Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size

Philippe Jégou, Hanan Kanso, and Cyril Terrioux

Aix-Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296
13397 Marseille Cedex 20 (France)
{philippe.jegou, hanan.kanso, cyril.terrioux}@lsis.org

Abstract. In this paper, we address two key aspects of solving methods based on tree-decomposition. First, we propose an algorithm computing decompositions that allows to bound the size of separators, which is a crucial parameter to limit the space complexity, and thus the feasibility of such methods. Moreover, we show how it is possible to dynamically modify the considered decomposition during the search. This dynamic modification can offer more freedom to the variable ordering heuristics. This also allows to better use the information gained during the search while controlling the size of the required memory.

1 Introduction

The solving methods of CSPs based on tree-decomposition have shown a theoretical significance because they guarantee complexity bounds in $O(exp(w))$ in time as well as in $O(exp(s))$ in space where w and s are parameters induced by the structural properties of the constraint network. When w is bounded by a constant, these methods ensure a polynomial runtime. Moreover, in practice, such approaches are quite justified by numerous real-world problems for which w is relatively small [1]. However, two major problems occur sometimes in practice. First, controlling the value s is not always guaranteed, especially for decomposition methods like $Min\text{-}Fill$ [2] which can be seen as the state of the art [3]. This sometimes makes this type of approach completely ineffective because this parameter is crucial in practice [4]. On the other hand, ensuring a time complexity in $O(exp(w))$ requires a traversal of the search space that imposes strong constraints on the variable assignment ordering, which can lead to a strong deterioration of practical efficiency.

To answer the question of memory, we propose a new configurable algorithm for computing decompositions. It takes as an input a parameter S to compute decompositions that guarantee separator sizes at most S. Its time complexity is less than that of $Min\text{-}Fill$ and it offers performances which are widely better in practice (around 1,000 times faster on a large set of benchmarks). This algorithm fits perfectly into the framework proposed in [5] and can then be considered as a refinement of the heuristics proposed in this framework. The second part of the paper proposes a framework to dynamically change the decomposition during the search, enabling to offer more freedom to heuristics while continuing exploiting decompositions. This approach relies on the fact that, to be efficient in practice, the solving methods must take into account the context of the search and the knowledge gained gradually during the search. This is done by

CSP solvers using adaptive heuristics (e.g. [6, 7]) and by CDCL SAT solvers (e.g. [8]) through clause learning and restart techniques. In the case of decomposition methods, the fundamental difficulty is linked to the variable ordering imposed by the decomposition. To overcome this difficulty, we propose to adapt dynamically the decomposition by merging clusters during the search. Such an approach have been introduced in [9] but mainly from a theoretical viewpoint. Thus, we show here how it is feasible. In addition, we extend it by integrating restarts techniques as proposed in [10]. Moreover, we describe how to dynamically change the decomposition, taking advantage of the knowledge acquired during the search while proposing to keep a bound on the size of separators all along the search. The last part of this paper presents an experimental analysis on a large set of instances, to assess the practical value of this approach.

In Section 2, we recall notions about solving methods based on tree-decompositions while in Section 3, we present the computation of tree-decompositions taking into account the size of separators. Section 4 introduces a variant of the algorithm BTD able to adapt the decomposition during search while Section 5 presents experiments that assess the relevance of this approach, before concluding.

2 Preliminaries

The *Constraint Satisfaction Problem* (CSP) provides a strong framework to formulate problems in computer science [11]. An instance of a finite CSP is given by a triple (X, D, C), with $X = \{x_1, \ldots, x_n\}$ a set of n variables, $D = \{d_{x_1}, \ldots, d_{x_n}\}$ a set of finite domains, and $C = \{c_1, \ldots, c_e\}$ a set of e constraints. Each constraint c_i is a pair $(S(c_i), R(c_i))$, where $S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ defines the *scope* of c_i, and $R(c_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ is its *compatibility relation*. The *arity* of c_i is $|S(c_i)|$. If the arity of each constraint is two, the instance is a *binary* CSP. The structure of a constraint network (other name of a CSP) is given by a hypergraph (a graph for a binary CSP), called the *constraint (hyper)graph*, whose vertices correspond to variables while edges correspond to the scopes of the constraints. To simplify notations, we denote the hypergraph $(X, \{S(c_1), \ldots S(c_e)\})$ by (X, C). An assignment on a subset of X is called *consistent* if all the constraints are satisfied. Checking whether a CSP has a *solution* (i.e. a consistent assignment of X) is well known to be NP-complete. So, many works have been done to improve the solving in practice such as algorithms exploiting heuristics, constraint learning, non-chronological backtracking or filtering-based algorithms. Nevertheless, the complexity of these approaches remains exponential, at least in $O(n.d^n)$ where d is the maximum size of domains. To circumvent this theoretical intractability, other approaches have been proposed. Some of them rely on a structural tractable class [12] based on the notion of *tree-decomposition of graphs* [13].

Definition 1 *A* tree-decomposition *of a graph $G = (X, C)$ is a pair (E, T) with $T = (I, F)$ a tree (I is the set of nodes and F the set of edges of T) and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of T and satisfies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, k \in I$, if k is in a path from i to j in T, then $E_i \cap E_j \subseteq E_k$. The width of a tree-decomposition (E, T) is equal to $max_{i \in I}|E_i| - 1$. The* tree-width *w of G is the minimal width over all the tree-decompositions of G.*

This notion is only defined for graphs but can be considered for a hypergraph by exploiting its *2-section*[1]. Their primary advantage is related to their theoretical time complexity in d^{w+1} [3] while their space complexity is in d^s where s is the maximum size of intersections (called *separators* in the sequel) between clusters. Thus, these methods can be efficient on large instances of small tree-width as it is the case for example for well known optimization problems of radio frequency allocations [15]. These methods run in two steps: (1) computing a tree-decomposition, and (2) solve the instance exploiting the decomposition. Since computing optimal decompositions (i.e. of width w) is NP-hard [16], in practice, the first step generally computes tree-decompositions whose width is $w^+ \geq w$, that is an approximation of the tree-width. In this context, $Min\text{-}Fill$ [2] appears as the best compromise between the computation time ($O(n^3)$) and the quality of the obtained decompositions. It can be considered as the state of the art for such algorithms [3], even if, for graphs with more of tens of thousands of vertices, it may be unusable in practice.

