

A compression algorithm to improve generalized hypertree decomposition-based solving approaches for large extensional non binary CSP

Kamal AMROUN¹, Zineb HABBAS², and Wassila AGGOUNE-MTALAA³

¹ LIMED, Faculté des Sciences Exactes, Université de Bejaia, 06000 Bejaia, Algeria
`k.amroun25@yahoo.fr`

² LCOMS, University of Lorraine, France
`zineb.habbas@univ-lorraine.fr`

³ Public Research Centre Henri Tudor, Luxembourg
`wassila.mtalaa@tudor.lu`

Abstract. Many approaches exploiting Generalized Hypertree Decomposition are proposed in the literature for solving extensional Constraint Satisfaction Problems on finite domains. The most recent one, called FC-GHD, combines the merits of an enumerative search algorithm which is memory efficient with those of the Hypertree Decomposition which is time efficient. Although FC-GHD is proved to be efficient, some large instances could remain unresolved because of the memory fault problem. This present article proposes to enhance the FC-GHD method with the notion of compressed table constraints to deal with memory explosion problems.

Keywords : Constraint satisfaction problems, Generalized Hypertree Decomposition, Compression, Backtracking.

1 Introduction

Many important real-world problems can be represented as Constraint Satisfaction Problems (CSPs) on Finite Domains. CSP is NP-complete. The usual method that guarantees to find a solution if it exists is based on a backtracking search. This method has been combined with different filtering techniques and with different heuristics for choosing the next variable to be instantiated or the next value for a given variable. Unfortunately, this approach has an exponential time complexity. In order to provide better theoretical complexity bounds, structural decomposition methods have received considerable interest these last decades [4, 2, 5, 12, 7]. The computation time of a solving algorithm informed by a structural decomposition is exponential with respect to the tree-width or

Proceedings of the 21st RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2014).
Vienna, Austria, July 17–18, 2014.

hypertree-width and the memory space consumption of this algorithm is unfortunately also exponential with respect to these bounds while it is linear in terms of the size of the problem for direct enumerative search algorithms. This memory space consumption problem is a real drawback which prevents the practical efficiency of structural decomposition methods to solve a CSP. To cope with this main drawback, some propositions have been done. In [14], the authors have proposed an algorithm called *w-CDBT* combining both merits of Constraint-Directed BackTracking and the structural decomposition method *w-graph*. To date, this method has not been exploited in practice. In [9] the authors have proposed the BTB (Backtracking with tree decomposition) method, an enumerative search algorithm guided by an order induced by a Tree Decomposition. In [8], a Forward Checking algorithm guided by an order induced by Generalized Hypertree Decomposition (*FC-GHD*) was proposed. The idea behind this approach is that rather than solving the problem by instantiating the variables one by one (FC is variable oriented), the *FC-GHD* algorithm solves one by one the subproblems resulting from the Generalized Hypertree Decomposition (*FC-GHD* is cluster oriented). To improve the *FC-GHD* algorithm, two extended variants were introduced. The first one, called *FC-GHD+NG*, explores the use of structural *NoGoods* [9]. The second one called *FC-GHD+NG+DR* extends the previous one by considering a *Dynamic Reordering* (DR) of the subtrees of the tree, in order to detect inconsistencies earlier.

Although the experimental results obtained with all these algorithms are very encouraging, some large instances remain unsolvable again for memory fault reasons. The main objective of this paper is to address this problem. To this aim a new version of the recent approach is proposed. It uses the compressed representation of relations originally presented in [10]. This gives rise to three extended algorithms called *CFC-GHD*, *CFC-GHD+NG* and *CFC-GHD+NG+DR* where C refers to a compressed version. Using a compressed representation of constraint tables requires naturally to extend all the procedures corresponding to the basic operations used by enumerative search algorithms such as backtracking, filtering and nogoods. It should be noted that the notion of compressed tuple does not correspond to a simple tuple but to a meta-tuple or a set of tuples. One of the main contributions of this paper lies in the extension of these operations and their integration in different algorithms. The present paper presents formally the three extended algorithms. In order to be concise, only the experimental results obtained with *CFC-GHD+NG+DR* the enhanced version of *FC-GHD+NG+DR* are presented. Indeed *FC-GHD+NG+DR* was proved experimentally to be the best performing algorithm in [8].

The rest of this article is organized as follows. The useful technical background is presented in Section 2. Section 3 presents the CFC-GHD algorithm while Section 4 is devoted to its improved variants: the CFC-GHD+NG using *Nogoods* and the FC-GHD+NG+DR using *Nogoods* and *Dynamic Reordering* of the subtrees of the tree. Section 5 gives the results on the time and space complexities of the algorithm. Experiments performed in order to validate our approach are presented in Section 6. Finally, this paper is concluded by Section 7.

2 Background

This section presents some definitions on Constraint Satisfaction Problems and the techniques to solve these problems by using structural decompositions.

