A generalized Cyclic-Clustering Approach for Solving Structured CSPs

Cédric Pinto and Cyril Terrioux
LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
cedric.pinto@lsis.org, cyril.terrioux@univ-cezanne.fr

Abstract

We propose a new method for solving structured CSPs which generalizes and improves the Cyclic-Clustering approach [1]. First, the cutset and the tree-decomposition of the constraint network, which are used for taking advantage of the CSP structure, are computed independently of the notion of triangulated induced subgraph. Then, unlike Cyclic-Clustering, our method can try to solve the tree-decomposition part of the problem without having assigned all the variables of the cutset. Regarding the solving of the tree-decomposition part, we use the BTD method [2] like in [3]. As BTD records and exploits structural (no)goods, we provide some conditions which make possible the use of structural (no)goods recorded during previous calls of BTD and we implement them in a dedicated version of BTD. By so doing, from a theoretical viewpoint, we can provide a theoretical time complexity bound related to parameters of the cutset and the tree-decomposition and, from a practical viewpoint, we expect to detect failures earlier and to avoid more redundancies in the search. This practical interest is assessed in some preliminary experiments.

1 Preliminaries

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems, in particular, many academic or real problems (e.g. graph coloring, planning, frequency assignment problems, ...). A finite constraint satisfaction problem \((X, D, C, R)\) is defined as a set of variables \(X = \{x_1, \ldots, x_n\}\), a set of domains \(D = \{d_1, \ldots, d_m\}\) (the domain \(d_i\) contains all the possible values for the variable \(x_i\)), and a set \(C\) of constraints. A constraint \(c_i \in C\) on an ordered subset of variables, \(c_i = (x_{i_1}, \ldots, x_{i_k})\) is defined by an associated relation \(r_{c_i} \in R\) of allowed combinations of values for the variables in \(c_i\) \((r_{c_i} \subseteq d_{i_1} \times \ldots \times d_{i_k})\). Note that we take the same notation for the constraint \(c_i\) and its scope. Let \(Y = \{x_1, \ldots, x_k\}\) be a subset of \(X\). An assignment \(A\) on \(Y\) is a tuple \(\langle v_1, \ldots, v_k\rangle\) of \(d_1 \times \ldots \times d_k\). We also write \(A\) in the form \(\{x_1 \leftrightarrow v_1, \ldots, x_i \leftrightarrow v_i, \ldots\}\). Then we denote \(A_1 \subseteq A_2\) if the assignment \(A_2\) is an extension of \(A_1\) (i.e. we have \(A_1 = \{x_1 \leftrightarrow v_1, \ldots, x_i \leftrightarrow v_i, \ldots\}\) and \(A_2 = \{x_1 \leftrightarrow v_1, \ldots, x_i \leftrightarrow v_i, \ldots, x_j \leftrightarrow v_j\}\) with \(j \geq 0\)). An assignment \(A\) on \(Y\) satisfies a constraint \(c \in C\) s.t. \(c \subseteq Y\) if \(A[c] = r_{c}\) with \(A[c]\) the restriction of \(A\) to the variables involved in \(c\). \(A\) is said consistent if it satisfies each constraint \(c \subseteq Y\). A solution is an assignment of each variable which satisfies all the constraints. Determining if a solution exists is an NP-complete problem. We denote \(Sol(\mathcal{P})\) the set of solutions of the CSP \(\mathcal{P}\). In the following, for sake of simplicity, we only consider binary CSPs (i.e. CSPs whose each constraint involves exactly two variables). Of course, this work can be extended to non-binary CSPs.

The usual methods for solving CSPs (e.g. Forward Checking [4] or MAC [5]) are based on backtracking search. This approach, often efficient in practice, has an exponential theoretical time complexity in \(O(m^{d^n})\) (denoted \(O(exp(n))\)) for an instance having \(n\) variables and \(m\) constraints and whose largest domain has \(d\) values. Several works have been developed to improve this theoretical complexity bound thanks to particular features of the instance. Generally, they exploit some structural properties of the CSP. The structure of a CSP \((X, D, C, R)\) can be represented by the graph \((X, C)\), called the constraint graph. In this context, the tree-decomposition notion [6] plays a central role. A tree-decomposition of a graph \(G = (X, C)\) is a pair \((T, E)\) where \(T = (I, F)\) is a tree with nodes \(I\) and edges \(F\) and \(E = \{E_i : i \in I\}\) a family of subsets of \(X\), s.t. each subset (called cluster) \(E_i\) is a node of \(T\) and verifies: (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 $E_i \cap E_j \subseteq E_k$. We will denote $S_j$ the separator $E_i \cap E_j$ between the clusters $E_i$ and $E_j$ such that $E_j$ is a son of $E_i$, and $\text{Desc}(E_j)$ the set of variables belonging to the descent of the cluster $E_i$ rooted in $E_j$. The width $w$ of a tree-decomposition $(E, T)$ is equal to $\max_{ij \in I}|E_j| - 1$. The tree-width $w^*$ of $G$ is the minimal width over all the tree-decompositions of $G$. On the one hand, it leads to one of the best known theoretical time complexity bounds, namely $O(exp(w^* + 1))$ with $w^*$ the tree-width. Different methods (e.g. [7, 2]) have been proposed to reach this bound. They aim to cluster variables so that the cluster arrangement is a tree.