However, the computed decompositions are not necessarily really suitable from a solving viewpoint [17, 18]. First, $Min\text{-}Fill$ does not take into account explicitly the topological properties of the considered graph which can make the solving inefficient. For example, the obtained decompositions may contain disconnected clusters [18]. Secondly, $Min\text{-}Fill$ can generate decompositions such that s is often close to w^+. Indeed, in order to minimize the width, $Min\text{-}Fill$ produces clusters with few proper vertices (i.e. vertices belonging to the cluster but not to its parent cluster in the tree-decomposition) or even only one proper vertex. This explains why s is often close to w^+. This can lead to a prohibitive cost for space memory. Thus, the minimization of s is crucial to be efficient in practice [17].

Secondly, to guarantee the time complexity in d^{w^+}, efficient structural methods such as BTD [19] use an ordering for the assignment of the variables which is partially determined by the considered decomposition. When $Min\text{-}Fill$ is used, this freedom is even more restricted because of the limited number of proper vertices in the clusters. But we know that to have an efficient search, it is desirable to have maximum freedom when choosing the next variable to assign.

To circumvent these difficulties, several approaches are possible. A first approach is to have a decomposition with small separators, while having larger clusters thereby releasing the constraints on the ordering [17]. Another possibility is to exploit restarts like in [10]. This approach works by restarting the search from a first variable which does not necessarily belong to the previous root cluster. This leads, while retaining the same decomposition (except for the root cluster), to give more freedom to the ordering and its relevance has been shown experimentally. Another possibility is to dynamically change the decomposition during the search while maintaining guarantees for time complexity. This approach was proposed in [9], but mainly on a theoretical level. It consists in expanding the cluster size by merging some neighboring clusters. However, its relevance has never been demonstrated. Moreover, as defined in [9], it is only guided by structural criteria, without taking into account explicitly the state of search, and the knowledge gained during the solving. Note that exploiting the structure of instances dynamically

[1] The 2-section of a hypergraph (X, C) is the graph (X, C') where $C' = \{\{x, y\} | \exists c \in C, \{x, y\} \subseteq c\}$ [14].

Algorithm 1: $H\text{-}TD\text{-}WT$

Input: A graph $G = (X, C)$
Output: A set of clusters $E_0, \dots E_m$ of a tree-decomposition of G

1 Choose a first cluster E_0 in G
2 $X' \leftarrow E_0$
3 Let $X_1, \dots X_k$ be the connected components of $G[X \backslash E_0]$
4 $F \leftarrow \{X_1, \dots X_k\}$
5 **while** $F \neq \emptyset$ **do** `/* find new cluster` E_i `*/`
6 Delete X_i from F
7 Let $V_i \subseteq X'$ be the neighborhood of X_i in G
8 Find a subset $X_i'' \subseteq X_i$ such that there is at least one vertex $v \in V_i$ such that $N(v, X_i) \subseteq X_i''$
9 $E_i \leftarrow X_i'' \cup V_i$
10 $X' \leftarrow X' \cup X_i''$
11 Let $X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}$ be the connected components of $G[X_i \backslash E_i]$
12 $F \leftarrow F \cup \{X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}\}$

has already been proposed for SAT in [20], but without guarantee for the complexity bounds, contrary to what we offer here.

To propose new alternative ways, we introduce in the following, first an algorithm which computes decompositions with clusters of bounded size. Secondly, we present a new solving algorithm based on BTD allowing to dynamically adapt the decompositions during the search, using information obtained from the beginning of the solving.

3 Decomposition controlling separators

3.1 A general framework to compute specific tree-decompositions

In this part, we recall the framework *H-TD-WT* (*Heuristic Tree-Decomposition Without Triangulation* [5]) that computes a tree-decomposition of the graph $G = (X, C)$ without triangulation in polynomial time, more precisely in $O(n(n + e))$. Like $Min\text{-}Fill$, no warranty about the optimality of the computed width is given. However, it allows to compute decompositions depending on the features we want to fulfill. Notably, different parameterizations are conceivable depending on the wanted criteria for the obtained tree-decompositions. For example, these criteria may be related to w^+ and/or s or the connectivity of clusters [18]. By designing such a framework, we have many goals. First, in order to manage dynamically the decompositions during the solving, efficient decomposition algorithms are needed from a theoretical and practical viewpoint. Second, the complexity of these algorithms should be at most in $O(n(n + e))$ to be more efficient than $Min\text{-}Fill$. To do so, the time-consuming step of triangulation performed by $Min\text{-}Fill$ must be avoided. Beyond that, limiting the maximum size of the separators (i.e. intersection between clusters), as well as the size of clusters, is also crucial.

The first step of *H-TD-WT* (line 1 in Algorithm 1) computes a first cluster, denoted E_0, thanks to a heuristic. X' which denotes the set of already considered vertices is initialized to E_0 (line 2). We denote $X_1, X_2, \dots X_k$ the connected components of the subgraph $G[X \backslash E_0]$ induced by the deletion in G of vertices of E_0[2]. Each one of these

[2] For any $Y \subseteq X$, the subgraph $G[Y]$ of $G = (X, C)$ induced by Y is the graph (Y, C_Y) where $C_Y = \{\{x, y\} \in C | x, y \in Y\}$.