2.1 Constraint Satisfaction Problems

Definition 1. (CSP from [13])

A *Constraint Satisfaction Problem (CSP)* is defined as a triple $P = \langle X, D, C \rangle$ where:

- $X = \{X_1, \dots, X_n\}$ is a set of n variables.
- $D = \{D_1, \dots, D_n\}$ is a set of finite domains. Each variable X_i takes its value from its corresponding domain D_i .
- $C = \{C_1, \dots, C_m\}$ is a set of m constraints. Each constraint C_i is a pair $(Scope(C_i), Rel(C_i))$ where $Scope(C_i) = \{X_{i_1}, \dots, X_{i_k}\}$ is a list of variables called the *scope* of C_i ($\{X_{i_1}, \dots, X_{i_k}\} \subseteq X$), and $Rel(C_i) \subseteq D_{i_1} \times \dots \times D_{i_k}$ (subset of the cartesian product) is the relation of C_i that specifies the allowed combinations of values for the variables in $Scope(C_i)$.

The size of $Scope(C_i)$ is called the *arity* of the constraint C_i . Binary CSPs are CSPs with constraints involving at most two variables. Constraints of arity n greater than 2 are called *non binary* or *n-ary*. A CSP with at least one n -ary constraint is called *non binary* or *n-ary CSP*. A tuple $\tau \in Rel(C_i)$ is a list of values $(a_{i_1}, \dots, a_{i_k})$ where $k = |Scope(C_i)|$ such that $a_{i_j} \in D_{i_j} \forall j = 1, \dots, k$.

Definition 2. (Solution of a CSP)

A solution of a CSP is a complete assignment $\mathcal{A} = (a_1, a_2, \dots, a_n)$ of values to all the variables in X , such that $\mathcal{A}[Scope(C_i)] \in Rel(C_i) \forall C_i \in C$. $(\mathcal{A}[Scope(C_i)])$ is the projection of the tuple solution $\mathcal{A}$ on the variables of $Scope(C_i)$

Definition 3. (Hypergraph of a CSP)

A hypergraph is a pair $\mathcal{H} = \langle V, H \rangle$ which consists of a set of vertices $V = \{v_1, \dots, v_n\}$ and a set of subsets of these vertices $H = \{h_1, \dots, h_m\}, h_i \subseteq V$ called hyperedges. The hyperedges differ from simple edges since they may connect more than two vertices. Note that any graph may be regarded as a hypergraph whose hyperedges connect two vertices.

The constraint hypergraph associated with a CSP instance $P = \langle X, D, C \rangle$ is the hypergraph $\mathcal{H} = \langle V, H \rangle$ where the set of vertices V is the set of variables X and the set of hyperedges H corresponds to the set of constraint scopes in C .

For any subset of hyperedges $K \subseteq H$, let us denote by $var(K) = \bigcup_{e \in K} e$ the set of the vertices occurring in the hyperedges of K , and let $var(\mathcal{H}) = V$ and $edges(\mathcal{H}) = H$ be the set of vertices of $\mathcal{H}$ and the set of hyperedges of $\mathcal{H}$ respectively.

2.2 Generalized Hypertree Decomposition

Gottlob et al.[6] proposed a new decomposition method called generalized hypertree decomposition and showed that it can be applied to CSPs.

Definition 4. (Hypertree)

A hypertree for a hypergraph $\mathcal{H} = \langle V, H \rangle$ is a triple $\langle T, \chi, \lambda \rangle$. $T = (N, E)$ is a rooted tree where N is a finite set of nodes (we call the vertices of T nodes, in order to avoid confusion with the vertices in $\mathcal{H}$), and E is a set of edges of T . χ and λ are labelling functions which associate each node $p \in N$ with two sets $\chi(p) \subseteq \text{var}(\mathcal{H})$ and $\lambda(p) \subseteq \text{edges}(\mathcal{H})$. If $T' = (N', E')$ is a subtree of T , we define $\chi(T') = \bigcup_{v \in N'} \chi(v)$ and $\lambda(T') = \bigcup_{v \in N'} \lambda(v)$. We denote the set of nodes N of T by $\text{nodes}(T)$ and the root of T by $\text{root}(T)$. $\forall p \in N(T)$, T_p denotes the subtree of T rooted at p . $\text{Parent}(p)$ is the parent node of p in N .

Definition 5. (Generalized Hypertree Decomposition)

A Generalized Hypertree Decomposition of a hypergraph $\mathcal{H} = \langle V, H \rangle$ is a hypertree $GHD = \langle T, \chi, \lambda \rangle$ which satisfies the following conditions:

1. $\forall h \in H$, $\exists p \in N$ such that $h \subseteq \chi(p)$;
2. $\forall v \in V$, the set $\{p \in N \mid v \in \chi(p)\}$ induces a connected subtree of T ;
3. $\forall p \in N$, $\chi(p) \subseteq \text{var}(\lambda(p))$.

The width of a GHD $\langle T, \chi, \lambda \rangle$ is $\text{Max}_{p \in \text{vertices}(T)} |\lambda(p)|$.