From a theoretical viewpoint, reach the best theoretical complexity bound requires to compute an optimal tree-decomposition (i.e. a tree-decomposition with a minimum width), which is an NP-hard problem [8]. In practice, it is clear that solving an NP-hard problem as a preliminary step of the solving of an NP-complete problem is not reasonable. So heuristic methods are generally used. They often provide a relevant approximation of an optimal tree-decomposition when the constraint graph has a small tree-width. Methods like BTD [2] are then well-suited for solving such problems. In contrast, when the constraint graph does not have a small tree-width, heuristic methods may often produce a poor approximation of an optimal tree-decomposition. In such a case, instead of running a structural method on a tree-decomposition with an excessive width, exploiting a method like Cyclic-Clustering [1] may be more interesting and more adapted. Cyclic-Clustering relies on a subset of vertices, called a cutset of the graph $(X, C)$, such that the graph $(X - V, \{x, y\} \in C$ s.t. $x, y \in X - V)$ induced by $X - V$ is triangulated (i.e. it has no cycle of length greater than 3 without an edge joining two non consecutive vertices in the cycle). The triangulated part of the constraint graph corresponds to a tree-decomposition. For instance, Figure 1(a) presents a graph having 19 vertices. The set $\{y_1, y_2\}$ forms a cutset of this graph s.t. the induced graph involving the vertices $x_1, \ldots, x_7$ is triangulated, which corresponds to a tree-decomposition with 7 clusters $E_1, \ldots, E_7$. We have $S_2 = E_1 \cap E_2 = \{x_3\}$, $\text{Desc}(E_1) = \{x_1, x_2, x_3, x_4, x_5\}$ and $\text{Desc}(E_2) = \{x_3, x_5\}$.

In [3], two implementations of Cyclic-Clustering, called CC-BTD1 and CC-BTD2 are proposed. They solve the cutset part of the problem with a classical enumerative algorithm and the triangulated part with BTD. CC-BTD2 differs from CC-BTD1 in calling BTD before solving the cutset part. By so doing, the nogoods recorded during this preliminary call can be exploited in the following calls of BTD. Unfortunately, the Cyclic-Clustering approach has some limits. For instance, informations recorded during the search are not fully exploited to avoid redundant parts of the search space. Moreover, the triangulated part must be computed thanks to the notion of Triangulated Induced Subgraph (TIS).

In this paper, we propose a generalization of CC-BTD, called CC-BTD-gen. Like the Cyclic-Clustering approach, CC-BTD-gen relies on a cutset and a tree-decomposition. Yet, it uses a tree-decomposition computed thanks to any method and so not necessarily related to the TIS notion, unlike Cyclic-Clustering. Regarding the solving, CC-BTD-gen exploits a specialized version of BTD which allows it to exploit some part of (no)goods recorded in previous calls to BTD, what leads to avoid more redundancies in practice. Finally, we have noted that CC-BTD assigns consistently all the variables of the cutset before solving the triangulated part even if after having assigned some of them, the triangulated part has no solution. So, in order to avoid this drawback, CC-BTD-gen can call BTD after having assigned consistently some variables of the cutset. If the subproblem associated to the tree-decomposition has a solution, the search keeps on the remaining variables of the cutset. Otherwise a backtrack occurs. In both cases, some (no)goods are recorded and may be exploited later.

The paper is organized as follows. Section 2 presents the theoretical framework of CC-BTD-gen while section 3 describes the CC-BTD-gen algorithm. Then section 4 deals with the experimental results. Finally, we conclude and discuss about related or future works in section 5.

2 Theoretical framework

In this section, we describe the theoretical framework required to present formally CC-BTD-gen. This framework is presented in a general way before focusing in the next section on a special case where $Y$ will be the cutset and $X - Y$ the variables belonging to the associated tree-decomposition used by CC-BTD-gen. In the following, we consider a CSP $P = (X, D, C, R)$. First, we define the notion of subproblem induced by a subset $Y$ of variables.

Definition 1 Let $Y \subseteq X$ be a subset of variables. The CSP induced by $Y$ is the CSP $(Y, D_Y, C_Y, R_Y)$ where $D_Y = \{d_i \in D | x_i \in Y\}$, $C_Y = \{c_{ij} = \{x_i, x_j\} \in C | x_i, x_j \in Y\}$ and $R_Y = \{r_{c_{ij}} \in R | c_{ij} \subseteq C_Y\}$.
In the next definitions and properties, we consider the following notations:

- \( Y_1, Y_2, Y \) and \( Z \) will be subsets of \( X \) such that \( Y_1 \subseteq Y, Y_2 \subseteq Y, Y \subseteq X \) and \( Z \subseteq X - Y \).
- \( A_1, A_2 \) and \( A \) will be assignments respectively on \( Y_1, Y_2 \) and \( Y \).
- \( TD \) will be the considered tree-decomposition of the CSP \( \mathcal{P}(X - Y) \).

