary variable x_{uk} with every pair $(u,k) \in V \bigtimes [1, \Delta(G)+1]$, so that x_{uk} equals 1 iff node u uses colour k. The objective is to minimize the sum of the integers corresponding to used colours, *i.e.* $\sum_{u=1}^{|V|} \sum_{k=1}^{\Delta(G)+1} k \cdot x_{uk}$ so that each node $u \in V$ is assigned one colour, *i.e.*, $\sum_{k=1}^{\Delta(G)+1} x_{uk} = 1$, and for each edge $\{u,v\} \in E$, u and v have different colours, *i.e.*, $x_{uk} + x_{vk} \leq 1, \forall k \in [1, \Delta(G)+1]$. This model was evaluated with CPLEX, showing that it is very efficient for small graphs, but that the memory cost was too high for larger ones.

3 New CP models for the SCP

The CP model of [22] is very limited, as it only propagates binary difference constraints, and bounds the objective function with the sum of minimal values in domains. In this section, we propose and compare several improvements.

3.1 Initial domain reduction

Instead of using the same domain $[1, \Delta(G)+1]$ for all variables, we propose to tighten domains by using the following property:

Property 1. For every optimal sum colouring c of a graph $G = (V, E)$, we have $\forall v \in V, \mathrm{c}(v) \leq \deg(v) + 1$.

To prove this property, let us suppose that it does not hold for a given optimal colouring c of a graph G. It follows that there exists a vertex v in V such that $c(v) > \deg(v) + 1$. In such a case, there exists $x \in [1, \deg(v) + 1]$ such that every neighbour of v has a colour different from x (since v only has $\deg(v)$ neighbours). As a consequence, a better colouring than c can be obtained by colouring v with x instead of $c(v)$. Therefore, c is not optimal, which contradicts our initial claim.

Hence, we define $D(x_u) = [1, \deg(u) + 1]$ for each $u \in V$. This is a minor but natural improvement, with a negligible cost.

3.2 Dominant colourings

As pointed out in [22] and recalled in Section 2, colourings found during the search may be dominated, and can be improved simply by swapping colours. This makes the upper bound go down faster at a low computing cost. Besides, these swappings break symmetries by forbidding, thanks to the update of the upper bound, the computation of colourings that are dominated by the ones already found.

3.3 *AllDifferent* constraints

Instead of using only binary disequality constraints to prevent neighbour vertices from being assigned the same colour, we propose to use *AllDifferent* constraints. This may be done in several different ways. A first possibility is to compute a clique decomposition of the graph, as defined in Def. 1. In this case, we post a global *AllDifferent* constraint for each clique, and a binary disequality constraint for each edge such that no clique contains its two endpoints. A second possibility is to compute a set of maximal cliques such that, for each edge, there exists at least one clique that contains its two endpoints. In this case, we post a global *AllDifferent* constraint for each maximal clique. This introduces redundancies in *AllDifferent* constraints and may prune more values, but at a higher cost.

For both approaches, we may consider different heuristics to build cliques. Several tests (not reported due to lack of space) showed us that a simple greedy construction of maximal cliques yields a good tradeoff between the time spent building the cliques, the time spent to propagate *AllDifferent* constraints, and the reduction of the search space. More precisely, for each vertex of the graph, we build a maximal clique in a greedy way: starting from a clique that contains this vertex, we iteratively choose the vertex with the largest degree among the vertices that can correctly extend the clique, until no such vertex exists. We then post a global *AllDifferent* constraint for each of these maximal cliques.

3.4 Lower bound

The main drawback of the CP model proposed in [22] is due to the poor lower bound, which is the sum of minimal values in domains. This lower bound does not take into account the disequality constraints. A better lower bound is obtained

by using a clique decomposition $\mathcal{C}$, as proposed in the B&B approach of [22]: it is defined by the sum, for each clique C_i of $\mathcal{C}$, of the sum of the $|C_i|$ smallest values in the union of the domains of the variables associated with vertices of C_i. This new bound takes into account some disequality constraints (those between pairs of variables that belong to a same clique). However, the sum of the $|C_i|$ smallest values may be a bad approximation of the chromatic sum of the subgraph induced by C_i when variables of C_i have different domains. Let us consider for example a clique $C_i = \{a, b, c\}$ with $D(a) = \{1, 2, 3\}$ and $D(b) = D(c) = \{7, 8\}$. The sum of the 3 smallest values in $D(a) \cup D(b) \cup D(c)$ is $1 + 2 + 3 = 6$, whereas the chromatic sum of the subgraph induced by $\{a, b, c\}$ is $1 + 7 + 8 = 16$.

Combining *AllDifferent* and sum constraints. Better bounds may be computed by considering the global constraint that combines an *AllDifferent* constraint with a sum constraint on the same set of variables [2]. In particular, [2] proposed a bound consistency algorithm for this global constraint. The main idea relies on the notion of "blocks" of variables, defined in such a way that, for a given *AllDifferent* constraint, variables of the same block are interchangeable. Each block also has a set of values. An initial lower bound is computed in $\mathcal{O}(n \log(n))$, where n is the number of variables in the *AllDifferent* contraint. During the search, if a variable of a block is assigned with a value of this block, the lower bound is left unchanged, otherwise, it is updated in $\mathcal{O}(1)$.