sets X_i is inserted in a queue F (line 4). For each element X_i deleted from F (line 6), V_i denotes the set of vertices of X' which are adjacent to at least one vertex of X_i (line 7). One can note that V_i is a separator in the graph G since removing V_i from G makes G disconnected (X_i being disconnected from the rest of G). We then consider the subgraph of G induced by V_i and X_i, that is $G[V_i \cup X_i]$. The next step (line 8) can be parameterized. It looks for a subset of vertices $X_i'' \subseteq X_i$ such that $X_i'' \cup V_i$ will be a new cluster E_i of the decomposition. This can be ensured if there is at least one vertex v of V_i s.t. all its neighbors in X_i appear in X_i''. More precisely, if $N(v, X_i) = \{x \in X_i : \{v, x\} \in C\}$, we must ensure that $\exists v, N(v, X_i) \subseteq X_i''$. We then define a new cluster $E_i = X_i'' \cup V_i$ (line 10). This process is repeated until the queue is empty. In [5], this framework implements several heuristics. The first (denoted H_1), tries to minimize the size of the clusters while the second (H_2) guarantees that clusters will be connected (see [18]). The third heuristic (H_3) aims to identify independent parts of the graph and to separate them as soon as possible using a breadth-first search starting from the vertices of V_i. The fourth heuristic (H_4), which introduces the one we will present in this contribution, aims to limit the size of the separators of the decomposition. To do so, it considers a parameter S which represents the maximum allowed size for a separator. This heuristic adds new vertices to the next cluster E_i similarly to H_3. Nevertheless, the heuristic stops progressing through levels at $l = L$ when $G[X_i \backslash E_{i_L}]$ does not contain any connected component with separator's size greater than S.

3.2 A heuristic for controlling separators

We hereby introduce a new heuristic (denoted H_5) controlling the separator size. It aims to refine the heuristic H_4 by detecting more separators of size at most S. If H_4 stops adding vertices when it arrives to a level where all separators are of size at most S, H_5 may stop earlier. If at level l, the separator associated to one of the connected components has at most size S, the separator will be taken into account and included in the obtained tree-decomposition. In fact, when the separator is detected the corresponding connected component is added to the queue in order to be managed later. Hence, the computation of the current cluster continues only on the remaining part of X_i after the removal of the connected component having a suitable separator. Consider the example given in Figure 1(a). We first show the computation of E_1, the second cluster (after E_0) during the first pass through the loop and we set S to 2. We consider then the set $V_1 = \{x, y, z\}$. The vertices of the first level are then a, b and c. There is no subset of $\{a, b, c\}$ of size at most 2 that induces a separator of the graph. The next level that is visited contains the vertices d, e, f and g, At this level, we obtain two minimal separators, $\{d, e, f\}$ and $\{f, g\}$. If H_4 is used, the search would continue. However with H_5, the search is modified because of the detection and the exploitation of the separator $\{f, g\}$. Since $\{f, g\}$ is of size 2, the induced connected component containing the vertices i, j and m is removed and added to the queue F (line 8). Hence, the search continues only in the remaining part of the connected component X_1 which includes the vertices h, k, l, and n. The next level only contains the vertex h which is then a separator of size 1. Therefore, the search stops and a new cluster E_1 is created with: $E_1 = \{x, y, z, a, b, c, d, e, f, g, h\}$ and $X_1'' = \{a, b, c, d, e, f, g, h\}$. We obtain then a

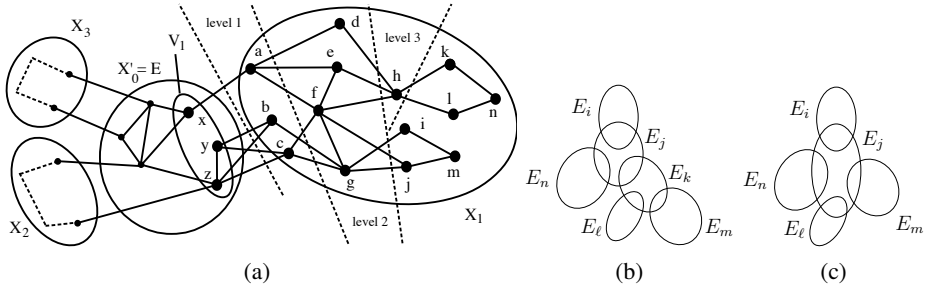


Fig. 1. View of $H5$ (a), a set of clusters of the decomposition before merging E_k with E_j (b) and after (c).

new connected component $X_{1_1} = \{k, l, n\}$ that is added to F. Note that the instantiation of H-TD-WT by H_5 is integrated in line 8. The conditions required by the approach are thus respected. In particular, a new subset $X_i'' \subseteq X_i$ is created where there exists at least one vertex $v \in V_i$ with $N(v, X_i) \subseteq X_i''$. With this in mind and the proof given in [5], the validity of H_5 is ensured.

Theorem 1 H_5 *computes the clusters of a tree-decomposition.*

Also, the analysis of its complexity is similar to the one given in [5].

Theorem 2 *The time complexity of the algorithm* H_5 *is* $O(n(n+e))$.

4 The dynamic decomposition

4.1 Context

The thesis that we defend in this paper is that changing the decomposition dynamically during the solving allows to adapt the decomposition to the nature of the instance to solve. The decomposition is modified according to the knowledge acquired during the solving, especially the one related to the semantics of the problem. The approach can be thus classified among the *adaptive* methods. These methods make choices depending on the current state of the problem as well as previous states. In practice, they have shown their benefit (like in [6, 21, 7]) w.r.t. conventional methods. For example, with conflict-driven variable ordering heuristics, the most problematic variables are identified during the search thanks to learned information from the part of the problem already explored. Hence, this allows to consider the identified variables earlier in the search and so to solve the problem efficiently. Nevertheless, in the case of solving methods based on tree-decompositions like BTD, the variable ordering is partially induced by the used decomposition. In other words, the next chosen variable should be allowed by the tree-decomposition. For this reason, the structural methods suffer from the restrictions imposed by the tree-decomposition with regard to the variable ordering. In order to circumvent this problem, we propose to adapt the tree-decomposition during the solving by merging dynamically some clusters.