Now, we propose a limited (but sufficient) definition of the deletion of some values by Forward-Checking (FC [4]).

**Definition 2** The resulting filtering of an assignment \( A \) performed by FC is the operation which consists in deleting the values from the domain \( d_i \) of each unassigned variable \( x_i \), which become incompatible with respect to at least a constraint \( \{ x_i, y \} \) where \( y \) is an assigned variable in \( A \). More formally, \( d_i^A = \{ v \in d_i | \forall c = \{ x_i, y \} \in C, (v, w) \in r_c \text{ with } w \text{ the value assigned to } y \text{ in } A \} \).

In other words, \( d_i^A \) is the current domain of the unassigned variable \( x_i \) obtained thanks to the filtering achieved after each assignment of a variable in the assignment \( A \). We then define the set of deleted values by the filtering.

**Definition 3** Let \( Y \subseteq X \) be such that \( |Y| = k \) and \( A = \{ x_1 \leftarrow v_1, ..., x_k \leftarrow v_k \} \) an assignment on \( Y \). The set of deleted values of \( \mathcal{P}(X - Y) \) by the filtering related to \( A \) is \( \mathcal{F}_A(X - Y) = \{ (x_i, v) \in (X - Y) \times (d_i - d_i^A) \} \).

Next, we refine the definition 1 by introducing the notion of filtered subproblem.

**Definition 4** The filtered subproblem \( \mathcal{P}_A(X - Y) \) refers to the induced CSP \( (X - Y, D_{X - Y}^A, C_{X - Y}, R_{X - Y}^A) \) with \( D_{X - Y}^A = \{ d_i^t | x_i \in X - Y \} \) and \( R_{X - Y}^A = \{ r_c^A = r_c \cap (d_i^A \times d_i^A) | c = \{ x_j, x_k \} \in C_{X - Y} \text{ and } r_c \in R \} \).

We can note that the filtering of FC does not change the structure defined by the constraint graph of a problem.

**Property 1** A tree-decomposition of \( \mathcal{P}(X - Y) \) is a tree-decomposition of \( \mathcal{P}_{A_1}(X - Y) \) and conversely.

**Proof:** A tree-decomposition is only dependent of the considered graph. As \( \mathcal{P}(X - Y) \) and \( \mathcal{P}_{A_1}(X - Y) \) have the same constraint graph (namely \( (X - Y, C_{X - Y}) \)), they have the same tree-decompositions. \( \square \)

Henceforth, thanks to the following property, we aim to measure the effect of a filtering on the domains and relations of a given problem.

**Property 2** If \( \mathcal{F}_{A_1}(Z) \subseteq \mathcal{F}_{A_2}(Z) \), then \( \forall z_i \in Z, d_i^{A_2} \subseteq d_i^{A_1} \) and \( \forall c, j, k \in C_Z, r_{c,j,k}^{A_2} \subseteq r_{c,j,k}^{A_1} \).

**Proof:** For each pair \( (z_i, v_i) \in \mathcal{F}_{A_1}(Z) \), the filtering consists in deleting the value \( v_i \) from the domain of \( z_i \). Therefore \( \forall d_i^{A_1} \in D_{X - Y}^{A_1}, d_i^{A_2} = d_i - \{ v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{A_1}(Z) \} \). Likewise, we have \( \forall d_i^{A_2} \in D_{X - Y}^{A_2}, d_i^{A_2} = d_i - \{ v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{A_2}(Z) \} \). Yet, as \( \mathcal{F}_{A_1}(Z) \subseteq \mathcal{F}_{A_2}(Z) \), we have \( \{ v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{A_1}(Z) \} \subseteq \{ v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{A_2}(Z) \} \). So, we have \( d_i^{A_2} \subseteq d_i^{A_1} \).
Let \( e_{jk} \in C_Z \) be a constraint between two variables \( x_j, x_k \in Z \) and let \( r_{e_{jk}}^{A_1} \) and \( r_{e_{jk}}^{A_2} \) be the associated relations obtained after the resulting filtering of \( A_1 \) and \( A_2 \). According to the definition 4, we have \( r_{e_{jk}}^{A_1} = r_{e_{jk}} \cap (d_{e_{jk}}^{A_1} \times d_{e_{jk}}^{A_1}) \) and \( r_{e_{jk}}^{A_2} = r_{e_{jk}} \cap (d_{e_{jk}}^{A_2} \times d_{e_{jk}}^{A_2}) \). Furthermore, \( d_{e_{jk}}^{A_2} \times d_{e_{jk}}^{A_2} \subseteq d_{e_{jk}}^{A_1} \times d_{e_{jk}}^{A_1} \) since \( \forall z_i \in Z, d_{e_{jk}}^{A_1} \subseteq d_{e_{jk}}^{A_1} \). So, \( r_{e_{jk}}^{A_2} \subseteq r_{e_{jk}}^{A_1} \).