Note that the clique decomposition used to compute lower bounds is different from the set of maximal cliques used to propagate disequality constraints as proposed in Section 3.3. In the clique decomposition used to compute lower bounds, some disequality constraints are missing (those between vertices that belong to different cliques). Hence, the propagation of the conjunctions of *AllDifferent* and sum global constraints (to compute the lower bound) does not ensure a proper colouring. It must be combined either with binary disequality constraints between neighbour vertices that belong to different cliques, or with *AllDifferent* constraints as defined in Section 3.3.

Computation of the clique decomposition. We may consider different heuristics to build clique decompositions, and different clique decompositions may lead to different bounds. To build a clique decomposition, we consider a basic greedy approach very similar to the one described in Section 3.3 to compute a set of maximal cliques: the only difference is that each time a vertex is added to a clique, it is removed from the graph so that it cannot be selected for another clique. A key point to obtain a good tradeoff between the time spent to compute bounds and the reduction of the search space lies in the frequency of the computation of a clique decomposition. In the B&B approach of [22], a new clique decomposition is computed at each node of the search tree, on the subgraph induced by uncoloured vertices. This allows to compute more accurate bounds, but clique decomposition computations are rather expensive. Hence, we propose to compute a clique decomposition only once, at the root of the search tree. This partition is then used at each node of the tree to compute a new bound.

Triggering of the bound computation. Trying too early to prune branches with bound computations often leads to a loss of efficiency: when only a few vertices are coloured, we do not have enough information as to how the colouring will turn out. The computed lower bound is thus too low to be of any use. To prevent unnecessary computations, we set a lower limit for the triggering of the bound computation: if the distance between the sum of currently assigned colours and the current upper bound amounts to more than *gap* %, we refrain from computing the lower bound. In addition to this lower triggering limit, we added an upper one: we refrain from using this bound when *unc* or less vertices are uncoloured. The reason behind this is that when only a few vertices are left uncoloured, it may be faster to explore what remains in this part of the search space rather than using a bound to try to prune this very small branch.

3.5 Experimental comparison

Experimental setup and benchmark. Programs are executed on an Intel® Xeon® CPU E5-2670 at 2.60 GHz processor, with 20,480 KB of cache memory and 4 GB of RAM. We consider 126 instances which are classically used for sum colouring, as in [31,12]. Some are from COLOR02/03/04[1], but most of them are DIMACS instances designed for the classical colouring problem[2]. The timeout was set to 24 hours. For each instance, the *reference solution* is the best known upper bound, either available in the literature (mainly [31,12]), or previously computed by one of our approaches. It gives an overview of the state of the art. Tables also give detailed results for a set of ten instances that we chose to highlight the peculiarities of each approach.

Configurations. We implemented CP models in Gecode (version 4.2.1) [29]. We cannot report results for all possible combinations of the different improvements described in Sections 3.1–3.4. We have chosen the following configurations:

– *Base*: Basic model, with binary disequality constraints, a lower bound defined as the sum of smallest values in variable domains, and bound consistency ensured.

– *AllDiff+Bound*: Model with *AllDifferent* constraints (as defined in Section 3.3), a lower bound defined by using a clique partition and computing for each clique C_i the sum of the $|C_i|$ smallest values in variables' domains, and bound consistency ensured. Parameters *gap* and *unc* are set to 20 % and 5, respectively. Experiments not detailed in this paper showed these values offer the best compromise.

– *AllDiff+Bound+Swap*: Same as *AllDiff+Bound*, but with colour swapping.

– *AllDiff+Bound+Swap+Dom*: Same as *AllDiff+Bound+Swap*, but with domain consistency instead of bound consistency.

– *AllDiff+SumBound+Swap*: Same as *AllDiff+Bound+Swap*, but lower bound computation is done by using the bound consistency algorithm of [2]. *gap* and *unc* are respectively set to 50 % and 0 (the best setting for this configuration).

[1] `http://mat.gsia.cmu.edu/COLOR02`

[2] `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/`

Table 1. Comparison of CP models. The first ten lines detail results for ten representative instances: best upper bound found within the time limit (UB), time needed to find UB (t_{UB}) and to prove optimality (t_{proof}), if optimality has been proven. The last three lines give the average distance between UB and the reference solution (in percentage), and the number of instances for which the reference solution has been found (# Ref. sol.), and optimality has been proven (# Optim. proofs) for the 126 instances.