Algorithm 2: BTD-MAC+Merge (**InOut**: $P = (X, D, C)$: CSP; **In**: Σ: sequence of decisions, E_i: Cluster, V_{E_i}: set of variables; **InOut**: G: set of goods, N: set of nogoods)

```
1  if V_{E_i} = ∅ then
2      result ← true
3      S ← Sons(E_i)
4      while result ∉ {false, unknown} and S ≠ ∅ do
5          Choose a cluster E_j ∈ S
6          S ← S\{E_j}
7          if Pos(Σ)[E_i ∩ E_j] is a nogood in N then result ← false
8          else
9              if Pos(Σ)[E_i ∩ E_j] is not a good of E_i w.r.t. E_j in G then
10                 result ←BTD(P,Σ,E_j,E_j\(E_i ∩ E_j),G,N)
11                 if result = true then Record Pos(Σ)[E_i ∩ E_j] as good of E_i w.r.t. E_j in G
12                 else
13                     if result = false then Record Pos(Σ)[E_i ∩ E_j] as nogood of E_i w.r.t. E_j in N
14                     else
15                         if merge then Merge E_j with one of its sons
16                         if not restart then
17                             S ← S ∪ {E_j}
18                             result ← true
19     return result
20 else
21     Choose a variable x ∈ V_{E_i}
22     Choose a value v ∈ d_x
23     d_x ← d_x\{v}
24     if AC (P,Σ ∪ ⟨x = v⟩) then result ← BTD(P, Σ∪ ⟨x = v⟩, E_i, V_{E_i}\{x}, G, N)
25     else result ← false
26     if result = false then
27         if restart then
28             Record nld-nogoods w.r.t. the decision sequence (Σ ∪ ⟨x ≠ v⟩)[E_i]
29             return unknown
30         else
31             if AC (P,Σ ∪ ⟨x ≠ v⟩) then return BTD(P,Σ ∪ ⟨x ≠ v⟩,E_i,V_{E_i},G,N)
32             else return false
33     else return result
```

4.2 The algorithm BTD-MAC+RST+Merge

The algorithm BTD-MAC+RST+Merge (see Algorithm 3) is an adaptation of the algorithm BTD-MAC+RST [10] in order to take into account the dynamic merging. For both algorithms, the use of a tree-decomposition having a root cluster E_r induces a partial variable ordering. If E_j is the current cluster, the freedom of variable ordering is limited to either choose among the unassigned variables of the cluster E_j or to choose the next cluster among the children of E_j when all its variables are assigned. Both algorithms compute first a tree-decomposition before the beginning of the solving. The main difference between them is that BTD-MAC+RST uses the initial decomposition during the solving (the same set of clusters but the root can change) while BTD-MAC+RST+Merge updates dynamically the tree-decomposition depending on the needs of the solving. Therefore, more different partial orderings can be exploited during the solving. The operation that permits to change the decomposition in this context is the *merging*. It consists in putting together the variables of two different clusters to create one cluster. Figure 1(c) shows the merging of two clusters E_j and E_k of the

Algorithm 3: BTD-MAC+RST+Merge (**In**: $P = (X, D, C)$: CSP)

```
1 G ← ∅; N ← ∅
2 repeat
3 |   Choose a root cluster E_r
4 |   result ← BTD-MAC+Merge (P,∅,E_r,E_r,G,N)
5 until result ≠ unknown
6 return result
```

decomposition of Figure 1(b). Note that, the children of the merged cluster become the children of the cluster resulting from the merging. For instance, E_ℓ and E_m, the children of E_k in Figure 1(b), become the children of E_j in Figure 1(c). Consider D the initial decomposition and D' the obtained decomposition after the merging. Any variable ordering allowed by D is also allowed by D'. Nonetheless, by exploiting D' we obtain more possible orderings than by using D. We then deduce that the merging preserves the orders initially allowed but also permits more freedom. Deciding to merge or not two clusters is only conditioned by the information learned during the solving. Also, the behavior of BTD-MAC+RST+Merge ranges from BTD-MAC+RST with a variable ordering partially imposed by the tree-decomposition (if no merging occurs) to MAC (for *Maintaining arc consistency* [22]) with a totally free variable ordering (if after several mergings, the decomposition contains only one cluster). So, the advantage of this new algorithm is its ability to find the right compromise thanks to learned information during the solving.

BTD-MAC+RST+Merge exploits the algorithm BTD-MAC+Merge (see Algorithm 2). The difference between BTD-MAC+Merge and BTD-MAC+NG [10] is located at lines 14-18. Initially, the sequence of decisions Σ as well as the set of goods G and nogoods N are empty. BTD-MAC+Merge (like BTD-MAC+NG) begins the solving by assigning consistently the variables of the root cluster E_r before moving to one of its children. By exploiting the new cluster E_i, only unassigned variables of E_i are assigned. In other words, only the variables of E_i that do not belong to $E_i \cap E_{p(i)}$ (where $E_{p(i)}$ is the parent cluster of E_i) are assigned. In order to solve each cluster, both algorithms rely on MAC (lines 21-26 and 31-33). During the solving MAC can make two kind of decisions: positive decisions $x_i = v_i$ which assign the value v_i to the variable x_i and negative decisions $x_i \neq v_i$ which ensure that x_i cannot be assigned with v_i. Let us consider $\Sigma = \langle \delta_1, ..., \delta_i \rangle$ as the current decision sequence where each δ_j may be either a positive or a negative decision. A new positive decision $x_{i+1} = v_{i+1}$ is chosen and AC filtering is achieved (line 24). If no dead-end occurs, the search goes on by choosing a new positive decision (line 24). Otherwise, the value v_{i+1} is deleted from the domain $d_{x_{i+1}}$, and an AC filtering is realized (line 31). If a dead-end occurs again, we backtrack and change the last positive decision $x_\ell = v_\ell$ to $x_\ell \neq v_\ell$. When the cluster E_i is chosen as the next cluster, the next positive decision involves a variable of the current cluster E_i. Since $E_i \cap E_{p(i)}$ is a separator and all its variables are already assigned, only the domains of future variables in $Desc(E_i)$ are impacted by the AC filtering (where $Desc(E_i)$ is the set of variables belonging to the union of the descendants E_k of E_i). When the variables of the cluster E_i are consistently assigned (line 1), each subproblem rooted in each child cluster E_j of E_i is solved (line 10). More

precisely, for a child E_j and a current decision sequence Σ, it attempts to solve the subproblem rooted in E_j induced by $Pos(\Sigma)[E_i \cap E_j]$ (where $Pos(\Sigma)[E_i \cap E_j]$ is the set of positive decisions involving the variables of $E_i \cap E_j$ in Σ). Once this subproblem solved, if a solution has been found by consistently extending Σ on $Desc(E_j)$ then $Pos(\Sigma)[E_i \cap E_j]$ is recorded as a *structural good*[3] (line 11). Otherwise if no solution exists, $Pos(\Sigma)[E_i \cap E_j]$ is recorded as a *structural nogood* (line 13). These structural (no)goods are exploited later during the solving to avoid redundancies (lines 7 and 9).