We compare now the set of solutions of two subproblems induced by the same set of variables but with any different filtering.

**Property 3** If \( F_{A_1}(Z) \subseteq F_{A_2}(Z) \), then we have \( \text{Sol}(P_{A_1}(Z)) \subseteq \text{Sol}(P_{A_2}(Z)) \) and \( |\text{Sol}(P_{A_1}(Z))| \leq |\text{Sol}(P_{A_2}(Z))| \).

**Proof:** Let \( S \) be a solution of \( P_{A_2}(Z) \). Let us prove that \( S \) is solution of \( P_{A_1}(Z) \). By definition, \( S \) is a consistent assignment on the set of variables \( Z \) such that \( \forall c_{jk} \in C_Z, S[c_{jk}] \in r_{e_{jk}}^{A_2} \). Yet, \( F_{A_1}(Z) \subseteq F_{A_2}(Z) \) and according to property 2, \( \forall c_{jk} \in C_Z, r_{e_{jk}}^{A_2} \subseteq r_{e_{jk}}^{A_1} \). So, \( \forall c_{jk} \in C_Z, S[c_{jk}] \in r_{e_{jk}}^{A_1} \). So, \( S \) is a solution of \( P_{A_1}(Z) \). Therefore, \( \text{Sol}(P_{A_1}(Z)) \subseteq \text{Sol}(P_{A_2}(Z)) \) and so \( |\text{Sol}(P_{A_1}(Z))| \leq |\text{Sol}(P_{A_2}(Z))| \). □

In the next corollary, we present the specific case where \( A_2 \) is an extension of \( A_1 \).

**Corollary 1** If \( A_2[Y_1] = A_1 \), then \( \text{Sol}(P_{A_2}(Z)) \subseteq \text{Sol}(P_{A_1}(Z)) \) and \( |\text{Sol}(P_{A_2}(Z))| \leq |\text{Sol}(P_{A_1}(Z))| \).

**Proof:** Let us observe that the assumption \( A_2[Y_1] = A_1 \) implies \( F_{A_1}(Z) \subseteq F_{A_2}(Z) \). So, using the property 3, we have \( \text{Sol}(P_{A_2}(Z)) \subseteq \text{Sol}(P_{A_1}(Z)) \) and \( |\text{Sol}(P_{A_2}(Z))| \leq |\text{Sol}(P_{A_1}(Z))| \). □

We will then exploit these properties and corollary in order to decide whether structural (no)goods can be reused validly. But, first, we remind the notion of structural (no)good which is used in the BTD algorithm [2].

**Definition 5** Given a cluster \( E_i \) and \( E_j \) one of its sons, a **good** (resp. **nogood**) of \( E_i \) with respect to \( E_j \) is a consistent assignment \( A \) on \( S_j = E_i \cap E_j \) such that \( A \) can (resp. cannot) be extended to a consistent extension of \( A \) on \( \text{Desc}(E_j) \).

We see now the cases where the (no)goods for the subproblem \( P(X-Y) \) can stay valid if we change the assignment on \( Y \).

**Theorem 1** If \( F_{A_i}(X-Y) \subseteq F_{A_2}(X-Y) \) and \( \text{ng}(S_j) \) is a nogood for the problem \( P_{A_i}(X-Y) \) then \( \text{ng}(S_j) \) is a nogood for \( P_{A_2}(X-Y) \).

**Proof:** \( TD \) is a tree-decomposition associated to the CSPs \( P_{A_i} \) and \( P_{A_2} \). Furthermore, \( S_j = E_i \cap E_j \) is a separator of \( TD \) between the cluster \( E_i \) and one of its sons \( E_j \). We know that \( \text{ng}(S_j) \) is a nogood for \( P_{A_i}(X-Y) \), so \( |\text{Sol}(P_{A_i}(\text{Desc}(E_j)))| = 0 \) if \( S_j \) is assigned by this nogood. Yet, \( F_{A_i}(X-Y) \subseteq F_{A_2}(X-Y) \) and \( \text{Desc}(E_j) \subseteq X-Y \). So, we can apply the property 3 which results \( |\text{Sol}(P_{A_2}(\text{Desc}(E_j)))| \leq |\text{Sol}(P_{A_i}(\text{Desc}(E_j)))| \) and so \( |\text{Sol}(P_{A_2}(\text{Desc}(E_j)))| = 0 \). Therefore, \( \text{ng}(S_j) \) is a nogood for \( P_{A_2}(X-Y) \). □

The previous theorem lays a condition (inclusion) on the set of values which are filtered to deduce the validity of a nogood already recorded. However, from an algorithmic and practical viewpoint, exploiting this theorem may lead to an expensive check (with respect to time). Hence, in the next corollary, we propose a restriction on the resulting filtering of the two assignments.