The generalized hypertree width $\text{ghw}(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is the minimum width over all its generalized hypertree decompositions. A generalized hypertree decomposition $GHD = \langle T, \chi, \lambda \rangle$ of a hypergraph $\mathcal{H} = \langle V, H \rangle$ is complete if for each hyperedge h of $\mathcal{H}$ there exists $p \in \text{vertices}(T)$ such that the vertices of $h \subseteq \chi(p)$ and $h \in \lambda(p)$.

3 A Compressed Forward Checking algorithm guided by a Generalized Hypertree Decomposition (CFC-GHD)

In order to present formally the CFC-GHD, the following definitions are given.

Definition 6. (Compressed tuple, compressed relation)

Let $P = \langle X, D, C \rangle$ be a CSP, consider C_i a constraint in C with $\text{Scope}(C_i) = \{X_{i_1}, \dots, X_{i_k}\}$ and $\text{Rel}(C_i) = R$ is the relation associated with it. A compressed tuple (ctuple for short) of C_i is a tuple $(D'_{i_1}, \dots, D'_{i_k})$ where $D'_{i_j} \subseteq D_{i_j} \forall j \in \{1, \dots, k\}$. D'_{i_j} is called a cvalue of X_{i_j} . A compressed relation (crelation for short) cR associated with R is a set of ctuples representing all and only all the tuples of R , thanks to the fact that the compression algorithm is correct and complete. Note that this representation has already been used in [11].

Let cR_i be a crelation defined on the scope S_i and let ct_j be a ctuple of cR_i . If $X_k \in S_i$ then $\text{cvalue}(X_k, ct_j)$ denotes the values that are in the cvalue of X_k in ct_j .

Example 1. Let C_1 be a constraint with $\text{Scope}(C_1) = \{X, Y, Z\}$ and let R be the relation associated with C_1 where $R = \{(0, 9, 1), (1, 9, 2), (2, 9, 0), (3, 9, 3), (4, 9, 2), (5, 9, 3), (6, 9, 1), (7, 9, 1), (8, 9, 1), (9, 9, 3), (10, 9, 3)\}$. One compressed relation associated with R can be $cR = \{(\{3, 5, 9, 10\}, \{9\}, \{3\}), (\{0, 6, 7, 8\}, \{9\}, \{1\}), (\{1, 4\}, \{9\}, \{2\}), (\{2\}, \{9\}, \{0\})\}$.

Notice that each ctuple is a compact representation of a set of tuples or classical tuples. The ctuple $(\{3, 5, 9, 10\}, \{9\}, \{3\})$ represents the subset of tuples $\{(3, 9, 3), (5, 9, 3), (9, 9, 3), (10, 9, 3)\}$ of the relation R .

Remark 1. A tuple $t = (v_{i_1}, \dots, v_{i_k})$ belongs to a ctuple $ct = \{D'_{i_1}, \dots, D'_{i_k}\}$ if $v_{i_j} \in D'_{i_j} \forall j \in \{1, \dots, k\}$

Definition 7. (Compressed CSP)

Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, a compressed CSP associated with P is any CSP $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}' \rangle$ such that $|\mathcal{C}| = |\mathcal{C}'|$ and $\forall C_i \in \mathcal{C}, \exists C'_i \in \mathcal{C}' | \text{Scope}(C_i) = \text{Scope}(C'_i)$ and the compressed relation associated with C'_i is a compressed version of $\text{Rel}(C_i)$.

Definition 8. (Compatible compressed tuples)

Let C_1 and C_2 be two constraints with $\text{Scope}(C_1) = S_1$, $\text{Scope}(C_2) = S_2$, $\text{Rel}(C_1) = R_1$ and $\text{Rel}(C_2) = R_2$. Let cR_1 and cR_2 be two crelations associated with R_1 and R_2 respectively. Let ct_1 and ct_2 be two compressed tuples in cR_1 and cR_2 respectively .

- ct_1 is all-compatible with ct_2 if $\forall X_i \in S_1 \cap S_2 \ ct_1[X_i] \subseteq ct_2[X_i]$
- ct_1 and ct_2 are compatible if there are at least two tuples t_1 and t_2 accepted respectively by ct_1 and ct_2 such that $\forall X_i \in S_1 \cap S_2 \ t_1[X_i] = t_2[X_i]$,
- ct_1 and ct_2 are incompatible otherwise.

Definition 9. A sub-problem associated with a node n_i , denoted P_{n_i} , is a compressed CSP defined by the variables in $\chi(n_i)$ and the constraints in $\lambda(n_i)$. Each constraint C_i in $\lambda(n_i)$ is a pair $(\text{Scope}(C_i), cR_i)$ such that cR_i is a compressed version of $\text{Rel}(C_i)$. The current solution for P_{n_i} is noted $\text{Sol}(P_{n_i})$. c'_{sol} is the solution of the sub-problem at parent node of the considered node.

Notation

Let σ be a list of the nodes of a GHD $= \langle T, \chi, \lambda \rangle$ and let n_i be a node in σ . We denote by $\text{prec}(n_i)$ the node before n_i in σ and by $\text{succ}(n_i)$ the node following n_i in σ . Note that $\text{prec}(n_i)$ differs from $\text{Parent}(n_i)$ which is defined as the parent node of n_i in T .