Regarding the restarts, they are managed like in [10]. If a restart occurs (line 27), the search is suspended and some reduced nld-nogoods [23] are recorded in order to avoid exploring again parts of the search tree already explored. The efficiency of restarts relies on the acquired knowledge and the exploitation of stored information via the structural (no)goods and the reduced nld-nogoods [23]. The restart condition may involve global parameters (related to the whole problem) or local parameters (related to the current cluster) or both. Lines 15-18 deal with the dynamic merging performed by BTD-MAC+Merge. The dynamic merging, as explained above in Section 2, aims to relax the constraints imposed by the decomposition on the variable ordering. Deciding to merge clusters or not depends on the current state of the problem as well as on all or part of its intermediate states. If no merging is required ($merge$ returns $false$), the search continues normally. On the contrary, if merging is judged relevant for the solving (e.g. by making possible the early assignment of important variables), $merge$ returns $true$ and BTD-MAC+Merge changes the current decomposition by merging the current cluster E_j with one of its children (line 15). To do so, all the assigned variables of $E_j \backslash (E_{p(j)} \cap E_j)$ are unassigned and the reduced nld-nogoods are recorded (line 28) as with conventional restarts. Once the search backtracks to the parent cluster, the merging is performed. In this case (line 16), either the backtracking through clusters continues if $restart$ returns $true$ or the search is resumed by exploring a child of the parent cluster. Note that, it is not mandatory to unassign the variables of the current cluster before merging. However, this would permit to consider the newly added variables in the cluster earlier in the search. BTD-MAC+Merge can be parameterized by a merging heuristic (we propose one in Section 5).

4.3 Theoretical foundations

We now show the validity of our approach. First, we prove that the merging operation does not influence the validity of the structural (no)goods and the reduced nld-nogoods.

Proposition 1 *Let (E', T') be the tree-decomposition of the graph G obtained from the decomposition (E, T) of G after merging the cluster E_y with the cluster E_x (where E_y is a child of E_x in (E, T)). The recorded structural (no)goods of E_i w.r.t. E_j ($E_j \neq E_y$) and the recorded reduced nld-nogoods for (E, T) remain valid for (E', T').*

Proof: Consider Δ a structural good of E_i w.r.t. its child E_j recorded for (E, T). Knowing that E_j differs from E_y, the subproblem of P rooted in E_j in (E, T) is identical to the subproblem of P rooted in E_j in (E', T'). Hence, if Δ can be consistently

[3] A *structural good* (resp. *nogood*) of E_i w.r.t. E_j (with E_j a child of E_i) is a consistent assignment of $E_i \cap E_j$ which can (resp. cannot) be consistently extended on $Desc(E_j)$ [19].

extended on the first subproblem then it can also be extended on the second one. Consequently, Δ is a structural good of E_i w.r.t. E_j recorded for (E', T'). The reasoning is similar for a structural nogood. A reduced nld-nogood Δ is a nogood[4] whatever the considered tree-decomposition. We should only verify then if a nld-nogood Δ is valid for the decomposition (E', T'), that is to say that there exists a cluster of E' including all the variables of Δ. By construction, there exists necessarily a cluster E_k of (E, T) covering Δ. If $E_k \neq E_x$ and $E_k \neq E_y$ then $E_k \in E'$. Otherwise, after merging, we have $E_k \subset E_x$ and $E_x \in E'$. Therefore, in both cases, the variables of Δ are all covered by one cluster of E' and Δ is valid for (E', T'). □

Then, we prove the validity of our algorithm.

Theorem 3 *BTD-MAC+RST+Merge is sound, complete and terminates.*

Proof: Consider first BTD-MAC+Merge which differs from BTD-MAC+NG by exploiting the merging. Assume that we obtain (E', T') from the decomposition (E, T) by merging two clusters. Let Σ_f be the sequence of decisions for which $merge$ becomes $true$. Some reduced nld-nogoods are then recorded and the search backtracks to the parent cluster E_i of the current cluster E_j. Thus, we obtain the sequence Σ'_f that corresponds to the sequence of decisions Σ_f restricted to the variables of the clusters present in the branch going from the root cluster E_r to E_i. BTD-MAC+Merge continues the search from E_i with Σ'_f by exploiting the decomposition (E', T'). The cluster resulting from the merging can be the next visited cluster or can be visited later. The search tree explored by BTD-MAC+Merge between its first call with an empty sequence of decisions and the sequence of decisions Σ_f is the same as the one developed by BTD-MAC+NG under the same circumstances on the decomposition (E, T). Also, after the merging, the search tree developed by BTD-MAC+Merge between the sequence of decisions Σ'_f and its termination is identical to the one developed by BTD-MAC+NG under the same circumstances on the decomposition (E', T'). We know that BTD-MAC+NG is complete (if no restart occurs), correct and terminates [10]. Also, according to the proposition 1, the structural (no)goods and the reduced nld-nogoods recorded for (E, T) remain valid for the new decomposition (E', T'). So, the correction, the termination and the completeness of the algorithm are not endangered. Furthermore, recording reduced nld-nogoods at each restart prevents from exploring a part of the search space already explored. Hence, BTD-MAC+Merge is complete (if no restart occurs), correct and terminates. In addition, when many merging operations are performed, the same reasoning can be applied for every merging by splitting the search tree. Note that, restarts stop the search without changing the fact that if a solution exists in the search space visited by BTD-MAC+Merge, BTD-MAC+Merge would find it. As BTD-MAC+RST+Merge only performs several calls to BTD-MAC+Merge, it is sound. Regarding the completeness, if the call to BTD-MAC+Merge is not stopped by a restart (what is necessarily the case of the last call to BTD-MAC+Merge if BTD-MAC+RST+Merge terminates), the completeness of BTD-MAC+Merge implies the