**Corollary 2** If \( A_2[Y_1] = A_1 \) and \( \text{ng}(S_j) \) is a nogood for the problem \( P_{A_i}(X-Y) \) then \( \text{ng}(S_j) \) is a nogood for \( P_{A_2}(X-Y) \).

**Proof:** By observing that \( A_2[Y_1] = A_1 \) implies \( F_{A_1}(Z) \subseteq F_{A_2}(Z) \), we apply the theorem 1. □

Then, we are interested in preserving the validity of goods.

**Theorem 2** If \( F_{A_2}(\text{Desc}(E_j)) \subseteq F_{A_1}(\text{Desc}(E_j)) \) and \( g(S_j) \) is a good for the problem \( P_{A_i}(X-Y) \) then \( g(S_j) \) is a good for \( P_{A_2}(X-Y) \).
had been recorded. In order to check easily this property, we need to record the extension of the good on the returns true.

Proof:
As ChoiceBTD(V) or V = ∅ then
G ← ∅ ; NG ← ∅
Cons ← BTD-gen(∅, E1, V E1, NGp, Gp, NG, G)
Gp ← Gp ∪ G ; NGp ← NGp ∪ NG
if Cons and V = ∅ then
Choose x1 ∈ V ; d1 ← D1 ; Cons ← false
while d1 ≠ ∅ and ¬Cons do
Choose v ∈ d1 ; d1 ← d1 − {v}
if Filtering(A ∪ {x1 ← v}, V − {x1}, NGp, Gp) then
Cons ← CC-BTD-gen(A ∪ {x1 ← v}, V − {x1}, NGp, Gp)
Unfiltering(A, x1)
return Cons

All these properties can be applied by the CC-BTD-gen algorithm to deduce the informations remaining true between different calls to BTD. For that, Y will be the cutset and so X − Y the variables belonging to the associated tree-decomposition. The theorem 1 allows to conclude that considering two partials assignments A1 and A2 on the cutset such that A2 filters at least the same values as A1, then the nogoods recorded by BTD on A1 stay valid on A2. However, due to the limited memory space, we cannot record the effects of resulting filtering generated by each consistent partial assignment on the cutset. Therefore, we exploit the corollary 2 which allows to record and reuse the nogoods in the case where we extend a consistent partial assignment of cutset. Likewise, for the reuse of goods, we keep all recorded goods and check their validity when we use them. In the next section, we describe and study the CC-BTD-gen algorithm.

3 A generalization of Cyclic-Clustering

The CC-BTD-gen algorithm (algorithm 1) relies on a cutset and a tree-decomposition of the constraint graph. The tree-decomposition and the cutset can be computed thanks to any method, and so are not necessarily related to the TIS notion, unlike in Cyclic-Clustering. The CC-BTD-gen algorithm consists in assigning consistently the variables of the cutset while checking, thanks to a dedicated version of BTD, whether the current partial assignment can be extended consistently on the tree-decomposition part. As this check can be expensive, after having assigned a value to a variable of the cutset, CC-BTD-gen decides thanks to the heuristic function ChoiceBTD if it must be performed or not. If BTD returns true, CC-BTD-gen keeps on the search on the cutset. Otherwise, it tries a new value for the current variable (if any) or a backtrack occurs. We iterate this process until a solution is found (i.e. a consistent assignment of the cutset which can be consistently extended to the tree-decomposition part) or the whole search space is explored.

First, in order to be able to reuse (no)goods recorded by different executions of BTD, we propose a variant of BTD, called BTD-gen (algorithm 2), which implements the properties highlighted in the previous section. BTD-gen only differs from BTD in its ability to exploit (no)goods recorded during previous calls to BTD-gen. So, it has two additional parameters, namely the set Gp of goods and the set NGp of nogoods recorded by previous calls to BTD-gen while G and NG denote respectively the set of goods and nogoods recorded by the current execution to BTD-gen. As we keep all the goods recorded previously, some of them cannot be reuse validly into some calls to BTD-gen. Therefore, before reusing such a good, BTD-gen must first check its validity for the current problem in order to respect the theorem 2. This test is performed by the function CheckGood (algorithm 3). This function returns true whether each variable of the descent of E is assigned with the value it had when the good g had been recorded. In order to check easily this property, we need to record the extension of the good on the
remaining variables of the cluster. Like BTD, BTD-gen returns the consistency of the subproblem associated to the tree-decomposition TD and rooted in the cluster \( E_i \).

This dedicated version of BTD is exploited in CC-BTD-gen to check if the current partial assignment on the cutset can be extended consistently on the tree-decomposition part of the problem. If BTD-gen(0, \( E_1, V_{E_1}, NG_p, G_p, NG, G \)) returns \textit{false}, then CC-BTD-gen tries another value for the last assigned variable in the cutset (if any) or a backtrack occurs. Otherwise, it keeps on the search by assigning a new variable of the cutset. In both cases, the set \( G \) of goods recorded by BTD-gen is added into the set \( G_p \). The process is similar for the nogoods except that \( NG_p \) cannot be modified out of the current call to CC-BTD-gen. In other words, when we come back from a call to CC-BTD-gen, we forget the nogoods recorded, during this call, by BTD-gen in order to respect the corollary 2.