		Base			*AllDiff+Bound*			*AllDiff+Bound +Swap*			*AllDiff+Bound +Swap+Dom*			*AllDiff+SumBound +Swap*		
Name	Ref. sol.	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}
DSJC250.5	3,210	3,598	24,823		3,580	16,179		3,591	840		**3,540**	**51,076**		3,577	9,505	
DSJC1000.1	8,991	10,323	84,001		10,339	29,570		10,328	3,340		10,315	66,581		**10,295**	**61,921**	
ash331GPIA	1,432	1,437	14,439		1,440	55,543		1,438	30,122		**1,437**	**6,755**		1,442	14,258	
le450_5b	1,350	1,518	46,685		1,510	63,361		1,509	8,819		1,487	6,439		**1,457**	**3,827**	
3-Insert_3	92	92	0		92	0	18,650	**92**	**0**	**17,838**	92	0	19,542	92	0	
qg.order60	109,800	109,800	404		109,800	592	593	109,800	203	203	**109,800**	**138**	**139**	109,800	218	218
r125.1	257	258	4		**257**	**1,472**	**4,193**	257	1,561	4,797	257	1,570	4,713	258	0	
inithx.i.3	1,986	1,988	17,571		1,988	84,043		**1,986**	**142**		1,986	463		1,987	4	
school1	2,674	3,646	77,581		3,646	52,114		**3,531**	**75,519**		3,644	48,434		3,648	1,072	
school1_nsh	2,392	3,067	1,329		3,139	42,737		3,031	1,047		3,031	2,604		**2,992**	**13,581**	
Average dist. (%)			5.57			5.58			5.32			5.44			5.34	
# Ref. sol.			43			45			49			48			45	
# Optim. proofs			5			11			11			11			8	

For these five configurations, the Branch and Bound (BAB) search engine was selected. As the goal is to minimize the sum of the variables, the value ordering heuristic chooses the smallest value. We have designed and compared different variable ordering heuristics (including well-known ones such as *Activity* and *wDeg*) and the best results are obtained with *minElim*, that chooses the variable that has the smallest value in its domain, and break ties by choosing the variable for which this smallest value would be removed from the fewest domains. *Luby* was used as a restart policy, with a scale of 500.

Results. Table 1 compares CP models on 10 representative instances, and then gives global results for the whole benchmark. *AllDiff+Bound* outperforms *Base* on 6 instances out of the 10, and adding colour swapping (*+Swap*) allows an overall improvement of bounds and generally faster proofs. Replacing bound consistency (in *AllDiff+Bound+Swap*) with domain consistency (in *AllDiff+Bound+Swap+Dom*) pays off on some instances, but degrades the solution process on some others. Finally, using the global constraint that combines a sum and an *AllDifferent* constraint improves the solution process on some instances, but also often degrades it. Actually, we noticed that, in many cases, all variables in the global constraint have very similar domains. Therefore, the bound computed for their sum is very close to the sum of the smallest values in the union of the domains.

As a conclusion, none of the proposed CP model appears to be competitive with state-of-the-art incomplete approaches, as the best model (*AllDiff+Bound+Swap*) is able to reach the reference solution for only 49 instances.

Making proofs with CP. None of the configurations considered above are good at proving optimality, as proofs were only made for 11 instances. This may be due to the variable ordering heuristic *minElim*, which aims at quickly finding

good solutions. Hence, we conducted experiments with a new CP configuration, designed to prioritize proof-making. It employs a hybrid restart policy, coupled with a scheduled heuristic change. In this configuration, we use the same setting as for *AllDiff+Bound+Swap+Dom* but we change the variable ordering heuristic during the search: as soon as the search endured 50 consecutive restarts without having improved the global upper bound, *minElim* is replaced by a heuristic that aims at proving optimality (it chooses the variable that has the highest number of uncoloured neighbours, and break ties by first choosing the variable that has the smallest value in its domain, and then the smallest current domain). When the variable ordering heuristic is changed, we also change the restart policy for a geometric policy, with a scale value of 100 and a base of 2.

Using this configuration, we are able to prove optimality for 15 instances instead of 11. Though this is an improvement of 36 %, this is still far from the state of the art. For example, using all known bounds computed with heuristic approaches, optimality was proven for 21 of the 94 instances considered in [12].

4 Integer Linear Programming

Two improvements introduced in the previous section for CP may be easily adapted to ILP. Firstly, initial domain reduction is enacted simply by declaring less variables and by removing difference constraints between nodes when the considered colour is not in the domains of both nodes. Secondly, *AllDifferent* constraints are modeled by adding the constraint $\sum_{v \in C_i} x_{vk} \leq 1$, for each clique C_i and each colour k (instead of binary disequalities).

Implementation. We used ILOG CPLEX (version 12.6.2) [4]. To help CPLEX to avoid running out of memory, the two following parameters were added: Depth-First Search was forced as a node selection strategy, and the cuts factor was set to 1.5. Previous experiments showed us that these parameters did not significantly lessen CPLEX's ability to solve the instances we use.

Experimental results. Table 2 compares results of the initial ILP model as proposed in [31], denoted *ILP*, with the ILP model that includes the two improvements, denoted *ILP+*. The most notable improvement is due to domain reduction, since reducing domains for ILP also removes variables and constraints. Overall, improvements allowed us to increase the number of optimality proofs from 61 to 65, and the number of times the reference solution has been found from 66 to 73. Besides, the number of memory outs goes down from 28 to 23.

When comparing ILP+ to CP, we note that they perform very differently. For four instances (`DSJC*` and `school*`), ILP+ ran out of memory. Therefore, the best solution found is far from the reference solution, and from the best solution found with CP models. For `qg.order60`, the best solution found by ILP is far from optimality, whereas all CP models are able to reach the optimum, with two of them even proving optimality. However, for the five remaining instances, ILP either

finds better solutions (ash331GPIA, le450_5b), proves optimality quicker (3-Insertions_3, r125.1), or proves optimality while CP cannot (inithx.i.3). When considering global results (on the whole benchmark), ILP+ is able to find reference solutions and to prove optimality much more often than CP, but the average distance to reference solutions is much larger, mostly because of the times it ran out of memory.