[4] Given a CSP $P = (X, D, C)$ and a sequence of decisions Σ, Δ is a *nogood* of P if $P_{|\Delta}$ has no solution where $P_{|\Delta}$ is the CSP (X, D', C) with $D' = (d'_{x_1}, \ldots, d'_{x_n})$ and for each positive decision $x_i = v_i$, $d'_{x_i} = \{v_i\}$ and for each negative decision $x_i \neq v_i$, $d'_{x_i} = d_{x_i} \backslash \{v_i\}$. If x_i does not appear in Δ then $d'_{x_i} = d_{x_i}$ [23].

one of BTD-MAC+RST+Merge. Furthermore, recording reduced nld-nogoods at each restart prevents from exploring a part of the search space already explored by a previous call to BTD-MAC+Merge. It ensues that, over successive calls to BTD-MAC+Merge, one has to explore a more and more reduced part of the search space. Hence, the termination and completeness of BTD-MAC+RST+Merge are ensured by the unlimited nogood recording achieved by the different calls to BTD-MAC+Merge and by the termination and the completeness of BTD-MAC+Merge. □

Finally, we give its time and space complexities.

Theorem 4 *BTD-MAC+RST+Merge has a time complexity in* $O(R.((n.s^2.e.\log(d) + w'^+.N).d^{w'^+ +2} + n.(w'^+)^2.d))$ *and a space complexity in* $O(n.s.d^s + w'^+.(d+N))$ *with* w'^+ *the width of the final obtained decomposition,* s *the size of the largest intersection* $E_i \cap E_j$ *of the initial decomposition,* R *the number of restarts and* N *the number of recorded reduced nld-nogoods.*

Proof: BTD-MAC+RST has a time complexity in $O(((n.s^2.e.\log(d) + w^+.N).d^{w^+ +2} + n.(w^+)^2.d).R)$ and a space complexity in $O(n.s.d^s + w^+.(d+N))$ [10]. Regarding BTD-MAC+RST+Merge, applying the merging operations implies that the size of the clusters may increase. Hence, the theoretical complexities are expressed in terms of w'^+ instead of w^+. The merging operations do not create new clusters but, on the contrary, some are removed. Thus, the maximum size of separators in the initial decomposition represents an upper bound on the size of separators. Therefore, the time and space complexities of the elements related to the size of separators are not modified. Regarding the reduced nld-nogoods recorded after a merging operation, even though they induce additional time and space costs, these costs are already taken into account by the costs of recorded reduced nld-nogoods of restarts. Thereby, the time complexity is in $O(R.((n.s^2.e.\log(d) + w'^+.N).d^{w'^+ +2} + n.(w'^+)^2.d))$ and the space complexity is in $O(n.s.d^s + w'^+.(d+N))$. □

Note that, we can limit the increase of the width of the obtained tree-decomposition regarding the width of the initial decomposition by using a suitable merging heuristic.

5 Experiments

In this section, we first present our experimental protocol before assessing H_5 w.r.t. the solving and comparing the dynamic decomposition with the static one and MAC+RST.

5.1 Experimental protocol

Regarding the exploited tree-decompositions, we consider $Min\text{-}Fill$ (as the state of the art heuristic known for its good tree-width approximation), H_2 (which guarantees the connectivity of the clusters), H_3 (whose clusters have many children) and H_5 (which controls the size of the separators of the decomposition). We discard H_4 since it computes less elaborate decompositions than H_5. For H_2, the clusters are computed by

Table 1. Number of solved instances and runtime for BTD-MAC, BTD-MAC+RST, BTD-MAC+Merge and BTD-MAC+RST+Merge depending on the exploited decompositions.

Algorithm	$Min\text{-}Fill$		H_2		H_3		H_5 ($S = 50$)	
	#solved	time	#solved	time	#solved	time	#solved	time
BTD-MAC	1,344	43,272	1,405	31,429	1,466	31,469	1,469	33,564
BTD-MAC+RST	1,495	43,557	1,518	35,042	1,529	30,187	1,543	33,049
BTD-MAC+Merge	1,481	42,505	1,518	37,440	1,523	35,101	1,534	34,048
BTD-MAC+RST+Merge	1,544	41,622	1,547	32,547	1,554	33,736	1,567	34,432

choosing the vertices of the considered connected component by decreasing degree order until the cluster becomes connected (i.e. the heuristic $NV2$ of [18]). For H_5, the decomposition is exploited with different bounds on the size of separators.

The dynamic decomposition exploits a merging heuristic. This latter relies on the advices of the variable ordering heuristic to assess the need of clusters merging. More precisely, given a current cluster E_j, each time we choose the next variable to assign in E_j, we test whether the variable ordering heuristic would choose another variable if it has the opportunity to choose among the unassigned variables of $(\bigcup_{E_k \in Children(E_j)} E_k) \cup E_j$. If a variable of a child E_k of E_j is preferred to a variable of E_j, a counter related to E_k is incremented. When the counter related to E_k reaches the limit L (namely 100 in these experiments), the cluster E_k is merged with its parent E_j.

Regarding the solving, we consider BTD-MAC and BTD-MAC+RST as reference structural methods based on a static decomposition, BTD-MAC+RST+Merge and BTD-MAC+Merge (i.e. BTD-MAC+RST+Merge without restarts) for the methods exploiting dynamic decompositions, and MAC+RST as the reference conventional enumerative method. We choose as root cluster the cluster having the maximum ratio number of constraints to its size minus one. The arc-consistency is enforced by $AC3^{rm}$ for the preprocessing and $AC8^{rm}$ for the solving [24]. We use the heuristic dom/wdeg [6] to choose the next variable to assign and the geometric restart policy based on the number of performed backtracks with a ratio of 1.1 and an initial number of backtracks of 100.