Finally, in algorithm 1, the Boolean heuristic function \textit{ChoiceBTD} defines, after each assignment of a variable in the cutset, if BTD-gen must be called or not. If it returns \textit{false} and some cutset’s variables are not yet assigned, CC-BTD-gen tries to assign one of them with Forward-Checking algorithm (lines 7-12). If \textit{ChoiceBTD} returns \textit{true}, we run BTD-gen and we keep the new goods and nogoods recorded (lines 3-5). Note that this heuristic can be entirely dynamic since it can decide to call BTD-gen anytime during the assignment of the cutset.

Now, we illustrate the CC-BTD-gen algorithm with an example. Let us consider the constraint graph of Figure 1(a) and a possible tree-decomposition with 5 connected components and a cutset with 2 variables \((y_1 \text{ and } y_2)\) as depicted in Figure 1(b). CC-BTD-gen assigns some variables of the cutset. For instance, if it only assigns \( y_1 \), the filtering of FC can reduce the domains of the unassigned neighboring variables of \( y_1 \), namely \( x_1, x_2, x_4, x_5, x_7 \) and \( x_{16} \). Next, the \textit{ChoiceBTD} heuristic can decide to solve the tree-decomposition part of the problem with
BTD-gen. If BTD-gen returns false then CC-BTD-gen will change the assignment on \( y_1 \). Otherwise CC-BTD-gen will assign the variable \( y_2 \) of the cutset. In this case, if it succeeds in assigning consistently \( y_2 \), a new call to BTD-gen is performed since the cutset is entirely assigned. If BTD-gen returns true then the CSP is consistent. Otherwise CC-BTD-gen looks for a new assignment for \( y_2 \).

**Theorem 3** BTD-gen is sound, complete and finishes.

**Proof:** BTD is sound, complete and finishes [2]. As BTD-gen only differs from BTD in exploiting (no)goods recorded during previous calls to BTD-gen, we only have to prove that the use of these (no)goods is valid. If we denote \( Y \) the cutset, \( X - Y \) corresponds to the variables of the tree-decomposition. According to corollary 2, if \( A \) is a consistent assignment on \( Y \) then each nogood for \( P_A(X - Y) \) is a nogood for \( P_{A'}(X - Y) \) where \( A' \) is a consistent extension of \( A \) on the cutset. Moreover, when it backtracks from \( A' = A \cup \{x_k \leftarrow w\} \) to \( A \), CC-BTD-gen forgets the nogoods recorded since \( x_k \) is assigned to \( w \). So the corollary 2 holds and using a nogood from \( NG_p \) is valid. Regarding the use of the goods of \( G_p \), if the function \( CheckGood \) returns true, it ensues that the use of the good is valid thanks to theorem 2. As BTD-gen uses only valid (no)goods, it is sound, complete and finishes.

**Theorem 4** CC-BTD-gen is sound, complete and finishes.

**Proof:** In outline, CC-BTD-gen solves the cutset part of the problem with a modified version of FC and the tree-decomposition part with BTD-gen when all the variables in the cutset are assigned. The modified version of FC consists in using FC and, when ChoiceBTD returns true, BTD-gen. This call to BTD-gen can be seen as an additional consistency check (we check whether the current assignment of the cutset can be extended consistently to the variables of the tree-decomposition). As BTD-gen and FC are sound, complete and finish, it is the same for CC-BTD-gen.

In the following theorems, \( n \) denotes the number of variables of CSP, \( m \) the number of constraints, \( d \) the size of the largest domain, \( k \) the size of the cutset, \( w \) the width of the considered tree-decomposition, and \( s \) the size of the largest intersection between two clusters.

**Theorem 5** BTD-gen has a time complexity in \( O(n(n+m)d^{w+1}) \) and a space complexity in \( O(nwd^s) \).

**Proof:** The proof is similar to one of BTD [2]. We have just to take into account the additional time required for checking the validity of goods of \( G_p \) and the additional space required for recording the extension of each good on the related cluster.

**Theorem 6** CC-BTD-gen has a time complexity in \( O(n(n+m)d^{w+k+2}) \) and a space complexity in \( O(nwd^s) \).

**Proof:** In the worst case, CC-BTD-gen calls BTD-gen for each partial assignment of the cutset. As the number of partial assignments on the cutset is bounded by \( d^{k+1} \), CC-BTD-gen has a time complexity in \( O(n(n+m)d^{w+1}d^{k+1} + nm.d^{k+1}) \), i.e. in \( O(n(n+m)d^{w+k+2}) \). Its space complexity only depends on one of BTD-gen. So CC-BTD-gen has a space complexity in \( O(nwd^s) \).