5 Combining CP and ILP

Experiments reported previously showed us that CP and ILP have complementary abilities: ILP is very efficient to solve small instances, but it runs out of memory for 23 instances; it never occurs with CP, but its solutions are often far from the optimum, despite a rather good average distance. We therefore propose to decompose the problem into smaller independent subproblems, and to combine CP and ILP to solve these subproblems. The goal is to identify a subset $K_r \subseteq V$ of nodes, for which we compute all proper colourings with CP. From each of these colourings, we derive an independent subproblem. Given the optimal sum colouring of each of these independent subproblems, we deduce the optimal solution of the original problem in a straightforward way. If the subproblems are small enough, they can be solved with ILP. Furthermore, when instances are well structured, we may choose the subset K_r so that, for each colouring of K_r, we obtain several independent subproblems (instead of a single one), even easier to solve with ILP. This idea is reminiscent of the approach called *Backtracking on Tree Decomposition* (BTD) [9].

In Section 5.1, we recall the basic principles of BTD and show how to use it to solve the SCP. In Section 5.2, we introduce a new decomposition, as well as a way to use it. This approach is called BFD, and can be employed to combine CP and ILP. Decomposition methods are experimentally compared in Section 5.3.

5.1 Tree decomposition

Definition 2. *[27] A* tree decomposition *of a graph (V, E) is a couple (K, T) where $T = (I, F)$ is a tree, and $K : I \to \mathcal{P}(V)$ is a function which associates a subset of variables $K_i \subseteq V$ (called a cluster) with every node $i \in I$ such that the following conditions are satisfied:* (i) $\cup_{i \in I} K_i = V$*;* (ii) *for each edge $(v_j, v_k) \in E$, there exists a node $i \in I$ such that $\{v_j, v_k\} \subseteq K_i$; and* (iii) *for all $i, j, k \in I$, if k is in a path from i to j in T, then $K_i \cap K_j \subseteq K_k$. The width of a tree decomposition is* $\max_{i \in I} |K_i| - 1$*. Intersections of neighbour clusters are called separators.*

BTD is a generic approach that exploits a tree decomposition of the constraint (hyper)graph of a CSP to identify independent subproblems which are solved separately. More precisely, given a tree decomposition (K, T) and a root node $r \in I$, BTD first assigns the variables of the root cluster K_r. Then, BTD recursively solves, for each child i of r, the independent subproblem that contains all variables occurring in the clusters associated with the subtree rooted in i. To avoid the repeated exploration of same parts of the search space, BTD records

structural (no)goods, that allow to reduce time complexity to $\mathcal{O}(nd^{w+1})$ with n the number of variables, d the maximum domain size and w the width of T. The space complexity is $\mathcal{O}(nsd^s)$ with s the size of the largest separator.

BTD may be used to solve optimization problems provided that the objective function is decomposable, *i.e.*, once all variables of a root cluster r are assigned, the optimal solution that extends this partial assignment may be obtained by computing separately, for each child of r, the optimal solution of the subproblem associated with this child [9,5]. In this case, we have to record *structural valued goods*, *i.e.*, pairs composed by an assignment of the variables of a separator and the optimal solution of the subproblem associated with this child for this assigment. To avoid solving to optimality a subproblem if it is obvious its optimal solution cannot be extended to a global solution, an upper bound is added to the subproblem. As soon as it is proved that the optimal solution of the subproblem cannot be lower than the upper bound, the solving of the subproblem is stopped.

BTD may be used to solve the SCP as its objective function is decomposable: given a tree decomposition of the graph to colour, once the nodes of a root cluster r are coloured, the best sum colouring that extends this partial colouring may be obtained by searching separately the best sum colouring of each child of r. Note that this is not the case, for example, of the classical colouring problem, as we cannot colour children separately when the goal is to minimize the number of used colours (as we must know the colours used by other clusters).

However, experiments on our 126 instances have shown us that many instances are poorly structured: when computing a tree decomposition with *MinFill* [16], 65 instances are not decomposed at all (*i.e.*, there is only one cluster, which contains all variables), and 103 instances have at least one cluster that contains more than 90 % of the variables. For these instances, BTD behaves poorly.

5.2 Backtracking with Flower Decomposition (BFD)

Even when the tree decomposition only contains one cluster, we may decompose it into two subsets (K_r and $V \setminus K_r$): we use CP to enumerate all proper colouring of K_r, and then, for each of these colourings, we use ILP to find the optimal sum colouring of $V \setminus K_r$ (given the colours assigned to K_r).

The idea of BFD is to exploit instance structure so that, for each colouring of K_r, we may split $V \setminus K_r$ into several subsets that may be solved to optimality with ILP independently. In other words, we propose to compute a tree decomposition with a height of 1, composed of a root cluster K_r, and a set of leaf clusters which are all children of K_r. The key point is to choose the nodes of K_r so that we obtain leaves that are small enough to be solved to optimality with ILP. To this end, we introduce a parameter l, that enforces a limit on the size of the leaves. More precisely, for each leaf cluster K_i, we ensure that $|K_i \setminus K_r| \leq l \times |V|$. Besides this hard constraint on the leaf sizes, we also aim at favoring small roots (as we have to enumerate all its proper colourings).