3.1 Presentation of CFC-GHD

The following sections present formally the contribution of this work. Due to a lack of space, some procedures cannot be presented here. However, all the

detailed procedures can be found in [1]. *CFC-GHD* proceeds globally like *FC-GHD*. First, the nodes of T are grouped in a sequential list σ obtained by passing through the tree rooted at n_1 using a depth first preorder. Then the sub-problems are solved one by one. The main difference between *CFC-GHD* and *FC-GHD* comes from the manipulation of compressed relations. *CFC-GHD* is formally described by algorithm 1 where $tempsol_c_n$ is a stack of solutions (or ctuples) (computed for P_{c_n}) that satisfies all the constraints occurring in descendant nodes of c_n .

Algorithm 1 *CFC-GHD*

Input: a complete and compressed $\mathcal{GHD} = \langle T, \chi, \lambda \rangle$ associated to a given CSP
Output: a solution $\mathcal{A}$ of the CSP if it is satisfiable

- 1: $\sigma = (n_1, n_2, \dots, n_{m'})$ /* a node ordering with n_1 the root and each node precedes all its children in σ , m' is the number of nodes */
- 2: $tempsol_c_n \leftarrow \emptyset \ \forall c_n \in \sigma$
- 3: $c_n \leftarrow n_1$
- 4: **while** $c_n \neq \emptyset$ **do**
- 5: **if** $tempsol_c_n \neq \emptyset$ **then**
- 6: $c_{sol} \leftarrow Pop(tempsol_c_n)$
- 7: filt-descendant-nodes $\leftarrow$ false
- 8: **else**
- 9: $c_{sol} \leftarrow Solve_sub_pb(P_{c_n})$
- 10: filt-descendant-nodes $\leftarrow$ true
- 11: **end if**
- 12: **if** c_{sol} is empty **then**
- 13: $BackTrack(c_n)$
- 14: **else**
- 15: **if** filt-descendant-nodes **then**
- 16: $Filter(c_n)$
- 17: **else**
- 18: **if** all-compatible (c'_{sol}, c_{sol}) **then**
- 19: $Filter_children(c_n, c_{sol})$
- 20: $c_n \leftarrow succ(c_n)$
- 21: **else**
- 22: $Const_all_compatible_ctuple(c'_{sol}, c_{sol}, ct'_{comp})$
- 23: Push($ct'_{comp}, tempsol_Parent(c_n)$)
- 24: **end if**
- 25: **end if**
- 26: **end if**
- 27: **end while**
- 28: $\mathcal{A} \leftarrow \bowtie_{i=1}^{i=m'} sol(P_{n_i})$ /* $\bowtie$ denotes the join operation */

The algorithm 1 proceeds as follows. For each node c_n , two cases can be distinguished:

- the stack $tempsol_c_n$ is empty (line 8): No temporary solution is stored in the stack. A new solution must then be found by the function *Solve_sub_pb*

which computes a (another) solution for the current sub-problem. As soon as a new solution c_{sol} is obtained, the procedure **Filter** (see algorithm 2) is invoked. This procedure uses the function **Filter_descendant**, formally de-

Algorithm 2 Procedure *Filter*(c_n)

```

1: if Filter_descendant ( $c_n, c_{sol}$ ) then
2:    $c_n \leftarrow succ(c_n)$ 
3: end if

```

scribed by algorithm 3, to remove from c_{sol} all the classical tuples that have no compatible ones in the ctuples of each crelation at descendant nodes. To this aim, for each descendant node and for each crelation of the sub-problem associated with this node, the function **Filter_descendant** calls the function **compressed-semi-join** (not presented here) which is the classical semi-join operation on compressed relations in order to remove from the ctuples of the first crelation, initialized with c_{sol} all the tuples which have no compatible ones in the tuples represented by the ctuples of the second crelation. If the obtained crelation is empty, then this function returns false and another solution is computed for the sub-problem at the current node c_n . Note that this filtering operation can transform at each iteration a ctuple into a set of ctuples. For example, let us consider the sub-problem at the node n whose current solution is the ctuple $(\{1, 2\}, \{3, 4\})$ which corresponds to the set of classical tuples: $\{(1, 3), (1, 4), (2, 3), (2, 4)\}$. If the filtering operation removes the classical tuple $(1, 3)$, the current solution becomes $\{(\{1\}, \{4\}), (\{2\}, \{3, 4\})\}$. This new set of ctuples must be saved in the stack. Then the head of this stack is moved to c_{sol} . If the solution of the sub-problem at $Parent(c_n)$ c'_{sol} is all-compatible with the solution c_{sol} , then before solving the sub-problem at the next node in σ , all the crelations of the sub-problems at children nodes of c_n are filtered by the procedure **Filter_children** described by algorithm 4. In contrast, if c'_{sol} is not all-compatible with c_{sol} , then a ctuple ct'_{comp} all-compatible with c_{sol} is derived from c'_{sol} and saved in the stack of $Parent(c_n)$.