### 4 Experimental results

In this section, we assess the practical interest of CC-BTD-gen with respect to the classical Cyclic-Clustering algorithms (namely CC-BTD1 and CC-BTD2), BTD and FC. The tests are performed on random structured problems. More exactly, we use a generator of binary CSPs which constructs a triangulated constraint graph. Then, it constructs another graph which represents the cutset. Finally, it adds some constraints in order to link these two graphs. We need ten parameters to generate this kind of problems: \( n \) the number of variables of the triangulated graph, \( d \) the size of the largest domain, \( r \) the size of the largest clique of triangulated graph, \( t_1, t_2 \) and \( t_3 \) the number of forbidden tuples by the constraints respectively between two variables into the triangulated part, between two variables of the cutset and between a variable of the triangulated part and one of the cutset, \( s \) the size of the largest separator, \( k \) the size of the cutset, \( e_1 \) the number of constraints into the cutset and \( e_2 \) the number of constraints between the cutset and the triangulated graph.
So, a class of problems is defined by these ten parameters. For each considered class, the number of consistent problems is approximately equal to the number of inconsistent ones. In our experiments, BTD and BTD-gen exploit FC to solve the clusters. For ordering variables in FC or inside a cluster, we use the well-known \( \text{dom/deg} \) heuristic which first chooses the variable which minimizes the ratio \( \frac{|d_x|}{\Gamma_x} \) with \( d_x \), the current domain of \( x \), and \( \Gamma_x \), the set of the variables sharing a constraint with \( x \). The tree-decomposition is computed thanks to the well-known triangulation heuristic \( \text{min-fill} \) or thanks to the merging method proposed in [9]. This latter method computes a tree-decomposition from a cutset and a triangulated subgraph (TIS), with the aim of fully exploiting the informations recorded during the search, unlike CC-BTD or CC-BTD-gen. Regarding the computation of the cutset and the tree-decomposition part, we know that for efficiency reasons, the methods based on the classical Cyclic-Clustering approach, like CC-BTD, need a cutset with few solutions. Unfortunately, we do not have really any method to recognize such a structure. We exploit the method of Balas and Yu [10] which computes a TIS from which the cutset is deduced. We also consider the generated cutset and tree-decomposition part in order to assess the behavior of the used algorithms when the structure is ideally detected. Regarding CC-BTD-gen and, in particular, the function \( \text{ChoiceBTD} \), we have tested many heuristics called \( H_i \) with \( i = \{1, ..., k\} \). Each heuristic \( H_i \) decides to solve the tree-decomposition part if, since the last call to BTD-gen, at least \( i \) variables of the cutset have been consistently assigned and if at least a value has been deleted from a domain of a variable belonging to the tree-decomposition part. Initially, each heuristic performs a preliminary call to BTD-gen exactly like CC-BTD-gen, at least 1,200 s. In this case, as we do not know the real runtime, we add penalty of 20 minutes, for each unsolved instance. Table 1 presents the classes and the corresponding structural parameters while Tables 2 and 3 provide the runtime of the considered methods.

First, if we compare the results obtained by CC-BTD-gen according to the heuristic \( H_i \) used, we can generally observe that the runtime and the number of instances which cannot be solved within the time limit significantly increase with \( i \). In particular, while \( H_1 \) and \( H_2 \) seem to be close theoretically, in practice, CC-BTD-gen performs significantly better with \( H_1 \) than with \( H_2 \). From a theoretical viewpoint, in the worst case, if \( i < j \) then \( H_i \) is

---

**Table 1**: Parameters of the different classes and the corresponding structural parameters for the different considered algorithms. The cutset and the tree-decomposition part are computed from the algorithm of Balas and Yu (BY) or are ones used by the instance generator (CC-gen).

<table>
<thead>
<tr>
<th>Classes</th>
<th>FC</th>
<th>BTD</th>
<th>CC-BTD</th>
<th>CC-BTD-gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a))</td>
<td>( &gt;_1 )</td>
<td>( \geq 25 )</td>
<td>( \leq 10 )</td>
<td>( \leq 25 )</td>
</tr>
<tr>
<td>((b))</td>
<td>( &gt;_1 )</td>
<td>( \geq 260 )</td>
<td>( \geq 19 )</td>
<td>( \geq 25 )</td>
</tr>
<tr>
<td>((c))</td>
<td>( &gt;_1 )</td>
<td>( \geq 260 )</td>
<td>( \geq 264 )</td>
<td>( \geq 264 )</td>
</tr>
<tr>
<td>((d))</td>
<td>( &gt;_1 )</td>
<td>( \geq 195 )</td>
<td>( \geq 993 )</td>
<td>( \geq 195 )</td>
</tr>
<tr>
<td>((e))</td>
<td>( &gt;_1 )</td>
<td>( \geq 317 )</td>
<td>( \geq 986 )</td>
<td>( \geq 317 )</td>
</tr>
<tr>
<td>((f))</td>
<td>( &gt;_1 )</td>
<td>( \geq 605 )</td>
<td>( \geq 1088 )</td>
<td>( \geq 953 )</td>
</tr>
</tbody>
</table>

**Table 2**: Runtime (in seconds) for the different methods on the considered classes. The cutset and the tree-decomposition part are ones used by the instance generator (CC-gen).