This flower decomposition is built as follows. We first build a tree decomposition $(K, T = (I, F))$ of the graph $G = (V, E)$ with *MinFill*. Let S be the set of all separators, *i.e.*, $S = \{K_{i_1} \cap K_{i_2} | \{i_1, i_2\} \in F\}$. Given a subset $S' \subseteq S$, we

define a flower decomposition whose root is the cluster $K'_r = \cup_{s \in S'} s$. The other clusters of the flower (the leaves) are defined by the connected components of the subgraph of G induced by $V \setminus K'_r$. Each leaf cluster K'_i is then extended by adding to it any vertex of K'_r adjacent to a vertex of K'_i. A similar process was employed in [10], where it was also demonstrated that it results in a correct tree decomposition. Of course, the quality of the obtained flower decomposition depends on the initial subset S'. The goal is to find the subset S' such that the resulting flower decomposition satisfies the size limit l on the leaf clusters while minimizing the size of K'_r. We use Gecode to solve this problem. As it is $\mathcal{NP}$-hard, we limit the CPU time for computing it to 15 minutes, and use the best flower decomposition computed within this time limit. If Gecode has not found any flower decomposition that satisfies the size limit l on the leaf clusters, we build a flower decomposition which only contains two clusters and such that the root cluster contains the $|V| \times (1 - l)$ vertices with largest degrees.

As with BTD, we also enforce a limit on the size of cluster separators. In the context of BTD, this is done by merging clusters whose separators contain too many variables. In our case, however, we keep the clusters as they are: the only effect of the limit is that no valued good is recorded on the separators which exceed the limit, since doing so would be very likely to consume a large amount of memory. Moreover, if a separator corresponds to the full root cluster, there is no need to record any valued good on it, since affectations on such a separator cannot be produced more than once.

5.3 Experimental evaluation

We compare two approaches, denoted BTD and BFD. BTD refers to the classical BTD approach. In this case, the tree decomposition is built using the *MinFill* algorithm and CP is used to solve subproblems: leaf clusters are solved with the Gecode configuration *AllDiff+Bound+Swap* (that uses restarts), whereas non-leaf clusters are solved with the same configuration, but without restarts, as we need to enumerate all solutions.

BFD refers to our new approach, based on a flower decomposition. CP (*AllDiff+Bound+Swap* without restarts) is used to enumerate the solutions of the non-leaf clusters and ILP+ to solve the subproblems induced by the leaves for each assignment of the separators. The maximal size for the separators is set to 30. We report results with two values for the "l" parameter (that limits the size of leaf clusters): 75 % (denoted BFD 75) and 90 % (denoted BFD 90).

Table 2 reports experimental results of BTD, BFD 90 and BFD 75. When looking at the detailed results on our ten representative instances, we note that they have complementary results: BTD is better than BFD 90 and BFD 75 on `DSJC250.5` and `school1`, BFD 90 is better on `ash331GPIA`, `3-Insertions_3`, `inithx.i.3`, `school1_nsh` and `r125.1`, and BFD 75 is better on `DSJC1000.1`, `le450_5b`, `qg.order60` and `r125.1`. For two of these instances (`r125.1` and `school1_nsh`), the best results, over all considered approaches, are actually obtained by BFD 90.

When looking at global results over the whole benchmark, BTD is able to find the reference solution for only 17 instances (instead of 46 and 48 for BFD 90

Table 2. Comparison of ILP, ILP+, BTD, BFD 90 and BFD 75. The first ten lines detail results for as many representative instances. "#M#" in t_{proof} means a memory out occurred. The last four lines give, for the 126 instances: the average distance from UB to the reference solution (in percentage of the reference solution); the number of instances for which the reference solution was found (# Ref. sol.); the number of instances for which optimality was proved (# Optim. proofs); the number of times search was aborted due to a lack of memory (# Out of memory).

		ILP			ILP+			BTD			BFD 90			BFD 75		
Name	Ref. sol.	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}	UB	t_{UB}	t_{proof}
DSJC250.5	3,210	4,587	795		4,587	2,457	#M#	**4,377**	**1**		4,855	86,400		15,918	0	#M#
DSJC1000.1	8,991	12,892	1	#M#	12,892	8,138	#M#	12,879	62		50,629	0	#M#	**12,467**	**86,400**	
ash331GPIA	1,432	1,458	7,066		**1,432**	**29,870**		1,767	13		1,448	86,400		1,528	1,894	
3-Insert._3	92	92	0	1	**92**	**0**	**0**	92	25	1,796	92	261	331	92	857	1,007
le450_5b	1,350	1,450	53,963		**1,398**	**22,554**		2,227	16,237		1,914	86,400		1,883	86,400	
qg.order60	109,800	216,000	0	#M#	116,520	86,393		110,453	4,193		115,259	86,400		**110,198**	**86,400**	
r125.1	257	**257**	**0**	**0**	**257**	**0**	**0**	257	0	1	**257**	**0**	**0**	**257**	**0**	**0**
inithx.i.3	1,986	2,523	1	#M#	**1,986**	**9**	**20**	2,010	5		1,986	106	686	6,560	10,907	
school1	2,674	5,723	7,962	#M#	5,769	1	#M#	**5,556**	**82,466**		19,480	0	#M#	19,480	0	#M#
school1_nsh	2,392	5,069	3,573		4,980	7,990	#M#	4,740	10,747		**2,539**	**86,400**		3,996	86,400	
Average dist. (%)			73.36			63.89			24.42			83.40			85.05	
# Ref. sol.			66			73			17			46			48	
# Optim. proofs			61			65			13			39			26	
# Out of memory			28			23			6			18			16	