- the stack $tempsol_c_n$ is not empty (line 4): that means that some temporary solutions saved in the stack are waiting to be checked. In this case, a solution for the sub-problem at c_n is moved from the stack to c_{sol} . If the solution of the sub-problem at $Parent(c_n)$ c'_{sol} is all-compatible with c_{sol} , then before solving the sub-problem at the next node in σ , all the crelations of the sub-problems at children nodes of c_n are filtered by the procedure **Filter_children** described by algorithm 4. In contrast, if c'_{sol} is not all-compatible with c_{sol} , then a ctuple ct'_{comp} all-compatible with c_{sol} is derived from c'_{sol} and saved in the stack of the parent node of c_n .

The procedure **Filter_children** (algorithm 4) proceeds as follows. It calls the procedure **Const_all_compatible_ctuple** in order to obtain from each ctuple

Algorithm 3 Function $Filter_descendant(c_n, c_{sol})$: *boolean*

```

1:  $cR \leftarrow c_{sol}$ 
2: for each node  $n'$  descendant of  $c_n$  do
3:   for each constraint relation  $cR_i$  in  $\lambda(n')$  do
4:      $cR' = compressed\_semi\_join(cR, cR_i)$ ;
5:     if  $cR' = \emptyset$  then
6:       return false
7:     else
8:        $cR \leftarrow cR'$ 
9:     end if
10:  end for
11: end for
12:  $ct \leftarrow cR'$ 
13: while  $ct \neq \emptyset$  do
14:    $Push(ct, tempsol\_c_n)$ 
15:    $ct \leftarrow next(ct)$ 
16: end while
17:  $c_{sol} \leftarrow Pop(tempsol\_c_n)$ 
18: if all-compatible( $c'_{sol}, c_{sol}$ ) then
19:    $Filter\_children(c_n, c_{sol})$ 
20:   return true
21: else
22:    $Const\_all\_compatible\_ctuple(c'_{sol}, c_{sol}, ct'_{comp})$ 
23:    $Push(ct_{comp}, tempsol\_Parent(c_n))$ 
24:   return false
25: end if

```

ct_i of each relation at children nodes a tuple ct_{comp} all-compatible with the current solution c_{sol} of the sub-problem at c_n . Two numbers are then associated to ct_{comp} : **remove**, set to 0, meaning that ct_{comp} is not removed and **label** which is set to c_n , meaning that ct_{comp} is derived from ct_i in order to be all-compatible with the solution of the sub-problem at the node c_n . If ct_{comp} is empty, then **label** of ct_i is set to c_n and **remove** is set to 1 (it is removed).

Finally, if there is no possible solution for the sub-problem at the current node c_n , the algorithm backtracks chronologically to the parent node using the procedure **Backtrack** (see algorithm 5). This procedure restores the removed tuples of the compressed relations of the constraints in the λ label of each child node n of each node between $prec(c_n)$ and $Parent(c_n)$ in σ by using the procedure **Restore_removed_tuples**.

The presentation of the two procedures **Restore_removed_tuples** and **Solve_sub_pb** has been omitted for reasons of space restriction.

Proposition 1. *CFC-GHD is correct.*

Algorithm 4 Procedure *Filter_children*(c_n, c_{sol})

```
1: for each node  $n$  child of  $c_n$  do
2:   for each crelation  $cR_i$  in  $\lambda(n)$  do
3:     for each ctuple  $ct_i$  in  $cR_i$  do
4:        $ct_{comp} \leftarrow \emptyset$ 
5:       if compatible( $ct_i, c_{sol}$ ) then
6:         Const_all_compatible_ctuple( $ct_i, c_{sol}, ct_{comp}$ )
7:       end if
8:       if  $ct_{comp} \neq \emptyset$  then
9:          $ct_i.suiv \leftarrow ct_{comp}$ 
10:         $ct_{comp}.label \leftarrow c_n$ 
11:         $ct_{comp}.remove \leftarrow 0$ 
12:         $ct_{comp}.parent \leftarrow ct_i$ 
13:      else
14:         $ct_i.label \leftarrow c_n$ 
15:         $ct_i.remove \leftarrow 1$ 
16:      end if
17:    end for
18:  end for
19: end for
```

Algorithm 5 Procedure *Backtrack*(c_n)

```
1: if  $c_n = root$  then
2:   problem has no solution
3: else
4:   Restore_removed_ctuples()
5:    $c_n \leftarrow Parent(c_n)$ 
6: end if
```

4 Extensions of *CFC-GHD*

In this section we propose two extensions for *CFC-GHD*.