8
likely to call BTD-gen more often than \( H_j \). However, in practice, we remark that checking often the consistency of the tree part allows to prune the search space related to the cutset solving. For this reason, \( H_1 \) gives the best times compared to other heuristics and to CC-BTD. CC-BTD-gen with \( H_k \) and CC-BTD2 performed the same processing: they assign consistently all the variables of cutset before to solve the tree-decomposition part with BTD-gen. CC-BTD-gen with \( H_1 \) differs from CC-BTD2 only in using the goods recorded during the different calls to BTD-gen which is not possible for CC-BTD2. We can see that CC-BTD-gen with \( H_k \) and CC-BTD2 have a similar behavior, even if when we use the Balas and Yu’s method, CC-BTD-gen performs slightly better. More precisely, we have observed that CC-BTD-gen with \( H_k \) develops less nodes than CC-BTD2 when solving the tree-decomposition part. However, the gain in nodes is not sufficient to offset the additional cost of checking the validity of goods. As it was shown in [3], the use of nogoods recorded during the preliminary call to BTD makes it possible to solve more instances which explains the large deviation between CC-BTD1 and CC-BTD2. Concerning BTD, in most cases, the structure of problems is not well-suited for tree-decomposition based methods. This is the case, for instance, when the cutset is sparse and when the cutset and the tree-decomposition part are weakly connected. It ensues that BTD has a behavior significantly worse than most of the other structural methods except sometimes CC-BTD1 when the tree-decomposition used by BTD relies on the merging algorithm. Regarding FC, we can remark that it does not succeed in solving all the instances and that its runtime is greater than one obtained by CC-BTD-gen with \( H_1 \).

Finally, we can note that CC-BTD1 and CC-BTD-gen perform better when they exploit the cutset and the tree-decomposition part used during the generation step than when they use ones produced thanks to the algorithm of Balas and Yu. Of course, such a result was foreseeable. But, it allows us to point out the lack of methods for computing both relevant cutset and tree-decomposition with respect to CSP solving, what is clearly a main problem for such structural methods. If the Balas and Yu’s algorithm allows to compute a triangulated subgraph from which a cutset is then deduced, unfortunately, it does not take into account the solving. For instance, for the random structured instances we use, the value of structural parameters \( k, w \) and \( s \) obtained by applying the Balas and Yu’s algorithm are close to ones used for producing these instances. However, in practice, the solving produced from the cutset and the tree-decomposition produced with the Balas and Yu’s algorithm is less efficient. Moreover, in practice, this algorithm often leads to large cutsets with trivial tree-decompositions. For example, we have experimented it on some frequency assignments problems (namely the fapp instances of the last CSP competition [11]). It results that most of the variables are in the cutset and the size of the clusters of the tree-decomposition does not exceed 3. In addition to the Balas and Yu’s algorithm, we have tried many heuristics (which exploit or not the TIS notion) for computing the cutset and the tree-decomposition required by CC-BTD1 and CC-BTD-gen. Unfortunately, none of these heuristics has led to interesting results with respect to the CSP solving yet. So, we think that a study like one proposed in [12] for tree-decomposition must be achieved in order to improve the efficiency of such approaches. Likewise, if the heuristic \( H_1 \) produces good results, it could be interesting to look for more relevant or clever heuristics for ChoiceBTD.

### 5 Conclusions and future works

We have proposed a new method for solving structured CSPs. This method generalizes and improves the Cyclic-Clustering approach [1]. More precisely, it exploits a cutset and a tree-decomposition whose computation is made
independent of the notion of triangulated induced subgraph, what brings more freedom in a crucial step of the method. Then, CC-BTD-gen can check whether the current assignment on the cutset can be consistently extended on the tree-decomposition part, even if all the variables of the cutset are not assigned yet. By so doing, it has a more global view of the problem than CC-BTD
i. Finally, it exploits a dedicated version of BTD which implements some properties which make it possible to exploit some (no)goods recorded during previous calls to BTD and so to avoid more redundancies in the search. Our preliminary experiments show the practical interest of our approach. Namely, CC-BTD-gen often outperforms CC-BTD
i.

In the CSP framework, few works related to cutset have been achieved. [13] presents a method close to Cyclic-Clustering except that the tree-decomposition part is solved with Adaptive-Consistency. Likewise, in our knowledge, the computation of both relevant cutset and tree-decomposition with respect to CSP solving has not been studied yet. Such a work, like [12] for tree-decomposition, must be performed to improve the efficiency of such approaches. It could turn to be very useful for solving efficiently structured real-world instances. Finally, exploiting dynamic cutset and tree-decomposition could be promising.

Bibliography