and BFD 75), and it proves optimality for 13 instances only (instead of 39 and 26). Actually, as pointed out previously, most instances have no structure at all, or only a very poor structure, and BTD is generally outperformed on them by the CP approaches introduced in Section 3.

BFD 90 and BFD 75 are able to find reference solutions and to prove optimality for much more instances than BTD. Actually, even if the instance is not structured at all (*i.e.*, there is only one cluster in the tree decomposition, which happens 40 times for our 126 instances), BFD is still able to build a flower decomposition with one root cluster (that contains $|V| \cdot (1-l)$ nodes) and one leaf (that contains the remaining $|V| \cdot l$ variables). In this case, BFD often behaves much better than BTD. However, BFD also suffers from a relatively high average distance to reference solutions. Actually, on some instances, BFD spends a lot of time to enumerate colourings for the root cluster which cannot be extended to good solutions. However, for each of these colourings, BFD wastes a lot of time solving to optimality useless subproblems.

Comparing BFD 90 and BFD 75 proves that allowing larger leaf clusters increases the memory needs but also eases the computation of upper bounds, as it makes the root cluster smaller (less enumeration) and gives ILP a more global view of the problem, preventing it in some cases to spend too much time solving a useless subproblem to optimality. When comparing BFD with the CP approaches of Section 3, we note an increase in the number of proofs (39 and 26 instead of 11), but the average distance to the reference solution is larger. Compared with ILP+, BFD finds the reference solution less often, as with optimality proofs, but there also are less failures due to memory. Actually, BFD and ILP+ have complementary performance: BFD 90 (resp. BFD 75) performs strictly better than CPLEX on 21 (resp. 28) instances, and strictly worse on 91 (resp. 85) instances.

6 Portfolio approach

We have introduced different approaches for the SCP in the previous sections. Some of them are dominated, in the sense that, for each instance, there is always another approach that performs better on this instance (it finds the same solution quicker, or a better solution). This is the case of the CP configurations *Base* and *AllDiff+Bound*, as well as ILP, BTD and BFD 75. The five other approaches (namely, the three remaining CP configurations, BFD 90 and ILP+) are complementary, and we propose to combine them in a portfolio approach.

More precisely, given a solver portfolio [8,6], the per-instance algorithm selection problem [26] consists in selecting the solver of the portfolio which is expected to perform best on a given instance. Algorithm selection systems usually build machine learning models to forecast which solver should be used in a particular context. Using the predictions, one or more solvers from the portfolio may be selected to be run sequentially or in parallel. In our SCP context, solver performance is highly constrained by memory bandwidth, in particular for ILP. Therefore, we cannot simply run our different solvers in parallel, and we consider the case where exactly one solver is selected.

One of the most prominent and successful systems that employs this approach is SATzilla [34], which defined the state of the art in SAT solving for a number of years. Other application areas include constraint solving [25], the travelling salesperson problem [19], subgraph isomorphism [20] and AI planning [28]. The reader is referred to a survey [18] for additional information on algorithm selection.

The selection process is composed of two steps: given an SCP instance to be solved, we first extract features from instances; then, we run algorithm selection to choose a solver. Finally we run the selected solver on the instance.

Feature extraction. Given a graph $G = (V, E)$ for which we are looking for the chromatic sum, we compute the following features (a "*" denoting the use of the minimum, maximum, mean and standard deviation): number of nodes $|V|$ and edges $|E|$, degrees of the vertices in V*, size of connected components in G*, number of constraints and variables in the ILP+ model, number of *AllDifferent* constraints (arity of more than 2) in the *AllDiff+Bound* CP model, arity of these *AllDifferent* constraints*. We also added features computed from the largest connected component G' of G: density, theoretical upper and lower bounds of $\Sigma(G')$, number of clusters in the tree decomposition computed with *MinFill*. Moreover, this tree decomposition is used to compute a flower decomposition (with $l = 90$), which gives additional features: size of the root cluster, Cartesian product of the sizes of the domains in the root cluster, number of clusters, distance between the theoretical upper and lower bounds of the root cluster, density of the root cluster, density of leaf clusters*, number of proper variables in clusters*, separator density*, separator sizes*, the distance between theoretical upper and lower bounds on leaf clusters*, the number of binary variables* and constraints* in the ILP+ model associated with leaf clusters.

Table 3. Detailed results, for the virtual best solver and our portfolio approach. $t_{\text{feat.}}$ is the time needed to compute the features. For each method, "Algo" gives the chosen algorithm. Gec_1 denotes *AllDiff+Bound+Swap*, Gec_2 *AllDiff+Bound+Swap+Dom*, and Gec_3 *AllDiff+SumBound+Swap*. Times t_{UB} and t_{proof} for the portfolio include $t_{\text{feat.}}$.