4.1 *CFC-GHD* with structural nogoods *CFC-GHD+ NG*

Consider a CSP $P = \langle X, D, C \rangle$ and its complete generalized hypertree decomposition $GHD = \langle T, \chi, \lambda \rangle$ and consider n_i and n_j two nodes in T such that n_j is the parent of n_i and $S = (\chi(n_j) \cap \chi(n_i))$ is their separator. If sol_j is the current solution of P_{n_j} then a new solution sol'_j of P_{n_j} where the variables of the separator take the same values, will necessarily lead to the same conflict. Such instantiations of the variables of S may be considered as nogoods for n_j . To avoid such partial solutions, we propose to extend *CFC-GHD* with these structural recording. This version is called *CFC-GHD+NG*. This notion has been introduced in *FC-GHD+NG*. However, in the present work, a nogood is a tuple and not a classical tuple. *CFC-GHD+NG* algorithm corresponds to algorithm 1 where the procedures *Backtrack* and *Filter* are respectively replaced by *Backtrack_NG*

and *Filter_NG*. The following algorithm 6 presents the procedure *Backtrack_NG*. In addition to the previous *Backtrack* procedure, *Backtrack_NG* needs the pro-

Algorithm 6 Procedure *Backtrack_NG*(c_n)

```

1: if  $c_n = n_1$  then
2:   the problem has no solution
3: else
4:   Restore_removed_ctuples()
5:   Add_nogoods(Parent( $c_n$ ))
6:    $c_n \leftarrow \text{Parent}(c_n)$ 
7: end if

```

cedure **add_nogoods**, not presented here, for recording nogoods at the parent of the current node. Note that the nogoods are not recorded at parent node when there is a solution for c_n which is not a nogood and satisfies all the constraints at descendant nodes but only because $\text{Sol}(P_{\text{Parent}(c_n)})$ is not all-compatible with the current solution c_{sol} for c_n . The procedure **Filter_NG** is described by algorithm 7. It is similar to the procedure **Filter**, but it additionally checks if

Algorithm 7 Procedure *Filter_NG*(c_n)

```

1: if nogood then
2:   if Verify( $c_{sol}$ ) then
3:     if filter_descendant_nodes( $c_n$ ) then
4:        $c_n \leftarrow \text{succ}(c_n)$ 
5:     end if
6:   end if
7: else
8:   if filter_descendant_nodes( $c_n$ ) then
9:      $c_n \leftarrow \text{succ}(c_n)$ 
10:  end if
11: end if

```

a current ctuple is not a nogood by calling the function **Verify** described by algorithm 8, and then constructs a stack of ctuples which are incompatible with all the nogoods of the current node.

The function **Verify** (algorithm 8) removes from the current ctuple c_{sol} returned by the function **Solve_sub_pb** each tuple which is a nogood for the current node giving rise to the ctuples in the stack *sol-not-nogoods- c_n* . The set *sol-not-nogoods- c_n* is computed by the function **Inverse-compressed semi-join** not presented here. This procedure takes as input a compressed relation cR_1 and returns another crelation cR_2 organized as a stack. It removes from each ctuple of cR_1 each tuple, compatible with one nogood recorded for the current node.

Algorithm 8 Function $Verify(c_{sol}):boolean$

```
1:  $cR_1 \leftarrow c_{sol}$ 
2:  $cR_2 \leftarrow c_n.nogoods$ 
3:  $sol - not - nogoods_{c_n} \leftarrow \text{Inverse-compressed semi-join}(cR_1, cR_2)$ 
4: if  $c_n.sol - nogoods \neq \emptyset$  then
5:   return true
6: else
7:   return false
8: end if
```

Proposition 2. $CFC - GHD + NG$ is correct.

4.2 $CFC - GHD + NG$ with dynamic reordering of subtrees: $CFC - GHD + NG + DR$

This extension is similar to $FC - GHD + NG + DR$. This variant corresponds to algorithm $CFC - GHD + NG$ where the procedures $Backtrack_NG$ is replaced by $Backtrack_DR$ (see algorithm 9).

Algorithm 9 Procedure $Backtrack_DR(c_n)$

```
1: if  $c_n = n_1$  then
2:   the problem has no solution
3: else
4:   Restore_removed_tuples()
5:   Add_nogoods(Parent( $c_n$ ))
6:   Reorder_hypertree( $c_n, \text{Parent}(c_n)$ )
7:    $c_n \leftarrow \text{Parent}(c_n)$ 
8: end if
```

As compared with the $Backtrack_NG$ algorithm, this new procedure calls the procedure $Reorder_hypertree$ (algorithm 10) which reorders the hypertree such that the next sub-problem to solve after the current one is the sub-problem associated with the node from which the backtracking has occurred.

Note that all the nodes of the subtree T_{c_n} rooted at c_n are moved with c_n and all these nodes are put just after c_r (with respect to the order in T_{c_n}) in σ .

5 Complexity analysis

5.1 Notations

S : the size of the input CSP,
 r : the size of the largest relation,
 w : the width of the decomposition,

Algorithm 10 Procedure *Reorder_hypertree*($c_n, \text{Parent}(c_n)$)

```

1:  $c'_n \leftarrow \text{Parent}(c_n)$ 
2:  $c_n'' \leftarrow c_n$ 
3: while  $c_n'' \neq \emptyset$  do
4:   if  $c_n'' \notin \text{vertices}(T_{c_n})$  then
5:     break
6:   end if
7:    $c_r \leftarrow \text{next}(c_n'')$ 
8:   insert  $c_n''$  just after  $c'_n$  in  $\sigma$ 
9:    $c_n'' \leftarrow c_r$ 
10:   $c'_n \leftarrow \text{next}(c'_n)$ 
11: end while

```

d : the largest size of a cvalue

a : the highest arity.