All the algorithms are implemented in C++ in our own library. The experiments were performed on blade servers running Linux Ubuntu 14.04 each with two Intel Xeon processors E5-2609 v2 2.5GHz and 32 GB of memory. We consider 1,859 CSP instances (the same as [10, 5]) from the CSP 2008 competition[5]. Regarding the instances selection, we have excluded the instances having a trivial tree-decomposition (e.g. instances having a complete constraint graph) and the instances having global constraints (because global constraints are not taken into account yet by our CSP library). The solving is performed with a timeout of 15 minutes (including the computation of the decomposition).

5.2 H_5 vs other decompositions

Table 1 provides the number of solved instances and the cumulative runtime of each mentioned algorithm with each considered tree-decomposition. First, we compare the

[5] See http://www.cril.univ-artois.fr/CPAI08.

Table 2. Runtime for BTD-MAC, BTD-MAC+RST, BTD-MAC+Merge and BTD-MAC+RST+Merge depending on the exploited decompositions for the 1,234 instances solved by all the algorithms.

Algorithm	$Min\text{-}Fill$	H_2	H_3	H_5 ($S = 50$)
BTD-MAC	34,669	18,018	18,951	18,243
BTD-MAC+RST	24,026	17,233	17,758	16,288
BTD-MAC+Merge	25,238	17,575	18,753	17,837
BTD-MAC+RST+Merge	23,832	16,803	17,602	15,718

decomposition heuristics w.r.t. the solving efficiency of BTD-MAC. Regarding the number of solved instances, BTD-MAC with H_5 (with $S = 50$) solves the largest number of instances (namely 1,469) while with $Min\text{-}Fill$, it solves the least number of instances (namely 1,344). Clearly, decompositions aiming to minimize the width are not necessarily the most efficient w.r.t. the solving. Other parameters have more impact on the solving such as the connectivity of the clusters (with H_2), the number of children of a cluster (with H_3) as well as the maximum size of separators (with H_5). Note that these results are consistent with ones of [18, 5]. Besides, of course, with H_5, the efficiency of the solving depends on the chosen value for S. For instance, BTD-MAC solves more instances with $S = 15$ (namely 1,514). Choosing $S = 50$ is more interesting for exploiting dynamic decompositions.

Regarding the runtime, for a fair comparison, we consider, in Table 2, the 1,234 instances solved by all the algorithms. Again, BTD-MAC obtains the best results with H_5 (and H_2) and the worst ones with $Min\text{-}Fill$. We must point out that computing tree-decompositions thanks to H_i ($i = 2, 3, 5$) is significantly faster than with $Min\text{-}Fill$. For instance, H_5 (with $S = 50$) only requires 7 s to compute the tree-decompositions for all the 1,234 instances while $Min\text{-}Fill$ needs 7,582 s.

Note that the benefits of H_5 observed here for BTD-MAC are valid whatever the variant of BTD we use as shown in Tables 1-4.

5.3 Dynamic decompositions vs static decompositions

First, if we compare BTD-MAC to BTD-MAC+Merge (resp. BTD-MAC+RST to BTD-MAC+RST+Merge) in Tables 1 and 2, whatever the used decomposition, we can observe that the methods exploiting dynamic decompositions solve more instances than their corresponding variants exploiting a static decomposition while their runtime is either similar or better. This clearly highlights the benefits of dynamically merging clusters during the solving.

Nevertheless, the concept of merging may also be performed statically, as advocated in [17], after having computed first a tree-decomposition thanks to any algorithm (e.g. $Min\text{-}Fill$, H_2, H_3 and even H_5). Tables 3-4 provide the corresponding results for BTD-MAC(+RST). Here, we limit the size of the separators by merging with its parent any cluster whose separator with its parent exceeds a given value (namely 15 in Tables 3-4). We can note that, regarding the number of solved instances, BTD-MAC+Merge is significantly better than BTD-MAC while BTD-MAC+RST+Merge is comparable or slightly better than BTD-MAC+RST. Again, the exploitation of dynamic decomposition

Table 3. Number of solved instances and runtime for BTD-MAC and BTD-MAC+RST depending on the exploited decompositions with a static merging limiting the size of separators to 15.

Algorithm	$Min\text{-}Fill$		H_2		H_3		H_5 ($S = 50$)	
	#solved	time	#solved	time	#solved	time	#solved	time
BTD-MAC	1,450	45,988	1,493	35,871	1,504	31,612	1,511	32,097
BTD-MAC+RST	1,537	41,722	1,549	33,328	1,553	33,164	1,564	33,145

Table 4. Runtime for BTD-MAC and BTD-MAC+RST depending on the exploited decompositions for the 1,234 instances solved by all the algorithms.

Algorithm	$Min\text{-}Fill$	H_2	H_3	H_5 ($S = 50$)
BTD-MAC	32,641	17,914	17,813	16,503
BTD-MAC+RST	24,456	17,514	17,050	16,235

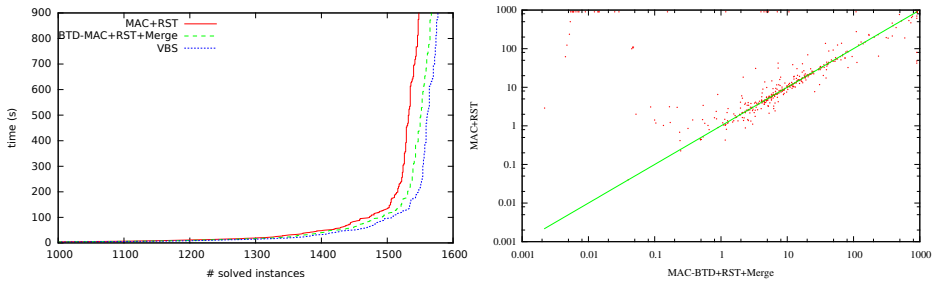
leads to obtain one of the best results. Beyond, by dynamically merging some clusters during the solving, we adapt the decomposition depending on some semantic knowledge about the instance whereas the static merging relies only on structural criteria and requires to choose a limit for the separator size, what may be a difficult task.