		Virtual best solver				Portfolio approach				
Name	Ref. sol.	Algo	UB	t_{UB}	t_{proof}	$t_{\text{feat.}}$	Algo	UB	t_{UB}	t_{proof}
DSJC250.5	3,210	Gec_2	3,540	51,076		6	Gec_1	3,591	845	
DSJC1000.1	8,991	Gec_3	10,295	66,842		126	Gec_1	10,328	3,466	
ash331GPIA	1,432	ILP+	1,432	29,870		20	ILP+	1,432	29,890	
3-Insert._3	92	ILP+	92	0	0	0	ILP+	92	0	0
le450_5b	1,350	ILP+	1,398	22,554		4	ILP+	1,398	22,558	
qg.order60	109,800	Gec_2	109,800	138	139	4,866	Gec_1	109,800	5,070	5,070
r125.1	257	BFD	257	0	0	0	BFD	257	0	0
inithx.i.3	1,986	ILP+	1,986	9	20	8	BFD	1,986	114	694
school1	2,674	Gec_1	3,531	75,519		18	Gec_3	3,648	1,149	
school1_nsh	2,392	BFD	2,539	86,400		10	Gec_3	2,992	14,166	
Average dist. (%)			4.25					5.51		
# Ref. sol.			83					76		
# Optim. proofs			66					65		
# Out of memory			0					0		

Selection Model. We use LLAMA [17] to build our solver selection model. LLAMA supports the most common algorithm selection approaches used in the literature. We performed a set of preliminary experiments to determine the approach that works best here, *i.e.*, a pairwise regression approach with random forest regression. This approach trains a model that predicts the performance difference between every pair of solvers in the portfolio, similarly to what is done in [34]: if the first solver is better than the second, the difference is positive, otherwise negative. The solver with the highest cumulative performance difference (*i.e.*, the most positive difference over all other solvers) is chosen to be run. As this approach already gives very good performance, we did not tune the parameters of the random forest machine learning algorithm. It is possible that overall performance can be improved by doing so, and we make no claims that the particular solver selection approach we use in this paper cannot be improved.

Experimental results. We use leave-one-out cross-validation to determine the performance of our portfolio approach, as we only have 126 instances in our benchmark (which is not much for training a learning model): for each instance i, we train the selection model on all instances but i, and evaluate it on i.

The set of features is computed in 5.4 minutes in average, with 110 of our instances actually being under 5 minutes. Some instances, such as latin_square_10, DSJC1000.9 or flat1000_50_0 take a prohibitive amount of time when computing our set of features, mostly because of the two decompositions needed.

Table 3 shows us that our portfolio approach obtains results that are close to those of the Virtual Best Solver (VBS), which considers the best solver for

each instance separately. It often selects either the best solver, or a solver which behaves well. The VBS (resp. our portfolio approach) uses ILP+ for 67 (resp. 78) instances, BFD 90 for 10 (resp. 13) instances, *AllDiff+Bound+Swap* for 20 (resp. 11) instances, *AllDiff+Bound+Swap+Dom* for 14 (resp. 11) instances, and *AllDiff+SumBound+Swap* for 15 (resp. 13) instances. Even when the portfolio selects the best solver, the solving time is increased by the time needed to compute features (which may be large on some instances). Note that the time needed by the model to select a solver is negligible (0.15 seconds on average).

Our portfolio is able to prove optimality for more than half of the 126 instances. In [12], best upper and lower bounds are reported for a set of state-of-the-art heuristic approaches, on a subset of 92 instances. Using the best of these bounds (computed with different heuristic approaches), they can prove optimality for 21 of these instances, whereas our portfolio approach is able to prove optimality for 42 of these 92 instances, *i.e.*, twice more. Finally, our portfolio approach has improved (resp. reached) the best upper bounds reported in [12] for 2 (resp. 48) of the 92 instances: for `DSJR500.1` the new upper bound is 2,142 instead of 2,156, and for `le450_25b` it is 3,349 instead of 3,365.

7 Conclusion and future work

We proposed some improvements for solving the SCP with CP and ILP, and demonstrated that they have complementary advantages: ILP is efficient on small instances, but fails to solve large instances due to its large memory needs; CP never runs out of memory but is not able to compute as good solutions as ILP on small instances. We proposed a CP / ILP combination that may be used as a compromise between CP and ILP. Besides, since it makes use of tree decomposition, this combination is better than CP or ILP alone to solve some well-structured instances. We combined those methods in a portfolio approach which obtains results close to those of the virtual best solver. It has been able to prove optimality for more than half of the considered instances. It has also been able to improve the best known upper bounds for two instances.

In the future, the use of a dedicated decomposition algorithm might be studied, in order to build a flower decomposition from scratch rather than resorting to an initial tree decomposition. The goal would be to make it more straightforward to obtain a balanced decomposition. Being able to automatically fine-tune BFD's parameters (l as well as the maximal size of separators) could also prove useful.