Proposition 3. *The space complexity of CFC-GHD+NG+DR is in $O(S \cdot (d \cdot a \cdot r)^w)$*

Proposition 4. *The time complexity of CFC-GHD+NG+DR is in $O(S \cdot r^w)$.*

6 Experimental results

In this section we present the experiments carried out in order to assess the interest of *CFC-GHD+NG+DR*.

The *CFC-GHD+NG+DR* algorithm is implemented using the C++ language and the Bucket Elimination (*BE*) algorithm [3] is used to compute a GHD for each CSP instance. *BE* was evaluated in [3] as the best algorithm giving a nearly optimal GHD within a reasonable CPU time. The experiments presented in this section were run on a Core (TM) 2 Duo CPU T5670 @ 1.80 GHZ with 2 GB of RAM under Linux.

In table 1, $|\mathcal{X}|$ is the number of variables, $|\mathcal{C}|$ is the number of constraints, w is the width of the GHD returned by *BE* for the constraint hypergraph and r is the maximum cardinality of the constraint relations. The times are given in seconds.

Definition 10. (*Compression gain*)

Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP instance and let $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}' \rangle$ be one of its compressed representations. Let C_i be a constraint where $\text{Rel}(C_i) = R_i$, cR_i is a compressed relation associated with R_i in P' and let ct_j be a compressed tuple in cR_i . Let us consider the ctuple extension $G_{ct_j} = \sum_{k=1}^{|\text{Scope}(C_i)|} |\text{cvalue}(X_k, ct_j)|$ then we have:

– the size of the compressed relation cR_i : $G_{C_i} = \sum_{j=1}^{|cR_i|} G_{ct_j}$

- **the compression gain:** $G_P = 1 - \frac{\sum_{i=1}^m G_{C_i}}{\sum_{i=1}^m |\text{scope}(C_i)| \times |R_i|}$, where m is the number of constraints.

Remark 2. G_P (also called memory gain) measures the memory savings achieved when the relations are compressed. The closer this gain is to 0, the worse the compression. The optimal gain is reached when the gain $G_P = 1$.

Definition 11. (Compression ratio)

Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP instance and let $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}' \rangle$ be one of its compressed representations. Let R_i be the relation of C_i and let cR_i be one of its compressed versions. The compression ratio ρ measures the performance of the compression algorithm with regard to the total number of tuples covered by the instance.

- **the compression ratio:** $\rho = \frac{\sum_{i=1}^m |cR_i|}{\sum_{i=1}^m |R_i|}$, m is the number of constraints.

Remark 3. ρ measures the degree of compression with regard to constraint relations. The closer this number is to 0, the better the compression. The optimal ratio is reached when $\rho = \frac{|C|}{\sum_{i=1}^m |R_i|}$ meaning that each relation is represented by one tuple.

In the sequel, ρ and G will denote respectively for each instance its compression ratio and compression gain.

6.1 Benchmarks description

The **Modified Renault series** used in this paper is downloaded from the web site <http://www.cril.univ-artois.fr/CPAI08>. This benchmark involves large constraint relations of high arity. The instances of "Modified Renault" series are very structured and involve domains with up to 42 possible values. The largest constraint relation contains 48 721 tuples.

6.2 Comparing *CFC-GHD+NG+DR* with *FC-GHD+NG+DR*

Table 1 compares *FC-GHD+NG+DR* [8] with *CFC-GHD+NG+DR* on some instances of the Modified-Renault series. These results clearly show the benefit of the new approach when seeking for many solutions. Indeed, the time results for the two methods are comparable but with *CFC-GHD+NG+DR* a solution is a tuple which represents many classical solutions. Note that the times reported for *CFC-GHD+NG+DR* include the time of the compression algorithm which is about 3.50 seconds for each instance.

The good results obtained on this benchmark are due to the fact that the compression ratio is very good for all instances.