Finally, we can remark that BTD-MAC+RST and BTD-MAC+Merge are relatively close w.r.t. the number of solved instances or the runtime. This can be explained by the choice of a new root cluster when BTD-MAC+RST restarts, what can be seen as a light form of dynamicity for the decomposition. Moreover, the exploitation of both restarts and dynamic decompositions is really relevant since BTD-MAC+RST+Merge outperforms both BTD-MAC+RST and BTD-MAC+Merge. At the end, we can note that BTD-MAC+RST+Merge with H_5 obtain the best results whatever the decomposition or the solving algorithm we use.

5.4 BTD-MAC+RST+Merge versus MAC+RST

We now compare BTD-MAC+RST+Merge versus MAC+RST w.r.t. the solving efficiency. For this, we consider here $S = 50$. Figure 2(a) presents the cumulative number of solved instances for MAC-BTD+RST+Merge, MAC+RST and VBS (i.e. the Virtual Best Solver among the two algorithms). First, BTD-MAC+RST+Merge solves more instances than MAC+RST (1,567 instances against 1,548). Then, we can note that the behavior of BTD-MAC+RST+Merge is closer to one of VBS than one of MAC+RST, what clearly shows that BTD-MAC+RST+Merge performs better than MAC+RST.

Now we focus our observations on the hardest instances. Among the 1,859 considered instances, some of them are easily solved by MAC+RST (e.g. 284 instances are solved in backtrack-free manner). Exploiting structural methods like BTD or its variants for solving such instances is not necessarily relevant. So, we exploit here the number of nodes developed by MAC+RST as a hardness criterion. An instance is considered as difficult if the number of nodes developed by MAC+RST is greater than $100n$ (with n the number of variables). By so doing, we have 577 instances considered as difficult. Figure 2(b) provides a runtime comparison for MAC-BTD+RST+Merge and MAC+RST for these instances. Globally, we can observe that MAC-BTD+RST+Merge



(a) (b)

Fig. 2. (a) The cumulative number of solved instances for MAC-BTD+RST+Merge with H_5 ($S = 50$), MAC+RST and VBS, (b) runtime comparison for MAC-BTD+RST+Merge and MAC+RST for the 577 difficult instances.

and MAC+RST have a similar behavior on a large part of these instances. Indeed, for about 60% of the instances, the runtime gap between the two methods is less than 10%. However, for the remaining instances, MAC-BTD+RST+Merge often outperforms MAC+RST. For 16% of them, MAC-BTD+RST+Merge is at least 10 times faster than MAC+RST while MAC+RST performs 10 times faster for only 1%. Finally, the exploitation of the structure plays here a central role. Indeed, we can note that 86% of the instances unsolved by MAC+RST but solved by BTD-MAC+RST+Merge are structured instances having a ratio $n/(w+1)$ greater than 5.

6 Conclusion

In this paper, we proposed two complementary contributions. On the one hand, we presented a new algorithm for computing tree-decompositions (namely H_5) allowing us to bound the size of separators, which is a crucial parameter for the practical efficiency of structural solving methods like BTD. Its time complexity is better than the one of $Min\text{-}Fill$ and it runs about 1,000 times faster than $Min\text{-}Fill$ on a large set of instances. On the other hand, we described a non straightforward extension of BTD, namely BTD-MAC+RST+Merge, which has the ability of adapting the tree-decomposition by dynamically merging some clusters depending on the semantics of the instance and the knowledge acquired during the solving. By so doing, our method exploits more flexible variable orderings and may correct some drawbacks of the initial tree-decomposition whose computation relies only on structural parameters. In practice, we showed that BTD-MAC+RST+Merge outperforms BTD-MAC+RST whatever the exploited decompositions. Moreover, its use jointly with H_5 leads to obtain the best results.

For future investigations, first, other merging heuristics are possible by exploiting different information related to the semantics learned during the solving. Then, the fact that H-TD-WT is much more faster than $Min\text{-}Fill$ allows to compute more elaborate decompositions during the solving and on restarts. Beyond, more difficult problems can be tackled (e.g. optimization, counting or compilation).

References

1. S. de Givry, T. Schiex, and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *Proceedings of AAAI*, pages 22–27, 2006.
2. D. J. Rose. A graph theoretic study of the numerical solution of sparse positive definite systems of linear equations. In *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.
3. R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.
4. D. Allouche, S. de Givry, and T. Schiex. Towards Parallel Non Serial Dynamic Programming for Solving Hard Weighted CSP. In *Proceedings of CP*, pages 53–60, 2010.
5. P. Jégou, H. Kanso, and C. Terrioux. An Algorithmic Framework for Decomposing Constraint Networks. In *Proceedings of ICTAI*, pages 1–8, 2015.
6. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI*, pages 146–150, 2004.
7. P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP*, pages 557–571, 2004.
8. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proceedings of SAT*, pages 502–518, 2003.
9. P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *Proceedings of CP*, pages 364–378, 2007.
10. P. Jégou and C. Terrioux. Combining Restarts, Nogoods and Decompositions for Solving CSPs. In *Proceedings of ECAI*, pages 465–470, 2014.
11. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.
12. G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:243–282, 2000.
13. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Algorithms*, 7:309–322, 1986.
14. C. Berge. *Graphs and Hypergraphs*. Elsevier, 1973.
15. C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.
16. S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embeddings in a k-tree. *SIAM Journal of Disc. Math.*, 8:277–284, 1987.
17. P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of CP*, pages 777–781, 2005.
18. P. Jégou and C. Terrioux. Tree-decompositions with connected clusters for solving constraint networks. In *Proceedings of CP*, pages 407–423, 2014.
19. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
20. W. Li and P. van Beek. Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition. In *Proceedings of ICTAI*, pages 542–548, 2004.
21. L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proceedings of CP-AI-OR*, pages 228–243, 2012.
22. D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of ECAI*, pages 125–129, 1994.
23. C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4):147–167, 2007.
24. C. Lecoutre, C. Likitvivatanavong, S. Shannon, R. Yap, and Y. Zhang. Maintaining Arc Consistency with Multiple Residues. *Constraint Programming Letters*, 2:3–19, 2008.