A major drawback of BFD is that some subproblems are solved to optimality even if they are useless due to poor assignments in the root. An interesting improvement that we shall investigate in further research would be to prevent ILP from spending more that a set amount of time on a leaf cluster, and to ask for a new assignment on the root if necessary. It could be seen as another form of restarts, as seen in [1].

Acknowledgements This work has been supported by the ANR project SoLStiCe (ANR-13-BS02-0002-01).

References

1. Allouche, D., Givry, S., Katsirelos, G., Schiex, T., Zytnicki, M.: Anytime hybrid best-first search with tree decomposition for weighted csp. In: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming - Volume 9255. pp. 12–29 (2015)
2. Beldiceanu, N., Carlsson, M., Petit, T., Régin, J.C.: An o (nlog n) bound consistency algorithm for the conjunction of an alldifferent and an inequality between a sum of variables and a constant, and its generalization. In: ECAI. vol. 12, pp. 145–150 (2012)
3. Benlic, U., Hao, J.K.: A study of breakout local search for the minimum sum coloring problem. In: Simulated Evolution and Learning, pp. 128–137. Springer (2012)
4. CPLEX, I.: High-performance software for mathematical programming and optimization (2005)
5. De Givry, S., Schiex, T., Verfaillie, G.: Exploiting tree decomposition and soft local consistency in weighted csp. In: AAAI. vol. 6, pp. 1–6 (2006)
6. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
7. Helmar, A., Chiarandini, M.: A local search heuristic for chromatic sum. In: Proceedings of the 9th metaheuristics international conference. vol. 1101, pp. 161–170 (2011)
8. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
9. Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence 146, 43–75 (2003)
10. Jégou, P., Kanso, H., Terrioux, C.: An algorithmic framework for decomposing constraint networks. In: Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on. pp. 1–8. IEEE (2015)
11. Jin, Y., Hao, J.K.: Hybrid evolutionary search for the minimum sum coloring problem of graphs, accepted to information sciences feb 2016 (2015)
12. Jin, Y., Hamiez, J.P., Hao, J.K.: Algorithms for the minimum sum coloring problem: a review. arXiv preprint arXiv:1505.00449 (2015)
13. Jin, Y., Hao, J.K., Hamiez, J.P.: A memetic algorithm for the minimum sum coloring problem. Computers & Operations Research 43, 318–327 (2014)
14. Jussien, N., Rochart, G., Lorca, X.: Choco: an open source java constraint programming library. In: CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08). pp. 1–10 (2008)
15. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: Isac-instance-specific algorithm configuration. In: ECAI. vol. 215, pp. 751–756 (2010)
16. Kjaerulff, U.: Triangulation of graphs - algorithms giving small total state space. Tech. rep., Judex R.R. Aalborg., Denmark (1990)
17. Kotthoff, L.: LLAMA: Leveraging learning to automatically manage algorithms. Tech. Rep. arXiv:1306.1031, arXiv (Jun 2013), `http://arxiv.org/abs/1306.1031`
18. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine 35(3), 48–60 (2014)
19. Kotthoff, L., Kerschke, P., Hoos, H., Trautmann, H.: Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In: LION 9 (2015)
20. Kotthoff, L., Mccreesh, C., Solnon, C.: Portfolios of subgraph isomorphism algorithms. In: Learning and Intelligent OptimizatioN Conference (LION 10). Springer (2016)

21. Kubale, M.: Graph colorings, vol. 352. American Mathematical Soc. (2004)
22. Lecat, C., Li, C.M., Lucet, C., Li, Y.: Exact methods for the minimum sum coloring problem. In: DPCP-2015. pp. 61–69. Cork, Ireland, Iran (2015), https://hal.archives-ouvertes.fr/hal-01323741
23. Moukrim, A., Sghiouer, K., Lucet, C., Li, Y.: Upper and lower bounds for the minimum sum coloring problem, submitted for publication
24. Moukrim, A., Sghiouer, K., Lucet, C., Li, Y.: Lower bounds for the minimal sum coloring problem. Electronic Notes in Discrete Mathematics 36, 663–670 (2010)
25. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (Jan 2008)
26. Rice, J.R.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
27. Robertson, N., Seymour, P.: Graph minors II: Algorithmic aspects of tree-width. Algorithms 7, 309–322 (1986)
28. Seipp, J., Braun, M., Garimort, J., Helmert, M.: Learning portfolios of automatically tuned planners. In: ICAPS (2012)
29. Team, G.: Gecode: Generic constraint development environment, 2006 (2008)
30. Thomassen, C., Erdös, P., Alavi, Y., Malde, P.J., Schwenk, A.J.: Tight bounds on the chromatic sum of a connected graph. Journal of Graph Theory 13(3), 353–357 (1989)
31. Wang, Y., Hao, J.K., Glover, F., Lü, Z.: Solving the minimum sum coloring problem via binary quadratic programming. arXiv preprint arXiv:1304.5876 (2013)
32. Wu, Q., Hao, J.K.: An effective heuristic algorithm for sum coloring of graphs. Computers & Operations Research 39(7), 1593–1600 (2012)
33. Wu, Q., Hao, J.K.: Improved lower bounds for sum coloring via clique decomposition. arXiv preprint arXiv:1303.6761 (2013)
34. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) 32, 565–606 (2008)