Problems	Size			w	ρ	G	Time(s)	
	$ \mathcal{X} $	$ \mathcal{C} $	r				CFC-GHD+NG+DR	FC-GHD+NG+DR
<i>ren-1_ext</i>	111	147	48721	2	0.03	0.93	5.33	6.94
<i>ren-4_ext</i>	111	147	48721	4	0.03	0.94	4.51	1.18
<i>ren-5_ext</i>	111	147	48721	3	0.03	0.94	8.70	11.85
<i>ren-6_ext</i>	111	147	48721	3	0.03	0.94	5.15	0.85
<i>ren-7_ext</i>	111	147	48721	3	0.03	0.94	4.74	1.76
<i>ren-8_ext</i>	111	147	48721	3	0.02	0.94	4.21	7.26
<i>ren-9_ext</i>	111	147	48721	3	0.04	0.93	4.76	1.08
<i>ren-11_ext</i>	111	149	48721	3	0.03	0.93	4.34	1.28
<i>ren-13_ext</i>	111	149	48721	3	0.03	0.93	5.52	1.03
<i>ren-14_ext</i>	111	149	48721	3	0.03	0.93	6.45	5.91
<i>ren-15_ext</i>	111	149	48721	3	0.03	0.93	9.01	4.30
<i>ren-17_ext</i>	111	149	48721	4	0.02	0.94	6.15	2.11
<i>ren-18_ext</i>	111	149	48721	3	0.02	0.94	6.27	1.71
<i>ren-22_ext</i>	111	159	48721	3	0.04	0.93	23.31	26.45
<i>ren-23_ext</i>	111	159	48721	3	0.04	0.92	10.34	2.08
<i>ren-24_ext</i>	111	159	48721	4	0.04	0.92	9.07	3.35
<i>ren-25_ext</i>	111	159	48721	3	0.04	0.93	27.48	36.25
<i>ren-27_ext</i>	111	159	48721	4	0.04	0.93	4.97	1.85
<i>ren-28_ext</i>	111	159	48721	3	0.04	0.92	71.35	75.30
<i>ren-29_ext</i>	111	159	48721	4	0.05	0.92	13.29	13.37
<i>ren-32_ext</i>	111	154	48721	5	0.03	0.94	4.20	4.46
<i>ren-34_ext</i>	111	154	48721	4	0.03	0.93	4.84	5.61
<i>ren-35_ext</i>	111	154	48721	4	0.03	0.93	6.50	16.56
<i>ren-36_ext</i>	111	154	48721	4	0.03	0.93	18.04	19.93
<i>ren-38_ext</i>	111	149	48721	4	0.03	0.93	4.46	1.34
<i>ren-41_ext</i>	108	149	48721	4	0.03	0.94	5.42	6.29
<i>ren-42_ext</i>	108	149	48721	3	0.02	0.94	4.44	1.48
<i>ren-43_ext</i>	108	149	48721	3	0.02	0.94	4.34	1.38
<i>ren-44_ext</i>	108	149	48721	4	0.02	0.94	4.13	0.94
<i>ren-45_ext</i>	108	149	48721	5	0.02	0.94	6.28	20.47
<i>ren-46_ext</i>	108	149	48721	4	0.02	0.94	4.17	4.88
<i>ren-47_ext</i>	108	149	48721	4	0.02	0.94	4.18	0.8
<i>ren-48_ext</i>	108	149	48721	4	0.03	0.93	4.41	47.74
<i>ren-49_ext</i>	108	149	48721	4	0.02	0.94	5.16	20.21

Table 1. Comparing *CFC-GHD+NG+DR* with *FC-GHD+NG+DR* on some instances of modified Renault series. ren is an abbreviation for normalized-renault-mod.

7 Conclusion and discussion

In this paper, a new method is presented to address CSPs with large extensional constraint relations. It combines the Generalized Hypertree Decomposition with a compression technique in order to optimize the memory space. This approach aims at enhancing a previous though recent approach without compression to address larger instances CSPs. The compressed problems solving handles at each node level compressed relations or compressed set of tuples. In this paper, the three algorithms deduced from this approach have been formally described. The experimental results are in line with the expectations. Indeed, the compressed approach enables solving the problems in similar computation times or even shorter times in certain cases. In fact, this approach is efficient when searching for several solutions because, as the solution is a compressed tuple, it includes a set of classical tuples. It can be interesting in the future to exploit this approach for tackling optimization problems not especially linked with decision aiding.

References

1. K. Amroun, Z. Habbas, and W. Aggoune-Mtalaa. Technique de compression pour résoudre efficacement les CSPs non binaires via l’hypertree décomposition généralisée, version draft. Rapport de recherche, Université de Lorraine, 2014.
2. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
3. A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decompositions. Technical report, DBAI-R, 2005.
4. E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32:755–761, 1985.
5. G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.
6. G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals and guards : Theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences*, 66:775–808, 2003.
7. M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *Proceedings of SODA 2006*, pages 289–298, 2006.
8. Z. Habbas, K. Amroun, and D. Singer. A cluster-oriented forward checking algorithm based on generalized hypertree decomposition. In *20th RCRA International Workshop on experimental evaluation of algorithms for solving problems with combinatorial explosion, RCRA 2013*, Rome-Italy, 2013.
9. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
10. G. Katsirelos and T. Walsh. A compression algorithm for large arity extentional constraints. In *Proc. of CP’07*, pages 379–393, 2007.
11. G. Katsirelos and T. Walsh. A compression algorithm for large arity extentional constraints. In *Proceedings of CP’07*, pages 379–393, 2007.
12. Z. Miklos. *Understanding tractable decompositions for constraint satisfaction*. PhD thesis, University of Oxford, 2008.
13. U. Montanari. Networks of constraints: Fundamental properties and applications to pictures processing. *Information Sciences*, 7:95–132, 1974.
14. W. Pang and S. D. Goodwin. A graph based backtracking algorithm for solving general CSPs. In *Lecture Notes in Computer Sciences of AI, 2671*, pages 114–128, Halifax, 2003.